UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Racyus Delano Garcia Pacífico

eBPFlow: a Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF

Belo Horizonte
2023

Racyus Delano Garcia Pacífico

eBPFlow: a Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF

Final Version

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marcos Augusto Menezes Vieira
Co-Advisor: José Augusto Miranda Nacif

Belo Horizonte
2023

Pacífico, Racyus Delano Garcia.

P117e eBPFlow: a hardware/software platform to seamlessly offload network functions leveraging eBPF [recurso eletrônico] / Racyus Delano Garcia Pacífico – 2023.
1 recurso online (70 f. il, color.) : pdf.

Orientador: Marcos Augusto Menezes Vieira.
Coorientador: José Augusto Miranda Nacif.

Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.62-69

1. Computação – Teses. 2. Arquitetura de redes de computador – Teses. 3. Linux (Sistema operacional de computador) – Teses. 4. Compiladores (Programas de computador) – Teses. I. Vieira, Marcos Augusto Menezes. II. Nacif, José Augusto Miranda. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. IV.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irenquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

eBPFlow: a Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF

RACYUS DELANO GARCIA PACÍFICO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. MARCOS AUGUSTO MENEZES VIEIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. JOSÉ AUGUSTO MIRANDA NACIF - Coorientador
Departamento de Informática - UFV

PROF. LUIZ FILIPE MENEZES VIEIRA
Departamento de Ciência da Computação - UFMG

PROF. ÍTALO FERNANDO SCOTÁ CUNHA
Departamento de Ciência da Computação - UFMG

PROF. FÁBIO LUCIANO VERDI
Departamento de Ciência da Computação - UFSCar

PROF. RONALDO ALVES FERREIRA
Faculdade de Computação - UFMS

Belo Horizonte, 26 de Outubro de 2023.

Acknowledgments

To God and Jesus Christ. They were my friends in the process of realizing this dream; without them, this would not be possible.

To my wife Pamela and family, thank you for all support and patience during the process.

To my friends and family at Florestal Minas Gerais for their support and guidelines.

To all my friends at UFMG and WINET laboratory for all the support, knowledge, and sharing of ideas.

To the advisors, Marcos Vieira and José Nacif, thank you for the knowledge and opportunities.

To the PPGCC's professors, for the knowledge and orientations.

To the DCC's staff, for all the support and orientations.

To the funding agencies of the research project - CAPES, CNPq, FAPEMIG procs. 02400-18, FAPESP procs. 2020/05183-0., which enabled this study.

"*If you can't fly then run, if you can't run then walk, if you can't walk then crawl, but whatever you do you have to keep moving forward.*"

(Martin Luther king Jr.)

Resumo

Virtualização de funções de rede (NFV) e redes definidas por software (SDN) habilitam flexibilidade e programabilidade no plano de dados. *Offloading* do processamento de pacotes em *hardware* minimiza o uso de núcleos de processamento. No entanto, cumprir requisitos atuais, como alta vazão e baixa latência combinados com *offloading* de funções de rede (NFs) com um plano de dados flexível e programável, ainda é uma tarefa desafiadora. Este trabalho propõe o eBPFlow, uma plataforma para acelerar a computação da rede. Ele baseia-se no eBPF, combinando flexibilidade e capacidade de programação em software com alto desempenho usando uma FPGA. O eBPFlow foi implementado na NetFPGA SUME. Experimentos com NFs foram realizados em um ambiente físico. Nossos resultados mostram que o eBPFlow suporta aceleração de NFs com vazão em taxa de linha, latência entre 20 μs e 40 μs, consumindo pouca energia 22 W. Além disso, o eBPFlow processa 12.05 Mpps mais que o kernel. Ele tem uma vazão de 2.59 Gbps maior que o hXDP, um sistema similar ao eBPFlow.

Palavras-chave: virtualização de funções de rede; redes definidas por software; plano de dados programáveis; filtro de pacote berkeley estendido; netfpga.

Abstract

Network Functions Virtualization (NFV) and Software-Defined Networking (SDN) enable flexibility and programmability at the data plane. In addition, offloading packet processing to a hardware saves processing resources to compute other workloads. However, fulfilling requirements such as high throughput and low latency with a flexible and programmable data plane is challenging. This thesis proposes eBPFlow, a platform for seamlessly accelerating network computation. It builds upon eBPF (extended Berkeley Packet Filter). eBPFlow combines flexibility and programmability in software with high performance using an FPGA. We implemented our system on the NetFPGA SUME, performing tests on a physical testbed. We built a range of NFs, including LPM forwarding, DDoS mitigation, stateful firewall, deep packet inspection, and application layer packet classifier. Our results show that the eBPFlow supports offloading of NFs with throughput at the line rate, latency between 20 μs and 40 μs, communication with host, and consumption of 22 W. Moreover, eBPFlow processes 12.05 Mpps more than the kernel. eBPFlow has a throughput of 2.59 Gbps higher than the hXDP, a system similar to eBPFlow.

Keywords: networking functions virtualization; software defined networking; programmable data plane; extended berkeley packet filter; netfpga.

List of Figures

2.1 Overview of the eBPF machine. . . . 19
2.2 FPGA design. . . . 22
2.3 Overview of the synthesis stages. . . . 23
2.4 Interface and interconnection between masters and slaves. . . . 25
2.5 NetFPGA SUME datapath. . . . 26
2.6 Modules synchronization of the datapath. . . . 27

4.1 eBPFlow design. . . . 32
4.2 eBPF engine design. . . . 34
4.3 Output crossbar design. . . . 36
4.4 Control and Datapath of the eBPF processor. . . . 38
4.5 Data memory division. . . . 40
4.6 Instruction memory - Double Buffer System. . . . 41
4.7 Coprocessor and map table. . . . 42
4.8 Overview of the configuration flow and data processing. . . . 45

5.1 SQL Tautology RegEx. . . . 47
5.2 SQL Sleep RegEx. . . . 48
5.3 Bittorrent Packets RegEx. . . . 48
5.4 Experiments topology. . . . 49
5.5 eBPFlow performance: throughput and latency. . . . 50
5.6 Communication with host. . . . 52
5.7 Systems performance on packet processing. . . . 54
5.8 Throughput: eBPFlow, CPU, and hXDP. . . . 55
5.9 Latency: eBPFlow, CPU, and hXDP. . . . 56

List of Tables

4.1 Metadata: Information retrieved from the input queue of the stored packet in the data memory of the eBPF processor. 35
4.2 Action performed on the packets. 36

5.1 Network Functions implemented on eBPFlow. 46
5.2 Time spent function call on coprocessor. 53

Contents

1 Introduction **12**

1.1 Contextualization . . . 12
1.2 Motivation . . . 13
1.3 Problem definition and objectives . . . 14
1.3.1 Specific goals . . . 14
1.4 Contributions . . . 15
1.5 Challenges to design eBPFlow . . . 16
1.6 Organization . . . 17

2 Approaches and Overview **18**

2.1 Extend Berkeley Packet Filter (eBPF) . . . 18
2.1.1 eBPF verifier . . . 20
2.2 Programmable Data Plane . . . 20
2.2.1 High-Level Languages . . . 21
2.3 Traffic classification . . . 21
2.4 FPGA . . . 22
2.5 NetFPGA . . . 24
2.5.1 Advanced eXtensible Interface (AXI) . . . 24
2.5.2 Datapath . . . 25
2.5.2.1 Synchronization . . . 27

3 Related work **28**

4 eBPFlow architecture **31**

4.1 Design . . . 31
4.1.1 How does eBPFlow work? . . . 32
4.1.2 How does eBPFlow provide flexibility and programmability of the data plane? . . . 33
4.1.3 eBPF engine . . . 34
4.1.4 Metadata . . . 34
4.1.5 Actions . . . 35
4.1.6 Output Crossbar . . . 35
4.2 Implementation . . . 37
4.2.1 Hardware Instance . . . 37

4.2.2 eBPF Processor with Pipeline . . . 37
4.2.3 Data memory (Optimized FIFO) . . . 38
4.2.3.1 eBPF stack . . . 39
4.2.4 Instruction Memory . . . 40
4.2.5 Maps . . . 41
4.2.6 Call instruction . . . 42
4.2.7 Bus, demux and output arbiter . . . 43
4.2.8 User space . . . 43
4.2.9 Re2c . . . 44

5 Offloading Stateless and Stateful NFs 46
5.1 Network Functions . . . 46
5.2 Evaluation . . . 49
5.2.1 Test environment . . . 49
5.2.2 Throughput . . . 49
5.2.3 Latency . . . 51
5.2.4 Communication with host . . . 51
5.2.5 Coprocessor measurement . . . 53
5.2.6 eBPFlow performance on packet processing . . . 54
5.2.7 eBPFlow performance compared to other systems . . . 55
5.2.8 Power . . . 56
5.2.9 Discussion . . . 56

6 Conclusion and future work 58
6.1 Results . . . 59
6.2 Future work . . . 60
6.3 Publications . . . 60

References 62

Appendix A eBPFlow: ALU instructions 70

Chapter 1

Introduction

Network Function Virtualization (NFV) and Software-Defined Networking (SDN) provide flexibility and programmability on the network data plane. Combining these technologies improves manageability, reliability, and agility, enabling network operators to adapt to both upgrades and service demands. However, NFVs processing typically occurs in software on virtual machines or containers of commodity servers. Such software dataplanes, while much faster today than a decade ago, struggle to support today's traffic demands.

Numerous recent studies have aimed to mitigate the poor performance of software, while retaining their flexibility, by offloading network functions (NFs) to hardware accelerators such as programmable switches and SmartNICs [33]. NFs can be partially or entirely offloaded and accelerated. Programmable data planes provide programmability and flexibility to implement different tasks on network devices, enabling adaptability for new headers and protocols. Also, they support NF offloading, improving processing performance. However, each offload platform brings with it major limitations on generality (e.g., P4 can only support a narrow range of types of NF) and expressiveness. Thus to date most efforts have focused on bespoke implementations for a specific offload platform rather than developing a fast, general-purpose approach to NF offload [25, 6].

1.1 Contextualization

This brief history of recent programmable data planes illustrates the industry's trend of adopting eBPF. Pacifico et al. [48] proposed a simplified version of the eBPFlow on NetFPGA SUME. In this version, the system contains four eBPF engines shared between all the ports. Each eBPF engine has a pipeline with 5-stages, providing parallelism of instructions. However, this system does not support parallelism in forwarding packets and per-port with a number of exclusive cores, harming the system's performance due to lost packets and processing overhead. Here, we extended this work by providing new types of parallelism (per port and forwarding of packets), increasing the number of eBPF

cores, and adding an output crossbar. We have also realized new experiments to evaluate and compare the system with similar systems. Netronome [3] provides a SmartNIC that includes programming capabilities with eBPF instructions, showing the trend towards programmable data planes with eBPF. But, to program the SmartNIC, the code has to pass a verifier that disables back-edge jump (e.g., for, while loops), so the SmartNIC can not execute NFs that compute on the packet payload (e.g., DPI). Moreover, Netronome is firmware and kernel-dependent, making it challenging to manage the network; it has a very low port density (only 2 ports); it does not provide specific hardware modules, such as CAM or TCAM to handle stateful NFs. Finally, hXDP executes XDP code in hardware. Besides eBPF ISA, hXDP also provides new instructions. But, it does not support offloading of NFs in runtime. The eBPFlow is compatible with the eBPF standard [57]. Moreover, it presents more return codes of the design in hardware. This does not harm the compatibility with other eBPF systems.

1.2 Motivation

Software-Defined Networking (SDN) is a paradigm for the development of research in computer networks that has gained the attention of the scientific community and industry in the area. SDN is a paradigm that separates the control plane from the data plane and allows the administrator to program the devices [40]. In SDN, the control plane configures the routing rules of the network with a logically centralized entity called controller, while the data plane forwards the packets according to the actions defined by this controller. Due to the structure that SDN provides, research areas such as traffic engineering, quality of service (QoS), and virtualization have evolved rapidly [61].

The OpenFlow [42] standard is an example of SDN, which has seen significant growth since its first release in 2008 until the release of the current version (1.5). The first version of OpenFlow had a matching table of ten fields and evolved into multiple tables with 44 different fields [28]. However, the number of fields supported by OpenFlow is constantly being updated to support new fields and protocols, such as the IPv6 protocol. Unfortunately, OpenFlow has a protocol-dependent data plane which difficulties the adoption of new fields and protocols realeased [24].

1.3 Problem definition and objectives

The formulated research question that this thesis tries to solve in the context of programmability on networking is: *How to provide flexibility and programmability of the data plane with abstraction and performance?*

To solve this research question, this thesis proposes eBPFlow, a platform that supports offloading NFs using the standard, general-purpose eBPF (extended Berkeley Packet Filter) instruction set [68] already used widely in the Linux kernel. eBPF specifies a bytecode machine and an instruction set that we leverage to program general-purpose NFs on the data plane. eBPFlow is a platform for seamlessly accelerating network computation to deploy building upon eBPF. Moreover, it combines flexibility and programmability in software with high performance in hardware using an FPGA (Field Programmable Gate Array).

Furthermore, eBPFlow is protocol-independent, allowing the utilization of new dynamically defined fields and protocols without recompiling or restarting the device when the user changes the packet processing algorithm on the data plane at runtime. eBPFlow supports all network hardware requirements to offload and accelerate NFs, such as similar integration, performance, programmability, flexibility, lookup and pattern matching, forwarding, traffic shaping and control, serviceability, and data manipulation. eBPFlow runs on the NetFPGA SUME 40 Gbps platform [81]. The tests were performed in a physical testbed, demonstrating the eBPFlow performance to offload stateless and stateful NFs and accelerate processing. We present the feasibility of building NFs such as LPM forwarding, Stateful firewall, DDoS mitigation, and Deep Packet Inspection (DPI). The eBPFlow's repository is publicly available on Github [18].

1.3.1 Specific goals

To achieve the overall goal, the follows specific goals were to attend:

- Setup NetFPGA SUME's development environment;
- Put NetFPGA SUME to work in a physical testbed;
- Setup pktgen-DPDK to generate traffic;
- Design and implementation of the eBPFlow;

- Develop and integrate tools of the userspace with data plane;
- Support maps on eBPFlow using CAM/TCAM memories (32 lines x 64 bits);
- Optimize design and implementation of the architecture to improve performance (double buffer memory and DM_FIFO memory);
- Add instructions parallelism with multi-cores containing 5-stages pipeline;
- Support parallelism per port through a cores group reserved per port;
- Insert parallelism on the forwarding of packets through an output crossbar coupled on the data path;
- Establish communication with the host via PCI express bus;
- Develop test scripts to evaluate the system;
- Evaluate eBPFlow to offloading of NFs;
- To overcome problems of simulation, synthesis, and frequency;
- Evaluate other similar systems (kernel, Netronome, and hXDP) to compare with eBPFlow;

1.4 Contributions

This section presents the contributions of this thesis on the scenario of programmable networking. The main contributions of this thesis are:

(i) Offloading network functions and accelerating packet processing by leveraging eBPF and integrating existing eBPF environments and projects;

(ii) eBPFlow allows users with little hardware expertise to develop functions that operate on packet headers and payload, L2-L7 layers of the network stack with high throughput and low latency.

(iii) Logic design and hardware implementation of eBPFlow, built on top of the NetFPGA SUME [81] with three parallelism types: instructions parallelism with a multi-core hardware design containing 5-stages pipeline; parallelism per port through a cores group reserved per port; and parallelism on the packet forwarding through an output crossbar coupled on the data path;

(iv) eBPFlow enables the programming of stateful and stateless NFs and the use of dynamically defined new fields and protocols at runtime.

1.5 Challenges to design eBPFlow

Offloading allows network functions to be loaded and executed in general-purpose hardware. This feature provides fast adoption of new networking protocols and services, flexibility on maintenance and management operations, and reduces operational costs because the hardware is generic instead of dedicated to a specific network function. There are many benefits of using NetFPGA to offload network functions and accelerate packet processing on the fly. However, to achieve these goals, some challenges need to be overcome. We present two challenges found on programming FPGA for network functions.

Programmability and flexibility: FPGAs are hardware platforms that combine flexibility in software with high processing power. Due to these features, FPGAs are attractive platforms to accelerate packet processing and offload NFs. Moreover, they are reprogrammable with power efficiency. On the other hand, hardware programming occurs through low-level hardware description languages (HDLs) such as Verilog and VHDL, which do not offer high productivity rate. eBPFlow overcomes this challenge by combining NetFPGA's features with eBPF technology. This combination allows users to accelerate packet processing on userspace using eBPF NFs.

Achieve high-performance: Hardware to offload NFs and accelerate packet processing must support many stateless and stateful functions (e.g., tunneling, forwarding, traffic shaping, monitoring, access control list (ACL), firewall, and DDoS protection) with a throughput of 40-200 Gbps. Moreover, it should minimize processing overheads and performance bottlenecks. All these points directly affect the hardware performance and quality of services. eBPFlow improves the system's performance by adding a processing cores group per queue composed of four eBPF engines on the data plane to support parallelism in packet processing per queue. Each eBPF engine contains an eBPF processor with a 5-stage pipeline, which provides instruction-level parallelism. Moreover, we included an output crossbar connected to output arbiters of the eBPF engine groups to supply parallelism on the forwarding of packets.

eBPFlow design and implementation on NetFPGA solves all the challenges presented in this section.

1.6 Organization

The remainder of this thesis is organized as follows. Chapter 2 approaches an overview of themes covered in this thesis. Chapter 3 describes and compares the related work of the literature. Chapter 4 introduces details of design and implementation of the eBPFlow built on top of the NetFPGA SUME platform. Chapter 5 shows the evaluation and results of offloading stateless and stateful NFs, in a physical environment. Chapter 6 presents the conclusion and future work. Finally, Appendix A lists the ALU instructions supported by system.

Chapter 2

Approaches and Overview

This chapter presents approaches and overview of the themes eBPF, programmable data plane, traffic classification, FPGA, and NetFPGA covered by this thesis. The chapter is organized as follows: the Section 2.1 introduces eBPF technology. Section 2.2 describes the motivation behind the use of FPGAs on programmable networking. Section 2.3 defines traffic classification and deep packet inspection. Section 2.4 introduces concepts about FPGA. Finally, Section 2.5 approaches concepts about NetFPGA.

2.1 Extend Berkeley Packet Filter (eBPF)

McCanne and Jacobson [41] in 1992 proposed *Berkeley Packet Filter* (BPF), a virtual machine composed of a small instruction set for filtering and analyzing packets. Since its launch, BPF has been a library used in several network applications, for example, `libpcap` and `Wireshark`. Over the years and adoption by the area's community, BPF has undergone several revisions that contributed to the emergence of eBPF (extended BPF). It is an extension of BPF integrated into the Linux kernel in version 3.18, allowing the compilation and insertion of eBPF programs into the kernel at runtime.

eBPF is an improvement of BPF in which the architecture has been expanded from 32 to 64 bits, increased the number of registers from 2 to 11, added a stack, and supports maps operations in userspace via function calls [57]. In addition, whereas BPF has only forward jumps, eBPF can have jumps in both directions. Figure 2.1 presents the overview of the eBPF machine.

Addresses [0 to 9] are the general-purpose registers [r0 to r9], and address 10 is the register [r10] that to stores the address of the top of the stack. The stack is a data structure of 512 bytes that stores data local variables of eBPF programs. Finally, maps are generic data structures that provide eBPF programs to share collected information and store state. eBPF allows designing architectures independent of platforms or protocols. No prior knowledge of the protocol or packet structure is required. The parsing starts when

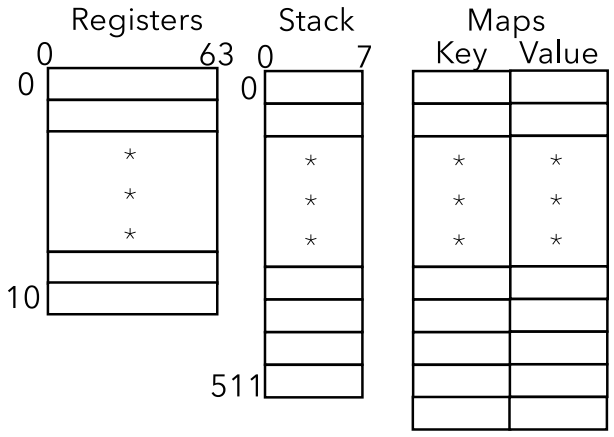


Figure 2.1: Overview of the eBPF machine.

the system loads a packet into the eBPF data memory, and the execution of the eBPF instructions moves the packet fields into registers and makes the necessary comparisons.

eBPF supports expressive flow rules. It enables flow rules supporting inequality, complement (`not` operation), or range matching. For instance, it can do logical operations such as $\geq$ or $<$, needed for range expressions. Moreover, by allowing the negation rule, packets with destinate different of the port 80 (web traffic), for example, can be matched directly without the need to create rules for the other complement values, reducing the number of flow rules.

The software implementation of eBPF in the Linux kernel supports a set of functions to handle maps ($< key; value >$ data structure). The most important functions are: `lookup`, `update`, and `delete`. The eBPF program can invoke these functions executing the eBPF call instruction. The maps include arrays, hashmaps, the Least Recently Used (LRU) replacement policy, and the longest prefix match (LPM) using trie. These maps allow eBPF programs to keep state between packet arrivals. For example, classic Layer-2 switches use a hashmap to map the port associated with a MAC address. LPM maps generally can be used to quickly implement IP routing, mapping the port associated with a sub-network. Thus, maps are essential to store states. In our hardware implementation, our design choice was to use hardware modules. Thus, the maps are content-addressable memory (CAM) for exact-matching and ternary content-addressable memory (TCAM) for LPM, and Dynamic Random Access Memory (DRAM) for arrays.

2.1.1 eBPF verifier

The Linux kernel implementation provides an eBPF verifier. The verifier checks the validity, security, and performance of eBPF programs. If desired, the verifier allows eBPF programs with only bounded-loops to enable static analysis. The eBPF verifier checks whether a program terminates, whether the memory accesses are in the range of memory space, and the greatest depth of the execution path (critical path). This critical path can provide an upper bound on the execution time. Thus, the verifier aims to guarantee eBPF kernel-safe code execution. In eBPFlow, the verifier is an auxiliary tool of the system. After the compiled code, the programmer can use the verifier before loading it into the data plane for static analysis. We leave the programmer the responsibility to check for infinite loops.

2.2 Programmable Data Plane

Programmability on data plane has been an active research field that gained highlight inside SDN. Hardware devices that support programmability on data plane provide expressiveness and flexibility on packet processing without requiring knowledge on low-level commands or device specifications. Currently, hardware devices such as FPGAs, ASICs, GPUs, and SmartNICs are the main options for performing packet processing, offloading, and NF acceleration. This use aims to meet processing demands, achieve energy efficiency, and abstract the data plan by providing programmability [16].

FPGAs are the most attractive hardware devices for developing network applications because it combines software flexibility with high power processing in a reconfigurable and energy-efficient way [69]. An FPGA is a device composed of programmable logic blocks, memories (TCAM, CAM, and SRAM), and I/O devices. Developers can program it through hardware description languages such as Verilog and VHDL. Building network services on FPGAs consists of synthesizing the circuit described in HDL and loading the bitstream file generated after the synthesis on the hardware device. This process is complex, time-consuming, and has not zero downtime. The network operator must also know much about the hardware besides being an expert in hardware description languages.

2.2.1 High-Level Languages

High-level languages can be used to write code to the data plane and compile it into the eBPF instruction set. A subset of C already exists, which excludes some external libraries, system calls, and pointer arithmetic while providing functions for defining and manipulating tables. Since version 3.7, the LLVM compiler collection has a backend for the eBPF platform, allowing programming in this subset of C and generating executable code in eBPF format. Many projects use eBPF, e.g., Facebook[20] built a layer 4 load balancing forwarding plane using eBPF to provide fast packet processing in-kernel. Moreover, problems such as interdependence, distribution, and heterogeneous hardware can be solved due to the features and environments available in this technology.

2.3 Traffic classification

Traffic classification (TC) allows network operators to characterize packet flows better, manage resources, and improve network security. It is fundamental to applications such as traffic engineering, network analytics, and Quality of Service (QoS) [56]. However, the traditional TC is a straightforward approach that fails when fields are inconclusive or unavailable. For example, in peer-to-peer (P2P) traffic, the applications do not have default ports, and the HTTP server runs on different ports from port 80 with encrypted connections. L7 classification (Application layer packet classification) rises as a solution to overcome these limitations. In this approach, the classification occurs based on patterns often shared with other applications [55].

TC uses Deep Packet Inspection (DPI) to examine each byte of the packet's payload at runtime and returns when it finds one or more patterns (e.g., malicious traffic or attacks) previously defined. DPI is a crucial component of classification, and it uses the Regular Expressions (RegExs) power to increase performance to find patterns. However, some limitations and challenges remain open in the context of L7 classification and DPI, such as programmability, performance, efficiency, security, scalability, and usability [50].

2.4 FPGA

Field-Programmable Gate Array (FPGA) is an integrated circuit that can be programmed or reprogrammed to execute a specific application or functionality after manufacturing [29]. FPGA combines the flexibility of software with hardware performance has been applicable in many areas. For example, in computer networking, FPGAs are employed on packet acceleration to improve the throughput and reduce the latency of data centers [52]. Moreover, they are vital components in deploying new services and functionality due to the flexibility by being programmable [13, 47].

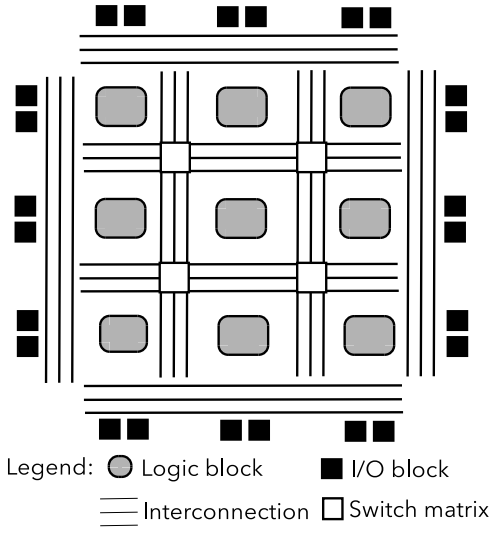


Figure 2.2: FPGA design.

The architecture of an FPGA consists of three main parts: configurable logic blocks (CLB), programmable interconnects, and programmable input/output (I/O) blocks. Figure 2.2 demonstrates the basic structure of an FPGA. CLBs implement different logic functions creating a physical array of logic ports. Although they work as separate modules that operate in parallel, they are configurable, allowing each internal state to be controlled. Moreover, they can be connected, programming the interconnections to build a specific function. CLBs consists of several components such as flip-flops, look-up tables (LUT), and multiplexers. Programmable interconnects are responsible for implementing the routing. They allocate the resource among CLBs. Routing paths have wire segments with different lengths that are connected using anti-fuse or memory-based techniques. Each CLB connects a switch matrix to access the routing structure. The switch matrix selects the signal of a routing channel connecting vertical and horizontal lines. Finally, I/O blocks allow communication with the external components. CLBs and interconnects use the I/O blocks to receive and transfer data. I/O blocks are responsible for data transfers in and out of the FPGA [17].

Register Transfer Level (RTL) design is a high-level logic design abstraction constructed using hardware description languages (HDL), VHDL or Verilog. Simulation is the stage to debug errors on RTL design before of the synthesis process. Synthesis is the process of converting an RTL design into a logic gates representation to set up the behavior of the FPGA. The synthesis occurs using a synthesis tool, where the design is synthesized. The synthesis tool receives an HDL program as input and generates as output a bitstream file that will be loaded on FPGA. During the processing of the design, the tool interprets the HDL program into implementations of digital elements in the FPGAs, such as lookup tables (LUT), flip flops, RAM blocks using boolean algebra operations [76].

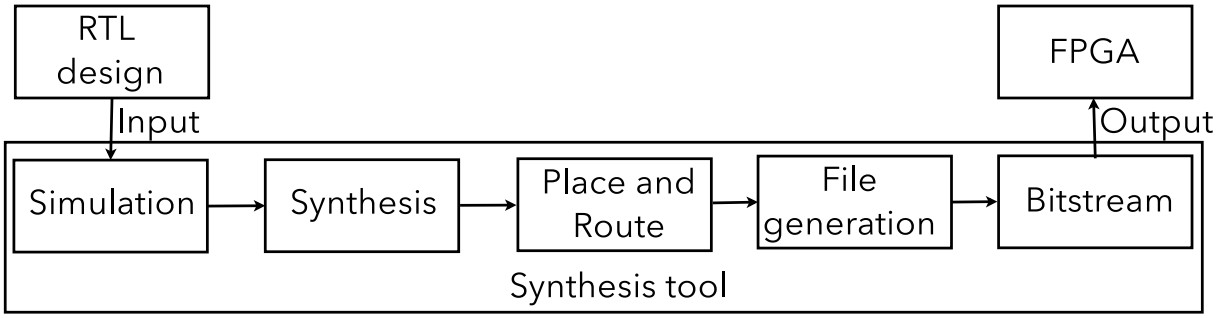


Figure 2.3: Overview of the synthesis stages.

The synthesis of the design (Figure 2.3) consists of three main steps: synthesis, place and route, and file generation. The synthesis tool transforms the functional RTL design into an array of gate-level macros in the synthesis stage, creating a flat hierarchical circuit diagram that implements the RTL design. Macros are models of the FPGA's logic cells. Logic cells are implemented as digital elements. The synthesis process requires at least two inputs: the source code and timing constraints. The source code has files that define the configuration of the synthesis tool. These files are responsible for telling the tool which FPGA to target, the pinout of the design, and which strategy to use when running the synthesis. The timing constraints define timing details about the FPGA. This stage also optimizes the netlist of the design. Moreover, it removes or replaces any elements of the netlist which are redundant or duplicated. Netlist contains information as nets, sequential and combinational cells, and their connectivity. This stage terminates with the netlist of the generated and optimized design.

With the generated netlist begins the place and route stage. The first process executed in this stage is known as placement, in which the synthesis tool maps the optimized netlist to physical cells in the FPGA. In the following, the routing process defines the interconnection between the logic cells of the FPGA. In this stage, the synthesis tool executes the placement and routing operations several times to meet the timing requirements of the design. Moreover, the tool is responsible for scheduling these multiple runs based

on the configuration of the placement and routing algorithms. This stage terminates when the tool meets the best time based on time constraints.

The last stage occurs the generation of the programming file, called bitstream. The synthesis tool generates this file after the place route stage. With the generated bitstream, the user can load it on FPGA using the synthesis tool.

2.5 NetFPGA

It is an open-source project that leverages research and development of new networking applications using a SmartNIC based on FPGA. It provides a platform composed of software and hardware. All platforms' infrastructure aims to simplify development tasks such as design, simulation, and testing of high-speed networking applications in hardware. The NetFPGA's development environment allows the creation of new designs reusing the base code of reference projects (e.g., NIC, Switch, and IPv4 Router). In addition, it has support from a broad research community [43]. We chose the NetFPGA to demonstrate the system due to the platform's benefits. Moreover, the NetFPGA platform allows bypass challenges of the system's design, creating new circuits to provide parallelism and using the resources available in the platform, such as IP cores, FIFOs, and memories, to optimize the performance of the system.

2.5.1 Advanced eXtensible Interface (AXI)

It is a protocol designed for on-chip communication, released in 2003 as part of ARM AMBA, a family of microcontroller buses introduced in 1996. In 2010, AMBA released the most current version of the AXI, AXI4 [2]. AXI4 [75] uses types different protocols based on an interface. An interface is composed of communication channels between a single AXI master and AXI slave, representing Intellectual Property (IP) cores exchanging information. In addition, the design can connect multiple AXI masters and slaves using AXI Interconnect, which is a traditional monolithic crossbar approach. Figure 2.4 shows an interconnection between masters and slaves.

Currently, there are three types of interfaces: AXI4, AXI4-Lite, and AXI4-Stream. Memory-mapped interfaces of designs that demand high-performance use AXI4 interface. It allows high throughput bursts of up to 256 data transfer cycles with just a single

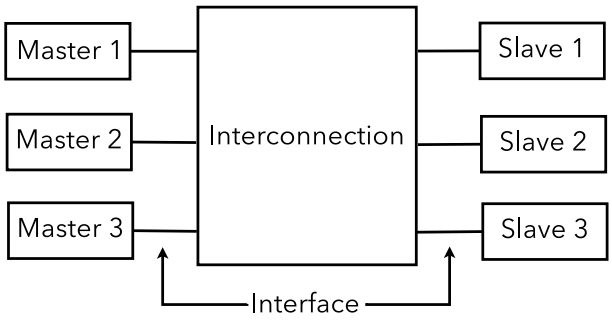


Figure 2.4: Interface and interconnection between masters and slaves.

address. AXI-Lite manages the control and status of registers using memory-mapped communication with low throughput. Moreover, It is composed of a single transaction memory-mapped interface. Finally, AXI-Stream is an interface for streaming data, not memory-mapped of high speed, that does not use the address on the transference of data. Therefore, this interface allows unlimited data burst size.

All protocol interfaces consist of five different channels: read address, write address, read data, write data, write response. The data in reading operations move only in a direction according to the read address. In write operations, the data can move in both directions between the master and slave simultaneously, and data transfer sizes can vary. For example, in AXI4, there is a limit of a burst transaction of up to 256 data transfers. On AXI4-Lite occurs only one data transfer per transaction.

This protocol added in hardware designs provides benefits like productivity, flexibility, availability. AXI4 contributes to the productivity of the design due to standardizing on the AXI interface, where developers need to learn only a single protocol for the IP core. Moreover, flexibility occurs because the correct interface (e.g., AXI, AXI-Lite, and AXI-Stream) can be used according to the design requirements. Many IP providers support the AXI protocol, which provides availability to the design if released as a product to the industry.

2.5.2 Datapath

Figure 2.5 presents the datapath of the NetFPGA SUME. The packets in the NetFPGA are processed in the form of 256-bit words and 128 bits control to identify the word type (data or metadata). The datapath transmits packets words utilizing data signals and control signals of the AXI4 Stream interface. If the control signal is zero, the words transmitted are words of the packet. Otherwise, the words are from the metadata.

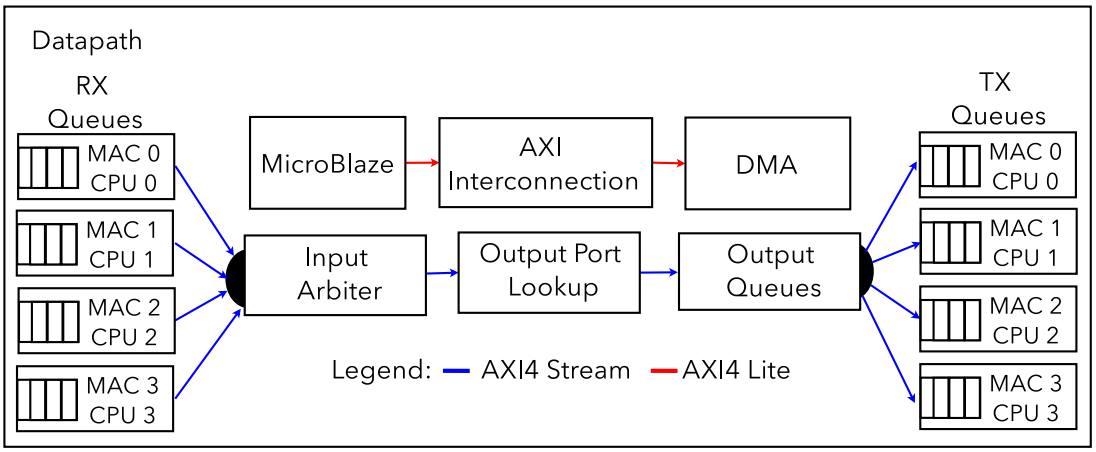


Figure 2.5: NetFPGA SUME datapath.

The standard NetFPGA library provides the following modules: input arbiter, output port lookup, and output queue. The input arbiter receives the packets from the network (via the MAC interface) or the PCI bus (via CPU) in the format of 256-bit words and stores them in internal queues of the module. Then, the words are taken from the queues using the round-robin algorithm, and it routes them to the output port lookup module. The output port lookup module defines which port the packet will be sent to based on the packet properties, for example, an input port or destination address. This operation happens on the metadata output port field. Output queues receive the packet with the output port marked and send it to packets TX queues. The output port value defines if the output queues module will forward the packet to the network interface or host machine.

The NetFPGA datapath uses the AXI4 protocol to communicate among modules, operations with registers, and direct memory access (DMA). The standard NetFPGA modules send and receive packets through the AXI4 Stream interface. Operations with registers and DMA happen from the AXI4 Lite interface. It also has auxiliary modules such as Microblaze, AXI Interconnection, and DMA. Microblaze module is a Xilinx Microblaze subsystem used only for clock configuration of the design. AXI Interconnection is responsible by to manage the communication among IP cores of the design. Finally, the DMA module works as a DMA engine that communicates between hardware and the host machine through Xilinx's PCIe core.

We implemented the eBPFlow's design inside the output port lookup module to allow the system to perform parsing, matchings, and actions on packets. We chose this module because it is the module of the NetFPGA's datapath that defines the output port to where the packet will be forward on hardware. The following section (Section 4.1) describes more details about eBPFlow design.

2.5.2.1 Synchronization

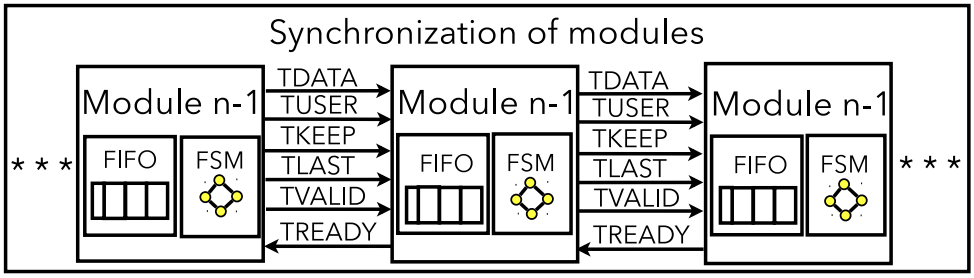


Figure 2.6: Modules synchronization of the datapath.

The synchronization of the NetFPGA modules operates using the AXI-4 stream Xilinx protocol [73]. Figure 2.6 shows the modules' synchronism with their respective signals. TDATA represents the data stream. TUSER is out of band metadata. TKEEP enables which bytes of TUSER are active on the data path (by standard, NetFPGA projects enable all bytes of TUSER). TLAST indicates the end of the packet/burst. The write (TVALID) and ready (TREADY) signals control the communication between the modules. The TVALID signal informs when the previous module is ready to transmit (when it has content to transmit), and the TREADY signal indicates that the next module is ready to receive. The data transfer between modules only occurs if the two signals are active. Each module contains an input queue and a finite state machine. The input queue temporarily stores the TDATA, TUSER, TKEEP, and TLAST signals until the state machine module can remove the word from the queue. The state machine of each module has specific behavior.

Chapter 3

Related work

This chapter presents a discussion about related work that support programmability using different technologies in hardware and software targets. The works are divided in six topics: programmable networks, high level domain-specific languages, BPF related, FPGA related, Smart NICs, NIDS and DPI.

Programmable networks. The OpenFlow [42] standard, although being the most adopted SDN architecture, has limitations. Its matching structure cannot do inequality, complement (not operation), or range matching. On the other hand, eBPFlow allows for logical expressions (not, and, or) and range comparison (>,<). Liu et al. [36] developed the CLARA, a network slicing architecture that uses NFV concepts and reinforcement learning algorithms for resource allocation management. Finally, Yen et al. [78] proposed the Lemur, a system that places and executes NF chains across heterogeneous hardware while meeting service-level objectives (SLOs) in NFV. It receives as input a high-level description of multiple NF chain DAGs and their associated SLOs. As output, the system returns a placement configuration for each NF chain and coordination code, ensuring that the NF executes on the appropriate hardware element specified by the placement.

High level domain-specific languages. The P4 programming language P4 [8] adopts the match-action abstraction model. Therefore, it is possible to use the P4 language to generate eBPF instructions using the compiler from P4 to eBPF [10]. Domino [59] is a high-level language compiled into Banzai, a low-level machine model designed for line-rate switches. Although P4 and Domino include small and fast registers to store states, they provide a restricted functionality for many stateful functions. Kfoury et al. [26] present an exhaustive survey about P4 programmable data plane switches highlighting subjects like taxonomy, applications, challenges, and trends.

BPF related. BPFabric [24] proposed a software platform that allows protocol-independent packet processing. It uses eBPF instructions to define the packet processing and forwarding in the data plane. BPFabric was initially implemented over a Linux raw socket interface and later adapted over the DPDK. hXDP [9] is a system to run Linux's XDP programs on an FPGA. hXDP is similar to the eBPFlow, and it executes XDP code. The hXDP design changed the load/store instructions and introduced three-

operand instructions. The hXDP design does not support network functions offloading at runtime like the eBPFlow, which has many processing cores with an instruction memory containing a double buffer system. FFShark [66] is an implementation of the Wireshark in an FPGA. It contains eBPF cores to execute written filters in the PCAP filtering language. However, FFShark does not provide instructions parallelism with eBPF cores containing a 5-stages pipeline coupled on the cores. Katran [20] is an open-source eBPF load-balancer application provided by Facebook, showing eBPF's adoption trend in the industry. Chaining-Box [12] is a Service Function Chaining (SFC) architecture where all the SFC functionality are implemented, in a fully transparent manner, as a sequence of eBPF stages.

FPGA related. P4FPGA [69] is a platform developed in hardware that performs conversion of P4 programs to Verilog. P4-To-VHDL [4] is a tool that converts a P4 description to a synthesizable VHDL code suitable for the FPGA implementation. ClickNP [31] also focuses on increasing programmability flexibility. It provides a declarative language called ClickNP. ClickNP can be compiled into an intermediate hardware description language (HDL) and synthesized on the FPGA. Zang et al. [79] proposed a distributed-agent NFV system that supports Service Function Chaining (SFC) of FPGAs and microprocessors. The system works with an agent helping the partial reconfiguration core to control the dynamic reconfiguration of middlebox functions on FPGAs. FlowBaze [51] is an FPGA-based SmartNIC that allows stateful packet processing in hardware by programming using Extended Finite State Machines (EFSM). However, it cannot operate on the packet payload and only supports storing 64 states. eBPFlow does not have these limitations because the states and transitions of an FSM are transformed into instructions. PANIC [32] is an FPGA based on Reconfigure Match Action (RMT) switches that schedule the order in which the packets are processed and distribute the packets across the different compute units. Finally, Eran et al. [19] proposed the NICA, an FPGA-based NIC server acceleration system that supports software abstractions via functional units for application acceleration in cloud systems. NICA was implemented on Mellanox and integrated with an abstraction denominated ikernel (inline kernel), which represents an Acceleration Functional Unit (AFU) in a user program.

Smart NICs. Netronome [3] provides a SmartNIC programmed with eBPF instructions. Some features and commands are specific to the kernel and firmware version, generating incompatibility on the network. When an update is released, firmware or kernel needs to be updated manually, generating failures and hindering the management of the network. Nonetheless, eBPFlow does not have these dependencies. It is independent and seamless with other technologies, e.g., the Linux kernel. Furthermore, to load a program into Netronome NIC, the code has to pass a verifier which does not allow back-edge jump (e.g., for, while), so the SmartNIC can not compute DPI NFs with different packet sizes. Netronome NICs have very low port density, with at most two ports per

NIC. Moreover, it does not support CAM and TCAM memories. On the other hand, the eBPFlow prototype has 12 physical ports. Moreover, eBPFlow design includes specific memory hardware, such as CAM and TCAM, to handle stateful NFs.

NIDS and DPI. Zeek[63], and L7 Filter[60] are NIDS that use RegEx to identify content inside packets on the application layer. They run on the Linux kernel or in userspace, sharing CPU resources between other processes. When the number of packets that arrive on the host increases, processing overhead occurs due to the busy CPU, harming the performance of applications L7. Wang et al. [70] implemented an IDS that uses eBPF technology for pattern matching in packets. The system runs on Linux kernel and userspace. On kernel, the system uses eBPF filters for pattern matching and dropping packets that do not match the matching rule. The system runs a program on userspace that examines the packets not dropped by the kernel. The system has processing overhead with processing bottleneck by sharing CPU cores with other processes running in parallel. Pigasus [80] is an IDS/IPS architecture FPGA-based that performs TCP reassembly on hardware and supports regular expressions to detect attacks in packets. DeepMatch [22] presents a DPI system implemented on the SmartNic Netronome that uses P4 and RegExs to detect attacks in suspicious packets. NetFilterOffloader [14] proposed a NetFilter Firewall on NetFPGA 1G using Iptables. It does not support RegExs and L7 classification. Moreover, it uses hardware deprecated with a throughput of 4 Gbps. Some works exploit specialized hardware such as TCAMs, GPUs, and FPGAs [27, 30, 35, 39, 53, 58, 49, 62, 65]. However, none of these works presents the set of eBPFlow contributions together in one system.

Chapter 4

eBPFlow architecture

This chapter shows details of the overall architecture, design, and implementation of eBPFlow built on top of the NetFPGA SUME platform. The eBPFlow architecture has a logically centralized controller that communicates through a socket TCP/IP with the network elements (*e.g.,* routers, switches). The controller can install programs in the data plane at run time without network interruption, and it includes a scheduler to support multi-tenants and to move code in and out of the data plane. Section 4.1 presents the eBPFlow design, describes the system, and how it works. Finally, Section 4.2 shows details of implementation on hardware and optimizations to improve the performance of the system.

4.1 Design

Figure 4.1 gives an overview of the eBPFlow's design and implementation built on top of the NetFPGA SUME [81] platform. The system has two components: data plane and userspace tools.

The data plane contains sixteen processing cores divided into groups composed of four eBPF engines, with each group responsible for packet processing and forwarding for each RX and TX queue. We chose the number of eBPF engines (four) per group based on the design's consumed resource (logic cells) and the maximum frequency obtained after the synthesis. All groups share an instruction memory, a timer, a coprocessor, and an output crossbar. The instruction memory includes a system with a double buffer that changes programs without stopping the processing. The timer allows measurement of network performance (for example, through EWMA, latency, and jitter) when storing the timestamp of packets on metadata. The coprocessor works as eBPF maps in hardware using TCAM/CAM memories to store pairs <key, value>. The output crossbar provides parallelism in the forwarding of packets, improving the throughput and latency of the system. Moreover, each group has one demux and output arbiter reserved per RX and TX

queue, which allows the system to receive and send packets exchanging cores in runtime. We divided the eBPF engines into groups by queue to provide per-port parallelism on packet processing. Moreover, we added an output crossbar to receive the processed packets per each group to all output queues, providing parallelism on forwarding.

On eBPFlow, four parallel engines per port can cause packet reordering in a single flow. We do not treat the packet reordering on the system, leaving the TCP protocol responsible for this task because it treats the packet reordering on the transport layer.

The userspace includes a controller that opens a socket connection TCP/IP to the device and applications created at the user level as a loader to compile/load programs and handle maps, an eBPF disassembler to convert binary code to eBPF instructions, a software emulator to debug, and a CLI application to interact with eBPF engines.

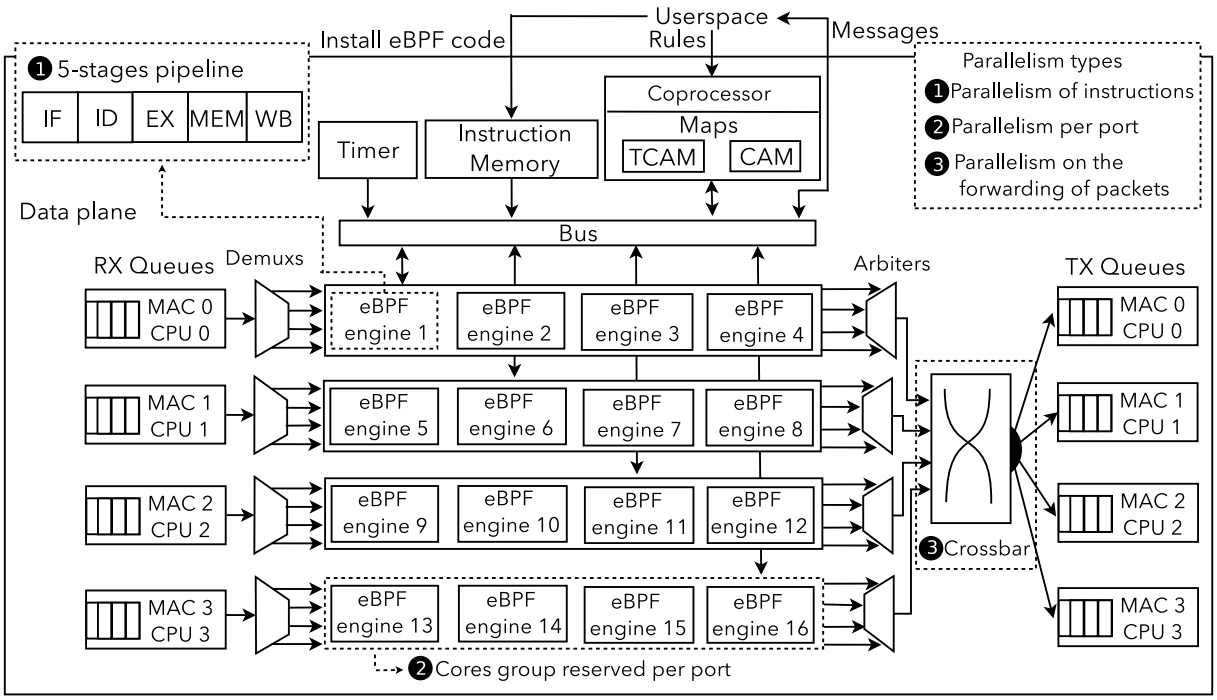


Figure 4.1: eBPFlow design.

4.1.1 How does eBPFlow work?

The packet processing on eBPFlow begins with the user-generated eBPF instructions via C eBPF or P4 code on userspace. Once generated, the instructions take their course from userspace to the data plane, where the system loads them into the instruction memory. The communication between userspace and data plane occurs through

userspace tools loader, and PCIe bus. All processing on the platform occurs without the user knowing specific low-level commands or having experience with hardware targets.

The processing on the data plane begins with the packet's arrival in an Rx queue. RX's queue demux forwards the packet to the current eBPF engine according to the register value that controls which eBPF engine has access priority. The access priority algorithm between eBPF engines is a standard Round Robin (RR) algorithm. If an eBPF engine cannot receive a packet, the system updates the register to the next eBPF engine. Each eBPF engine waits for the first packet word to arrive. In the next step, eBPFlow processes and forwards the packet to the output arbiter. It selects the current eBPF engine packet that terminates the processing and sends it to the output crossbar. The output arbiter chooses the priority of access of the current eBPF equal to the demux using the standard RR algorithm. The output crossbar receives packets of the output arbiters simultaneously, it parallelizes the forwarding of packets, and decides which TX queue will store the packet. In its turn, the TX queue sends or drops the packet according to the value stored in the eBPF r0 register. Each engine has one action module. Thus, for 16 eBPF engines, we have 16 action modules. The action module updates the packet's metadata for the TX queue to process it. The TX queue is responsible for drop or send the packet based on r0 value. We present more details about eBPFlow in Section 4.2.

4.1.2 How does eBPFlow provide flexibility and programmability of the data plane?

The eBPFlow is not tied to specific network protocols, enabling programmers to perform runtime parse, match, and action operations dynamically. On eBPFlow, programmers can change how the system processes packets after the design is synthesized and loaded on hardware. This feature allows the system to provide the flexibility of the data plane, defining the packet processing logic in two ways: (i) Reconfigurable in the field; and (ii) processing protocol-independent packets. The combination of these functionalities allows programmers to insert new fields and protocols. The eBPF technology is responsible for the system's flexibility using eBPF instructions generated from programs in the C language.

eBPFlow supports the standard eBPF ISA, allowing similar integration with other existing eBPF environments and projects on the network. In addition, the eBPFlow provides programmability, enabling programmers to describe packet processing logic independent of the specifics of the underlying hardware. This feature becomes the target-independent eBPFlow. Programmers only need to know the eBPF technology to use

the eBPFlow. It is possible due to a combination of software and hardware technologies implemented on the userspace and data plane of the system. For example, hXDP [9] is similar to the eBPFlow, and it executes XDP code. However, hXDP is incompatible with eBPF because of the addition of new instructions, thus changing the standard eBPF ISA.

4.1.3 eBPF engine

Each eBPF engine comprises four hardware modules: a data memory FIFO (DM_FIFO), the eBPF processor, a Finite State Machine (FSM), and an action module. Figure 4.2 shows the eBPF engine design. The DM_FIFO stores packets on the fly, working as data memory and FIFO with no extra transfer. The eBPF processor is responsible for performing the parse, matching, and actions using instructions stored in the instruction memory. Also, the processor communicates with the control plane through a socket TCP/IP. The FSM controls the whole operation of packet processing. It removes the packet from the DM_FIFO of the module, starts executing the eBPF instructions, and forwards the packet to the next module (action packet) when the last instruction (`exit`) of eBPF finishes executing. The action module forwards or discards the packet according to the value stored in r0 after processing the eBPF instructions.

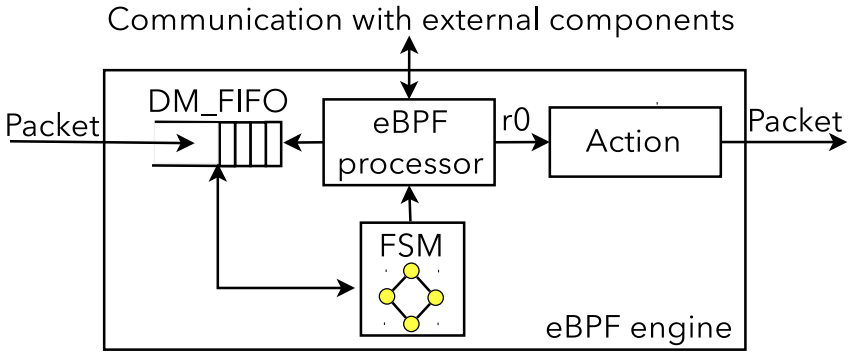


Figure 4.2: eBPF engine design.

4.1.4 Metadata

The data plane receives the packet through the input interface and stores the packet in the input queue with additional information called metadata. Table 4.1 shows

the metadata header. The first line indicates the byte order and size. The other lines show the stored structure. After the metadata is received, it comes the Ethernet frame. eBPF programs and any other protocol field can use the metadata header fields. The currently defined metadata is the destination port, packet size in multiples of 64-bit, source port, packet size in bytes, timestamp in nanoseconds, and seconds. The fields of packet size, in multiples of 64-bit and bytes, are included because the input queue module already provided this information.

Table 4.1: Metadata: Information retrieved from the input queue of the stored packet in the data memory of the eBPF processor.

0			bit			255
8 bits	8 bits	8 bits	32 bits	32 bits	16 bits	152 bits
Input Port	Source Queue	Destination Queue	Time-stamp (s)	Time-stamp (ns)	Length (bytes)	Free
Ethernet Frame						
Payload						

4.1.5 Actions

The register r0 stores the return value of the eBPF processor. In addition, it determines which action the processor will execute on the packet. Table 4.2 describes the return values of eBPF and their respective actions. After eBPF finishes the computation, the packet can be: forwarded to a port, forwarded to the controller, discarded, flooded to all ports except for the input port, or sent to the host machine via PCIe bus.

eBPFlow enables other dynamic actions such as modifying the packet header, packet payload, and adding or removing fields. With the packet stored in the data memory, a store instruction can modify the packet. The packet content can also be used for arithmetic and logical operations, for example, decrementing TTL or recomputing checksum.

4.1.6 Output Crossbar

On eBPFlow, we added an output crossbar to provide parallelism on the forwarding of packets. The output crossbar (Figure 4.3) allows connecting the eBPF engine outputs

Table 4.2: Action performed on the packets.

Action	Code	Description
Forwarding	0 - 0xFFEF	Forward packet to a specific port.
Controller	0xFFF3	Send packet to the controller.
Drop	0xFFF0	Drop packet.
Flood	0xFFFF	Send packet to all ports except for the input port.
Host	0xFFF[2-5]	Send packet to host. (CPU Queue: 0-3).

to TX queues using the output queues (OQ) module of the NetFPPGA's datapath as a buffer. Packets processed by eBPF engines are forwarded to output queues modules and TX queues via crossbar interconnect. TX queues receive the packets based on the destination queue metadata field generated by eBPF engines. This field receives the r0 value updated of the eBPF engine after the eBPF program finishes. The output crossbar works on non-blocking mode, allowing multiple simultaneous packet forwarding for different TX queues. We used an N-to-M uni-directional crossbar interconnection architecture to connect output queues to TX queues. N is the number of output queues modules, and M is the number of TX queues. The crossbar interconnection has a size equal to $N \times M$ ($4 \times 4 = 16$ *points*).

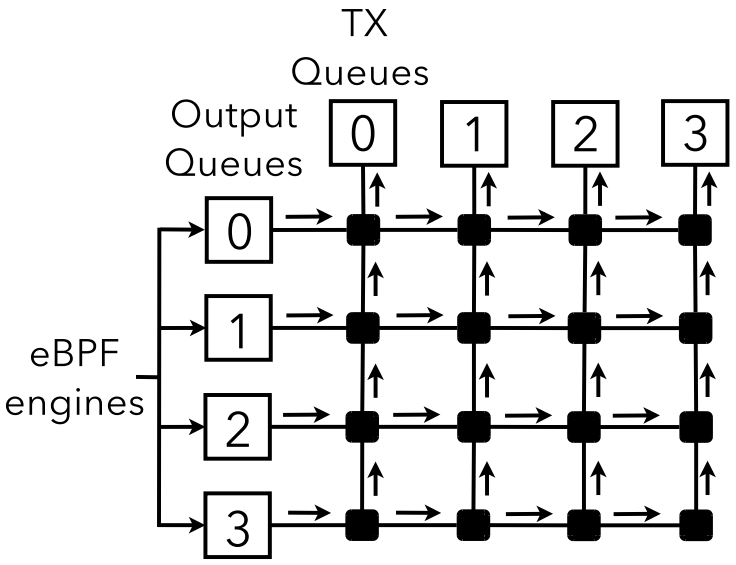


Figure 4.3: Output crossbar design.

4.2 Implementation

Here, we describe the implementation details of eBPFlow. We built eBPFlow data plane in Verilog HDL on the top of the NetFPGA SUME platform and created tools on userspace to manage operations of the system [38].

4.2.1 Hardware Instance

FPGA enables the building of hardware logic systems. The NetFPGA SUME hardware has four SFP+ transceivers that support 10 Gbps Ethernet ports. It connects to a motherboard through a PCIe Gen 3 x8 adapter. In addition, it contains a Xilinx Virtex-7 690T FPGA [72], which has approximately 693,120 logic cells, a 27 MiB SRAM, and a 5 ns (200 MHz) clock cycle. After synthesis, eBPFlow consumed 20.71% of the logical slices and 11.35% of the register slices on the NetFPGA SUME. The maximum frequency is 166.67 MHz (cycle of 6.172 ns).

4.2.2 eBPF Processor with Pipeline

The eBPF processor performs the parse, matching, and actions according to the user-generated C-code or P4-generated eBPF instructions. When starting the device operation, the user must load the eBPF instructions into the instruction memory to define the behavior of the data plane. Figure 4.4 presents the data and control paths in register transfer level (RTL), containing five data functional units (program counter, instruction memory, register file, arithmetic logic unit – ALU, and data memory) and three control units (hazard detection, forwarding, and control).

After the instruction memory returns the instruction pointed by the current program counter, eBPFlow divides the instruction into five parts: operation code, destination register address, source register address, offset, and immediate value. Each specific unit of the datapath receives part of the instruction. The control unit receives the operation code and forwards the control signals to the functional units, defining the behavior of each unit. For example, the ALU class instructions do not use the data memory, so the read and write signals from the data memory are not activated.

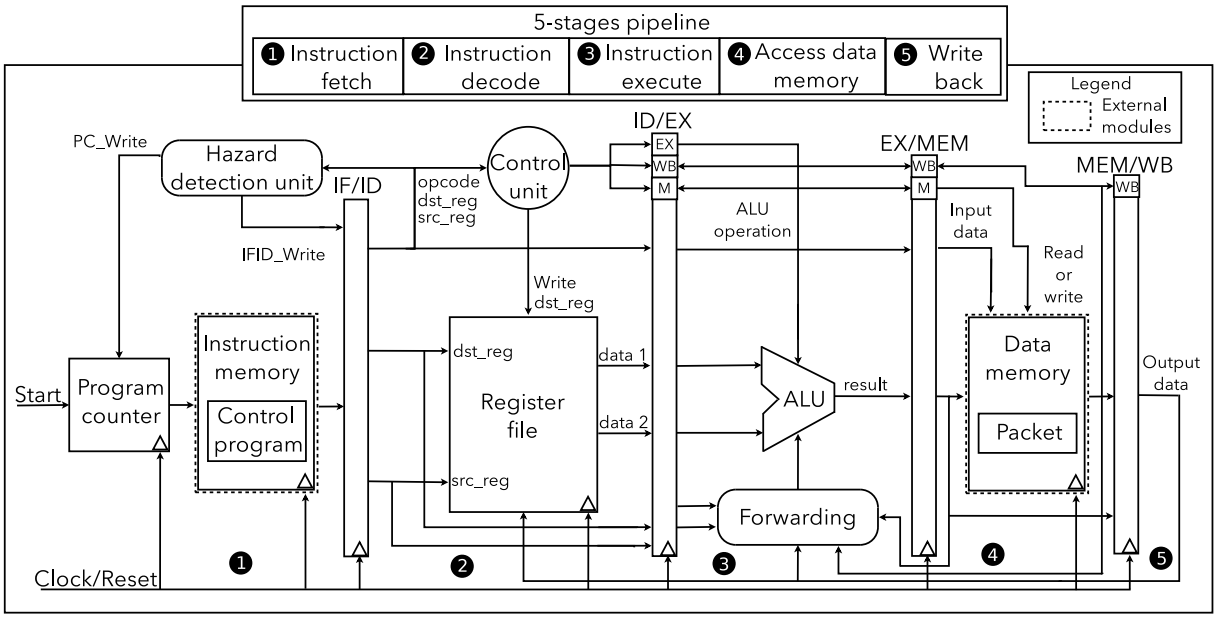


Figure 4.4: Control and Datapath of the eBPF processor.

We design the eBPF processor with a 5-stage pipeline: instruction fetch (IF), instruction decode (ID), execute (EXE), memory (MEM), and write back (WB). IF stage gets instruction from memory and increments program counter (PC). ID stage translates opcode into control signals and reads registers from the register file. EXE stage performs ALU operation and computes jump/branch targets. MEM stage accesses data memory if needed. Finally, the WB stage updates the register file. This design follows the MIPS load-store pipeline architecture [21]. We add four pipeline registers (between the stages), the forwarding, and hazard units.

We implemented the eBPF engines with a 5-stage pipeline because it allows multiple instructions to be executed simultaneously, each in a different stage of the pipeline, thus improving the overall performance of the engines. Moreover, we added the forwarding and hazard units to ensure the correct execution of the instructions on the pipeline when there is data dependence between instructions in execution on the datapath.

4.2.3 Data memory (Optimized FIFO)

To avoid the overhead of copying the packet from the transfer FIFO to the processor data memory, we designed a new abstract data type called Data Memory FIFO (DM_FIFO), which works as a FIFO and as well a Data Memory. DM_FIFO enables the eBPF processor to access the packet's data without waiting for all the packets to

arrive with no extra transfer and running load and store operations with high efficiency. Moreover, DM_FIFO synchronizes with the NetFPGA datapath's modules (input arbiter and output queues) and the eBPF processor. Therefore, the packet does not need to be initially stored on FIFO and forwarded to data memory to be processed by the processor.

We added two-way communication to communicate with NetFPGA's modules and eBPF processor. Thus, DM_FIFO works as a FIFO receiving and forwarding packets using the AXI4 stream interface signals. At the same time, DM_FIFO also operates as a data memory that communicates with the eBPF processor using control signals sent by the control unit of the eBPF processor's data path and load and store instructions. As a result, DM_FIFO has a capacity of 2.048 bytes (64 depth lines x 256 bits width) and can store up to 32 packets of 64 bytes.

The eBPF engine can start processing the packet even if the packet has not fully arrived yet. Inside DM_FIFO, for each word, we added a valid bit to indicate if the word contains data from the new incoming packet. Thus, DM_FIFO brings two advantages: It does not has no extra transfer and allows the eBPF engine to begin the packet processing before waiting for the entire packet to arrive.

Each engine has its own DM_FIFO. The DM_FIFO can simultaneously receive multiple packets of the datapath. But, it can only process one packet at a time. When a new packet is inserted in the DM_FIFO, and an older packet is in processing, it must wait for the processing to finish. The metadata region can not be overwritten because we use a FSM. Each FSM has its own metadata register.

4.2.3.1 eBPF stack

On eBPFlow, the stack is part of the data memory on the last bytes (2,112 to 2,624). We include the stack in data memory to facilitate the eBPF engine access to the stack. eBPFlow's stack has a size of 512 bytes, the same as the Linux's kernel, following the standard of the eBPF virtual machine. Figure 4.5 presents the data memory structure with spaces reserved for metadata, packet, and stack. The loader is responsible for defining the value r10 on the system. Byte 2,624 (`0xa40`) is the first byte of the stack. After the generated eBPF instructions by the eBPF compiler, the loader initializes and adds one instruction with r10 value (`mov r10, 0a40`) on the eBPF program before loading the instructions on the instruction memory of the eBPFlow. If there is a local variable on the eBPF program, one of the r6 to r9 registers receives the r10 value minus the number of bytes of the local variable size to define the reserved space on the stack. With the address of the local variable defined in a register, the eBPF processor can access the local variable

data on the stack through load and store instructions.

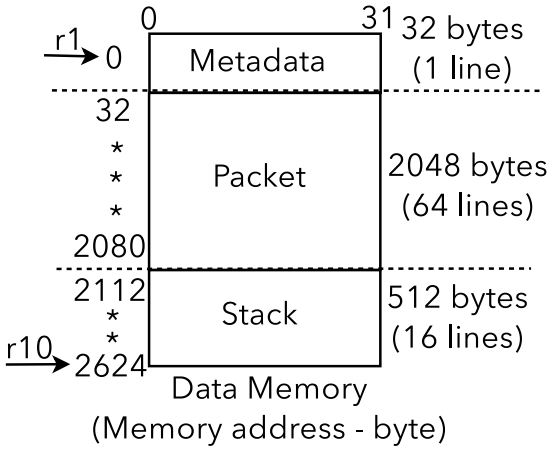


Figure 4.5: Data memory division.

4.2.4 Instruction Memory

The eBPF instructions define the behavior of how the eBPF processor handles the packets. First, we created software registers to insert the eBPF instructions into the instruction memory through NetFPGA's register interface. The controller is responsible for sending the instructions to the data plane of the eBPFlow. They are then written to the software registers using the loader and forwarded to instruction memory through the PCIe bus. Instruction memory (Figure 4.6) uses a double buffer system - DBS. We added a DBS on eBPFlow to not stop the execution of the system when a new program is loaded on instruction memory, avoiding the loss of packets due to wait time to load a new program. This system contains two memories (M_1 and M_2). Both memories never assume the same state (writing or reading) simultaneously. It means that while a memory receives eBPF program instructions, the eBPF processor reads the instructions of other memory.

As a design decision, we have put the instruction memory outside the eBPF processor to enable the connection of multiple processors using a shared instruction memory to reduce the number of used logic resources in the design. Moreover, with the increasing number of eBPF processors, it is possible to process more packets simultaneously, thus also increasing the throughput.

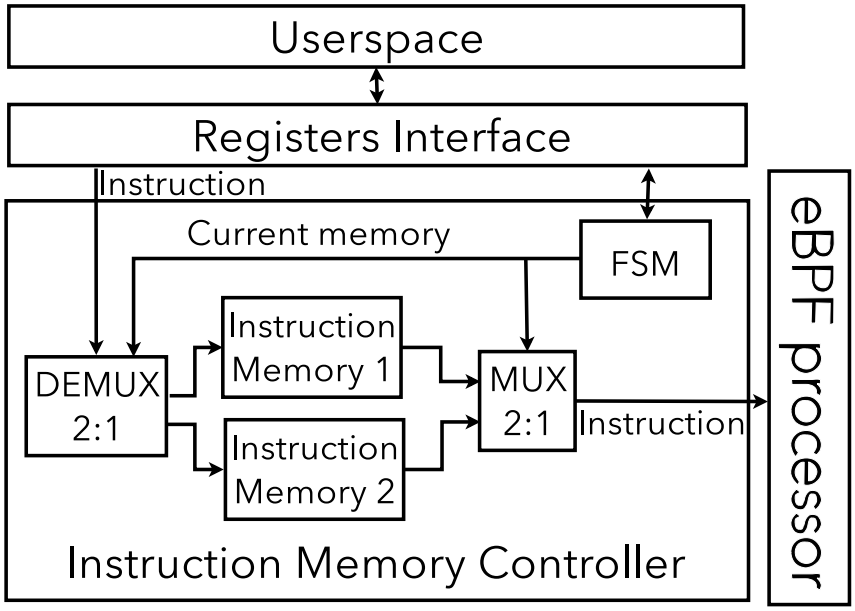


Figure 4.6: Instruction memory - Double Buffer System.

4.2.5 Maps

eBPF Linux kernel implementation allows for maps. A map is a generic data structure that stores different data types in the form of key-value pairs. Our design currently provides three types of maps: longest prefix matching (LPM), exact-match, and array, as hardware components of the system. For LPM, we use the Ternary Content Addressable Memory (TCAM) module combined with the BRAM (Block RAM).

Our TCAM module contains one TCAM memory with 32 lines of 64 bits. In addition, it spends 16 cycles for a write operation and only one cycle for the read operation. We use the Content Addressable Memory (CAM) module combined with the BRAM for the exact match. The CAM module contains one CAM memory with 32 lines of 64 bits. The TCAM is implemented using Xilinx SRL16e primitives [37]. It is generated using Xilinx's IP core generator *coregen* [77]. The CAM is implemented using block RAM (BRAM) instead of SRL16e. This option enables writing on CAM using two cycles instead of 16 cycles. We defined the size of 64 bits to CAM and TCAM memories to optimize the system's design to insert other functionalities on eBPFlow. The size of these memories can be extended to support keys greater than 64 bits. However, it consumes more logical resources that can harm the system's performance. For the array map, we use the DRAM memory.

There are three functions to manipulate the maps: update, delete, and lookup. The update operation updates an item on the map. If the item does not exist, it inserts the item. The delete operation removes the item with the given key. Finally, the lookup

operation searches for the key and returns an item.

The coprocessor needs to know the actual size of the data read/written on the memory map. This information is stored on fields type, key mask, value mask, and maximum number of the *maps table*, shown in Figure 4.7, which holds metadata about each map declared in the currently loaded program. A lookup on the map table is performed on every map operation to retrieve key and value masks used in a bitwise-AND operation with the data to clean any unwanted bits. The coprocessor also uses the map type to switch to the proper memory unit (CAM or TCAM). The r3 register stores a pointer to the item when used to operations with maps. The size of the r3 register is 64 bits based on standard eBPF architecture.

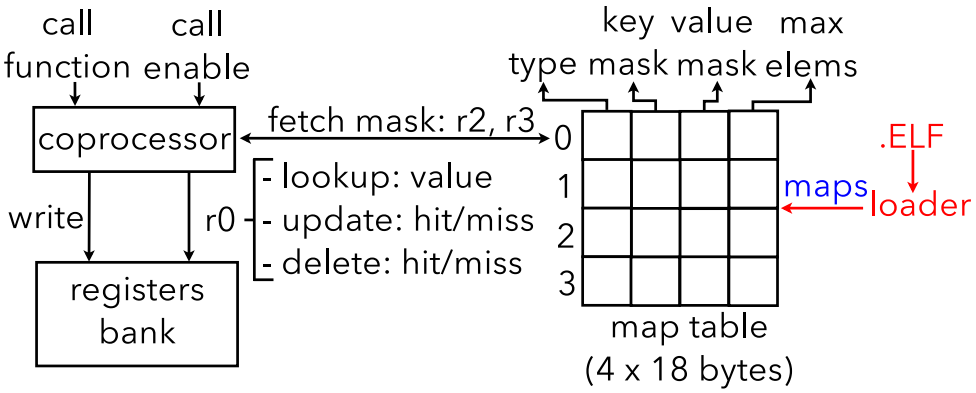


Figure 4.7: Coprocessor and map table.

On eBPFlow, the maps are loaded dynamically via the registers interface. It supports 64 flows simultaneously using static rules through the CAM and TCAM memories. However, if the user uses the wildcard mechanism of the TCAM memory with the operator (*), the number of flows monitored can be increased. Another mechanism to extend the number of flows is to forward the packets to userspace for offline processing via the PCIe bus. However, it is slow, decreasing the system's processing power due to the speed of the PCIe bus and context switch between hardware and userspace.

4.2.6 Call instruction

eBPF allows invoking functions to access tables. In our design, to manipulate maps, we decided to use the TCAM, CAM, and DRAM modules. The function call inside the processor establishes communication with the coprocessor hardware module to manipulate tables. Thus, the processor communicates with the coprocessor module where there is a call instruction. This module identifies what function (lookup, update, delete)

was called through the call instruction immediate opcode parameter. Registers 1 to 4 store the passed parameters on the call instruction. Register r1 indicates which hardware module to communicate (tables TCAM, CAM, DRAM, respectively). Register r2 provides the key. Register r3 stores the item of 64 bits. Register r4 has the TCAM mask item. The function return parameter is through register r0. Since the call instruction requires register values, it can also suffer from hazards in the pipeline. Therefore, the hazard unit has to stall to solve this issue.

4.2.7 Bus, demux and output arbiter

On eBPFlow, we implemented demux and output arbiter modules to manage the receiving of the packet from the RX queue to the eBPF engines and forward the packet from eBPF engines to the output crossbar. Each RX/TX queue has its demux and output arbiter. These modules use the AXI-4 stream interface signals to synchronize the receiving and forwarding of the packet from/to eBPF engines. Moreover, these modules are sequential circuits that depend on AXI-4 stream interface signals value and the hardware register state to control what eBPF engine has access priority. We used the Round Robin algorithm to schedule between eBPF engines using a finite state machine that controls the hardware register responsible for access priority between eBPF engines.

4.2.8 User space

It has a controller, a loader, and tools created at the user level. We implemented it all in the Python language.

Controller: It opens a socket connection TCP/IP to the device to exchange the messages. After establishing the connection, the operator can transmit the eBPF program already compiled as bytecode. Finally, the controller installs the bytecode in the hardware at runtime.

Loader: It is responsible for the following operations: loading code to the processors, appending two instructions, handling maps, and interacting with the processor register interface. *Loader* specially designed for the eBPF processor. At the beginning of every eBPF program, registers r1 and r10 must be initialized with two pointers: one to the packet and one to the stack's top. Since these are specific to the runtime environ-

ment (here, the processor), such initialization is not part of the code generated by *clang* compiler. To handle maps, the compiler adds map information to the eBPF ELF file as a relocation section, which needs to be processed before code execution. *Loader* adjusts all map *call* instructions with their corresponding map values according to the relocation table in the ELF file. Finally, the *loader* interacts with the system through the register interface of the hardware, allowing to update and query the content of maps from the user space at run-time independent from the loading operation of the program on eBPFlow. There is an option on the loader specific only for operations with maps. Operations with maps via user space do not harm the system's performance because it uses the register interface instead of the processing datapath. Moreover, it can query status information about the processor.

Tools: A set of tools were implemented as part of the eBPFlow infrastructure: an eBPF disassembler, a software emulator, and a CLI application to interact with the eBPF engines. The emulator leverages the uBPF [7]. Software emulator aims to replicate the processor's behavior in software. Furthermore, it enables code testing and debugging with well-known tools such as *gdb*, enabling faster and easier bug detection and correction even before deploying the code.

Communication with host: We chose the NetFPGA's interface nf0 to receive and forward the packets sent from hardware to the host. We do not define the number of cores on the host's CPU to process the packets on userspace. Instead, we leave the operator responsible for defining the number of cores according to the processing demand of the host.

4.2.9 Re2c

Re2c [54, 11] is a lexer generator capable of converting regular expressions into fast and optimized finite state machines (FSM). As a result, users can write new protocols in eBPF. We use re2c to convert regular expressions into eBPF-compatible C code. Figure 4.8 presents an overview of the configuration flow and data processing in eBPFlow. The process encapsulates the FSM generated by Re2c within a function with control pointers. It is also necessary for all states to verify that the pointer's current value is greater than the address at the end of the packet to prevent invalid memory access. After this, the code is packaged in a single file and compiled by the eBPF compiler. Then, a controller sends the generated instructions to eBPFlow, saving them in its instruction memory. From that moment on, the system waits for incoming packets, processes them according to the instructions present in the memory, and performs the appropriate actions according to

each packet's content. We do not create an interface between re2c and eBPF-compatible C code, being necessary adaptations for the generated code to work in the eBPFlow. We left the programmer responsible for this task.

We use the unrolling loop directive on eBPF-compatible C code to optimize the number of instructions generated from the loop to travel the packet payload. In some situations where the number of instructions exceeded the maximum number of instructions in the instruction memory, we perform optimizations on code to reduce the number of instructions.

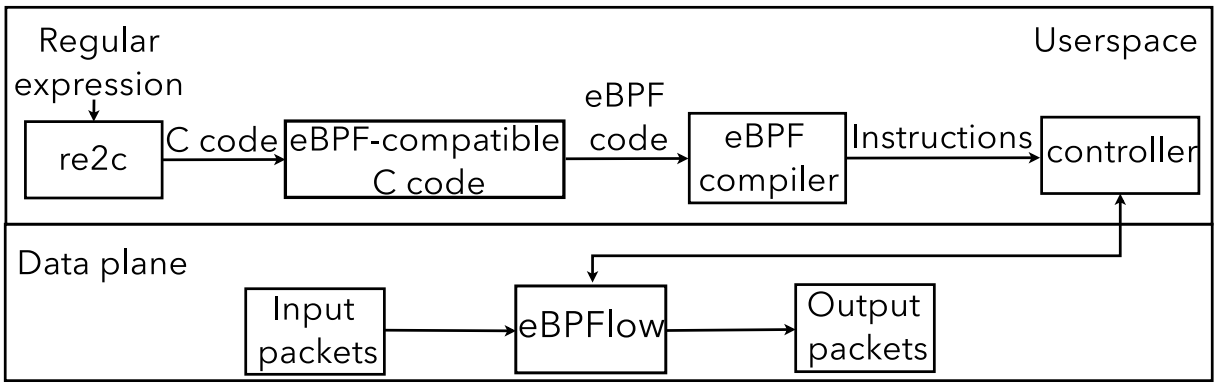


Figure 4.8: Overview of the configuration flow and data processing.

Chapter 5

Offloading Stateless and Stateful NFs

This chapter shows the performance of the eBPFlow in a physical environment to NFs offloading and processing of packets. This chapter is composed of two sections. In the first section (Section 5.1) describes about stateless and stateful NFs implemented to evaluate the system. In the second section (Section 5.2) shows the test environment, results of performance of the system, and a discussion about the eBPFlow design.

5.1 Network Functions

Table 5.1: Network Functions implemented on eBPFlow.

eBPF programs	**#Instructions**	**#LoC in C**	**State (Yes/No)**	**# States**
Wire	5	7	No	0
Stateful Firewall	23	24	No	0
LPM forwarding	30	26	No	0
DDoS Mitigation	35	57	No	0
BitTorrent Packets	181	86	Yes	5
SQL Injection (Tautology)	143	110	Yes	4
SQL Injection (Sleep)	183	117	Yes	2

We have implemented some NFs (Table 5.1) on eBPFlow to demonstrate the offloading of functions and the acceleration in packet processing. For each NF, we present the number of eBPF instructions (#Instructions), the number of C code lines (#LoC in C), is there state (Yes/No)?, and the number of states (#States). Here is the description of the NFs:

Wire: acts as a wire connecting adjacent ports in pairs of two. It performs an

XOR operation between the input port value and 1, which inverts the least significant bit. This value defines the outgoing packet port. It is the most straightforward application and serves as a performance baseline.

LPM Forwarding (LPMF): forwards packets using the NetFPGA's TCAM module, effectively speeding up longest prefix matching (LPM) operations. In addition, this NF can use up to 32 forwarding rules inserted by the user through the loader.

DDoS Mitigation (DDoS): tries to saturate broadband or overload networking equipment's computational resources, limiting the processing or making unavailable services, servers, and the target network. This NF can analyze random ports of UDP packets. Moreover, it can block the attack on a specific port, dropping the packet and not allowing the attack to have success [5].

Stateful Firewall (SFW): is a network firewall that tracks the status and characteristics of network connections, distinguishing packets for different types of communications and propagating only packets that match the active connections [51].

SQL Injection with Tautology (SQL_TAU): this attack is characterized by the insertion of tautologies in an SQL query, making them manipulable. For example, if the system has the query *SELECT * FROM Users WHERE Id = "username"* where *username* is a user-supplied parameter. If no input filter exists, the attacker can exploit the vulnerability by sending the string *"OR 1 = 1* as a parameter. The resulting query will be *SELECT * FROM Users WHERE Id = "" OR 1 = 1*, which is valid and returns all rows in the *Users* table, since *1 = 1* is always true. Figure 5.1 shows the FSM corresponding to the regular expression proposed to detect the threat.

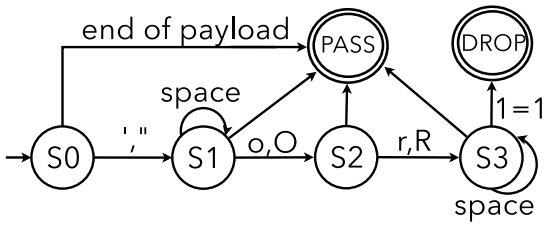


Figure 5.1: SQL Tautology RegEx.

It consists of four states that lead to two possible final actions: PASS if there is no attack or DROP if the malicious string is detected before the end of the payload. The first state detects the beginning of the attack, single or double-quotes. The second state detects the presence of spaces and the keyword OR. Finally, the last state detects the end of the attack, 1=1.

SQL Injection with Sleep function (SQL_SLEEP): this attack allows hackers to look for possible SQL vulnerabilities on a server. It uses the User-Agent field of HTTP requests to send an SQL query that calls the function *sleep*, applying a delay in seconds to the current operation. During the delay period, any further requests received run only

after the end of the first query, which indicates to the attacker that there are vulnerabilities that allow the insertion of other SQL attacks. Figure 5.2 shows the FSM corresponding to the regular expression of this example.

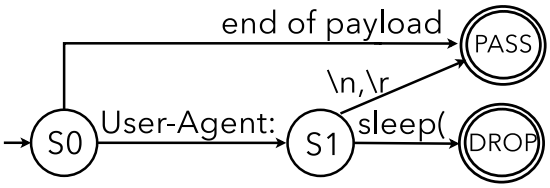


Figure 5.2: SQL Sleep RegEx.

The first state detects the presence of the User-Agent keyword or ends processing if it arrives at the end of the payload. The second state looks for the *sleep (* string, which indicates the presence of the attack within the specified field. The processing terminates if a line break occurs before this string. The SQL injection NFs presented above are examples of functions that use regular expressions to efficiently analyze packet payload. This type of analysis is critical today, in which servers store a large amount of valuable data, demanding protection against this and other types of attacks.

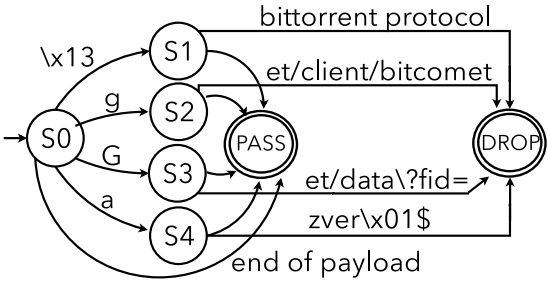


Figure 5.3: Bittorrent Packets RegEx.

BitTorrent Packets (BITP): BitTorrent can cause many simultaneous connections, which can overload the network. This NF detects four BitTorrent packet types based on Strait and Sommer [60]. It is an example of an Application Layer Packet Classifier. Figure 5.3 shows the FSM with five states to match the patterns containing 64 characters. The first state is responsible for detecting the patterns' initial character at the beginning of the packet payload. The following four states detect the rest of the strings. This NF only forwards the packet if the patterns are not present at the end of the payload.

5.2 Evaluation

Here, we present the experimental evaluation of eBPFlow.

5.2.1 Test environment

The testing environment contains one NetFPGA SUME, one server running pktgen-DPDK [64] as a traffic generator, and a custom controller to interact with the eBPFlow's data plane. Our server couples a Netronome Agilio CX SmartNIC and an Intel X710 DA-2 SmartNIC with two 10 Gbps interfaces directly connected to the four NetFPGA SUME ports. We add Intel and Netronome boards on pktgen-DPDK userspace to generate the traffic and to receive the traffic forwarded by NetFPGA SUME, running the eBPFlow design. In addition, the server and machine with coupled NetFPGA SUME have i7-7700 processors clocked at 3.60 GHz containing eight cores and 8 GB of RAM. Figure 5.4 presents the experiments topology of the system.

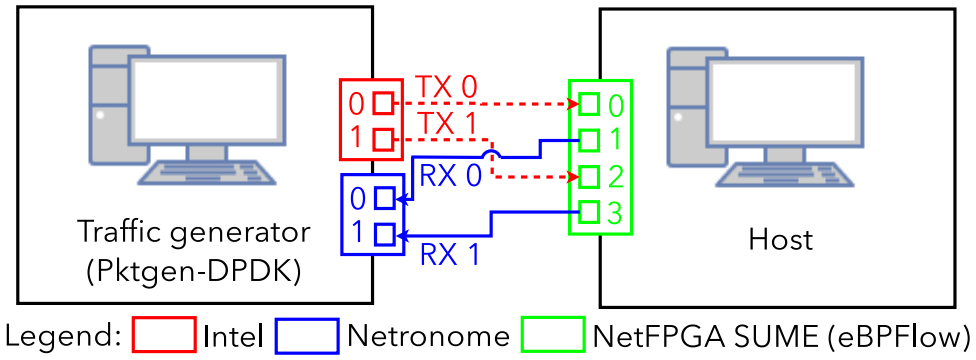
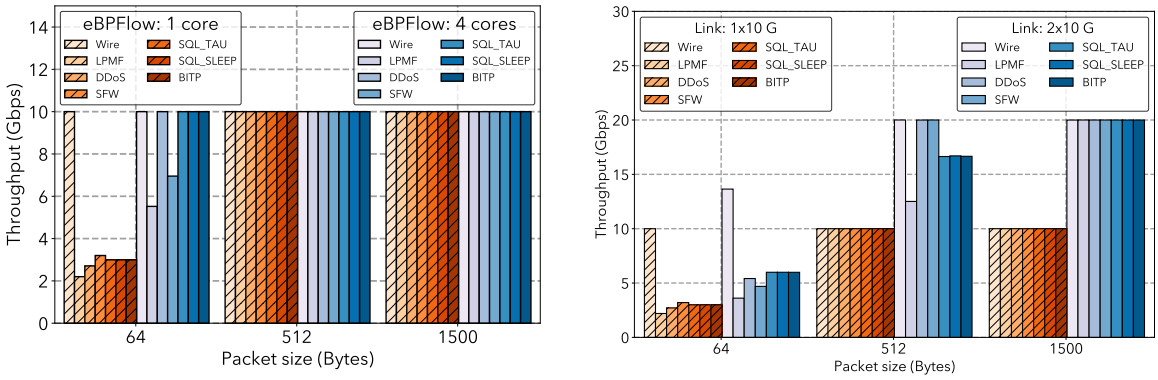


Figure 5.4: Experiments topology.

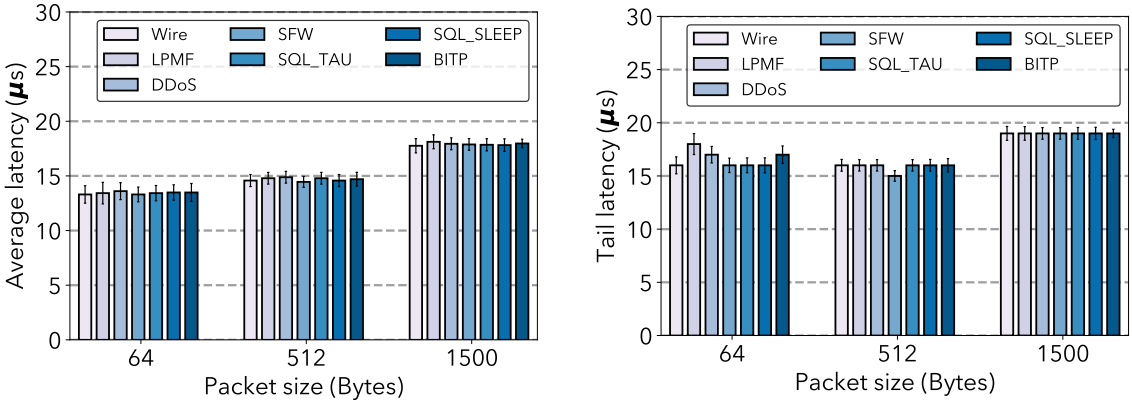
5.2.2 Throughput

We evaluated the performance of the eBPFlow to packet processing rates 64 bytes (minimum-sized), 512 bytes (middle-sized), 1,500 bytes (maximum-sized), respectively, for network functions described in Section 5.1. Moreover, we used the same parameters



(a) Throughput: Number of cores.

(b) Throughput: Link 1x10 and 2x10 G.



(c) Average latency.

(d) 99th-percentile latency.

Figure 5.5: eBPFlow performance: throughput and latency.

to evaluate the system's performance, the number of processing cores per port (one and four), and the number of links generating the traffic (1x10 and 2x10 G).

Figure 5.5a summarizes the system's throughput according to the number of cores. For packets of 64 bytes using one processing core, all network functions except the Wire achieve throughput less than 4 Gbps, demonstrating that the number of cores affects the system's performance. However, when the number of cores passes from one to four per port, the throughput of all NFs doubles or achieves line rate (10 Gbps). Wire, DDoS mitigation, SQL Injection attacks, and Bittorrent filter achieve line rate for 64 bytes packets. Moreover, these NFs do not realize operations with maps using CAM/TCAM memories. Stateful Firewall and LPM forwarding have improved throughput with the increase of the number of cores from one to four but yet had throughput reduction due to spent time with operations of access to maps (Section 5.2.5 presents the time in clock cycles and microsecond of each operation).

Figure 5.5b presents the throughput of the eBPFlow based on the number of links generating traffic per port. We generated traffic using one and two links of 10 Gbps to evaluate the system's performance when stressed. To packets of 64 bytes using one and

two links of 10 Gbps, the system had a throughput of less than 6 Gbps per port for all NFs, except the Wire. With two links of 10 Gbps for 512 bytes packets, Wire, DDoS mitigation, and Stateful firewall achieve a throughput of 20 Gbps. On the other hand, SQL Injection attacks and Bittorrent filter achieve a throughput of 16 Gbps, reducing throughput due to the number of instructions because they spend more time processing packets than other NFs and the bottleneck on the FIFOs of the datapath. LPM forwarding has been improved throughput using two links, but the spent time with operations of access to map TCAM harmed the throughput to this network function. Finally, to 1,500 bytes packets, using one and two links, the eBPFlow achieved the maximum throughput (10 and 20 Gbps), respectively, to all NFs without packet loss.

5.2.3 Latency

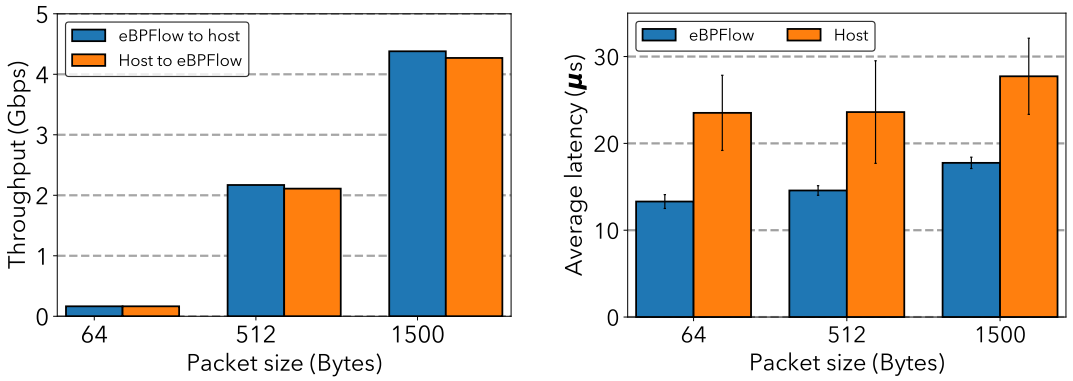
In addition to the throughput, we also measured the average and tail latencies for each NF (Figures 5.5c and 5.5d) using *pktgen-DPDK*, with 1 μs precision. pktgen-DPDK measures the end-to-end latency by adding a timestamp on the packet payload. It calculates latency stats, sends the packet to the network, and after the packet returns, gets the timestamp and calculates the time stats. In this experiment, we load the Intel smartNIC on pktgen-DPDK userspace and use one port to send packets and another port to receive the back packets. Moreover, we repeated each experience 33 times for each NF for packet sizes 64, 512, and 1,500 bytes. As expected, latency increased according to the packet size increase because the number of words on the data plane increased, taking more time to run the entire program. All experiments had a latency of less than 20 μs. This metric demonstrates little change in the processing time between same-sized packets for a single NF, leading to reduced jitter. Similarly, the tail latency is close to the average value in almost all cases. The bars in Figures 5.5c and 5.5d represent the standard deviation, which was close to zero in all cases.

5.2.4 Communication with host

It is a functionality important when network devices with packets processing in hardware are overloaded or arithmetics operations are not synthesizable (e.g., MULT and DIV 64 bits, MOD (64 and 32 bits)). When there is communication between the data

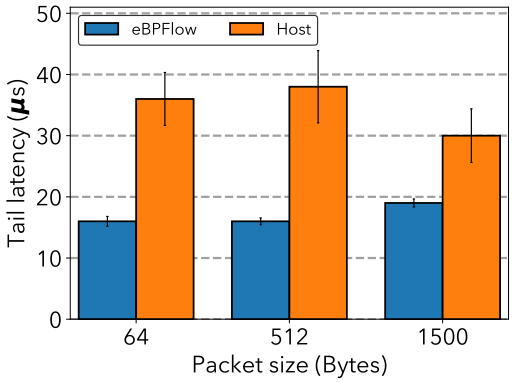
plane and control plane, the data plane can send packets to the control plane for offline processing and then forward them back to the data plane, dividing the workload between hardware and software. Moreover, it supports operations not synthesizable in hardware.

The communication with the host on NetFPGA SUME occurs through the PCIe generation 3 (PCIe v3) bus with the driver SUME RIFFA managing the communication between hardware and operating system. NetFPGA SUME design couples the IP core PCIe v3 module available by Xilinx [74] on code. We adapt the eBPFlow design, adding a new action (Table 4.2) responsible for sending and receiving packets between eBPFlow and host. In this experiment, we evaluated the throughput and latency of the communication between eBPFlow and the host running on userspace.



(a) Throughput.

(b) Average latency.



(c) 99th-percentile latency.

Figure 5.6: Communication with host.

Throughput: Figures 5.6a summarizes the throughput of the eBPFlow communicating with the host. We measured the throughput using iPerf3 tool [23] in two directions: eBPFlow communicating with the host (eBPFlow to host) and host communicating with eBPFlow (host to eBPFlow). We executed iPerf's client program on the server and iPerf's server program running on the machine with NetFPGA SUME. Moreover, we generated traffic with TCP packets of 64, 512, and 1,500 bytes in one second. The results show

that the system achieves a throughput of less than 5 Gbps to all packet sizes in both directions. It occurs due to context switch and the IP core PCIe v3 with 8-lane of the Xilinx to achieve a maximum transference speed of 5 Gbps [74].

Latency: In this experiment, we measured the average (Figure 5.6b) and tail (Figure 5.6c) latencies of the eBPFlow communicating with the host using *pktgen-DPDK*, with 1 μs precision. On a host (machine with coupled NetFPGA), we used Linux's traffic control subsystem called (tc) [34] to receive the packets from board to host and send them back from host to board. Tc is responsible for setup traffic control in the Linux kernel. Moreover, we repeated each experience 33 times for each NF for packet sizes 64, 512, and 1,500 bytes. eBPFlow's latencies (average and tail) had a latency of less than 20 μs with a standard deviation close to zero. While host average and tail latencies were less than 30 and 40 μs with a difference of 10 and 20 μs if compared with eBPFlow's latencies. These results demonstrate that sending the packet from board to host is slower than processing the packet only on the board. However, communication with the host allows dividing the workload between hardware and software and supporting operations not synthesizable.

5.2.5 Coprocessor measurement

This experiment evaluates the coprocessor time spent on each function call (lookup, delete, and update) on eBPFlow. We performed this experiment by adding time registers over the coprocessor's Verilog code. In addition, we created an application to read and add the time values to obtain the time spent after the function call execution. Table 5.2 presents the time in clock cycles (clks) and microseconds (μs) spent on the coprocessor to each eBPF function call. The measurement begins when the coprocessor triggers the call function processing. Each register on code increments its value according to the time spent executing a coprocessor's code-specific functionality. After the coprocessor finishes the function call execution, the application reads and obtains the total time spent on the function call. We compare the times obtained via simulation and tests in the real environment to validate the experiment.

Table 5.2: Time spent function call on coprocessor.

	Maps	
Function call	CAM	TCAM
Lookup	13 clks (0.065 μs)	13 clks (0.065 μs)
Delete	15 clks (0.075 μs)	29 clks (0.145 μs)
Update	19 clks (0.095 μs)	33 clks (0.165 μs)

5.2.6 eBPFlow performance on packet processing

We evaluated the packet processing capacity of the eBPFlow compared to kernel and Netronome that support offloading of eBPF programs. The kernel runs eBPF programs in software, while Netronome uses a SmartNIC. To compare both systems, we choose the network functions SQL Sleep because it is the eBPF program with more significant numbers of instructions (Table 5.1). We executed this NF on the three systems. Moreover, we measured the throughput in millions of packets per second (Mpps) to evaluate the packet processing power between software (using kernel Linux 5.0.4) and hardware (using Netronome and eBPFlow). We generated packet rates of 64, 512, and 1500 bytes using pktgen-DPDK.

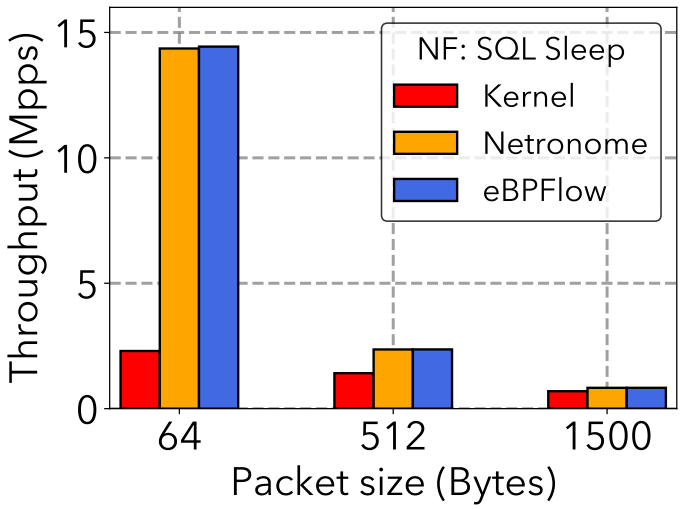


Figure 5.7: Systems performance on packet processing.

On the kernel, we insert a map structure in each network function to count the number of processed packets and measure this information using the xdp-stats tool available in [71] for each experiment. We evaluated Netronome SmartNIC with their stat_watch.py tool [45]. Figure 5.7 presents the throughput in Mbps of each system, respectively, to evaluate network function. Netronome and eBPFlow processes approximately 12.05, 0.87, 0.15 Mpps more than the kernel to packet sizes 64, 512, and 1500 bytes. The packet processing between Netronome and eBPFlow is similar to all packet sizes. However, eBPFlow provides functionalities not available on Netronome, such as parallelism per port using eBPF cores reserved per port and parallelism on packet forwarding using an integrated output crossbar on the system's data plane.

5.2.7 eBPFlow performance compared to other systems

We compared the performance of two systems (hXDP and CPU x86@3.7 GHz) that are similar to eBPFlow. hXDP runs eBPF code on the NetFPGA. CPU means we execute code in the kernel. Kernel executes eBPF code on the CPU. We evaluated the throughput and latency of the systems to 64 bytes packets (minimum-size) using a firewall as a network function. We choose the firewall as NF based on experiments of [9]. Moreover, to compare the systems fairly, for eBPFlow, we used one physical port to receive packets, one physical port to send packets, and four eBPF engines. This is the same setup applied to CPU and hXDP [9]. This is the same setup applied to CPU and hXDP [9]. Furthermore, we use the paper's throughput and latency results to compare the CPU and hXDP results with eBPFlow.

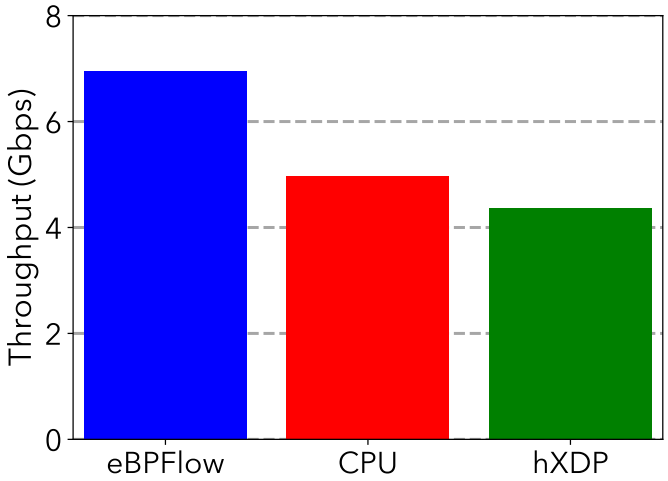


Figure 5.8: Throughput: eBPFlow, CPU, and hXDP.

Throughput: Figure 5.8 presents a throughput comparison of the systems. hXDP achieves a throughput of 4.36 Gbps. CPU x86@3.7 GHz gets a throughput of 4.97 Gbps. Finally, eBPFlow obtains a throughput of 6.95 Gbps. This result demonstrates that the eBPFlow has a processing performance improvement of 2.59 and 1.98 Gbps over hXDP and CPU x86@3.7 GHz, respectively.

Latency: Figure 5.9 presents a latency comparison of the systems on packet forwarding. CPU x86@3.7 GHz gets a latency of 11 μs. hXDP achieves a latency of 3 μs. Finally, eBPFlow obtains a latency of 13.30 μs. This result demonstrates that the eBPFlow spends more time on packet forwarding 2.3 μs and 10.3 μs about CPU x86@3.7 GHz and hXDP, respectively. eBPFlow overcomes this limitation by providing parallelism in design and processing more packets than both systems.

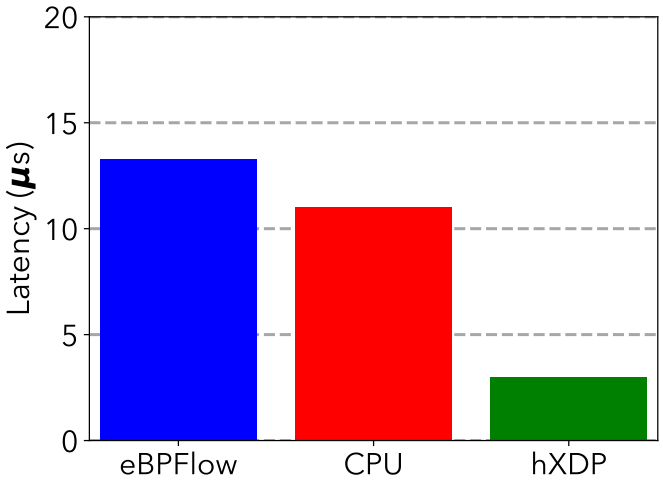


Figure 5.9: Latency: eBPFlow, CPU, and hXDP.

5.2.8 Power

When idle, the NetFPGA consumes 16 W. However, when we synthesize eBPFlow, the power consumption is 22 W regardless of the packet rate or running program [51]. Devices as Netronome [46, 22], Intel Core i7-7700 [1], 1U rackmount x86 [67], and P4 Wedge 100BF-32X [15] have power consumption of approximately 25-40 W, 65 W, 300-350 W, and 436 W, which demonstrates that eBPFlow saves power in comparison to the listed devices.

5.2.9 Discussion

Packet reordering: Network devices use multiple packet processing cores to provide parallelism. However, they can cause packet reordering contributing to the congestion of the network due to the number of packets out order stored on FIFOs of the eBPFlow until the TCP adjusts the send the packets to the receptor. On eBPFlow, we leave the TCP protocol responsible for managing packet reordering because the TCP guarantees that the packets will be delivered in the same order in which they were sent. Moreover, we do not spend the logic resources of the hardware with storage buffers, registers, and finite state machines to control and handle packets to put them in order, which simplifies the system's design. Similar systems to eBPFlow, such as Netronome and hXDP, also do

not do packet reordering, leaving the TCP responsible for this task.

RegExs processing in parallel: The eBPFlow supports the processing of one RegEx on the data path. It occurs because RegExs are transformed into eBPF instructions, loaded in one instruction memory with a double buffer system shared between eBPF engines. We used one shared instruction memory to reduce the number of used logic resources on design. However, to support RegExs processing in parallel new circuits on design need to be added, such as one instruction memory per eBPF engines group or to support the tail call function that executes eBPF programs in parallel, jumping between them.

Maps access race condition: All eBPF engines share the maps on eBPFlow. However, only an eBPF engine has access priority to the maps by time. We created an arbiter module to manage the access between eBPF engines. The arbiter defines the access priority based on the scheduling algorithm Round Robin. It defines the current eBPF engine with access priority and warns the other eBPF engines that the map is busy. The eBPF engines without access priority wait until each one of them has permission for access.

Extending maps capacity: The wildcard mechanism allows more flows to be processed on eBPFlow using the operator (*). However, a service provider or networking operator can classify or process more than 32 flows. Therefore, they can extend the eBPFlow design by adding the DDR3 DRAM memory as an array map, available on the NetFPGA SUME. This memory is suitable for packet buffering. In addition, it has a total capacity of 8 Gb with a clock frequency of 933 MHz. Another mechanism to extend the number of flows is to forward the packets to userspace for offline processing via the PCIe bus. However, it is slow, decreasing the system's processing power due to the speed of the PCIe bus and context switch between hardware and userspace.

Chapter 6

Conclusion and future work

Providing flexibility and programmability of the data plane on networking devices with abstraction and performance is a requirement that is necessary for the next generation of networking devices. This thesis proposes eBPFlow, a platform that support offloading of NFs and acceleration on processing of packets. eBPFlow, has hardware and software components, is a packet processing platform targeted for high-performance data plane. eBPFlow is built on top of the NetFPGA SUME platform.

We designed and implemented in hardware a system with multiple eBPF virtual machines at its core. This system allows executing parse, matching, and actions dynamically through eBPF instructions. The system is protocol independent, and it allows the use of new fields, facilitating the adoption of new protocols and services The eBPFlow allows changing the image of the eBPF program at runtime, allowing to modify how the eBPFlow will process flows. eBPFlow is capable of processing both the packet header and payload at line rate.

In short, Chapter 2 presented an overview of themes approached in this thesis, such as eBPF technology, programmable data plane, traffic classification, deep packet inspection, FPGA, and NetFPGA. Chapter 3 demonstrated a discussion about related work found in the literature that supports programmability using different technologies in hardware and software targets. In addition, this Chapter introduced the art state of the literature and highlighted functionalities and scientific contributions of the system does not present in similar systems to eBPFlow. Chapter 4 approached details about the eBPFlow design and implementation built on top of the NetFPGA SUME. Moreover, this Chapter presented optimizations on the data plane realized to improve the system performance and provide different parallelism types on NFs offloading and packet processing, such as parallelism of instructions, parallelism per port, and parallelism on the forwarding of packets. Chapter 5 presented an evaluation broad of the system with many experiment types and results of offloading stateless and stateful NFs in a physical test environment. Finally, this Chapter discusses the obtained results, future work, and publications.

6.1 Results

This research advances the art state of the networking community with scientific contributions and relevant results to the area. In addition, the obtained results demonstrate that the eBPFlow provides flexibility and programmability on the data plane with abstraction and performance.

The throughput results on packet processing show that the eBPFlow achieves line rate to most network functions and packet sizes. However, factors such as NFs that use maps, 64 bytes packet processing, and the number of cores can affect and reduce the system's throughput. The latency results on the packet process present latency between 20 μs and 40 μs, demonstrating little change in the processing time between same-sized packets for a single NF, leading to reduced jitter.

The throughput result of communication with the host shows that the system achieves a throughput of fewer than 5 Gbps to all packet sizes in both directions, not achieving at line rate. However, it occurred due to the context switch and the maximum transference speed supported by IP core PCIe. eBPFlow's latencies results communicating with the host (average and tail) had a latency of less than 20 μs with a standard deviation close to zero. While host average and tail latencies were less than 30 and 40 μs with a difference of 10 and 20 μs if compared with eBPFlow's latencies. These results demonstrate that sending the packet from board to host is slower than processing the packet only on the board. However, communication with the host allows dividing the workload between hardware and software and supporting operations not synthesizable in hardware.

We evaluate the time spent on each function call on eBPFlow, comparing the times obtained through simulation and tests in a physical environment on coprocessor measurement results. This result allows knowing how much time the coprocessor spent in each function call. The performance results comparing the eBPFlow with kernel and Netronome present that Netronome and eBPFlow processes approximately 12.05, 0.87, 0.15 Mpps more than the kernel for all packet sizes. The packet processing between Netronome and eBPFlow is similar to all packet sizes. However, eBPFlow provides parallelism per port and parallelism on packet forwarding not available on Netronome to improve the system's processing performance. The performance results comparing the eBPFlow with hXDP and CPU demonstrate that the eBPFlow has a processing performance improvement of 2.59 and 1.98 Gbps over hXDP and CPU, respectively. Finally, the power consumption result shows that the eBPFlow has a little power consumption (approximately 22 W) and saves power compared to evaluated devices.

6.2 Future work

We intend to create three versions of the eBPFlow: (1) A design that supports an eBPF CISC instruction set. CISC Instructions perform multiple operations in a single instruction, such as memory access and mathematical calculations. It, combined with eBPF, can simplify operations with packets using fewer instructions and improve the system's processing performance; (2) A design that supports Service Function Chaining (SFC). SFC allows the processing of NFs in parallel, providing parallelism in the execution of NFs; and (3) A design that supports measurement primitives and metrics of the networking performance. This system will combine the programmability and flexibility of the eBPF with accurate hardware measurements. These features together are essential to the traffic engineering of modern networks. All these designs will be implemented on the NetFPGA PLUS, it is a codebase built for Xilinx Alveo Data Center Accelerator Card based on Xilinx Virtex Ultrascale+ FPGAs to networks of 100 Gbps [44].

6.3 Publications

This section lists the scientific papers generated with this thesis.

- **PACÍFICO, R. D.**; DUARTE, L. F.; VIEIRA, L. F.; RAGHAVAN, B.; NACIF, J. A.; VIEIRA, M. A. eBPFlow: a Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF. In: IEEE/ACM Transactions on Networking, 2023.
- **PACÍFICO, R. D.**; DUARTE, L. F.; NACIF, J. A.; VIEIRA, M. A. Function as a Service Offloaded to a SmartNIC. In: IEEE Latin-American Conference on Communications - LATINCOM, 2022, Rio de Janeiro/RJ.
- **PACÍFICO, R. D.**; DUARTE, L. F.; CASTANHO, M. S.; VIEIRA, L. F.; NACIF, J. A.; VIEIRA, M. A. Application Layer Packet Classifier in Hardware. In: IFIP/IEEE International Symposium on Integrated Network Management, 2021, Bordeaux.
- **PACÍFICO, R. D.**; DUARTE, L. F.; VIEIRA, M. A.; NACIF, J. A. Sistema de Detecção de Intrusão Serverless em uma SmartNIC. In: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2021, Uberlândia/MG.

- **PACÍFICO, R. D.**; DUARTE, L. F.; CASTANHO, M. S.; NACIF, J. A.; VIEIRA, M. A. Sistema de processamento de pacotes serverless. In: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2020, Rio de Janeiro/RJ.

- VIEIRA, A. G. ; PANTUZA, G. ; FREIRE, J. H. F. ; DUARTE, L. F.; **PACÍFICO, RACYUS D. G.**; Vieira, M. A. ; VIEIRA, L. F.; NACIF, J. A. Computação Serverless: conceito, aplicações e desafios. Computação Serverless: conceito, aplicações e desafios. In: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2020, Rio de Janeiro/RJ.

- VIEIRA, M. A. M.; CASTANHO, M. S.; **PACÍFICO, R. D.** ; SANTOS, E. R.; JÚNIOR, E. P.; VIEIRA, L. F. Fast Packet Processing with eBPF and XDP. In: ACM Computing Surveys, 2020.

- VIEIRA, M. A.; **PACÍFICO, R. D.**; CASTANHO, M. S.; SANTOS, E. R.; VIEIRA, L. F. Processamento Rápido de Pacotes com eBPF e XDP. In: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2019, Gramado/RS.

- **PACÍFICO, R. D.**; COELHO, G. R.; NACIF, J. A.; VIEIRA, M. A. Roteador SDN em hardware independente de protocolo com análise, casamento e ações dinâmicas. In: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2018, Campos do Jordão/SP.

References

[1] CPU agent. Intel core i7-7700 review. https://www.cpuagent.com/cpu/intel-core-i7-7700/summary/nvidia-geforce-gtx-980-ti, 2016. Accessed on 12/27/2021.

[2] ARM. Axi4 specification. http://www.gstitt.ece.ufl.edu/courses/fall1\5/eel4720_5721/labs/refs/AXI4_specification.pdf, 2011. Accessed in: 10/28/2021.

[3] David Beckett, Jaco Joubert, and Simon Horman. Host dataplane acceleration (hda) tutorial. In *ACM SIGCOMM '18 - Special Interest Group on Data Communication Tutorial*, New York, NY, USA, 2018. ACM.

[4] Pavel Benáček, Viktor Puš, Hana Kubátová, and Tomáš Čejka. P4-to-vhdl: Automatic generation of high-speed input and output network blocks. *Microprocessors and Microsystems*, 56:22 – 33, 2018.

[5] Gilberto Bertin. XDP in practice: integrating XDP into our DDoS mitigation. In *Proceedings of the Technical Conference on Linux Networking*, page 5, 2017.

[6] Roberto Bifulco and Gábor Rétvári. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–7. IEEE, 2018.

[7] Big Switch Networks, Inc. Userspace ebpf vm. https://github.com/iovisor/ubpf, 2020. Accessed on 06/22/2020.

[8] 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. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[9] 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, 2020.

[10] Mihai Budiu. Compiling p4 to ebpf. https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4, 2015. Accessed in: 10/28/2021.

[11] Peter Bumbulis and Donald D. Cowan. Re2c: A more versatile scanner generator. *ACM Lett. Program. Lang. Syst.*, 2(1–4):70–84, March 1993.

[12] Matheus S. Castanho, Cristina K. Dominicini, Magnos Martinello, and Marcos A. M. Vieira. Chaining-box: A transparent service function chaining architecture leveraging bpf. *IEEE Transactions on Network and Service Management*, 19(1):497–509, 2022.

[13] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[14] Mou-Sen Chen, Ming-Yi Liao, Pang-Wei Tsai, Mon-Yen Luo, Chu-Sing Yang, and C Eugene Yeh. Using netfpga to offload linux netfilter firewall. *2nd North American NetFPGA Developers Workshop*, 2010.

[15] Edge Core. Wedge 10032x datasheet. https://www.edge-core.com/_upload/images/Wedge_100-32X_DS_R04_20170615.pdf, 2017. Accessed on 12/27/2021.

[16] Tooska Dargahi, Alberto Caponi, Moreno Ambrosin, Giuseppe Bianchi, and Mauro Conti. A survey on the security of stateful sdn data planes. *IEEE Communications Surveys & Tutorials*, 19(3):1701–1725, 2017.

[17] Digilient. Structure of an fpga. https://digilent.com/blog/structure-of-an-fpga/, 2021. Accessed in: 11/04/2021.

[18] eBPFlow. Repository with ebpflow source code. https://github.com/racyusdelano/ebpflow, 2021. Accessed on 12/28/2021.

[19] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. Nica: An infrastructure for inline acceleration of network applications. In *USENIX Annual Technical Conference*, pages 345–362, 2019.

[20] Facebook. Katran source code repository. https://github.com/facebookincubator/katran, 2018. Accessed in: 10/28/2021.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[22] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. Deepmatch: practical deep packet inspection in the

data plane using network processors. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 336–350, 2020.

[23] iPerf 3. Documentation iperf 3. https://iperf.fr/, 2014. Accessed on 12/31/2021.

[24] Simon Jouet and Dimitrios P. Pezaros. Bpfabric: Data plane programmability for software defined networks. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '17, pages 38–48, Piscataway, NJ, USA, 2017. IEEE Press.

[25] Enio Kaljic, Almir Maric, Pamela Njemcevic, and Mesud Hadzialic. A survey on data plane flexibility and programmability in software-defined networking. *IEEE Access*, 7:47804–47840, 2019.

[26] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 2021.

[27] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. Sax-pac (scalable and expressive packet classification). In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 15–26, 2014.

[28] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[29] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008.

[30] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary cams. *ACM SIGCOMM Computer Communication Review*, 35(4):193–204, 2005.

[31] 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. ACM.

[32] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. Panic: A high-performance programmable nic for multi-tenant networks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 243–259, 2020.

[33] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE*, 107(4):746–764, 2019.

[34] Linux. tc(8) linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html, 2021. Accessed on 01/03/2021.

[35] Alex X Liu and Mohamed G Gouda. Complete redundancy removal for packet classifiers in tcams. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):424–437, 2008.

[36] Yongshuai Liu, Jiaxin Ding, Zhi-Li Zhang, and Xin Liu. Clara: A constrained reinforcement learning based resource allocation framework for network slicing. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 1427–1437. IEEE, 2021.

[37] Kyle Locke. Parameterizable content-addressable memory. *Xilinx Application Note XAPP1151*, 2011.

[38] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, MSE '07, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[39] Yadi Ma and Suman Banerjee. A smart pre-classifier to reduce power consumption of tcams for multi-dimensional packet classification. *ACM SIGCOMM Computer Communication Review*, 42(4):335–346, 2012.

[40] D. F. Macedo, D. Guedes, L. F. M. Vieira, M. A. M. Vieira, and M. Nogueira. Programmable networks: From software-defined radio to software-defined networking. *IEEE Communications Surveys Tutorials*, 17(2):1102–1125, 2015.

[41] 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, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[42] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[43] Andrew W. Moore. Netfpga project. www.netfpga.org, 2007. Accessed on 12/21/2017.

[44] Andrew W. Moore. Netfpga plus project. https://netfpga.org/NetFPGA-PLUS.html, 2021. Accessed on 27/04/2022.

[45] Netronome. Netronome flow processor (nfp) kernel drivers. https://github.com/Netronome/nfp-drv-kmods/blob/master/tools/stat_watch.py, 2022. Accessed on 02/09/2022.

[46] Open-nfp. Smartnic programming models. https://open-nfp.org/static/pdfs/dxdd-eu-smartnic-prog-models-2017-06-07.pdf, 2017. Accessed on 12/27/2021.

[47] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for large-scale dnn systems. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–23. IEEE, 2014.

[48] Racyus DG Pacífico, Matheus S Castanho, Luiz FM Vieira, Marcos AM Vieira, Lucas FS Duarte, and José AM Nacif. Application layer packet classifier in hardware. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 515–522. IEEE, 2021.

[49] Racyus DG Pacífico, Lucas B Silva, Gerferson R Coelho, Pablo G Silva, Alex B Vieira, Marcos AM Vieira, Ítalo FS Cunha, Luiz FM Vieira, and José AM Nacif. Bloomtime: space-efficient stateful tracking of time-dependent network performance metrics. *Telecommunication Systems*, pages 1–23, 2020.

[50] Gabriel Arquelau Pimenta Rodrigues, Robson de Oliveira Albuquerque, Flávio Elias Gomes de Deus, Gildásio Antônio De Oliveira Júnior, Luis Javier García Villalba, Tai-Hoon Kim, et al. Cybersecurity and network forensics: Analysis of malicious traffic towards a honeynet with deep packet inspection. *Applied Sciences*, 7(10):1082, 2017.

[51] 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, 2019. USENIX Association.

[52] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal,

Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[53] Yun R Qu, Hao H Zhang, Shijie Zhou, and Viktor K Prasanna. Optimizing many-field packet classification on fpga, multi-core general purpose processor, and gpu. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 87–98. IEEE, 2015.

[54] re2c Organization. re2c. https://re2c.org/, 1993. Accessed on 06/18/2020.

[55] André Ribeiro and Helder Pereira. L7 classification and policing in the pfsense platform. *21st International Teletraffic Congress (ITC 21), Paris, France*, 2009.

[56] Davide Sanvito, Daniele Moro, and Antonio Capone. Towards traffic classification offloading to stateful sdn data planes. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–4. IEEE, 2017.

[57] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux socket filtering aka berkeley packet filter (bpf). www.kernel.org/doc/Documentation/networking/filter.txt, 1993. Accessed on 12/15/2017.

[58] 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*, pages 213–224, 2003.

[59] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM.

[60] Matthew Strait and Ethan Sommer. L7 filter - bittorrent. http://l7-filter.sourceforge.net/layer7-protocols/protocols/bittorrent.pat, 2003. Accessed on 06/22/2020.

[61] Henning Stubbe. P4 compiler & interpreter: A survey. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*, 47, 2017.

[62] 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. IEEE, 2013.

[63] The Zeek Project. Zeek. https://www.zeek.org/, 2020. Accessed in: 01/10/2020.

[64] D. Turull, P. Sjödin, and R. Olsson. Pktgen: Measuring performance on high speed networks. *Computer Communications*, 82:39–48, 2016.

[65] Matteo Varvello, Rafael Laufer, Feixiong Zhang, and TV Lakshman. Multilayer packet classification with graphics processing units. *IEEE/ACM Transactions on Networking*, 24(5):2728–2741, 2015.

[66] Juan Camilo Vega, Marco Antonio Merlini, and Paul Chow. Ffshark: a 100g fpga implementation of bpf filtering for wireshark. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 47–55. IEEE, 2020.

[67] Vertatique. Average power use per server. https://www.vertatique.com/average-power-use-server, 2015. Accessed on 12/27/2021.

[68] 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. *ACM Comput. Surv.*, 53(1), February 2020.

[69] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.

[70] Shie-Yuan Wang and Jen-Chieh Chang. Design and implementation of an intrusion detection system by using extended bpf in the linux kernel. *Journal of Network and Computer Applications*, page 103283, 2021.

[71] XDP. Xdp project: Xdp hands-on tutorial. https://github.com/xdp-project/xdp-tutorial, 2022. Accessed on 02/09/2022.

[72] Xilinx. Virtex-7 family overview. https://www.xilinx.com, 2010. Accessed on 12/31/2021.

[73] Xilinx. Axi4 reference guide. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, 2011. Accessed in: 10/28/2021.

[74] Xilinx. 7 series fpgas integrated block for pci express v3.0. https://www.xilinx.com/support/documentation/ip_documentation/pcie_7x/v3_0/pg054-7series-pcie.pdf, 2014. Accessed on 12/31/2021.

[75] Xilinx. Axi reference guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, 2017. Accessed in: 10/28/2021.

[76] Xilinx. Ug901 - vivado synthesis. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug901-vivado-synthesis.pdf, 2018. Accessed in: 10/29/2021.

[77] Xilinx. Xilinx core generator system. https://www.xilinx.com/products/design-tools/coregen.html, jan 2018. Accessed in: 10/28/2021.

[78] Jane Yen, Jianfeng Wang, Sucha Supittayapornpong, Marcos AM Vieira, Ramesh Govindan, and Barath Raghavan. Meeting slos in cross-platform nfv. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 509–523, 2020.

[79] Xuzhi Zhang and Russell Tessier. Service chaining for heterogeneous middleboxes. *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 263–267, 2020.

[80] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1083–1100, 2020.

[81] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sep. 2014.

Appendix A

eBPFlow: ALU instructions

Arithmetic 64 bits		
Opcode	**Mnemonic**	**Pseudocode**
0x07	add dst, imm	$dst = dst + imm$
0x0f	add dst, src	$dst = dst + src$
0x17	sub dst, imm	$dst = dst - imm$
0x1f	sub dst, src	$dst = dst - src$
0x47	or dst, imm	$dst = dst\ or\ imm$
0x4f	or dst, src	$dst = dst\ or\ src$
0x57	and dst, imm	$dst = dst\ and\ imm$
0x5f	and dst, src	$dst = dst\ and\ src$
0x67	lsh dst, imm	$dst = dst << imm$
0x6f	lsh dst, src	$dst = dst << src$
0x77	rsh dst, imm	$dst = dst >> imm$ (logical)
0x7f	rsh dst, src	$dst = dst >> src$ (logical)
0x87	neg dst	$dst = not\ dst$
0xa7	xor dst, imm	$dst = dst\ xor\ imm$
0xaf	xor dst, src	$dst = dst\ xor\ src$
0xb7	mov dst, imm	$dst = imm$
0xbf	mov dst, src	$dst = src$
0xc7	arsh dst, imm	$dst = dst >> imm$ (arithmetic)
0xcf	arsh dst, src	$dst = dst >> src$ (arithmetic)
Arithmetic 32 bits		
Opcode	**Mnemonic**	**Pseudocode**
0x04	add32 dst, imm	$dst = dst + imm$
0x0c	add32 dst, src	$dst = dst + src$
0x14	sub32 dst, imm	$dst = dst - imm$
0x1c	sub32 dst, src	$dst = dst - src$
0x24	mul32 dst, imm	$dst = dst * imm$
0x2c	mul32 dst, src	$dst = dst * src$
0x34	div32 dst, imm	$dst = dst\ /\ imm$
0x3c	div32 dst, src	$dst = dst\ /\ src$
0x44	or32 dst, imm	$dst = dst\ or\ imm$
0x4c	or32 dst, src	$dst = dst\ or\ src$
0x54	and32 dst, imm	$dst = dst\ and\ imm$
0x5c	and32 dst, src	$dst = dst\ and\ src$
0x64	lsh32 dst, imm	$dst = dst << imm$
0x6c	lsh32 dst, src	$dst = dst << src$
0x74	rsh32 dst, imm	$dst = dst >> imm$ (logical)
0x7c	rsh32 dst, src	$dst = dst >> src$ (logical)
0x84	neg32 dst	$dst = not\ dst$
0xa4	xor32 dst, imm	$dst = dst\ xor\ imm$
0xac	xor32 dst, src	$dst = dst\ xor\ src$
0xb4	mov32 dst, imm	$dst = imm$
0xbc	mov32 dst, src	$dst = src$
0xc4	arsh32 dst, imm	$dst = dst >> imm$ (arithmetic)
0xcc	arsh32 dst, src	$dst = dst >> src$ (arithmetic)
Byteswap		
Opcode	**Mnemonic**	**Pseudocode**
0xd4 (imm == 16)	le16 dst	$dst = htole16(dst)$
0xd4 (imm == 32)	le32 dst	$dst = htole32(dst)$
0xd4 (imm == 64)	le64 dst	$dst = htole64(dst)$
0xdc (imm == 16)	be16 dst	$dst = htobe16(dst)$
0xdc (imm == 32)	be32 dst	$dst = htobe32(dst)$
0xdc (imm == 64)	be64 dst	$dst = htobe64(dst)$

Memory		
Opcode	**Mnemonic**	**Pseudocode**
0x18	lddw dst, imm	$dst = imm$
0x61	ldxw dst, [src+off]	$dst = *(uint32\_t*)(src + off)$
0x69	ldxh dst, [src+off]	$dst = *(uint16\_t*)(src + off)$
0x71	ldxb dst, [src+off]	$dst = *(uint8\_t*)(src + off)$
0x79	ldxdw dst, [src+off]	$dst = *(uint64\_t*)(src + off)$
0x62	stw [dst+off], imm	$*(uint32\_t*)(dst + off) = imm$
0x6a	sth [dst+off], imm	$*(uint16\_t*)(dst + off) = imm$
0x72	stb [dst+off], imm	$*(uint8\_t*)(dst + off) = imm$
0x7a	stdw [dst+off], imm	$*(uint64\_t*)(dst + off) = imm$
0x63	stxw [dst+off], src	$*(uint32\_t*)(dst + off) = src$
0x6b	stxh [dst+off], src	$*(uint16\_t*)(dst + off) = src$
0x73	stxb [dst+off], src	$*(uint8\_t*)(dst + off) = src$
0x7b	stxdw [dst+off], src	$*(uint64\_t*)(dst + off) = src$
Branch		
Opcode	**Mnemonic**	**Pseudocode**
0x05	ja +off	$PC = PC + off$
0x15	jeq dst, imm, +off	$PC = PC + off\ \ if\ dst == imm$
0x1d	jeq dst, src, +off	$PC = PC + off\ \ if\ dst == src$
0x25	jgt dst, imm, +off	$PC = PC + off\ \ if\ dst > imm$
0x2d	jgt dst, src, +off	$PC = PC + off\ \ if\ dst > src$
0x35	jge dst, imm, +off	$PC = PC + off\ \ if\ dst >= imm$
0x3d	jge dst, src, +off	$PC = PC + off\ \ if\ dst >= src$
0xa5	jlt dst, imm, +off	$PC = PC + off\ \ if\ dst < imm$
0xad	jlt dst, src, +off	$PC = PC + off\ \ if\ dst < src$
0xb5	jle dst, imm, +off	$PC = PC + off\ \ if\ dst <= imm$
0xbd	jle dst, src, +off	$PC = PC + off\ \ if\ dst <= src$
0x45	jset dst, imm, +off	$PC = PC + off\ \ if\ dst\ and\ imm$
0x4d	jset dst, src, +off	$PC = PC + off\ \ if\ dst\ and\ src$
0x55	jne dst, imm, +off	$PC = PC + off\ \ if\ dst\ != imm$
0x5d	jne dst, src, +off	$PC = PC + off\ \ if\ dst\ != src$
0x65	jsgt dst, imm, +off	$PC = PC + off\ \ if\ dst > imm\ (signed)$
0x6d	jsgt dst, src, +off	$PC = PC + off\ \ if\ dst > src\ (signed)$
0x75	jsge dst, imm, +off	$PC = PC + off\ \ if\ dst >= imm\ (signed)$
0x7d	jsge dst, src, +off	$PC = PC + off\ \ if\ dst >= src\ (signed)$
0xc5	jslt dst, imm, +off	$PC = PC + off\ \ if\ dst < imm\ (signed)$
0xcd	jslt dst, src, +off	$PC = PC + off\ \ if\ dst < src\ (signed)$
0xd5	jsle dst, imm, +off	$PC = PC + off\ \ if\ dst <= imm\ (signed)$
0xdd	jsle dst, src, +off	$PC = PC + off\ \ if\ dst <= src\ (signed)$
0x85	call imm	*Function call*
0x95	exit	*return r0*
0x00	nop	do nothing
Instructions not synthesizable on NetFPGA SUME		
Arithmetic 64 and 32 bits		
Opcode	**Mnemonic**	**Pseudocode**
0x27	mul dst, imm	$dst = dst * imm$
0x2f	mul dst, src	$dst = dst * src$
0x37	div dst, imm	$dst = dst\ /\ imm$
0x3f	div dst, src	$dst = dst\ /\ src$
0x97	mod dst, imm	$dst = dst\ mod\ imm$
0x9f	mod dst, src	$dst = dst\ mod\ src$
0x94	mod32 dst, imm	$dst = dst\ mod\ imm$
0x9c	mod32 dst, src	$dst = dst\ mod\ src$