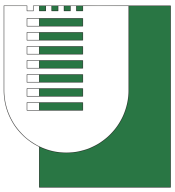


TOR VERGATA
UNIVERSITY OF ROME

PhD in Electronic Engineering

PhD Cycle XXXV

Design and Implementation of 100+ Gbps Packet Processing Accelerators

Marco Spaziani Brunella

A.Y. 2022/2023

Tutor: Prof. G. Bianchi

Coordinator: Prof. C. Di Natale

Abstract

One of the most disruptive ideas in computer network design is the introduction of Software-Defined Networking (SDN) and Network Function Virtualization (NFV). At the idea's core is the willingness to enhance the programmability of high-speed network devices without sacrificing performance. SDN and NFV are pure software approaches, relying on the underlying general-purpose (e.g., x86 CPUs) hardware to execute efficiently. General purpose computing infrastructure has been showing a consistent decline in performance growth over the last decade, providing diminishing returns at each new generation of devices. With 100gbps widely deployed in hyperscalers infrastructure, a software approach is no longer an option.

This dissertation aims to rethink packet processing architectures from the ground up by designing and implementing Domain-Specific hardware architectures to process line-rate traffic at 100+ Gbps. In particular, this work examines the requirements for line-rate processing and languages to describe packet processing tasks, proposing a Very-Long Instruction Word (VLIW) processor called Sephirot, able to execute native extended Berkeley Packet Filter (eBPF) code. Furthermore, the VLIW core is complemented by a set of Maps and Helper Functions, constituting a complete network processing offload system called hXDP. This system is then implemented on a NetFPGA-SUME and tested against an x86 CPU with a set of microbenchmarks and real-world use cases, such as Meta's Load Balancer Katran. Finally, I propose a TCAM-based engine called Program Warping as a frontend to hXDP to accelerate the packet parsing portion of every network program. This engine can dramatically reduce the number of instructions needed to process a packet to the extent of a 1-clock cycle decision on specific paths of Suricata, an open-source Intrusion Detection System (IDS).

Acknowledgements

First, I would like to thank my advisor, Prof. Giuseppe Bianchi, for all the feedback and guidance during and before my Ph.D. journey began. I sincerely thank Marco Bonola, with whom I have been working on this fantastic and long journey, for his passion and expertise in the field. A special thanks go to Roberto Bifulco from NEC Laboratories Europe, who mentored me throughout my Ph.D. journey and immensely contributed to the person I am today. Last but not least, I wanted to thank my partner Giulia for never letting me miss my true north.

Contents

Chapter 1

Introduction

Today, everything runs through the Internet, the largest network of devices. In the last couple of decades, this Network Infrastructure has become one of the critical backbones of our society. In this timeframe, this infrastructure needed to be always more capable of offering new and complex services, such as proxies, caches, routing, and firewalls. As a network operator, the ability to reconfigure and adapt the existing devices composing the network has become a must-have feature.

Given the flexibility of modern networks and the need to continuously support new applications, operators have turned to pure software implementations for such functions, using a mix of well-defined approaches: Software Defined Networks (SDN) and Network Function Virtualization (NFV). The goal of SDN is to logically separate the network control functions from the network forwarding functions, while NFV seeks to abstract network functions from the underlying hardware. The natural consequence of this solution is that servers in operator networks spend a significant fraction of their CPU's resources to process network traffic coming from their network interface cards (NICs).

A pure CPU-based approach has already started to provide diminishing returns. Modern CPUs must be faster to cope with 100+Gbps network traffic streaming in from the NIC without severely compromising peak throughput and latency, which are essential

elements of current workloads such as Virtual/Augmented Reality, the Metaverse, and live event streaming.

For the reasons mentioned above, modern network operators must challenge those paradigms and deploy new architectures to overcome the limitations of running networks on top of a general-purpose infrastructure. In the rest of this chapter, we will analyze and discuss the motivations beyond these challenges. We will then present the contributions of this Dissertation to the state of art. This chapter partially includes figures and verbatim copies of the text of my first-author papers.

1.1 Background and Motivations

Modern networks are called to efficiently and flexibly support an ever-growing variety of heterogeneous middlebox-type functions such as network address translation, tunneling, load balancing, traffic engineering, monitoring, intrusion detection, and so on. This flexibility, combined with the increasing demands in packet processing throughput, is hugely challenging. While a network's speed increases, it decreases the time budget to perform meaningful transformations to the data. Network speed has increased by 100x in the last decade. In contrast, CPUs processing this data stream has yet to match such a growth, mainly due to two reasons: the first one is related to the CMOS technology at the heart of every CPU and the other one is related to every program running on a CPU-based system.

End of Dennard's scaling Law. Dennard's scaling law can be reduced to: "as transistors become smaller, they become cheaper, faster and consume less power." This observation mainly drove the ability to cram more transistors inside a chip that, as transistors become smaller, they become faster, consume less power, and become easier to manufacture.

This observation started to shake in the last couple of decades, as shown in **Fig. 1.1**

, as transistors became smaller, their energy density ramped up, eventually leading to sophisticated power management techniques and limiting the maximum frequency at which a chip can run.

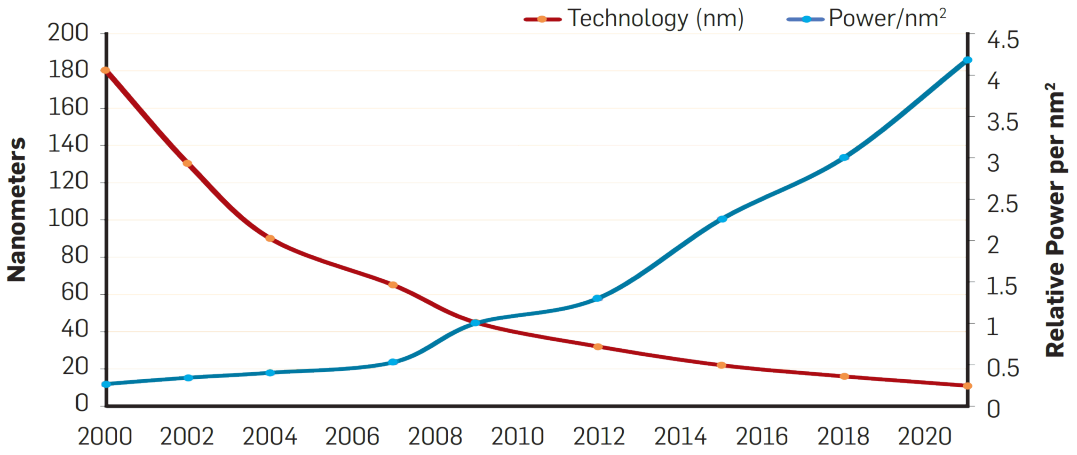


Figure 1.1: Transistor Scaling trends of Supply Voltage and Power Density

In fact, the powerd drawn by the supply of a CMOS chip is given by the following equation:

$$P_{dynamic} \propto \frac{1}{2} \times V_{DD}^2 \times f_{CLK}$$

The critical limit of the technology is thus the *power density* of the die. The industry has reduced the dynamic power by lowering V_{DD}. Still, we are now in a situation where no further decrease can be successfully achieved without violating the noise margins of the design. Moreover, we don't want to decrease the f_{CLK} since it's directly related to how many instructions we can execute in a single cycle. Thus, around the beginning of the 2000s, the so-called power wall was hit, pinning the clock frequency for desktop CPUs at $\sim 3-4GHz$.

Amdhal's Law If we take any program running on a CPU, it will be composed of a stream of logical instructions executed one after the other. Modern CPU architectures typically employ deep pipeline stages and multiple execution units (e.g., superscalar

processors) to increase the number of instructions churned per clock cycle, increasing the Instructions Per Cycle (IPC). On a higher abstraction level, modern workloads are divided into threads that can be executed independently on the different cores of a many-core CPU.

Leveraging Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) helped us to offset the need for better single-thread performance of the CPUs, ultimately bottlenecked by the end of Dennard's scaling law. ILP and TLP are metrics that heavily depend on the kind of workload we're running. More specifically, they rely on the amount of concurrency we can extract from the workload. The speedup we gain by parallelizing a portion x of a workload W is given by Amdahl's law:

$$speedup = \frac{1}{(1 - W) + \frac{W}{x}}$$

Plotting the equation mentioned above in **Fig. 1.2**, we can see that it doesn't make sense to deploy hundreds or thousands of cores on workloads that exhibit poor concurrency characteristics, which is the case of network processing tasks.

1.2 Strategy, Vision and List of Contributions

This dissertation aims to provide a more general and easy-to-use solution to program packet processing on Field-Programmable Gate Arrays (FPGAs) Netowrk Interface Cards (NICs), using little FPGA resources while seamlessly integrating with existing operating systems. Computers in datacenter and telecom operator networks employ a significant fraction of their CPU's resources to process network traffic coming from their NICs. Enforcing security, for example, using a firewall function, monitoring network-level performance, and routing packets toward their intended destinations, are just a few examples of the tasks being performed by these systems. With NIC's port speeds growing beyond 100Gigabit/s (Gbps), and given the limitations in further scaling CPUs performance, new hardware

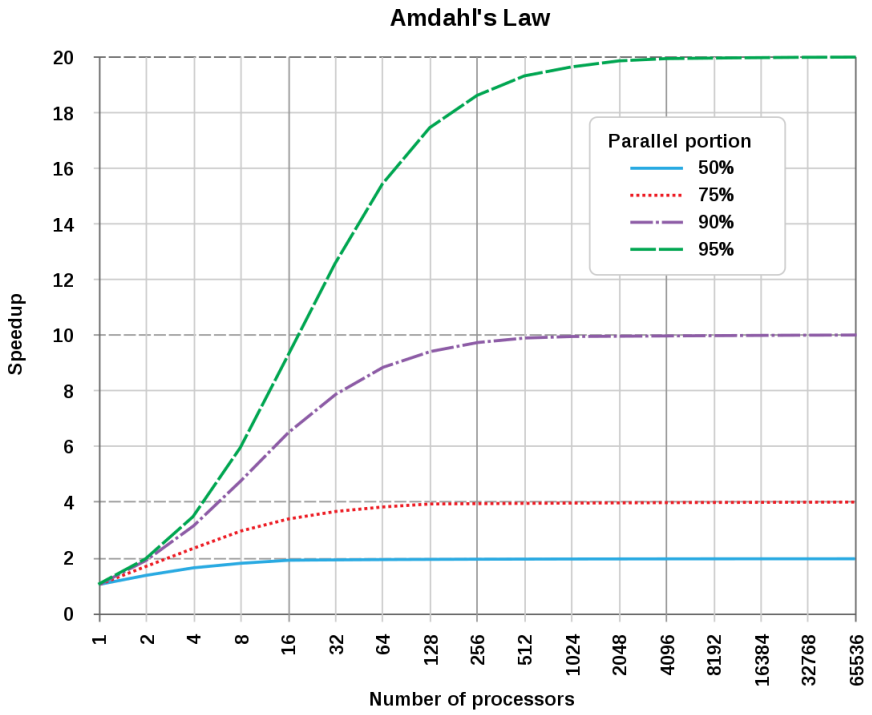


Figure 1.2: Diminishing returns as we increase the number of execution units given a fixed amount of intrinsic concurrency

architectures need to be introduced to handle these growing workloads.

From an OS interface perspective, this dissertation explores using the extended Berkeley Packet Filter (eBPF) [3] as a Domain-Specific Language to describe packet processing tasks. eBPF is a technology that enables running of small programs within the Linux kernel in a sandboxed environment, the eBPF virtual machine (VM). The eBPF VM has its Instruction Set Architecture (ISA), which implements a register architecture with a program counter (PC) register, 10 general purpose registers ($R0 - R9$), and a read-only stack pointer register ($R10$) that contains the address of a 512B memory area used as program's stack. The registers and the stack capture the current program's state, which resets every time a new run is triggered. To save state across program runs, eBPF pro-

vides unique data structures called maps. These are memory areas defined at compile time and organized as lookup tables. eBPF programs are usually written in a high-level language, such as C, and compiled to the eBPF bytecode that uses the eBPF Instruction Set Architecture (ISA). The eBPF bytecode can be loaded in the kernel using different *hooks*. Depending on the hook, the program execution triggers changes, and so does the eBPF input data structure. For instance, in this dissertation, we focus on the XDP hook, and call *XDP programs* an eBPF program attached to the XDP hook. The hook is provided at the NIC driver level. Whenever a packet is received, the XDP environment: (i) creates an `xdp_md` struct to contain the packet buffer pointers and metadata, such as the packet's input port id; (ii) sets $R0$ to point to the address of the memory area hosting the struct; (iii) and then starts the VM to run the XDP program. At the end of its execution, the program can return a forwarding decision for the packet by writing in $R0$ one of the following four values: (i) `XDP_DROP`, to drop it; (ii) `XDP_PASS`, to pass the packet to the next level in the network stack; (iii) `XDP_TX`, to transmit the packet to the same port it was received on; (iv) `XDP_REDIRECT`, to transmit the packet to a different port.

From a hardware architecture standpoint, this dissertation explores three hardware architectures, each building on top of the other, to achieve line-rate throughput with sub-microsecond latency. **Sephirot** is a VLIW processor with four parallel lanes that execute eBPF instructions. Sephirot is designed as a pipeline of four stages: instruction fetch (IF); instruction decode (ID); instruction execute (IE); and commit. A program is stored in a dedicated instruction memory, from which Sephirot fetches the instructions in order. The processor has another dedicated memory area to implement the program's stack, which is 512B in size, and 11 64b registers are stored in the register file. These memory and register locations match one-to-one the abovementioned eBPF virtual machine specification. We perform a design-space exploration of Sephirot growing and shrinking the number of parallel lanes, evaluating the amount of Instruction Level Parallelism (ILP)

we can extract from network packet processing workloads. We then synthesize the various design onto a NetFPGA-SUME [67].

hXDP is an entire eBPF runtime execution system on FPGA. It's built on top of Sephirot and features the minimum set of *Maps* and *Helper Functions* needed to support native eBPF/XDP program execution. To evaluate hXDP, we provide an open-source implementation for the NetFPGA. We tested our implementation using the XDP example programs provided by the Linux source code and using two real-world applications: a simple stateful firewall and Facebook's Katran load balancer. hXDP can match the packet forwarding throughput of a multi-GHz server CPU core while providing a 10x lower forwarding latency. This is achieved despite the low clock frequency of our prototype (156MHz) and using less than 15% of the FPGA resources.

Program Warping builds on the observation that a subset of the eBPF/XDP program's instructions implements typical packet processing tasks, such as packet header parsing, which can be efficiently implemented in architectures with a high degree of pipeline-level parallelism [13, 18]. Therefore, under the constraint of keeping transparency to the programmer, we address two main issues in our design: (i) identifying, from the eBPF bytecode, the tasks that can be efficiently parallelized; (ii) designing an FPGA pipeline that runs such tasks leveraging their parallelism, while providing runtime reconfigurability and using minimal hardware resources. We address the two issues extending technologies that run eBPF programs on FPGA, like hXDP, by adding a compilation step, the *Warp Optimizer*, and a new hardware module, the *Warp Engine*. The Warp Optimizer analyzes the eBPF bytecode, at compilation time. While the general purpose nature of the eBPF programming model complicates the identification of functional tasks from the bytecode, its target machine model, with registers assigned with specific functions, allows us to simplify the problem by introducing assumptions such as prior knowledge

of the memory areas hosting a packet's data. Therefore, we can identify common packet processing tasks, including packet header parsing and classification, and delegate their execution to the Warp Engine.

1.3 State of the Art

Including programmable accelerators on the Network Interface Card (NIC) is one of the promising approaches to offload the resource-intensive packet processing tasks from the CPU, thereby saving its precious cycles for tasks that cannot be performed elsewhere. Nonetheless, achieving programmability for high-performance network packet processing tasks is an open research problem, with solutions exploring different areas of the solution space that compromise in different ways between performance, flexibility, and ease-of-use [43].

As a result, today's accelerators are implemented using different technologies, including Application-Specific Integrated Circuits (ASICs), Field-Programmable Gate Arrays (FPGAs), and many-core System-on-Chip. FPGA-based NICs are especially interesting since they provide good performance and a high degree of flexibility, enabling programmers to define virtually any function, provided that it fits in the available hardware resources. Compared to other accelerators for NICs, such as network processing ASICs [14] or many-core System-on-Chip SmartNICs [46], the FPGA NICs flexibility also gives the additional benefit of supporting diverse accelerators for a broader set of applications. For instance, Microsoft employs them in datacenters for network and machine learning tasks [17, 20]. In telecom networks, they are also used to perform radio signal processing tasks [32, 63, 45].

A large number of new NIC designs appeared in the last few years [49, 24, 31, 47, 64, 38]. These solutions mostly combine different compute and network modules in a mix-and-match manner, e.g., regular NIC's switching ASICs with general-purpose compute clusters

based on RISC cores [47], or FPGA-enhanced switching combined with general-purpose clusters [31, 64]. In many solutions, a novelty factor enables P4-based programming of the switching ASIC. This effectively corresponds to replacing the fixed-function switching module with a programmable switching module [49]. Some of these designs offer (partial) eBPF support. However, they implement eBPF on top of the general purpose clusters, replicating the architecture commonly used in server machines, but on a smaller scale. In research, previous work addresses the challenges of moving data among these modules [36], and explores ways to leverage these new NIC designs to improve application performance [66, 21, 17, 35, 37, 50, 52]. Program warping focuses on the design of the packet switching module, targeting FPGA NICs and presenting a solution that integrates with Linux applications that leverage eBPF/XDP. We extend hXDP [56], which is the only solution providing full support for XDP on FPGA NIC directly within the switching module. Compared to hXDP, we provide better performance by introducing a new compilation step co-designed with a hardware module, the Warp Engine, which is pipelined to the hXDP processor.

Recent work addressed eBPF program's optimization at compile time, targeting x86 processors [42, 65]. These works focus on implementing compiler techniques targeting a fixed processor design, whereas we co-design the compiler and the hardware executor. Another related work is Gallium [66], which targets offloading a program's part to programmable switching ASICs. Also, in this case, it assumes a fixed set of executors, including programmable switching chips and processors.

1.4 Structure of the Dissertation

The remainder of this Dissertation is structured as follows.

- **Chapter 2, Modern Networks Overview**, reviews some of the breakthroughs

of the last decade in computer network organization and design.

- **Chapter 3, Sephirot: a Very-Long Instruction Word Processor for Networking Applications**, analyzes the details of a fully-pipelined, Very-Long Instruction Word (VLIW) processor executing native eBPF instructions.
- **Chapter 4, hXDP: Software Packet Processing on FPGA NICs**, introduces a novel datapath that allows the offloading of the eBPF kernel functionalities to an FPGA-based Smart NIC.
- **Chapter 5, Program Warping: Faster Hardware Packet Processing**, examines a new technique that improves throughput by replacing several instructions of a packet processing program described in eBPF with an equivalent runtime programmable hardware implementation based on TCAMs and programmable parsers.

We conclude the Dissertation with final remarks, summarizing the contributions and results of this work.

1.4.1 List of Published Material

I have (co-)authored the following publications and manuscripts:

- **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 f. *Commun. ACM*, 65(8):92–100, jul 2022
- **Marco Spaziani Brunella**, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020

- Marco Bonola, Giacomo Belocchi, Angelo Tulumello, **Marco Spaziani Brunella**, Giuseppe Siracusano, Giuseppe Bianchi, and Roberto Bifulco. Faster software packet processing on FPGA NICs with eBPF program warping. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 987–1004, Carlsbad, CA, July 2022. USENIX Association

- Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, **Marco Spaziani Brunella**, 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)*, Boston, MA, 2019. USENIX Association

- **Marco Spaziani Brunella**, Salvatore Pontarelli, Marco Bonola, and Giuseppe Bianchi. V-PMP: A VLIW packet manipulator processor. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–9. IEEE, 2018

- Marco Spaziani Brunella, Salvatore Pontarelli, Fabrizio Marrese, Marco Bonola, and Giuseppe Bianchi. Packet Manipulator Processor: A RISC-V VLIW core for networking applications. In *7th RISC-V Workshop*

- **Marco Spaziani Brunella**, G. Bianchi, S. Turco, F. Quaglia, and N. Blefari-Melazzi. Foreshadow-vmm: Feasibility and network perspective. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 257–259, 2019

- **Marco Spaziani Brunella**, Sara Turco, Giuseppe Bianchi, and Nicola Blefari Melazzi. Foreshadow-VMM: on the practical feasibility of L1 cache Terminal Fault attacks. In *2019 ITASEC*

- A. Nannarelli, M. Re, G. C. Cardarilli, L. Di Nunzio, **Marco Spaziani Brunella**, R. Fazzolari, and F. Carbonari. Robust throughput boosting for low latency dynamic

partial reconfiguration. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, 2017

Chapter 2

Modern Networks Overview

In this chapter, we are going to review some of the breakthroughs of the last 10 years in computer network design. A consistent part of the chapter will be dedicated to *eBPF*, which will be the target *Instruction Set Architecture* for V-PMP.

2.1 Software-Defined Networking

An IP network element (NE) appears to external entities as a monolithic piece of network equipment, e.g., a router, NAT, firewall, or load balancer. Internally, however, an IP network element (NE) is composed of numerous logically separated entities that cooperate to provide a given functionality, such as routing. Two types of network element components exist: the control element (CE) in the control plane and the forwarding element (FE) in the data plane. Forwarding elements are typically ASIC, network-processor, or general-purpose processor-based devices that handle data path operations for each packet. Control elements are typically based on general-purpose processors that provide control functionality, like routing and signaling protocols.

In Software Defined Networks [34], most of the control plane software is removed from the route processor card, and instead re-implemented for execution on external commod-

ity hosts, which are under the control of the cloud/Internet service provider. The SDN controller software can hence be maintained and upgraded to support new features by the provider itself, without having to wait for the switch vendor to make modifications.
Software Defined Networking is arguably one of the most influential innovations that emerged in the last years in the field of computer networks. The SDN concept is seen as a key enabling technology to provide flexibility and configurability of complex networks. In particular, the use of SDN devices allows easy upgrading of the current protocols and simplifies the adoption of new protocols without requiring to change in the actual hardware devices deployed in the network.

Unfortunately, the programmability of the Software Defined Network cannot be provided by using common hardware architectures such as general purpose CPU but requires the design of specific hardware-based architectures able to forward network traffic with an aggregated bandwidth up to multiple terabits/sec.
The most adopted SDN architecture is based on [41], a pragmatic and viable platform agnostic interface to the network switch hardware. In particular, this abstraction provides a match, action mechanism to modify a packet traversing the network switch pipeline. The use of multiple match stages in the pipeline permits great increases in the packet throughput since the device is able to perform several queries in parallel.

Considering that in each query the packet header must be checked against thousands of rules stored in the match tables, it is evident that a CPU-based architecture cannot sustain the same throughput that can be provided by specific hardware for pattern matching such as Ternary Content Addressable Memories (TCAMs), binary Content Addressable Memories (CAMs) [48] or exact matching engines based on multiple hash tables [53].

2.1.1 The OpenFlow Protocol

OpenFlow provides an open protocol to program the flow table in different switches and routers.
The datapath of an OpenFlow Switch consists of a Flow Table and an action associated with each flow entry, as depicted in Fig.2.1.
For high performance and low cost, the data path must have a carefully prescribed degree of flexibility. This means forgoing the ability to specify arbitrary handling of each packet and seeking a more limited, but still useful, range of actions.
An OpenFlow Switch consists of at least three parts:

- A Flow Table, with an action associated with each flow entry, to tell the switch how to process the flow
- A Secure Channel that connects the switch to the controller, allowing commands and packets to be sent between a controller and the switch
- The OpenFlow Protocol, which provides an open and standard way for a controller to communicate with a switch.

2.2 Network Function Virtualization

Network Function Virtualization (NFV) [7] is a term used to represent the implementation of data plane network functions in software that is executed on commodity hosts, as depicted in Fig.2.2 .
The hypothesis is that NFV will incur lower capital expenditures and operating expenditures when compared to traditional switches/routers and middlebox appliances in which data plane network functions are typically implemented in custom hardware.
The architectural components of NFV are:

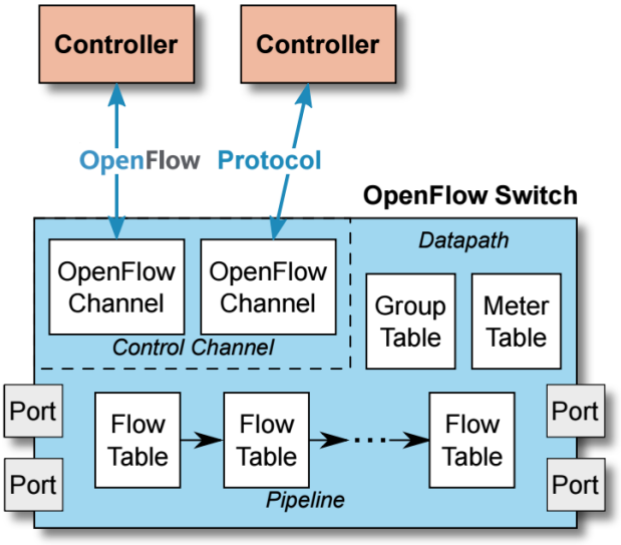


Figure 2.1: Typical OpenFlow architecture

- Network Function Virtualization Infrastructure (NFVI)
- Network Functions (NFs),
- Management And Network Orchestration (MANO)

The infrastructure consists of hardware (a single computer or a compute cluster), and framework software, which offers functions that are commonly required by NFs, such as NF placement, dynamic scaling, etc.

Data-plane network functions considered for software implementation in commodity hosts range from basic packet forwarding to complex middlebox functions such as intrusion prevention systems. When NFs are executed on Virtual Machines (VMs), they are referred to as Virtual Network Functions (VNFs). MANO components include management functions, such as Fault management, Configuration management, Accounting, Performance monitoring, and Security, and orchestrators, which manage service chains of multiple NFs.

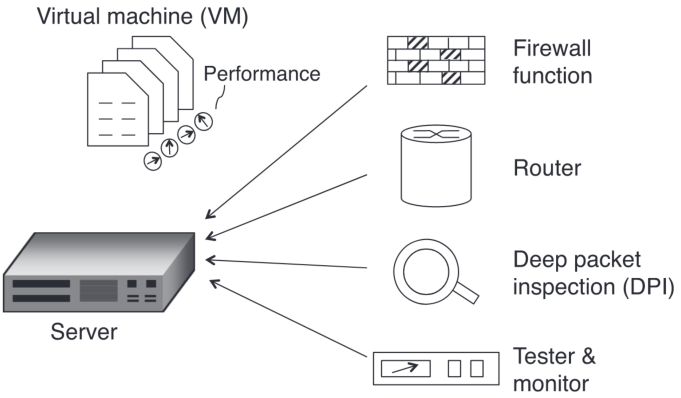


Figure 2.2: Network Function Virtualization concept

2.3 Berkeley Packet Filter

Berkeley Packet Filter (BPF) [40] is an in-kernel virtual machine for packet filtering. BPF has two main components: the network tap and the packet filter. The network tap collects copies of packets from the network device drivers and delivers them to listening applications, as depicted in **Fig.2.3**.

The filter decides if a packet should be accepted and, if so, how much of it to copy to the listening application. When a packet arrives at a network interface, the link-level device driver normally sends it up the system protocol stack. But when BPF is listening on this interface, the driver first calls BPF. BPF feeds the packet to each participating processfilter.

This user-defined filter decides whether a packet is to be accepted and how many bytes of each packet should be saved. For each filter that accepts the packet, BPF copies the requested amount of data to the buffer associated with that filter. The device driver then regains control. If the packet was not addressed to the local host, the driver returns from the interrupt. Otherwise, normal protocol processing proceeds.

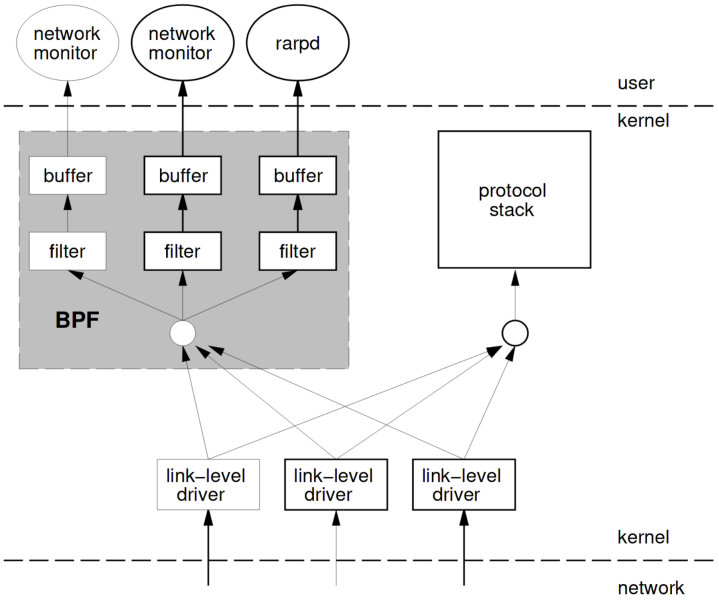


Figure 2.3: BPF architecture

BPF has been extended to exploit the full power of the 64-bit architectures of modern processors, thus leading to *extended BPF*. Whilst providing the same functionalities of classic BPF, there are several appealing new features in eBPF: it can execute arbitrary code, permit fast dynamic recompilation, and provides an efficient interface with the userland. The main characteristic of eBPF is to provide dynamic programmability at the kernel level without compromising the operating system's security and stability. This is possible since the eBPF programs must fulfill a set of tight requirements: eBPF program always terminates, does not contain loops and pointer arithmetic is prohibited. Furthermore, the use of helper functions (special functions restricted to a white list defined in the kernel) greatly extends the capability of eBPF to perform network processing tasks and to interact with the system.

The main components of the eBPF framework are:

- eBPF Virtual Machine

- Maps
- Helper Functions

The architecture of eBPF is illustrated in **Fig.2.4**.

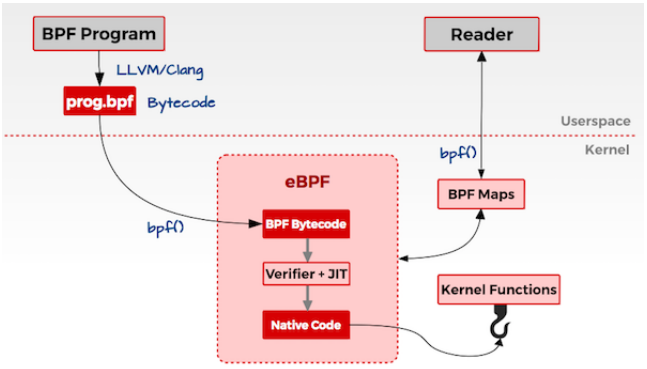


Figure 2.4: eBPF architecture

2.3.1 eBPF Virtual Machine

eBPF is a RISC register machine with a total of 11 64-bit registers, a program counter, and a 512-byte fixed-size stack. 9 registers are general-purpose read-write, one is a read-only stack pointer and the program counter is implicit, i.e. we can only jump to a certain offset from it. The VM registers are always 64-bit wide and support 32-bit subregister addressing if the most significant bits of the registers are zeroed.

Each function call can have at most 5 arguments in registers r1-r5; this applies to both ebpf-to-ebpf calls and to kernel function calls. Registers r1-r5 can only store numbers or pointers to the stack (to be passed as arguments to functions), never direct pointers to arbitrary memory. All memory accesses must be done by first loading data to the eBPF stack before using it in the eBPF program. This restriction helps the eBPF verifier, it simplifies the memory model to enable easier correctness checking.

2.3.2 eBPF Instruction Set Architecture

eBPF encodes its instructions in a similar way to what a standard MIPS architecture would do. They are fixed length, 64bit wide instructions, as depicted in **Fig. 2.5**.

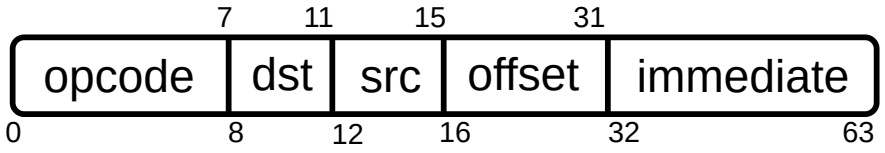


Figure 2.5: eBPF Instruction Encoding

Instructions are divide with respect to the functional unit they use:

- ALU64
- ALU32
- Memory
- Branch

The complete ISA is reported in the following tables.

Opcode	Mnemonic	Pseudocode
0x07	`add dst, imm`	$dst += imm$
0x0f	`add dst, src`	$dst += src$
0x17	`sub dst, imm`	$dst -= imm$
0x1f	`sub dst, src`	$dst -= src$
0x27	`mul dst, imm`	$dst *= imm$
0x2f	`mul dst, src`	$dst *= src$
0x37	`div dst, imm`	$dst /= imm$
0x3f	`div dst, src`	$dst /= src$
0x47	`or dst, imm`	$dst \mid= imm$
0x4f	`or dst, src`	$dst \mid= src$
0x57	`and dst, imm`	$dst \&= imm$
0x5f	`and dst, src`	$dst \&= src$
0x67	`lsh dst, imm`	$dst <<= imm$
0x6f	`lsh dst, src`	$dst <<= src$
0x77	`rsh dst, imm`	$dst >>= imm(logical)$
0x7f	`rsh dst, src`	$dst >>= src(logical)$
0x87	`neg dst`	$dst = -dst$
0x97	`mod dst, imm`	$dst \%= imm$
0x9f	`mod dst, src`	$dst \%= src$
0xa7	`xor dst, imm`	$dst \oplus= imm$
0xaf	`xor dst, src`	$dst \oplus= src$
0xb7	`mov dst, imm`	$dst = imm$
0xbf	`mov dst, src`	$dst = src$
0xc7	`arsh dst, imm`	$dst >>= imm(arithmetic)$
0xcf	`arsh dst, srs`	$dst >>= src(arithmetic)$

Table 2.1: ALU 64 Instructions

Opcode	Mnemonic	Pseudocode
0x04	`add32 dst, imm`	$dst += imm$
0x0c	`add32 dst, src`	$dst += src$
0x14	`sub32 dst, imm`	$dst -= imm$
0x1c	`sub32 dst, src`	$dst -= src$
0x24	`mul32 dst, imm`	$dst *= imm$
0x2c	`mul32 dst, src`	$dst *= src$
0x34	`div32 dst, imm`	$dst /= imm$
0x3c	`div32 dst, src`	$dst /= src$
0x44	`or32 dst, imm`	$dst \mid= imm$
0x4c	`or32 dst, src`	$dst \mid= src$
0x54	`and32 dst, imm`	$dst \&= imm$
0x5c	`and32 dst, src`	$dst \&= src$
0x64	`lsh32 dst, imm`	$dst <<= imm$
0x6c	`lsh32 dst, src`	$dst <<= src$
0x74	`rsh32 dst, imm`	$dst >>= imm(logical)$
0x7c	`rsh32 dst, src`	$dst >>= src(logical)$
0x84	`neg32 dst`	$dst = -dst$
0x94	`mod32 dst, imm`	$dst \%= imm$
0x9c	`mod32 dst, src`	$dst \%= src$
0xa4	`xor32 dst, imm`	$dst \oplus= imm$
0xac	`xor32 dst, src`	$dst \oplus= src$
0xb4	`mov32 dst, imm`	$dst = imm$
0xbc	`mov32 dst, src`	$dst = src$
0xc4	`arsh32 dst, imm`	$dst >>= imm(arithmetic)$
0xcc	`arsh32 dst, srs`	$dst >>= src(arithmetic)$

Table 2.2: ALU 32 Instructions

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`	$dst = htobe64(dst)$

Table 2.3: Byteswap Instructions

Opcode	Mnemonic	Pseudocode
0x18	`lddw dst`	$dst = imm$
0x20	`ldabsw src`	*See kernel documentation*
0x28	`ldabsh src`	...
0x30	`ldabsb src`	...
0x38	`ldabsdw src`	...
0x40	`ldindw src`	...
0x48	`ldindh src`	...
0x50	`ldindb src`	...
0x58	`ldinddw src`	...
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`	$*(uint64\_t*)(dst + off) = src$

Table 2.4: Memory Instructions

Opcode	Mnemonic	Pseudocode
0x05	`ja +off`	$PC+ = off$
0x15	`jeq dst, imm, +off`	$PC+ = off\ if\ dst == imm$
0x1d	`jeq dst, src, +off`	$PC+ = off\ if\ dst == src$
0x25	`jgt dst, imm, +off`	$PC+ = off\ if\ dst > imm$
0x2d	`jgt dst, src, +off`	$PC+ = off\ if\ dst > src$
0x35	`jge dst, imm, +off`	$PC+ = off\ if\ dst >= imm$
0x3d	`jge dst, src, +off`	$PC+ = off\ if\ dst >= src$
0xa5	`jlt dst, imm, +off`	$PC+ = off\ if\ dst < imm$
0xad	`jlt dst, src, +off`	$PC+ = off\ if\ dst < src$
0xb5	`jle dst, imm, +off`	$PC+ = off\ if\ dst <= imm$
0xbd	`jle dst, src, +off`	$PC+ = off\ if\ dst <= src$
0x45	`jset dst, imm, +off`	$PC+ = off\ if\ dst\&imm$
0x4d	`jset dst, src, +off`	$PC+ = off\ if\ dst\&src$
0x55	`jne dst, imm, +off`	$PC+ = off\ if\ dst! = imm$
0x5d	`jne dst, src, +off`	$PC+ = off\ if\ dst! = src$
0x65	`jsgt dst, imm, +off`	$PC+ = off\ if\ dst > imm(signed)$
0x6d	`jsgt dst, src, +off`	$PC+ = off\ if\ dst > src(signed)$
0x75	`jsge dst, imm, +off`	$PC+ = off\ if\ dst >= imm(signed)$
0x7d	`jsge dst, src, +off`	$PC+ = off\ if\ dst >= src(signed)$
0xc5	`jslt dst, imm, +off`	$PC+ = off\ if\ dst < imm(signed)$
0xcd	`jslt dst, src, +off`	$PC+ = off\ if\ dst < src(signed)$
0xd5	`jsle dst, imm, +off`	$PC+ = off\ if\ dst <= imm(signed)$
0xdd	`jsle dst, src, +off`	$PC+ = off\ if\ dst <= src(signed)$
0x85	`call imm`	*Function call*
0x95	`exit`	*return* $r0$

Table 2.5: Branch Instructions

2.3.3 Maps

The eBPF maps are a generic data structure for the storage of different data types. When the map is created the user specifies the key type (and thus the size of the key), the value type, and the number of entries supported by the map. eBPF maps are used to keep the state between invocations of the eBPF program, and allow sharing of data between eBPF kernel programs, and also between kernel and user-space applications. eBPF defines several types of maps (12 maps are listed in the Linux implementation of eBPF[1]).
The main ones are:

- `BPF_MAP_TYPE_HASH`: this map provides an hash table where the $\langle key, value \rangle$ pairs are stored.
- `BPF_MAP_TYPE_ARRAY`: this is an array of memory locations.
- `BPF_MAP_TYPE_PROG_ARRAY`: This is one of the more interesting eBPF-map types because it allows tail calling of eBPF programs.

2.3.4 Helper Functions

The eBPF helper is a special function that can be called from within eBPF programs to perform a variety of tasks. The helper function can be used by eBPF programs to interact with the system. This is useful for debugging the kernel and carrying out performance analysis. Helper functions can be also used to interact with eBPF maps or to handle network packets. The helper functions are restricted to a defined white list to guarantee kernel security and stability. Calling helpers functions in the Linux implementation introduces no overhead, because of the direct call into the compiled helper, and so excellent performance is offered.

[1] a complete list of the eBPF maps is available here: `https://ferrisellis.com/posts/ebpf\_syscall_and_maps/`

2.4 eXpress Data Path

XDP allows programmers to inject programs at the NIC driver level so that such programs are executed before a network packet is passed to the Linux's network stack. XDP programs are based on Linux's eBPF technology. eBPF provides an in-kernel virtual machine for the sandboxed execution of small programs within the kernel context. In its current version, the eBPF virtual machine has 11 64b registers: $r0$ holds the return value from in-kernel functions and programs, $r1 - r5$ are used to store arguments that are passed to in-kernel functions, $r6 - r9$ are registers that are preserved during function calls and $r10$ stores the frame pointer to access the stack. The eBPF virtual machine has a well-defined ISA composed of more than 100 fixed-length instructions (64b). Programmers usually write an eBPF program using the C language with some restrictions, which simplifies the static verification of the program.

eBPF programs can also access kernel memory areas called `maps`, i.e., kernel memory locations that essentially resemble tables. For instance, eBPF programs can use maps to implement arrays and hash tables. An eBPF program can interact with the map's locations by means of pointer deference, for unstructured data access, or by invoking specific helper functions for structured data access, e.g., a lookup on a map configured as a hash table. Maps are especially important since they are the only means to keep state across program executions and to share information with other eBPF programs and with programs running in user space.

Chapter 3

Sephirot: a Very-Long Instruction Word Processor for Networking Applications

In this chapter, we analyze the details of Sephirot: a fully-pipelined, Very-Long Instruction Word (VLIW) processor executing native eBPF instructions. A VLIW processor can execute multiple concurrent micro-instructions on its different execution *lanes*. In contrast to what happens in a super-scalar CPU, where two (or more) instructions are determined to be run concurrently at run-time, that is, inside the hardware, in a VLIW processor, the complexity of instruction's group formation is moved to the compiler, which *packs* multiple micro-instructions into, ipso facto, Very-Long Instruction Words.

To understand the principles behind a VLIW processor, let's suppose that these two micro-instructions need to be executed:

```
1    add $2, $3 ;add to register 2 the content of register 3
2    mov $1, $3 ;move the content of register 3 to register 1
```

To execute the `mov` instruction, no information about the result of the `add` is needed. Thus the two instructions can be executed concurrently, thus exploiting the **Instruction Level Parallelism** intrinsic of this stream of instructions.
To formalize this process, we can use Bernstein's conditions [9].

Bernstein's conditions

Suppose we have two instructions, P_1, and P_2. To tell whether the two instructions are independent - thus can be parallelized - they need to match 3 requirements:

$$I_1 \cap O_2 = \emptyset$$
$$I_2 \cap O_1 = \emptyset$$
$$O_2 \cap O_1 = \emptyset$$

Where I_i is the set of inputs required for the i^{th} instruction to execute, while the O_i is the set of output resources of the i^{th} instruction.
The first two conditions impose that the inputs of one of the two instructions do not depend on the outputs of the other one, while the last condition imposes that the two instruction do not write to the same resource - e.g., the same register -.
For more than two instructions, the number of conditions to be checked for dependencies grows with the square of the number of instructions.

Once two instructions, called *syllables*, can be grouped for concurrent execution, they are *packed* to form a very-long instruction, as depicted in **Fig. 3.1**. The very-long instruction word is then executed by the CPU, which takes the VLIW, chops that into *syllables*,

and then sends each syllable to an execution lane, as depicted in **Fig. 3.2**. Each lane implements an independent pipeline.

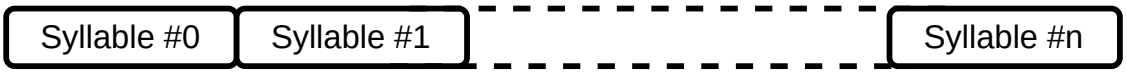


Figure 3.1: VLIW Instruction

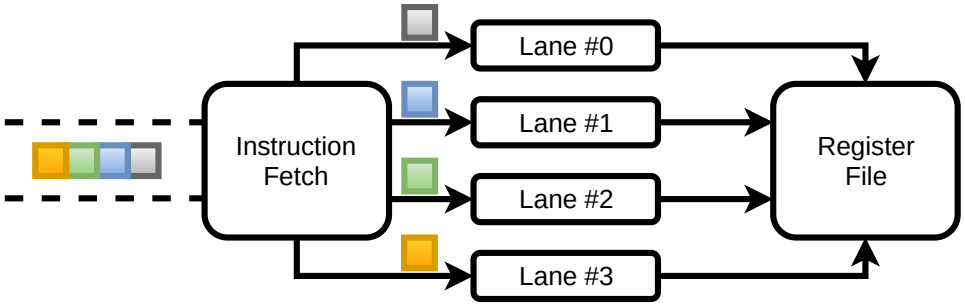


Figure 3.2: Very-Long Instruction Word architecture

The task of packing the instructions into parallelized syllables is entirely offloaded to the compiler in a VLIW architecture, allowing the hardware to be fast and efficient. When a package of syllables cannot be completed because the compiler wasn't able to find any other parallelizable instruction, it fills the syllable with a `nop` syllable, which instructs the relevant lane not to do anything.

Thus, the theoretical speedup is n times, where n is the number of lanes available in the architecture. More realistically, we estimate the speedup to be roughly $\frac{n}{2}$ since not always the compiler can parallelize syllables.

From the hardware side, the real problem is posed by the register file, which needs to implement an n port RAM. For this reason, the number of parallel lanes and registers is typically inversely proportional.

In the rest of the chapter, we detail the design choices made for a particular use case. Sephirot has been integrated as an execution unit inside a programmable network datapath called hXDP, which aims at offloading the eBPF functionalities from the Kernel to the FPGA. With this in mind, Sephirot has been crafted to map exactly the resources of the eBPF virtual machine.
We perform a Design-Space exploration of different numbers and kinds of parallel lanes, resulting in a critical analysis of the network workloads involved.

3.1 Sephirot architecture

As previously introduced, a VLIW CPU architecture can execute multiple instructions in parallel by using n *syllables*, each long m bits, concatenated to form a unique very-long word. All syllables are executed in parallel by a dedicated lane of the CPU, which is composed of a fetch stage (IF), a decode stage (SD), and an execute stage (SE).
In the particular case of Sephirot, the width of the instruction is 256 bits, allowing the simultaneous execution of four 64-bit syllables on four different lanes. The syllables represent a single eBPF instruction.

The top-view architecture of Sephirot is depicted in Figure 3.3.

Taking as reference a standard CPU, such as the MIPS CPU used in [51], this architecture can provide a theoretical 4x instruction throughput[1].

The Sephirot architecture differs from a standard VLIW processor for several design choices that we identified as mandatory to obtain high throughput in the specific task of packet manipulation, such as:

[1]The actual improvement can be less than 4x since the four syllables can be executed in parallel only if they are mutually independent. Suppose there are fewer than four independent syllables to execute. In that case, the VLIW instruction is filled by NOP operations, and the actual improvement of the VLIW architecture to a standard architecture decreases.

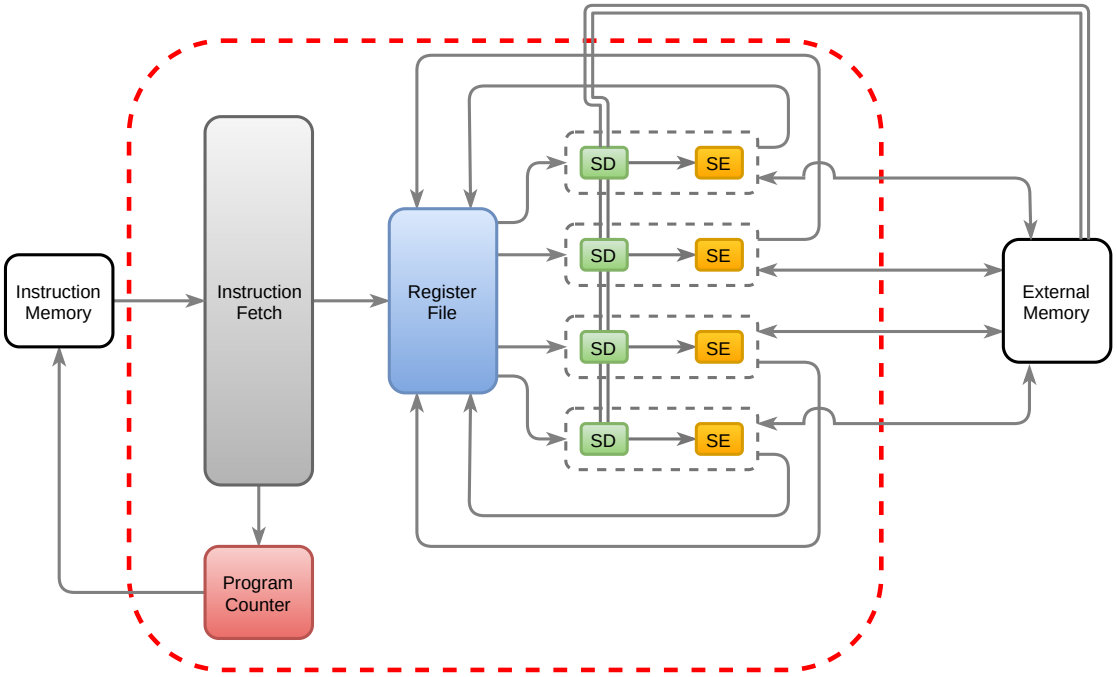


Figure 3.3: Sephirot architecture

- Memory & registers prefetch
- Multiple memory units
- Short pipeline

The need for memory and internal registers prefetch arises from the high memory pressure required in header rewriting and data movement. Since requesting data from memory and registers takes an additional clock cycle, data is queried to these units in advance to have the requested data ready when needed in the syllable execution phase. In particular, for the operations involving operands stored in the register file, operands are queried already in the fetch stage, directly latched in the pipeline register of the decode stage, and finally provided to the execute stage without stalls. Similarly, the data is queried from memory already in the decode stage for load operations from memory. For the same reason, PMP allows four memory units to execute operations on the memory in

parallel, allowing 256 bits per clock cycle to be moved from/to memory.

The core has been thoroughly simulated using Vivado 2017.4, and a snapshot of the simulation is depicted in **Fig. 3.4**, in which the pipeline timing is highlighted.

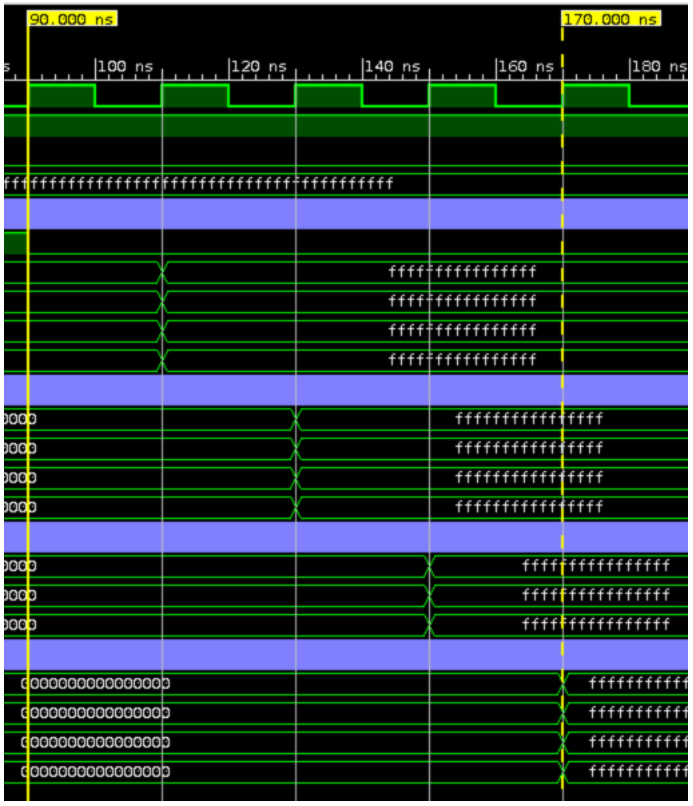


Figure 3.4: Sephirot Pipeline Simulation

As in almost all the pipelined CPU architectures, the management of branches can significantly affect the performance of Sephirot. If a branch is taken, the instructions that are in the fetch and decode stages could be invalid, resulting in what is usually called misprediction. When a branch instruction is processed, the next instruction to be executed is known only at the end of the execute stage.

Although stalling the pipeline and waiting for the result of the branch instruction is a possible solution, it decreases the microprocessor performance. Instead, we implemented

a simple speculative execution mechanism that always assumes that the branch is not taken, starting executing instruction right after the fetch of the branch instruction. The misprediction costs several clock cycles equal to the depth of the pipeline. For this reason, Sephirot implements a very short pipeline, having the branch misprediction penalty quantifiable in 4 clock cycles. Every stage takes exactly one clock cycle to be executed. Using pipelining registers between the fetch, decode, and execute stages allows the execution of a complete instruction (4 syllables) per clock cycle.

Another issue arising from the use of a pipelined architecture is the fact that consecutive dependent instructions will encounter a data hazard. If an instruction in the decode stage depends on the results of another instruction that is in the execute stage, it will get the old value of the data stored in the registers. The new one will only be written on the register file on the next clock cycle.

To overcome this issue, *lane forwarding* has been implemented. In this way, when a syllable is in the execute stage, it sends its result not only to the register file but also to the decode stage. This allows the instruction in the decode stage to always have the updated value. The details of each lane are depicted in **Fig. 3.5**.

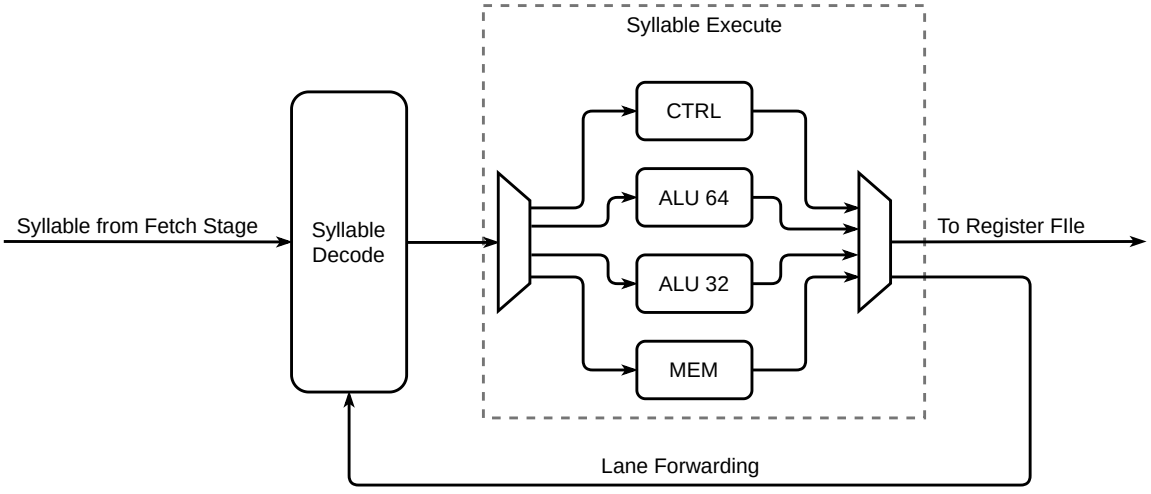


Figure 3.5: Sephirot Lane Architecture

The first optimization in area occupancy has been achieved by noticing that branch instruction cannot be parallelized. Thus, the *control* unit is present only on the first lane. The compiler for the VLIW architecture must take care of placing the branch instructions only on the first lane.

The *Program Counter* unit provides the address for the next instruction to be fetched from the instruction memory. It also implements a finite-state machine to handle control signals at each pipeline stage. In particular, when a branch is taken, the program counter unit flushes the pipeline. This unit is also responsible for stalling the pipeline and calling the correct eBPF helper function when a `call` syllable is executed.

The *Register File* is a 4-port RAM with 11 64-bit wide locations. That map precisely the ones of the eBPF virtual machine.

3.2 Synthesis Results

The core has been synthesized on a NetFPGA Sume [67], a Virtex-7-based board specifically designed for networking applications. The board is depicted in **Fig. 3.6**.

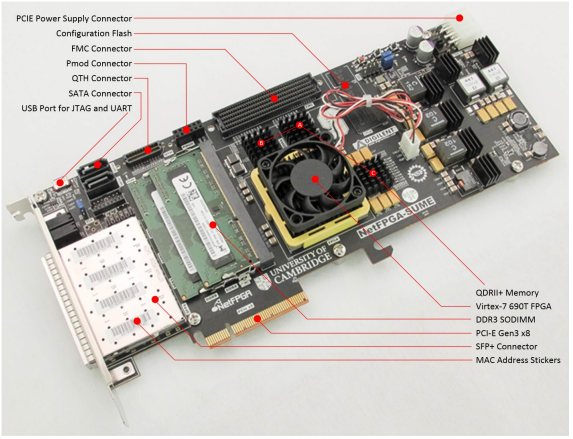


Figure 3.6: NetFPGA SUME

The NetFPGA SUME design aims to create a low-cost, PCIe host adapter card able to support 40Gb/s and 100Gb/s applications. The NetFPGA SUME uses a large FPGA, supporting high-speed serial interfaces of 10Gb/s or more presented both in standard interfaces (SFP+) and in a format that permits easy user-expansion, a large and extensible quantity of high-speed DRAM, alongside a quantity of high-throughput SRAM, and all this constrained by a desire for low cost to enable access by the wider research and academic communities. The block diagram of the NetFPFA SUME is reported in **Fig. 3.7**.

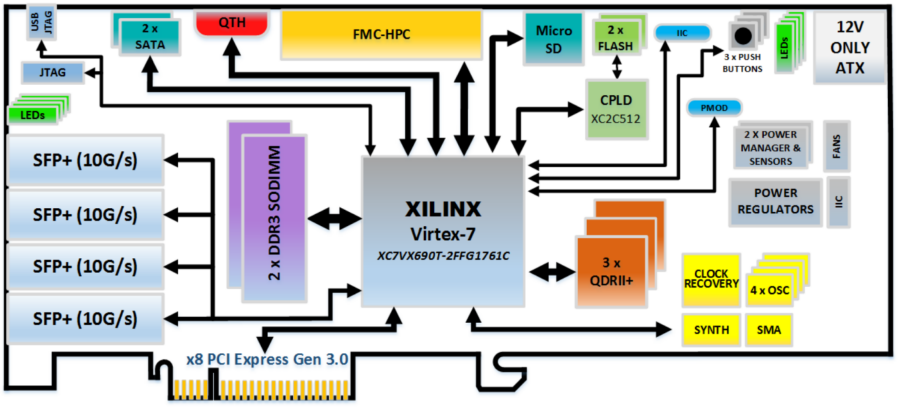


Figure 3.7: NetFPGA SUME block Diagram

3.2.1 Full-core Synthesis

The results of the synthesis of the core are given in **Table 3.1**. To achieve timing closure, *retiming* has been enabled in Vivados's synthesizer. This allowed the synthesizer to add pipelining registers to cut the critical path without altering the timing of the CPU.

Unsurprisingly, the critical path was due to the large adder inside the 64-bit ALU. In fact, the Virtex-7 provides specialized blocks called *DSP48* [62] to allow fast arithmetic operations in the FPGA fabric, without generating distributed adders.

Resource Type	Used	Available	Utilization %
Slice LUTs	19559	433200	4.52%
Slice FFs	5111	866400	0.59%
F7 Muxes	1030	216600	0.48%
F8 Muxes	512	108300	0.47%
DSP48E1	40	3600	1.11%

Table 3.1: Sephirot full core utilization

The problem is that the input size of those adders is limited to 48 bits, while our operands are of 64bits, thus having the remaining 16 bits implemented as a distributed adder. Despite this, a frequency of 123 MHz has been achieved in all the variants of Sephirot.

The results of the timing are given inf **Fig. 3.8**.

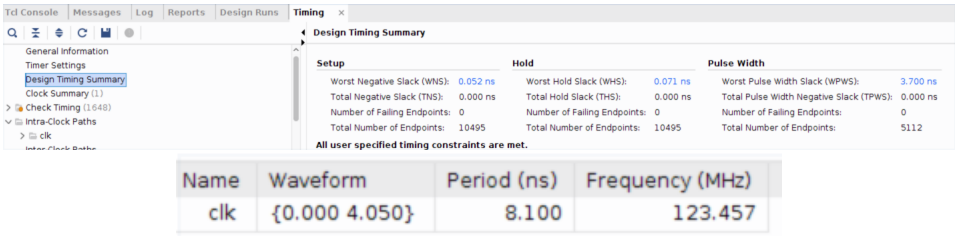


Figure 3.8: Full-core timing report

On the overall occupancy, the impact of the register file is detailed in **Table 3.2**, while the occupancy of a single lane is detailed in **Table 3.3**.

Resource Type	Used	Available	Utilization %
Slice LUTs	3977	433200	0.92%
Slice FFs	2016	866400	0.23%
F7 Muxes	1024	216600	0.47%
F8 Muxes	512	108300	0.47%

Table 3.2: 4-Port register file utilization

Resource Type	Used	Available	Utilization %
Slice LUTs	4394	433200	1.01%
Slice FFs	683	866400	0.08%
F7 Muxes	2	216600	<0.01%
F8 Muxes	0	108300	0%
DSP48E1	10	3600	0.28%

Table 3.3: Full lane utilization

From the reports, we can derive that the hardware footprint of Sephirot is indeed very low and that, even if the register file cannot be synthesized as a *Block RAM*, the impact of a distributed approach on area and performance is not remarkable.
The synthesized lane is depicted in **Fig. 3.9**, while the complete execution unit is depicted in **Fig. 3.10**.

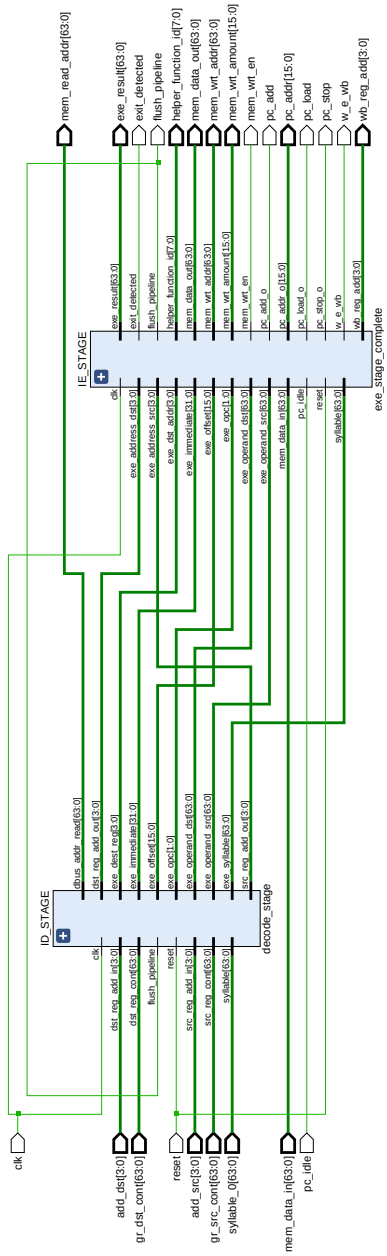


Figure 3.9: Synthesized Lane Schematic

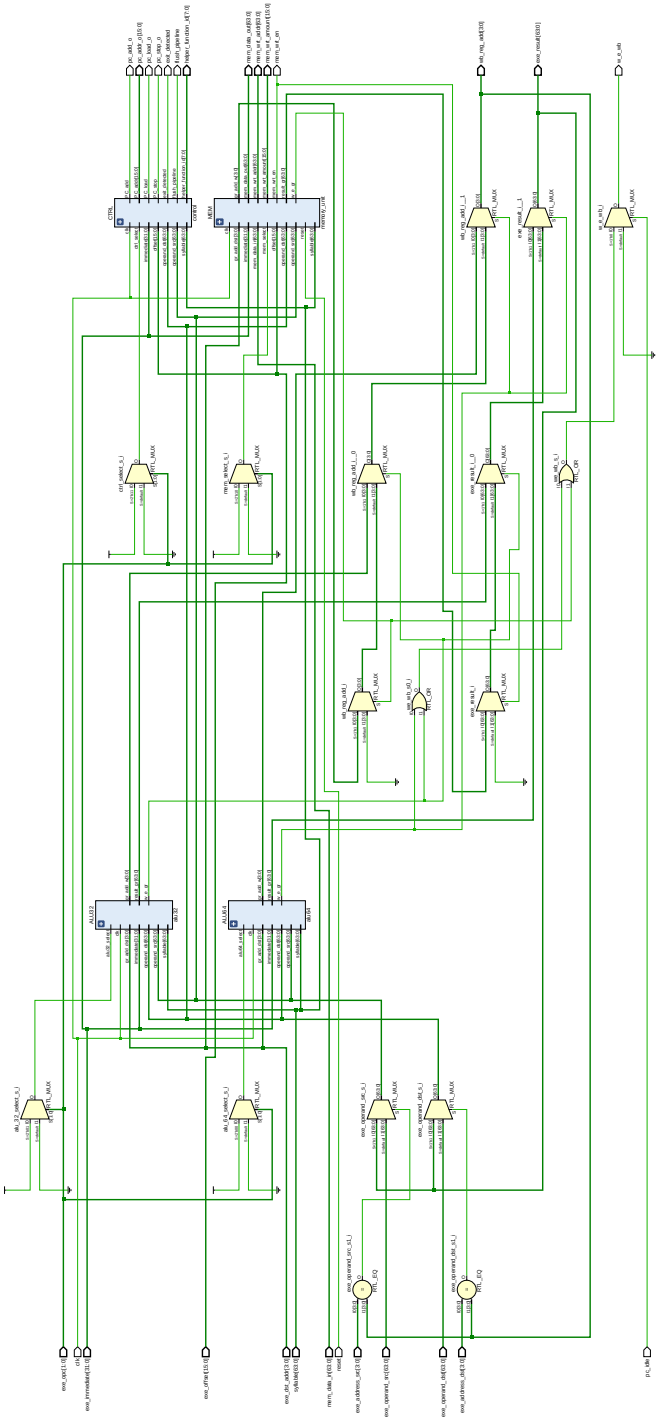


Figure 3.10: Synthesized Execution Unit Schematic

3.3 Performance-Area Tradeoff

In this section, we provide an analysis of the achievable Instruction Level Parallelism that can be reached on some of the examples of eBPF contained in the Linux kernel documentation.
This kind of analysis will drive us to the final architecture of Sephirot. In fact, the previous architecture was just a baseline, with a homogeneous structure of the 4 lanes.
For comparison, also a 2-lane architecture has been considered and implemented successfully.

3.3.1 ILP Analysis

In order to evaluate how much Instruction Level Parallelism is achievable, we need to apply Bernstein's condition. In the specific case of our architecture, one of the 3 conditions can be relaxed.
In fact, the $I_1 \cap O_2 = \emptyset$ condition does not need to be fulfilled since, even if the output of the second instruction it's equal to the input of the first, there is a one clock cycle delay between the two, thus they can be parallelized. This is the case of *load after store* sequences of instruction. One of the XDP examples on which the analysis has been carried out is reported in the following listing.

```
1   r6 = r1;
2   r7 = 1;
3   r1 = *(u32 *)(r6 + 4);
4   r2 = *(u32 *)(r6 + 0);
5   r3 = r2;
6   r3 += 14;
7   if r3 > r1 goto +108 <LBB0_9>;
8   r3 = *(u8 *)(r2 + 12);
9   r4 = *(u8 *)(r2 + 13);
10  r4 <<= 8;
11  r4 |= r3;
12  r7 = 2;
13  if r4 != 8 goto +102 <LBB0_9>;
14  r1 -= r2;
15  r1 <<= 32;
```

```
16      r1 s>>= 32;
17      r2 = 601;
18      if r2 s> r1 goto +97 <LBB0_9>;
19      r2 = 98;
20      r2 -= r1;
21      r1 = r6;
22      call 65;
23      r0 <<= 32;
24      r0 >>= 32;
25      if r0 != 0 goto +90 <LBB0_9>;
26      r1 = r6;
27      r2 = 4294967268 ll;
28      call 44;
29      r0 <<= 32;
30      r0 >>= 32;
31      r7 = 1;
32      if r0 != 0 goto +82 <LBB0_9>;
33      r1 = *(u32 *)(r6 + 4);
34      r6 = *(u32 *)(r6 + 0);
35      r2 = r6;
36      r2 += 126;
37      r7 = 1;
38      if r2 > r1 goto +76 <LBB0_9>;
39      r1 = *(u8 *)(r6 + 33);
40      *(u8 *)(r6 + 11) = r1;
41      r1 = *(u8 *)(r6 + 32);
42      *(u8 *)(r6 + 10) = r1;
43      r1 = *(u8 *)(r6 + 31);
44      *(u8 *)(r6 + 9) = r1;
45      r1 = *(u8 *)(r6 + 30);
46      *(u8 *)(r6 + 8) = r1;
47      r1 = *(u8 *)(r6 + 29);
48      *(u8 *)(r6 + 7) = r1;
49      r1 = *(u8 *)(r6 + 28);
50      *(u8 *)(r6 + 6) = r1;
51      r1 = *(u8 *)(r6 + 37);
52      *(u8 *)(r6 + 3) = r1;
53      r1 = *(u8 *)(r6 + 36);
54      *(u8 *)(r6 + 2) = r1;
55      r1 = *(u8 *)(r6 + 35);
56      *(u8 *)(r6 + 1) = r1;
57      r1 = *(u8 *)(r6 + 34);
58      *(u8 *)(r6 + 0) = r1;
59      r1 = *(u8 *)(r6 + 39);
60      *(u8 *)(r6 + 5) = r1;
61      r1 = *(u8 *)(r6 + 38);
62      *(u8 *)(r6 + 4) = r1;
63      r1 = *(u8 *)(r6 + 41);
64      *(u8 *)(r6 + 13) = r1;
65      r1 = *(u8 *)(r6 + 40);
66      *(u8 *)(r6 + 12) = r1;
67      r1 = 1027;
68      *(u32 *)(r6 + 34) = r1;
69      r1 = 18946;
70      *(u16 *)(r6 + 40) = r1;
71      r3 = r6;
```

```
72      r3 += 34;
73      r7 = 0;
74      r1 = 0;
75      r2 = 0;
76      r4 = 92;
77      r5 = 0;
78      call 28;
79      r1 = 1879048261;
80      *(u32 *)(r6 + 14) = r1;
81      r1 = 320;
82      *(u32 *)(r6 + 22) = r1;
83      r1 = *(u32 *)(r6 + 54);
84      *(u32 *)(r6 + 30) = r1;
85      r1 = *(u32 *)(r6 + 58);
86      *(u32 *)(r6 + 26) = r1;
87      r1 = r0;
88      r1 >>= 16;
89      r1 += r0;
90      r1 ^= -1;
91      *(u16 *)(r6 + 36) = r1;
92      r3 = r6;
93      r3 += 14;
94      r1 = 0;
95      r2 = 0;
96      r4 = 20;
97      r5 = 0;
98      call 28;
99      r1 = r0;
100     r1 >>= 16;
101     r1 += r0;
102     r1 ^= -1;
103     *(u16 *)(r6 + 24) = r1;
104     *(u64 *)(r10 - 8) = r7;
105     r2 = r10;
106     r2 += -8;
107     r1 = 0 ;
108     call 1;
109     if r0 == 0 goto +3 <LBB0_8>;
110     r1 = *(u64 *)(r0 + 0);
111     r1 += 1;
112     *(u64 *)(r0 + 0) = r1;
113     ;
114
115     LBB0_8:
116     r7 = 3;
117     ;
118
119     LBB0_9:
120     r0 = r7;
121     exit;
```

The same code, executed with 2-lanes can be compressed into the following code:

```
r6 = r1; r7 = 1;
r1 = *(u32 *)(r6 + 4); r2 = *(u32 *)(r6 + 0);
r3 = r2;
r3 += 14;
if r3 > r1 goto +108 <LBB0_9>;
r3 = *(u8 *)(r2 + 12); r4 = *(u8 *)(r2 + 13);
r4 <<= 8;
r4 |= r3; r7 = 2;
if r4 != 8 goto +102 <LBB0_9>;
r1 -= r2;
r1 <<= 32;
r1 s>>= 32; r2 = 601;
if r2 s> r1 goto +97 <LBB0_9>;
r2 = 98;
r2 -= r1; r1 = r6;
call 65;
r0 <<= 32;
r0 >>= 32;
if r0 != 0 goto +90 <LBB0_9>;
r1 = r6; r2 = 4294967268 ll;
call 44;
r0 <<= 32;
r0 >>= 32; r7 = 1;
if r0 != 0 goto +82 <LBB0_9>;
r1 = *(u32 *)(r6 + 4); r6 = *(u32 *)(r6 + 0);
r2 = r6;
r2 += 126; r7 = 1;
if r2 > r1 goto +76 <LBB0_9>;
r1 = *(u8 *)(r6 + 33);
*(u8 *)(r6 + 11) = r1; r1 = *(u8 *)(r6 + 32);
*(u8 *)(r6 + 10) = r1; r1 = *(u8 *)(r6 + 31);
*(u8 *)(r6 + 9) = r1; r1 = *(u8 *)(r6 + 30);
*(u8 *)(r6 + 8) = r1; r1 = *(u8 *)(r6 + 29);
*(u8 *)(r6 + 7) = r1; r1 = *(u8 *)(r6 + 28);
*(u8 *)(r6 + 6) = r1; r1 = *(u8 *)(r6 + 37);
*(u8 *)(r6 + 3) = r1; r1 = *(u8 *)(r6 + 36);
*(u8 *)(r6 + 2) = r1; r1 = *(u8 *)(r6 + 35);
*(u8 *)(r6 + 1) = r1; r1 = *(u8 *)(r6 + 34);
*(u8 *)(r6 + 0) = r1; r1 = *(u8 *)(r6 + 39);
*(u8 *)(r6 + 5) = r1; r1 = *(u8 *)(r6 + 38);
*(u8 *)(r6 + 4) = r1; r1 = *(u8 *)(r6 + 41);
*(u8 *)(r6 + 13) = r1; r1 = *(u8 *)(r6 + 40);
*(u8 *)(r6 + 12) = r1; r1 = 1027;
*(u32 *)(r6 + 34) = r1; r1 = 18946;
*(u16 *)(r6 + 40) = r1; r3 = r6;
r3 += 34; r7 = 0;
r1 = 0; r2 = 0;
r4 = 92; r5 = 0;
call 28;
r1 = 1879048261;
*(u32 *)(r6 + 14) = r1; r1 = 320;
*(u32 *)(r6 + 22) = r1; r1 = *(u32 *)(r6 + 54);
*(u32 *)(r6 + 30) = r1; r1 = *(u32 *)(r6 + 58);
*(u32 *)(r6 + 26) = r1; r1 = r0;
r1 >>= 16;
```

```
r1 += r0;
r1 ^= -1;
*(u16 *)(r6 + 36) = r1; r3 = r6;
r3 += 14; r1 = 0;
r2 = 0; r4 = 20;
r5 = 0;
call 28;
r1 = r0;
r1 >>= 16;
r1 += r0;
r1 ^= -1;
*(u16 *)(r6 + 24) = r1; *(u64 *)(r10 - 8) = r7;
r2 = r10;
r2 += -8; r1 = 0 ;
call 1;
if r0 == 0 goto +3 <LBB0_8>;
r1 = *(u64 *)(r0 + 0);
r1 += 1;
*(u64 *)(r0 + 0) = r1;

LBB0_8:
r7 = 3;

LBB0_9:
r0 = r7; exit;
```

While the same code executed with the 4-lane homogeneous architecture is:

```
r6 = r1; r7 = 1; r1 = *(u32 *)(r6 + 4); r2 = *(u32 *)(r6 + 0);
r3 = r2;
r3 += 14;
if r3 > r1 goto +108 <LBB0_9>;
r3 = *(u8 *)(r2 + 12); r4 = *(u8 *)(r2 + 13); r7 = 2;
r4 <<= 8;
r4 |= r3;
if r4 != 8 goto +102 <LBB0_9>;
r1 -= r2;
r1 <<= 32;
r1 s>>= 32; r2 = 601;
if r2 s> r1 goto +97 <LBB0_9>;
r2 = 98;
r2 -= r1; r1 = r6;
call 65;
r0 <<= 32;
r0 >>= 32;
if r0 != 0 goto +90 <LBB0_9>;
r1 = r6; r2 = 4294967268 ll;
call 44;
r0 <<= 32;
r0 >>= 32; r7 = 1;
if r0 != 0 goto +82 <LBB0_9>;
r1 = *(u32 *)(r6 + 4); r6 = *(u32 *)(r6 + 0);
r2 = r6;
```

```
26 r2 += 126; r7 = 1;
27 if r2 > r1 goto +76 <LBB0_9>;
28 r1 = *(u8 *)(r6 + 33);
29 *(u8 *)(r6 + 11) = r1; r1 = *(u8 *)(r6 + 32);
30 *(u8 *)(r6 + 10) = r1; r1 = *(u8 *)(r6 + 31);
31 *(u8 *)(r6 + 9) = r1; r1 = *(u8 *)(r6 + 30);
32 *(u8 *)(r6 + 8) = r1; r1 = *(u8 *)(r6 + 29);
33 *(u8 *)(r6 + 7) = r1; r1 = *(u8 *)(r6 + 28);
34 *(u8 *)(r6 + 6) = r1; r1 = *(u8 *)(r6 + 37);
35 *(u8 *)(r6 + 3) = r1; r1 = *(u8 *)(r6 + 36);
36 *(u8 *)(r6 + 2) = r1; r1 = *(u8 *)(r6 + 35);
37 *(u8 *)(r6 + 1) = r1; r1 = *(u8 *)(r6 + 34);
38 *(u8 *)(r6 + 0) = r1; r1 = *(u8 *)(r6 + 39);
39 *(u8 *)(r6 + 5) = r1; r1 = *(u8 *)(r6 + 38);
40 *(u8 *)(r6 + 4) = r1; r1 = *(u8 *)(r6 + 41);
41 *(u8 *)(r6 + 13) = r1; r1 = *(u8 *)(r6 + 40);
42 *(u8 *)(r6 + 12) = r1; r1 = 1027;
43 *(u32 *)(r6 + 34) = r1; r1 = 18946;
44 *(u16 *)(r6 + 40) = r1; r3 = r6;
45 r3 += 34; r7 = 0; r1 = 0; r2 = 0;
46 r4 = 92; r5 = 0;
47 call 28;
48 r1 = 1879048261;
49 *(u32 *)(r6 + 14) = r1; r1 = 320;
50 *(u32 *)(r6 + 22) = r1; r1 = *(u32 *)(r6 + 54);
51 *(u32 *)(r6 + 30) = r1; r1 = *(u32 *)(r6 + 58);
52 *(u32 *)(r6 + 26) = r1; r1 = r0;
53 r1 >>= 16;
54 r1 += r0;
55 r1 ^= -1;
56 *(u16 *)(r6 + 36) = r1; r3 = r6;
57 r3 += 14; r1 = 0; r2 = 0; r4 = 20; r5 = 0;
58 call 28;
59 r1 = r0;
60 r1 >>= 16;
61 r1 += r0;
62 r1 ^= -1;
63 *(u16 *)(r6 + 24) = r1; *(u64 *)(r10 - 8) = r7; r2 = r10; r1 = 0 ;
64 r2 += -8;
65 call 1;
66 if r0 == 0 goto +3 <LBB0_8>;
67 r1 = *(u64 *)(r0 + 0);
68 r1 += 1;
69 *(u64 *)(r0 + 0) = r1;
70
71 LBB0_8:
72 r7 = 3;
73
74 LBB0_9:
75 r0 = r7; exit;
```

In this example, when all four lanes are used, they do not require more than two

memory access. This justifies the implementation of a Sephirot core with four lanes, of which 2 of them perform only arithmetic instructions, thus reducing the area occupied. There are a few examples for which this is not true, as is the case of `xdp_redirect_map`. For this particular example, we are going to have a performance-area tradeoff analysis to determine if the two additional MEM units provide a boost in performance such that it will justify the additional area.
In all the XDP examples, only one would benefit from more than four lanes: `xdp_ip_tunnel`.

In the tables below, we report the parallelization analysis made on the XDP subset of examples of eBPF provided in the Linux Kernel documentation.

Example Name	**# Instructions**	**# Syllables**	**Speedup %**
xdp1	38	61	1.61%
xdp2	56	77	1.375 %
xdp2skb_meta	19	24	1.263 %
xdp_adjust_tail	76	111	1.461 %
xdp_drop_dump	10	12	1.200 %
xdp_fwd	167	268	1.605 %
xdp_monitor	101	150	1.485 %
xdp_redirect	16	26	1.625 %
xdp_redirect_map	21	33	1.571 %
xdp_router_ipv4	60	97	1.617 %
xdp_rxq_info	61	78	1.279 %
xdp_sample_pkts	23	40	1.739 %
xdp_sock	10	17	1.700 %
xdp_ip_tunnel	190	277	1.458 %

Table 3.4: Speedup with 2 lanes

Example Name	# Instructions	# Syllables	Speedup %	1 Lane Time %	2 Lanes Time %	3 Lanes Time %	4 Lanes Time %
xdp1	30	61	2.0333 %	56.67%	30.00%	13.3%	0.00%
xdp2	36	77	2.1388 %	47.22%	38.89%	5.56%	8.33%
xdp2skb_meta	13	24	1.846 %	38.46%	38.46%	23.0%	0.00%
xdp_adjust_tail	71	111	1.563 %	50.7%	38.03%	4.23%	7.04%
xdp_drop_dump	9	12	1.333 %	77.78%	11.11%	11.1%	0.00%
xdp_fwd	149	268	1.799 %	40.94%	42.95%	10.7%	5.37%
xdp_monitor	90	150	1.667 %	43.33%	46.67%	10.0%	0.00%
xdp_redirect	14	26	1.857 %	50.00%	28.57%	21.4%	0.00%
xdp_redirect_map	19	33	1.833 %	38.89%	44.44%	11.1%	5.56%
xdp_router_ipv4	59	97	1.644 %	42.37%	50.85%	6.78%	0.00%
xdp_rxq_info	51	78	1.529 %	58.82%	31.37%	7.84%	1.96%
xdp_sample_pkts	24	40	1.667 %	41.67%	50.00%	8.33%	0.00%
xdp_sock	9	17	1.889 %	33.33%	44.44%	22.2%	0.00%
xdp_ip_tunnel	161	277	1.720 %	45.96%	42.86%	4.35%	6.83%

3.3.2 2 Lanes Implementation

The Sephirot core has been reduced to only two lanes and synthesized as detailed in the previous chapter. The utilization report is given in **Tab. 3.5**, while the synthesized architecture is depicted in **Fig. 3.11**.

Resource Type	Used	Available	Utilization %
Slice LUTs	10280	433200	2.37%
Slice FFs	4114	866400	0.47%
F7 Muxes	533	216600	0.25%
F8 Muxes	260	108300	0.24%
DSP48E1	20	3600	0.56%

Table 3.5: 2-lanes Sephirot Utilization

As we expected from a rule-of-thumb analysis, the area occupied is roughly half of the full Sephirot implemented in the previous chapter. Also, the register file has shrunk by roughly half, as detailed in **Tab. 3.6**.

Resource Type	Used	Available	Utilization %
Slice LUTs	1949	433200	0.45%
Slice FFs	1616	866400	0.19%
F7 Muxes	512	216600	0.24%
F8 Muxes	256	108300	0.24%

Table 3.6: 2-ports Register File Utilization

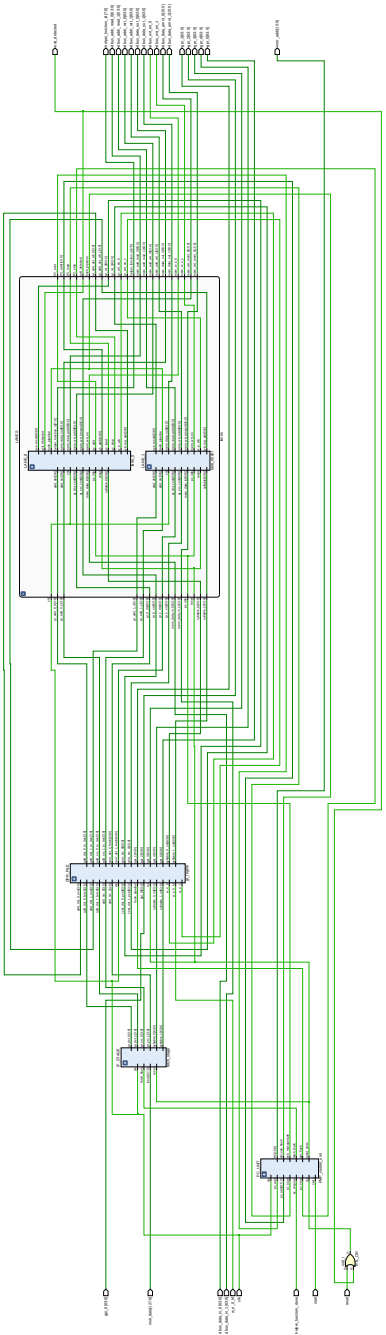


Figure 3.11: Synthesized 2-lane Sephirot Architecture

3.3.3 4 Lanes - Half Memory Unit Implementation

Considering the `xdp_redirect_map` example, the parallelized code for the 4 lane architecture with 4 memory units is:

```
r2 = *(u32 *)(r1 + 4); r6 = *(u32 *)(r1 + 0); r1 = 0; *(u32 *)(r10 - 4) = r1;
r0 = 1; r1 = r6;
r1 += 14;
if r1 > r2 goto +26 <LBB0_4>;
r2 = r10;
r2 += -4; r1 = 0 ll;
call 1;
if r0 == 0 goto +3 <LBB0_3>;
r1 = *(u64 *)(r0 + 0);
r1 += 1;
*(u64 *)(r0 + 0) = r1;

LBB0_3:;
r1 = *(u16 *)(r6 + 0); r2 = *(u16 *)(r6 + 6);
*(u16 *)(r6 + 0) = r2; r2 = *(u16 *)(r6 + 8); r3 = *(u16 *)(r6 + 2);
*(u16 *)(r6 + 8) = r3; *(u16 *)(r6 + 2) = r2; r2 = *(u16 *)(r6 + 10); r3 = *(u16 *)(r6 + 4);
*(u16 *)(r6 + 10) = r3; *(u16 *)(r6 + 4) = r2; *(u16 *)(r6 + 6) = r1; r1 = 0 ll;
r2 = 0; r3 = 0;
call 51;

LBB0_4:;
exit;

r0 = 2; exit;
```

We can thus evaluate the area occupied by a Sephirot implementation in which two of the four lanes are missing the memory unit. The utilization report is depicted in **Tab. 3.7**.

Resource Type	Used	Available	Utilization %
Slice LUTs	18843	433200	4.35%
Slice FFs	4487	866400	0.52%
F7 Muxes	1106	216600	0.51%
F8 Muxes	512	108300	0.47%
DSP48E1	40	3600	1.11%

Table 3.7: Sephirot 4 lanes - Half memory units utilization

The lane with only the ALU64 and ALU32 is depicted in **Fig. 3.12**, while its occupancy is detailed in **Tab. 3.8**.

Resource Type	Used	Available	Utilization %
Slice LUTs	3546	433200	0.82%
Slice FFs	382	866400	0.04%
F7 Muxes	0	216600	0%
F8 Muxes	0	108300	0%
DSP48E1	10	3600	0.28%

Table 3.8: Lane with only ALU's Utilization

The code for the `xdp_redirect_map` program needs to be modified in the following manner in order to be executed on the new architecture:

```
r2 = *(u32 *)(r1 + 4); r6 = *(u32 *)(r1 + 0);
r1 = 0; *(u32 *)(r10 - 4) = r1;r0 = 1;
r1 = r6;
r1 += 14;
if r1 > r2 goto +26 <LBB0_4>;
r2 = r10;
r2 += -4; r1 = 0 ll;
call 1;
if r0 == 0 goto +3 <LBB0_3>;
r1 = *(u64 *)(r0 + 0);
r1 += 1;
*(u64 *)(r0 + 0) = r1;

LBB0_3:;
r1 = *(u16 *)(r6 + 0); r2 = *(u16 *)(r6 + 6);
*(u16 *)(r6 + 0) = r2; r2 = *(u16 *)(r6 + 8);
r3 = *(u16 *)(r6 + 2);
*(u16 *)(r6 + 8) = r3;
*(u16 *)(r6 + 2) = r2;r2 = *(u16 *)(r6 + 10);
r3 = *(u16 *)(r6 + 4);
*(u16 *)(r6 + 10) = r3; *(u16 *)(r6 + 4) = r2;
*(u16 *)(r6 + 6) = r1; r1 = 0 ll; r2 = 0; r3 = 0;
call 51;

LBB0_4:;
exit;

r0 = 2; exit;
```

W.r.t. the full architecture, the half memory one has a performance loss of:

$$1 - \frac{\#instructions_{full}}{\#instructions_{half}} = 1 - \frac{19}{23} = 17,4\%$$

On the other hand, area has improved of:

$$LUTs: \ 1 \ - \ \frac{\#LUTs_{full}}{\#LUTs_{half}} \ = \ 1 \ - \ \frac{18843}{19559} \ = \ 3.66\%$$

$$FFs: \ 1 \ - \ \frac{\#FFs_{full}}{\#FFs_{half}} \ = \ 1 \ - \ \frac{4487}{5111} \ = \ 12.21\%$$

Based upon the example above, performance decreases a little more than what area can benefit from such an arrangement for the Sephirot lanes.

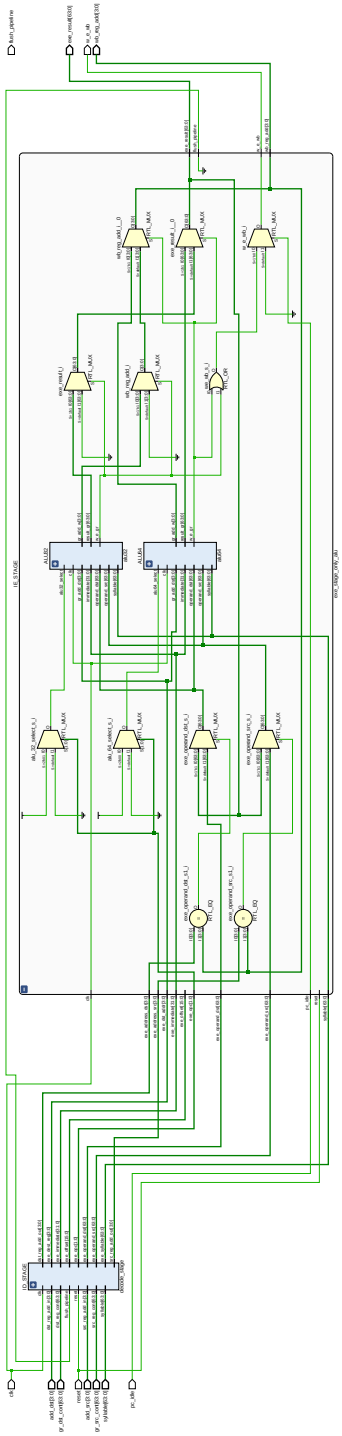


Figure 3.12: Synthesized Lane with only ALUs

Chapter 4

hXDP: Software Packet Processing on FPGA NICs

This chapter introduces a new datapath named hXDP that facilitates the transfer of eBPF kernel functionalities to the NetFPGA SUME. eBPF is a technology in Linux that enables the implementation of various functions such as load balancing [20], security [11], monitoring [2], deep packet inspection [6], policy enforcement [1], and more. The eBPF framework uses a virtual machine (VM) with its own Instruction Set Architecture (ISA) to run small programs within the Linux kernel. The VM has a register architecture, including a program counter (PC), ten general registers (R0 to R9), and a read-only stack pointer (R10), which contains the address of a 512B memory area that functions as the program's stack. These registers capture the current state of the program, which resets for every new run. eBPF provides data structure maps that save the state across program runs, and these are defined as memory areas and organized as lookup tables during compilation.

Programs written in high-level languages like C can be compiled to eBPF bytecode, which can then be loaded into the kernel using various hooks. This chapter concentrates on the XDP hook [30], and an eBPF program connected to the XDP hook is referred to

as an XDP program. The hook is provided at the level of the NIC driver. When a packet is received, the XDP environment carries out the following steps: (i) creates an xdp md struct to hold the packet buffer pointers and metadata, such as the input port id of the packet; (ii) sets R0 to indicate the address of the memory area that hosts the struct; (iii) and then starts the VM to execute the XDP program. At the conclusion of execution, the program can decide how to forward the packet by writing the forwarding action code in R0.

The bytecode is statically verified when loaded in the Linux kernel to ensure safety, e.g., guaranteed program termination. To enable verification, eBPF programs can only use a subset of the C expressive power. For instance, unbounded cycles and dynamic memory allocations are not allowed. To finally run on the target hardware, a second (just-in-time) compilation step translates the eBPF bytecode to the target machine code.

We have targeted eBPF for its appealing new features: it can execute arbitrary code, permit fast dynamic recompilation, and provides an efficient interface with the userland. The main characteristic of eBPF is to provide dynamic programmability at the kernel level without compromising the operating system's security and stability.

4.1 Challenges

To grasp an intuitive understanding of the design challenge involved in supporting XDP on FPGA, we now consider the example of an XDP program that implements a simple stateful firewall for checking the establishment of bi-directional TCP or UDP flows. A C program describing this simple firewall function is compiled into 71 eBPF instructions.

We can build a rough idea of the potential best-case speed of this function running on an FPGA-based eBPF executor, assuming that each eBPF instruction requires 1 clock cycle to be executed, that clock cycles are not spent for any other operation and that the FPGA has a 156MHz clock rate, which is common in FPGA NICs [67]. In such a

case, a naive FPGA implementation that implements the sequential eBPF executor would provide a maximum throughput of 2.8 Million packets per second (Mpps) under optimistic assumptions, e.g., assuming no additional overheads due to queues management. For comparison, when running on a single core of a high-end server CPU clocked at 3.7GHz, including operating system overhead and the PCIe transfer costs, the XDP simple firewall program achieves a throughput of 7.4 Million packets per second (Mpps). [1] Since it is often undesired or impossible to increase the FPGA clock rate, e.g., due to power constraints, in the lack of other solutions, the FPGA-based executor would be 2-3x slower than the CPU core.

Furthermore, existing solutions to speed-up sequential code execution, e.g., superscalar architectures, are too expensive in terms of hardware resources to be adopted. In a superscalar architecture, the speed-up is achieved by leveraging instruction-level parallelism at runtime. However, the complexity of the hardware required to do so grows exponentially with the number of instructions being checked for parallel execution. This rules out re-using general-purpose soft-core designs, such as those based on RISC-V [28, 25].

4.2 hXDP Overview

hXDP addresses the outlined challenge by taking a software-hardware co-design approach. In particular, hXDP provides both a compiler and the corresponding hardware module. The compiler takes advantage of eBPF ISA optimization opportunities, leveraging hXDP's hardware module features introduced to simplify exploiting such opportunities. Effectively, we design a new ISA that extends the eBPF ISA, specifically targeting the execution of XDP programs.

The compiler optimizations perform transformations at the eBPF instruction level: remove unnecessary instructions, replace instructions with newly defined, more concise

[1] Intel Xeon E5-1630v3, Linux kernel v.5.6.4.

instructions, and parallelize instructions execution. All the optimizations are performed at compile-time, moving most of the complexity to the software compiler, thereby reducing the target hardware complexity. Accordingly, the hXDP hardware module implements an infrastructure to run up to 4 instructions in parallel, implementing a Very Long Instruction Word (VLIW) soft processor. The VLIW soft-processor does not provide any runtime program optimization, e.g., branch prediction, instruction re-ordering, etc. We rely entirely on the compiler to optimize XDP programs for high-performance execution, freeing the hardware module of complex mechanisms that would use more hardware resources.

Ultimately, the hXDP hardware component is deployed to the FPGA as a self-contained IP core module. The module can be interfaced with other processing modules or placed as a bump-in-the-wire between the NIC's port and its PCIe driver towards the host system. The hXDP software toolchain, which includes the compiler, provides all the machinery to use hXDP within a Linux operating system.

From a programmer's perspective, a compiled eBPF program could be interchangeably executed in-kernel or on the FPGA.

4.3 Hardware Architecture

We design hXDP as an independent IP core, which can be added to a larger FPGA design. Our IP core comprises the elements to execute all the XDP functional blocks on the NIC, including `helper functions` and `maps`.

4.3.1 Architecture and components

The hXDP hardware design includes five components (see **Fig. 4.1**): the Programmable Input Queue (`PIQ`); the Active Packet Selector (`APS`); the Sephirot processing core; the Helper Functions Module (`HF`); and the Memory Maps Module (`MM`). All the modules work in the same clock frequency domain. The PIQ receives incoming data. The APS

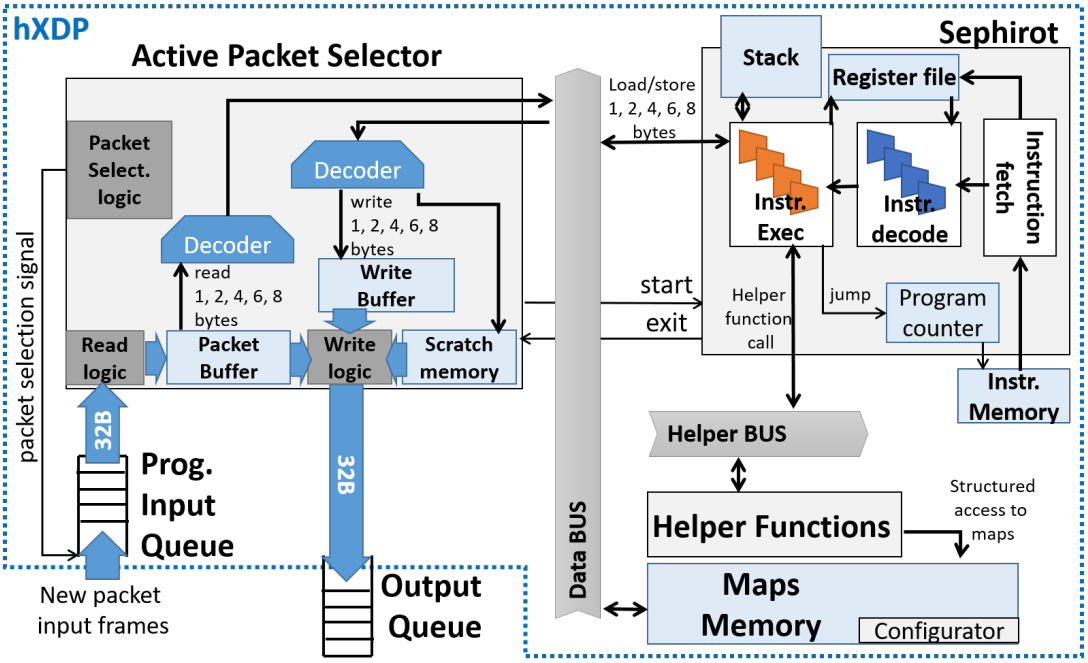


Figure 4.1: The logic architecture of the hXDP hardware design.

reads a new packet from the PIQ into its internal packet buffer. In doing so, the APS provides byte-aligned access to the packet data through a *data bus*, which Sephirot uses to read/write the packet content selectively. When the APS makes a packet available to the Sephirot core, the execution of a loaded eBPF program starts. Instructions are entirely executed within Sephirot, using 4 parallel execution lanes, unless they call a helper function or read/write to maps. In such cases, the corresponding modules are accessed using the *helper bus* and the *data bus*, respectively. Next, we detail the architecture's core component, i.e., the Sephirot eBPF processor.

Sephirot is a VLIW processor with 4 parallel lanes that execute eBPF instructions. Sephirot is designed as a pipeline of four stages: instruction fetch (IF); instruction decode (ID); instruction executes (IE), and commit. A program is stored in a dedicated instruction memory, from which Sephirot fetches the instructions in order. The processor has another dedicated memory area to implement the program's stack, which is 512B in size, and 11

64b registers are stored in the register file. These memory and register locations match one-to-one the eBPF virtual machine specification. Sephirot begins execution when the APS has a new packet ready for processing, and it gives the processor *start* signal.

On processor start (IF stage), a VLIW instruction is read, and the 4 extended eBPF instructions that compose it are statically assigned to their respective execution lanes. In this stage, the operands of the instructions are pre-fetched from the register file. The remaining 3 pipeline stages are performed in parallel by the four execution lanes. During ID, memory locations are pre-fetched if any of the eBPF instructions is a *load*, while at the IE stage, the relevant sub-unit are activated using the related pre-fetched values. The sub-units are the Arithmetic and Logic Unit (ALU), the Memory Access Unit and the Control Unit. The ALU implements all the operations described by the eBPF ISA, with the notable difference that it is capable of performing operations on three operands. The memory access unit abstracts the access to the different memory areas, i.e., the stack, the packet data stored in the APS, and the memory of the map. The control unit provides the logic to modify the program counter, e.g., to perform a *jump* and to invoke helper functions. Finally, during the commit stage, the results of the IE phase are stored back in the register file or to one of the memory areas. Sephirot terminates execution when it finds an exit instruction, in which case it signals to the APS the packet forwarding decision.

4.3.2 Pipeline Optimizations

We now list a subset of notable architectural optimizations we applied to our design.

Program state self-reset. As we have seen in previous sections, eBPF programs may perform zero-ing of the variables they will use. We automatically reset the stack and the registers at program initialization. This is an inexpensive feature in hardware, which improves security [19] and allows us to remove any such zero-ing instruction from the program.

Parallel branching. The presence of branch instructions may cause performance problems with architectures that lack branch prediction and speculative and out-of-order execution. For Sephirot, this forces a serialization of the branch instructions. However, in XDP programs, there are often series of branches in close sequence, especially during header parsing. We enabled the parallel execution of such branches, establishing a priority ordering of the Sephirot's lanes. All the branch instructions are executed in parallel by the VLIW's lanes. If more than one branch is taken, the highest priority one is selected to update the program counter. The compiler considers that when scheduling instructions and ordering the branch instructions accordingly.[2]

Early processor exit. The processor stops when an exit instruction is executed. The exit instruction is recognized during the IF phase, which allows us to stop the processor pipeline early and save the 3 remaining clock cycles. This optimization also improves the performance gain obtained by extending the ISA with parametrized exit instructions. XDP programs usually move a value to $r0$ to define the forwarding action before calling an exit. Setting a value to a register always needs to traverse the entire Sephirot pipeline. Instead, with a parametrized exit we remove the need to assign a value to $r0$, since the value is embedded in a newly defined exit instruction.

4.3.3 Implementation

We prototyped hXDP using the NetFPGA [67], a board embedding 4 10Gb ports and a Xilinx Virtex7 FPGA. The hXDP implementation uses a frame size of 32B and is clocked at 156.25MHz. Both settings come from the standard configuration of the NetFPGA reference NIC design.

The hXDP FPGA IP core takes 9.91% of the FPGA logic resources, 2.09% of the register resources, and 3.4% of the FPGA's available BRAM. The APS and Sephirot

[2]This applies equally to a sequence of `if...else` or `goto` statements.

are the components that need more logic resources since they are the most complex ones. Interestingly, even somewhat complex helper functions, e.g., a helper function to implement a hashmap lookup, have just a minor contribution in terms of required logic, which confirms that including them in the hardware design comes at little cost while providing good performance benefits. When including the NetFPGA's reference NIC design, i.e., to build a fully functional FPGA-based NIC, the overall occupation of resources grows to 18.53%, 7.3%, and 14.63% for logic, registers and BRAM, respectively. This relatively low occupation level enables the use of the largest share of the FPGA for other accelerators.

4.4 Compile-Time Optimizations

We now describe the hXDP instruction-level optimizations and the compiler design to implement them.

Instructions reduction. The eBPF technology is designed to enable execution within the Linux kernel, requiring programs to include several extra instructions, which the kernel's verifier then checks. When targeting a dedicated eBPF executor implemented on FPGA, most of such instructions could be safely removed, or cheaper embedded hardware checks can replace them. Two relevant examples are instructions for memory boundary checks and memory zero-ing.

`Boundary checks.` are required by the eBPF verifier to ensure that programs only read valid memory locations whenever a pointer operation is involved. In hXDP, we can safely remove these instructions, implementing the check directly in hardware.

`Zero-ing.` Is the process of setting a newly created variable to zero, and it is a typical operation performed by programmers both for safety and to ensure the correct execution of

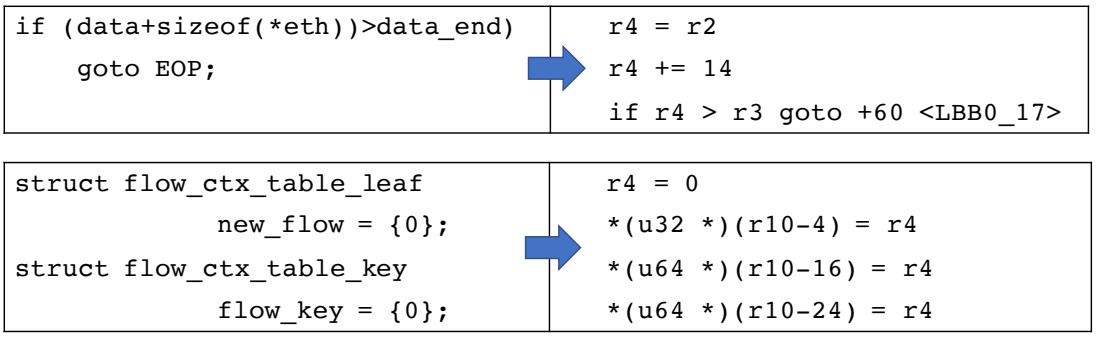


Figure 4.2: Examples of instructions removed by hXDP

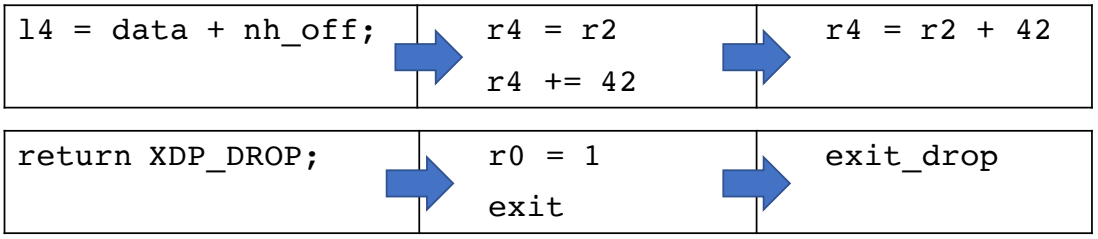


Figure 4.3: Examples of hXDP ISA extensions

their programs. A dedicated FPGA executor can provide hard guarantees that all relevant memory areas are zero-ed at program start, making the explicit zero-ing of variables during initialization redundant.

ISA extension. To effectively reduce the number of instructions, we define an ISA that enables a more concise description of the program. Here, there are two factors at play to our advantage. First, we can extend the ISA without accounting for constraints related to the need to support efficient Just-In-Time compilation. Second, our eBPF programs are part of XDP applications, and we can expect packet processing as the main program task. We use these two facts to define a new ISA that changes in three main ways from the original eBPF ISA.

`Operands number.` The first significant change has to involve the inclusion of three-

operand operations in place of eBPF's two-operand ones. Here, we believe that the eBPF's ISA selection of two-operand operations was mainly dictated by the assumption that an x86 ISA would be the final compilation target. Instead, using three-operand instructions allows us to implement an operation that would normally need two instructions with just a single instruction.

`Load/store size.` The eBPF ISA includes byte-aligned memory load/store operations, with sizes of 1B, 2B, 4B, and 8B. While these instructions are effective for most cases, we noticed that using 6B load/store can reduce the number of instructions in common cases during packet processing. 6B is the size of an Ethernet MAC address, a commonly accessed field. Extending the eBPF ISA with 6B load/store instructions often halves the required instructions.

`Parametrized exit.` The end of an eBPF program is marked by the exit instruction. In XDP, programs set the $r0$ to a value corresponding to the desired forwarding action (e.g., DROP, TX, etc.). When a program exits, the framework checks the $r0$ register to perform the forwarding action finally. While this extension of the ISA only saves one (runtime) instruction per program, it will also enable more significant hardware optimizations.

Instruction Parallelism. Finally, we explore the opportunity to perform parallel processing of an eBPF program's instructions. Since our target is to keep the hardware design as simple as possible, we do not introduce runtime mechanisms. Instead, we perform only static analysis of the instruction-level parallelism of eBPF programs at compile-time. We, therefore, design a custom compiler to implement the optimizations outlined in this section and to transform XDP programs into a *schedule* of parallel instructions that can run with hXDP. The compiler analyzes eBPF bytecode, considering both (i) the Data & Control Flow dependencies and (ii) the hardware constraints of the target platform. The

Program	Description
xdp1	parse pkt headers up to IP, and XDP_DROP
xdp2	parse pkt headers up to IP, and XDP_TX
xdp_adjust_tail	receive pkt, modify pkt into ICMP pkt and XDP_TX
router_ipv4	parse pkt headers up to IP, look up in routing table and forward (redirect)
rxq_info (drop)	increment counter and XDP_DROP
rxq_info (tx)	increment counter and XDP_TX
tx_ip_tunnel	parse pkt up to L4, encapsulate and XDP_TX
redirect_map	output pkt from a specified interface (redirect)

Table 4.1: Tested Linux XDP example programs.

schedule can be visualized as a virtually infinite set of rows, each with multiple available spots, which need to be filled with instructions. The number of spots corresponds to the number of execution lanes of the target executor. The compiler fits the given XDP program's instructions in the smallest number of rows, while respecting the three Bernstein conditions that ensure the ability to run the selected instructions in parallel [10].

4.5 Evaluation

We use a selection of the Linux's XDP example applications and two real-world applications to perform the hXDP evaluation. The Linux examples are described in Table 4.1. The real-world applications are the simple firewall described in the previous section and Facebook's Katran server load balancer [20]. Katran is a high-performance software load balancer that translates virtual addresses to actual server addresses using a weighted scheduling policy and providing per-flow consistency. Furthermore, Katran collects several flow metrics and performs IPinIP packet encapsulation.

Using these applications, we evaluate the impact of the compiler optimizations on the programs' number of instructions and the achieved level of parallelism. Then, we evaluate the performance of our NetFPGA implementation. We use the microbenchmarks also

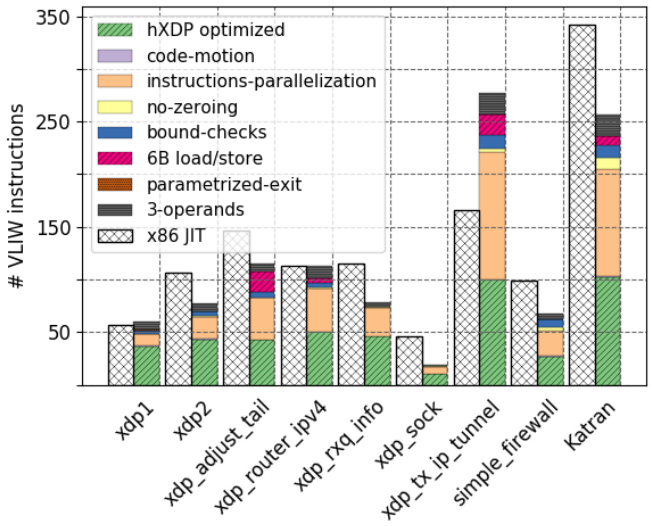


Figure 4.4: Number of VLIW instructions, and impact of optimizations on its reduction.

to compare the hXDP prototype performance with a Netronome NFP4000 SmartNIC. Although the two devices target different deployment scenarios, this can provide further insights into the effect of the hXDP design choices. Unfortunately, the NFP4000 offers only limited eBPF support, which does not allow us to complete the evaluation. We further include a comparison of hXDP to other FPGA NIC programming solutions before concluding the section with a brief discussion of the evaluation results.

4.5.1 Test Results

Compiler. **Fig. 4.4** shows the number of VLIW instructions produced by the compiler. We show the reduction provided by each optimization as a stacked column and report the number of x86 instructions, resulting in the output of the Linux's eBPF JIT compiler. In **Fig. 4.4**, we report the gain for instruction parallelization and the additional gain from *code movement*, which is the gain obtained by anticipating instructions from control equivalent blocks. As we can see, the compiler can provide several VLIW instructions that are often 2-3x smaller than the original program's number of instructions. Notice that, by

Program	# instr.	x86 IPC	hXDP IPC
xdp1	61	2.20	1.70
xdp2	78	2.19	1.81
xdp_adjust_tail	117	2.37	2.72
router_ipv4	119	2.38	2.38
rxq_info	81	2.81	1.76
tx_ip_tunnel	283	2.24	2.83
simple_firewall	72	2.16	2.66
Katran	268	2.32	2.62

Table 4.2: Programs' number of instructions, x86 runtime instruction-per-cycle (IPC) and hXDP static IPC mean rates.

contrast, the output of the JIT compiler for x86 usually grows the number of instructions.[3]

Instructions per cycle. We compare the parallelization level obtained at compile time by hXDP with the runtime parallelization performed by the x86 CPU core. Table 4.2 shows that the static hXDP parallelization often achieves a parallelization level as good as the one achieved by the complex x86 runtime machinery.[4]

Hardware performance. We compare hXDP with XDP running on a server machine and the XDP offloading implementation provided by an SoC-based Netronome NFP 4000 SmartNIC. The NFP4000 has 60 programmable network processing cores (called *micro-engines*), clocked at 800MHz. The server machine is equipped with an Intel Xeon E5-1630 v3 @3.70GHz, an Intel XL710 40GbE NIC, and Linux v.5.6.4 with the i40e Intel NIC drivers. During the tests, we use different CPU frequencies, i.e., 1.2GHz, 2.1GHz, and 3.7GHz, to cover a larger spectrum of deployment scenarios. Many deployments favor CPUs with lower frequencies and a higher number of cores [27]. We use a DPDK packet

[3]This is also due to the overhead of running on a shared executor, e.g., calling helper functions requires several instructions.

[4]The x86 IPC should be understood as a coarse-grained estimation of the XDP instruction-level parallelism since, despite being isolated, the CPU also runs the operating system services related to the eBPF virtual machine, and its IPC is also affected by memory access latencies, which more significantly impact the IPC for high clock frequencies.

generator to perform throughput and latency measurements. The packet generator can generate a 40Gbps throughput with any packet size and is connected back-to-back with the system-under-test, i.e., the hXDP prototype running on the NetFPGA, the Netronome SmartNIC, or the Linux server running XDP. Delay measurements are performed using hardware packet timestamping at the traffic generator's NIC and measure the round-trip time. Unless differently stated, all the tests are performed using packets with size 64B belonging to a single network flow. This is a challenging workload for the systems under test.

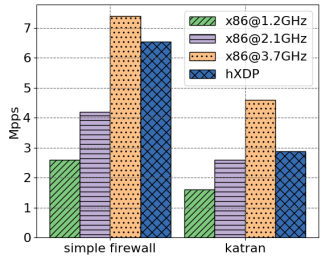


Figure 4.5: Throughput for real-world applications. hXDP is faster than a high-end CPU core clocked at over 2GHz.

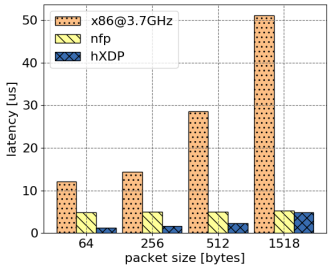


Figure 4.6: Packet forwarding latency for different packet sizes.

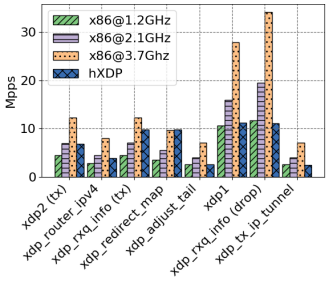


Figure 4.7: Throughput of Linux's XDP programs. hXDP is faster for programs that perform TX or redirection.

Applications performance. An optimistic upper bound for the simple firewall performance would have been 2.8Mpps. When using hXDP with all the compiler and hardware optimizations described in this dissertation, the same application achieves a throughput of 6.53Mpps, as shown in **Fig. 4.5**. This is only 12% slower than the same application running on a powerful x86 CPU core clocked at 3.7GHz, and 55% faster than the same CPU core clocked at 2.1GHz. Regarding latency, hXDP provides about 10x lower packet processing latency for all packet sizes (see **Fig. 4.6**). This is the case since hXDP avoids crossing the PCIe bus and has no software-related overheads. We omit latency results

for the remaining applications since they are not significantly different. While we are unable to run the simple firewall application using the Netronome's eBPF implementation, **Fig. 4.6** also shows the forwarding latency of the Netronome NFP4000 (`nfp` label) when programmed with an XDP program that only performs packet forwarding. Even in this case, we can see that hXDP provides a lower forwarding latency, especially for packets of smaller sizes.

When measuring Katran, we find that hXDP is instead 38% slower than the x86 core at 3.7GHz, and only 8% faster than the same core clocked at 2.1GHz. The reason for this relatively worse hXDP performance is the overall program length. Katran's program has many instructions; executors with a very high clock frequency are advantaged since they can run more instructions per second. However, notice the clock frequencies of the CPUs deployed at, e.g., Facebook's datacenters [27] have frequencies close to 2.1GHz, favoring many-core deployments in place of high-frequency ones. hXDP clocked at 156MHz can still outperform a CPU core clocked at that frequency.

`Linux examples.` We finally measure the performance of the Linux's XDP examples listed in Table 4.1. These applications allow us to better understand the hXDP performance with programs of different types (see **Fig. 4.7**). We can identify three categories of programs. First, programs that forward packets to the NIC interfaces are faster when running on hXDP. These programs do not pass packets to the host system; thus, they can live entirely in the NIC. For such programs, hXDP usually performs at least as well as a single x86 core clocked at 2.1GHz. Processing XDP on the host system incurs the additional PCIe transfer overhead to send the packet back to the NIC. Second, programs that always drop packets are usually faster on x86 unless the processor has a low frequency, such as 1.2GHz. Here, it should be noted that such programs are uncommon, e.g., those used to gather network traffic statistics receiving packets from a network tap. Finally, long programs, e.g., `tx_ip_tunnel` have 283 instructions and are faster on x86. As we

noticed in the case of Katran, with longer programs, the hXDP's implementation of low clock frequency can become problematic.

4.5.2 Comparison to other FPGA solutions.

hXDP provides a more flexible programming model than previous work for FPGA NIC programming. However, in some cases, simpler network functions implemented with hXDP could also be implemented using other programming approaches for FPGA NICs while keeping functional equivalence. One example is the simple firewall presented in this dissertation, which is also supported by FlowBlaze [52].

Throughput. Leaving aside the cost of re-implementing the function using the FlowBlaze abstraction, we can generally expect hXDP to be slower than FlowBlaze at processing packets. In fact, in the simple firewall case, FlowBlaze can forward about 60Mpps, vs. the 6.5Mpps of hXDP. The FlowBlaze design is clocked at 156MHz, like hXDP, and its better performance is due to the high level of specialization. FlowBlaze is optimized to process only packet headers using statically-defined functions. This requires loading a new bitstream on the FPGA when the function changes, enabling the system to achieve the reported high performance.[5] Conversely, hXDP has to pay a significant cost to provide full XDP compatibility, including dynamic network function programmability and processing of both packet headers and payloads.

Hardware resources. A second significant difference is the number of hardware resources required by the two approaches. hXDP needs about 18% of the NetFPGA logic resources, independently from the particular network function being implemented. Conversely, FlowBlaze implements a packet processing pipeline, with each pipeline's stage

[5]FlowBlaze allows the programmer to perform some runtime reconfiguration of the functions. However, this is a limited feature. For instance, packet parsers are statically defined.

requiring about 16% of the NetFPGA's logic resources. For example, the simple firewall function implementation requires two FlowBlaze pipeline stages. More complex functions, such as a load balancer, may require 4 or 5 stages, depending on the implemented load balancing logic [22].

In summary, FlowBlaze's pipeline leverages hardware parallelism to achieve high performance. However, it has the disadvantage of often requiring more hardware resources than a sequential executor, like the one implemented by hXDP. Because of that, hXDP is especially helpful in scenarios where a small amount of FPGA resources is available, e.g., when sharing the FPGA among different accelerators.

4.5.3 Discussion

Suitable applications. hXDP can run XDP programs with no modifications, however, the results presented in this section show that hXDP is especially suitable for programs that can process packets entirely on the NIC, and which are no more than a few 10s of VLIW instructions long. This is a common observation made also for other offloading solutions [29].

FPGA Sharing. At the same time, hXDP succeeds in using little FPGA resources, leaving space for other accelerators. For instance, we could co-locate on the same FPGA several instances of the VLDA accelerator design for neural networks presented in [16]. Here, one important note is about using memory resources (BRAM). Some XDP programs may need larger map memories. It should be clear that the memory area dedicated to maps reduces the memory resources available to other accelerators on the FPGA. As such, the memory requirements of XDP programs, which are known at compile time, is another important factor to consider when making program offloading decisions.

Chapter 5

Program Warping: Faster Hardware Packet Processing

XDP programs deal with packet processing, which always includes a few initial common steps for header parsing and classification. We can observe this in Listing 5.1, and in the program's flow diagram shown in Figure 5.1.a. The more complex program also includes these steps, even if one can expect them to be significantly more complex. This is shown in Figure 5.1.b, which reports the flow diagram for (a subset of) Katran [20], an L4 load balancer deployed in production by Facebook. In this initial part of the program, the program's control flow only depends on the input packet until a lookup into a map is performed. More interestingly, it is often the case that more diverse application-specific logic (e.g., additional lookups)

This is common in many programs: they perform parsing and classification and then, after the first lookup, they perform additional operations that are specific to the application logic.

From an implementation perspective, the programs' parts that depend only on the input packet are interesting, since they only need to perform reads of sets of bits from the packet, and comparisons on them. That is, read-only access to a small packet buffer

```
int l2_acl(struct xdp_md *ctx) {
 void *data_end =(void *)(long)ctx->data_end;
 void *data = (void *)(long)ctx->data;
 void *lookup_res = NULL;
 __u32 proto, nh_off;
 struct ethhdr *eth = data;
 __u8 key[6] = {0};

 nh_off = sizeof(struct ethhdr);
 if (data + nh_off > data_end) {
  return XDP_DROP;
 }
 proto = eth->h_proto;
 if (proto == BE_ETH_P_IP) {
  __builtin_memcpy(key, eth->h_source, 6);
  lookup_res =
        bpf_map_lookup_elem(&map, &key);
  if (lookup_res) {
   return XDP_PASS;
  } else {
   return XDP_DROP;
  }
 } else if (proto == BE_ETH_P_IPV6) {
  return XDP_DROP;
 } else {
  return XDP_PASS;
 }
}
```

Listing 5.1: A simple eBPF/XDP program example in C

and bit comparison operations is everything an executor needs, to fully implement such programs' parts. Therefore, in principle, these parts could be executed on a simpler engine than a full-fledged eBPF processor. Given the small set of simple operations, it may also be possible to specialize the engine implementation by making stronger assumptions on the execution model (e.g., lack of race conditions), and therefore provide faster execution of those program parts. An obvious candidate for such an engine is a (stateless) match-action pipeline similar to that employed by high-performance network switches [13].

Given that, we can rewrite the program from Listing 5.1 using the three match-action rules shown in Table 5.1. Rules #1 and #3 can be directly implemented by a match-action engine since they only involve reading some bits from the packet and checking their value.

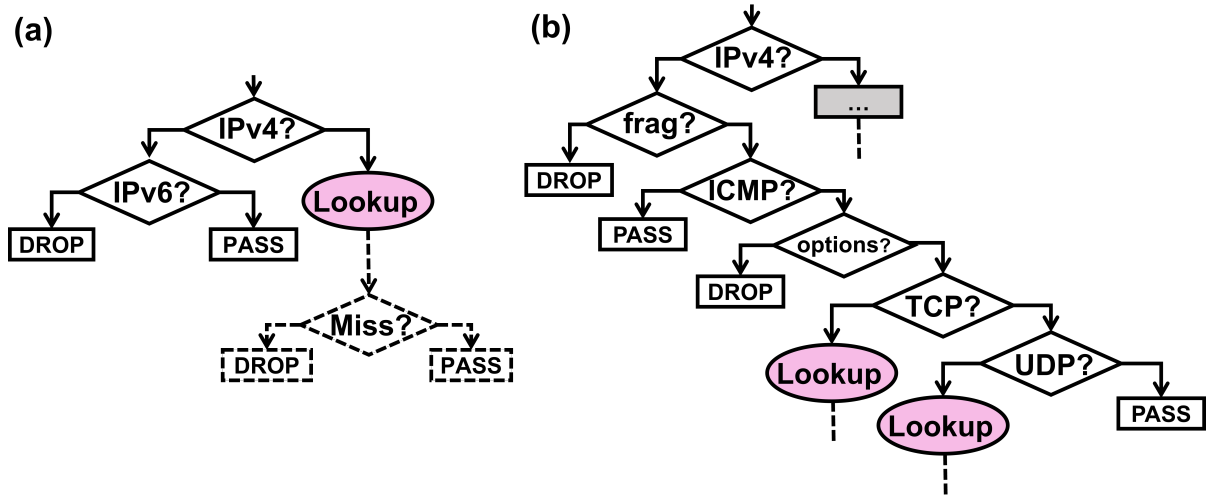


Figure 5.1: Flow diagrams for the program from Listing 5.1 (a) and for (part of) Katran (b). The bold solid lines highlight the part of the program that only needs reading packet data and comparison operations, without accessing any other state.

#	eth_proto	action
1	IPv6	DROP
2	IPv4	Continue Processing
3	*	PASS

Table 5.1: Match-action rules to implement part of the program from Listing 5.1. Rule #2 needs additional processing, which requires accessing data beyond the packet's content.

Rule #2 is more complex since it requires the execution of the program's part which includes the lookup and its downstream operations. More generally, this second part of the program requires reading/writing memory areas beyond that containing the packet data. The core idea of program warping is to anyway enable the execution of the first program's part on the simpler match-action engine. This requires designing a solution that is able to configure the downstream eBPF processor with any state needed to continue the XDP program's execution from where the match-action engine left it. For an eBPF executor, the state corresponds to the value of the general purpose registers ($R0 - 9$), the stack memory, and the PC.

5.1 Requirements and Challenges

From the discussion so far, it emerges that there is an opportunity to accelerate the execution of eBPF programs on FPGA NICs, by leveraging the common properties of XDP programs and purpose-built hardware designs. However, we are subject to requirements that make the implementation challenging.

Requirement 1: The eBPF/XDP programming experience should NOT change. That is, program warping should be transparent to the programmer, inferring any information about the program only from the unmodified eBPF bytecode.

Challenge: eBPF bytecode can implement arbitrary logic, and with the exception of using maps and helper functions, programmers are not constrained to any specific model, variable convention, or program structure. This complicates identifying the function implemented by the program's instructions.

Requirement 2: The eBPF/XDP deployment model should NOT change, and thus support dynamic program loading

Challenge: re-configurable match-action engines in the state-of-the-art do not provide support for dynamic reconfiguration. For instance, solutions that use P4 to describe match-action pipelines for FPGA NICs [61, 4] need to synthesize a new FPGA firmware and flash the FPGA anytime the packet parsing logic changes. This operation may take tens of minutes/hours, whereas in eBPF we expect the entire program compilation and loading process to last seconds at most.

Requirement 3: Program Warping should provide a significant packet processing speed improvement (¿2x) compared to existing systems (i.e., hXDP), while only using a small number of hardware resources, leaving the vast majority of the FPGA resources (¿80%)

available for other accelerators.

Challenge: systems like hXDP use already 5-10% of the FPGA resources. Program warping includes and extends such systems, therefore it can only use an additional 10%, at most.

5.2 System Design

In this Section we present the overall system design to support program warping, introducing the two components it comprises the *Warp Optimizer*; and the *Warp Engine*.

When co-designing the Warp Optimizer and the Warp Engine, we aim to reduce the set of operations the Warp Engine needs to implement. That is, our goal is to ensure the Warp Engine can run common program parts whose execution is needed in most cases, rather than implementing larger portions of the program that are executed less frequently. As we will see in Section 5.3, this will enable us to keep hardware complexity (hence, resources usage) under control, while gaining significant performance improvements.

Architecture Driven by this design goal, and without loss of generality, we design program warping as an extension to hXDP, the current state-of-the-art to run XDP programs on FPGA NICs [56]. hXDP provides the XDP environment for the FPGA NIC and a compiler that takes eBPF bytecode and outputs the machine code for the on-NIC XDP environment. The Warp Engine is a simplified packet parser and match-action unit that runs in parallel with hXDP, offloading from it the simpler, but time-consuming, parts of the program. We inherit the programming and deployment models from hXDP, which in turn implements the eBPF/XDP programming and deployment models. That is, from the perspective of an eBPF/XDP programmer, program warping does not introduce any change.

Compile time The programmer writes an XDP program and compiles to eBPF bytecode with their preferred toolchain, e.g., using C and LLVM to produce the bytecode. When loading the bytecode to the NIC, program warping extends the hXDP compiler by trig-

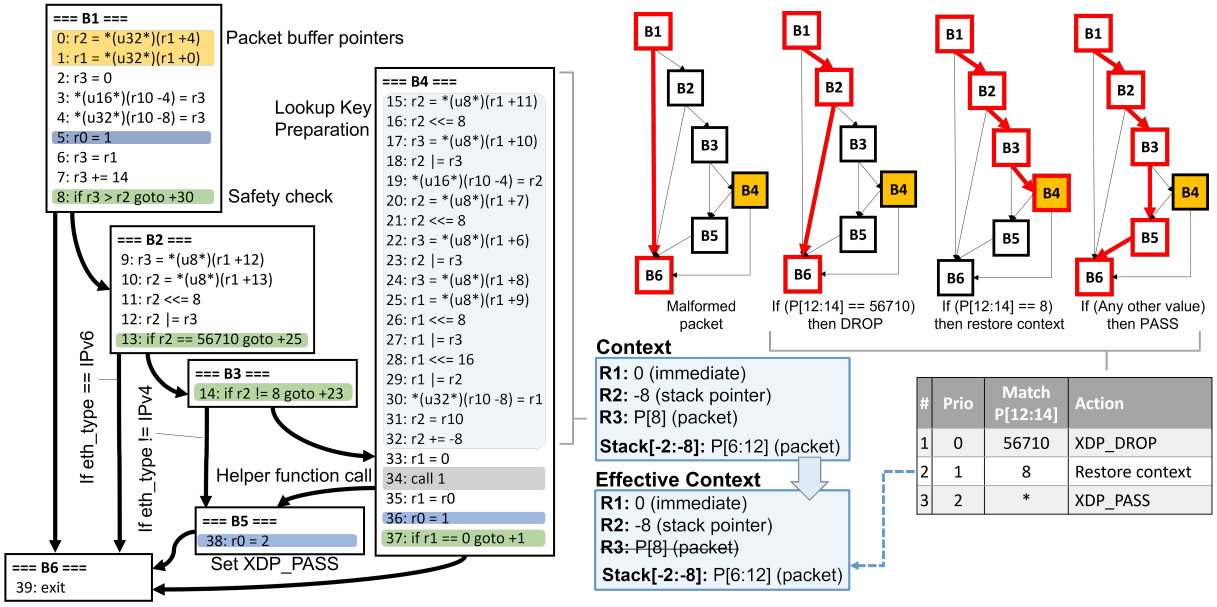


Figure 5.2: The operations of the Warp Optimizer for the program from Listing 5.1: (i) Control Flow Graph analysis; (ii) Match-action rules extraction; (iii) Context identification. All the operations are performed at compile time.

gering the Warp Optimizer first. The Warp Optimizer performs a static analysis on the bytecode to extract the program's parts that can *warped*, i.e., they can be run by the Warp Engine. The output of this process is a configuration for the Warp Engine in the form of match-action rules.

Runtime At runtime, the Warp Engine receives packets first and performs packet's bits extraction and match-action evaluation according to the received configuration. If a forwarding decision can be already taken within the match-action subsystem, the Warp Engine configures the value of the XDP Environment's $R0$ and transfers the packet to the XDP Environment that carries out the forwarding action. If instead, the program cannot run entirely in the Warp Engine, then a context restoration is triggered. This involves copying data from the packet to the XDP Environment's $R0 - R9$ and stack memory and setting the program counter to point to the next program's instruction. The Warp Engine performs such operations in parallel while also transferring the packet to the XDP

environment, where finally the processing continues to terminate the program's execution. Here, it is important to notice that the Warp Engine and the XDP Environment work in the pipeline, i.e., while the XDP Environment processes a packet, the Warp Engine is processing the next packets. This effectively ensures that the introduction of the Warp Engine *never* reduces the overall system throughput, and the only potentially negative runtime impact would be at most a negligible increase in the per-packet processing latency (i.e., few additional 10s of nanoseconds).

5.2.1 Warp Optimizer

The Warp Optimizer is a custom compiler that takes as input the eBPF/XDP bytecode generated from LLVM and produces as output: (i) the set of bits that should be extracted from the packet data; (ii) a set of match-action rules describing the part of the program that can be *warped*; (iii) the description of the context associated to *context restore* actions. The output is provided as a list of priority-ordered "if-then" statements, where the "if" part of contains the match conditions, and the "then" part is the corresponding action. The match conditions are described by a set of couples `(offset, length)`, which specify the bits of the packet that should be read. The actions can be of one of the following two types:

- A **forwarding decision** that neither modifies the packet nor the internal state of the system (e.g., the content of the maps), i.e., `DROP`, `PASS`, `TX` or `REDIRECT`;
- A **context restore** to continue execution in the XDP Environment, configured using the provided program counter and context (i.e., registers and stack content).

Here, we took two main design decisions, in order to meet our goal of reducing hardware complexity. First, the definition of the match conditions may be thought as roughly corresponding to the definition of packet header's fields, however, the Warp Optimizer (and the Warp Engine) have no knowledge of what a header field is. We could have implemented

a complete packet header parser logic, with header field parse graphs and corresponding state machines [26], but we purposely avoided that, in order to reduce the whole process to a simpler set of reads of a sequence of bit vectors from the packet data. This allows us to significantly simplify the hardware design, avoiding the implementation of state machines and enabling a fully pipelined execution of the bit vector extraction. Second, the Warp Optimizer only provides a forwarding decision action when there is no modification to the packet and no *side effects* due to the packet processing, i.e., helper function calling and map accesses. Modification to the packet would require additional hardware machinery, e.g., to compute values and write them in the specific packet's positions. Instead, accessing any internal state of the system would mean a tighter coupling with the XDP Environment, which would hinder the pipelining of the Warp Engine with the XDP Environment. More critically, it may introduce potential race conditions due to e.g., read-after-write for packets processed back-to-back in the Warp Engine pipeline and accessing the same state [54].

5.2.2 Program analysis

To extract the Warp Engine configuration, the Warp Optimizer performs static analysis of the input program. Here, recall that XDP programs can implement arbitrary computations, which generally complicates any static analysis task [42]. Nonetheless, the eBPF technology is designed to simplify static verification of programs loaded in the Kernel. This design goal helps also our case targeted at identifying the part of the program that can be *warped*. In particular, we benefit from the definition of three logically distinct memory areas: (i) the packet buffer; (ii) the stack; (iii) and maps. Each of these areas can be easily identified. The packet buffer is retrieved from the `struct xdp_md`, whose address is in eBPF VM's register $R1$ when a program starts. The stack base address is always stored in $R10$, which is a read-only register. Finally, maps are always accessed using a specific helper function. Using this information, the Warp Optimizer can trace the accesses to the different memory areas, and identify what reads/writes are performed to each of them.

In greater detail, the Warp Optimizer first builds the program's *Control Flow Graph* (CFG), e.g., see left part of Figure 5.2. The CFG is a directed graph, in which each *node* represents a code *block*, i.e., a set of instructions that are all executed if the program's control flow triggers the execution of the block's first instruction. The directed edges show how the different blocks might be executed one after the other, depending on the results of (conditional) jumps. Second, the Warp Optimizer converts the eBPF instructions, which use physical registers, into a Static Single Assignment (SSA) form. In this form, physical registers are substituted with variables that identify the instruction that has defined them. This helps the tracking of the values accessed by each instruction, and therefore it allows the Warp Optimizer to identify the accessed packet data, and the values stored in stack and registers.

After these two processing steps, the Warp Optimizer divides the CFG's blocks into three categories: start node; middle nodes; and terminal nodes. Terminal nodes are the blocks containing as last instruction `exit`, `call`, or instructions that write to the packet data. Middle nodes are all the nodes that are not the start or terminal nodes, and they are further categorized into matching and non-matching nodes. This depends on whether the block ends with a conditional jump instruction (matching) or with any other instruction (non-matching).

5.2.3 Match-action rules generation

Match-action rules generated by the Warp Optimizer are triples $\langle matches,\ action,\ priority \rangle$, where `matches` is a list of (offset, length) couples, `action` is either a forwarding decision or context restore, and `priority` is an integer value where the lower number encodes the higher priority. To generate these triples, the Warp Optimizer runs the Algorithm 1. The algorithm defines a zero-initialized current priority counter, a list of bit-vectors extracted from the packet (`fields`), their corresponding matching values (`matches`), and the current `stack` and `registers`. Then, it performs Depth-First Search (DFS) on the CFG, starting

```
priority ← 0
matches, fields, rules, registers, stack ← [ ]
Function get_MAT(block, matches, fields, priority, rules, stack, registers):
    evaluate_instructions(block, registers, stack)
    last_insn ← block.instructions[LAST]
    if is_terminal(block) then
        if is_exit(last_insn) then
            rule ← ⟨matches, Action(r0), priority⟩
        end
        else
            action ← Action(PC=last_insn.pc, registers, stack)
            rule ← ⟨matches, action, priority⟩
        end
        rules ← rules ∪ {rule}
        priority + +
    end
    else
        block+ ← block.tnext
        block− ← block.fnext
        if is_match(last_insn) then
            fields ← fields ∪ {PacketField(last_insn)}
            matches+ ← matches ∪ {Match(last_insn)}
            get_MAT(block+, matches+, priority, rules, stack, registers)
        end
        get_MAT(block−, matches, priority, rules, stack, registers)
    end
```

Algorithm 1: Warp Optimizer Algorithm

from the start node and stopping the descent when it reaches a terminal node. The paths explored with this approach capture the part of the program that can be *warped*.

When performing DFS, the algorithm evaluates all the instructions in the node, updating the current stack and registers state (i.e., the `registers` and `stack` arrays). For each middle node, if it is a matching node, the algorithm creates a copy of the `fields` and `matches`, and adds to them the bit vector checked by the current's block matching condition. That is the condition of the conditional jump, and the variable's value used in the condition, respectively. This corresponds to checking `if packet_data[s:e] == X`, where `[s:e]` identifies a vector of `e - s` bits in the packet starting at offset `s`, and X is an `e - s` long bit vector. The current `registers` and `stack` are also copied since the

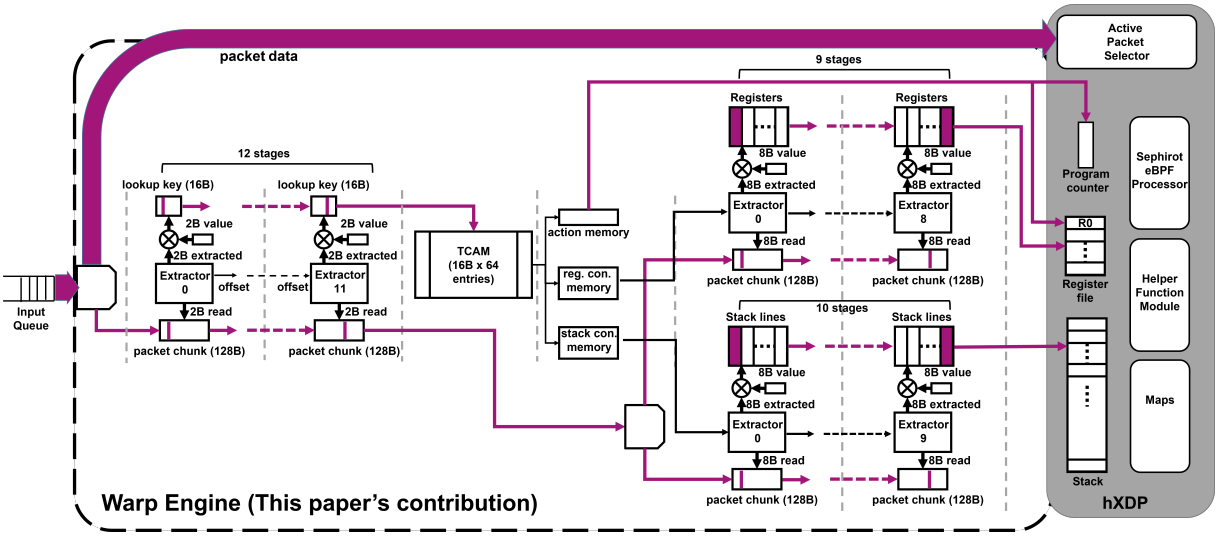


Figure 5.3: Warp Engine's architecture. The pipeline stalls only when hXDP is not ready to receive the next packet.

algorithm has to explore the two branches coming after the conditional jump, for which the program's state will evolve differently. When doing so, the algorithm first explores the branch corresponding to the *jump taken* case. When completing the exploration of that branch, the algorithm comes back to the latest encountered branching point, to explore the other branch, i.e., the one corresponding to the *jump not-taken* case (see right-top part of Figure 5.2).

A branch exploration terminates when there is a terminal node. After evaluating the instructions in the terminal node, the algorithm creates a `match-action rule` using the current list of `matches` and the current `priority` value. To define the action associated to the rule, the algorithm looks at the nodes' last instruction. If it is an `exit` instruction, the action is a forwarding decision, defined by the currently evaluated value of the `r0` register. Otherwise, the action is a restore context action, which includes $\langle pc,\ restored\_stack,\ restored\_registers \rangle$, where *pc* is the program counter of the instruction immediately following the node's last instruction, *restored_stack* and *restored_registers* are the evaluated current stack and registers, i.e., the *context* to be

restored (right-bottom part of Figure 5.2). After the rule creation, the priority counter is incremented. Since the CFG is explored by selecting first the branch-taken path, this ensures that the rules having the longer match list have higher priority, which is then useful to simplify the rule-matching logic implementation at runtime.

5.2.4 Warp Engine

The Warp Engine is a pipelined implementation of a *fused* packet parsing and the match-action unit, in principle similar to those implemented in switching ASICs, such as RMT [13], but with important conceptual differences and simplifications that are enabled by the co-design with the Warp Optimizer.

In hardware architectures such as RMT or in logical reference architectures used by languages like P4, e.g., PISA, there is a separation between the packet parsing subsystem and the match-action units. There, the packet parser usually implements finite state machines that navigate the parse graph performing iterative extraction of the packet's headers from the packet data. The parser identifies header types and their fields and extracts the corresponding values. Since the parser performs state tracking while iteratively exploring the packet data, it can understand what fields are to be extracted next, given the current field value [26]. The extracted fields are then used to compose a packet header vector (PHV). The downstream match-action units use a subset of the PHV to perform a lookup in a table and extract the corresponding action. The action is then implemented by action units that perform packet modification or update the internal state before delivering the packet to an output parser that re-constructs the packet headers before forwarding the packet.

In the Warp Engine, a large share of these subsystems is completely missing, e.g., there is no need for an output parser, and there is no distinction between the input parser and the match-action unit. Figure 5.3 shows an overview of the Warp Engine architecture. We can identify three conceptual sub-systems. First, there is a *key extraction unit*, which

comprises twelve stages and is in charge of building a 16B long vector extracting bits from the packet data. Second, a *match-action unit* uses the key to perform a lookup for a matching entry, which is associated with three areas in three distinct memories. These memory areas store the type of action associated with the packet, and the program context (register, stack) that should be restored in case of a context restore action. Finally, the last sub-system is the *context restoration unit*, which extracts the packet data required to build the context for a packet that needs to continue processing at the end of the Warp Engine. In our design, we use hXDP to implement the XDP environment on the FPGA, slightly modifying it to enable the Warp Engine to hook into the registers and stack memories.

An important aspect of the Warp Engine design is that the three sub-systems are part of a single pipeline that by design never stalls. In fact, the only case in which the pipeline stages do not advance processing is when hXDP is busy processing a previous packet, and therefore the hXDP's Active Packet Selector cannot host a new packet in its buffer memory. This design has also a second effect since hXDP is still in charge of the forwarding of each and any packet; the Warp Engine has no impact on the packet ordering.

5.2.5 Key Extractor

The Key Extractor is connected to the packet input queue through a *splitter*, which duplicates the first 128B of the packet to forward them to the Key Extractor's pipeline. The pipeline includes 12 stages, each implementing a configurable extractor module. The extractor reads up to 2Bs from the duplicated packet chunk, performs a simple bitwise operation on them, e.g., `and`, with a 2B long constant value, and finally writes the result of the such operation to a lookup key buffer. Which bytes to read, what operation to perform, and the value of the constant is all runtime-configurable parameters that are provided by the Warp Optimizer. Each extractor performs its operations in a single clock cycle and passes to the next extractor: (i) its modified lookup key buffer; (ii) the offset at which to write in such buffer; (iii) the packet chunk.

5.2.6 Match-action Unit

The match-action includes a ternary-addressable content memory (TCAM), and three memory areas: (i) the *Action Memory*, to store the actions associated to the TCAM entries, including action type, $R0$ value, and program counter; (ii) *Registers Configuration Memory*, which stores the Context Restoration Unit's configuration to extract the register values; (iii) *Stack Configuration Memory*, which provides a similar configuration but related to the stack. These three areas are organized in *lines* of different sizes, and for each memory, the number of lines is equal to the maximum number of TCAM entries. The lookup key provided by the key extractor is used to find the matching entry in the TCAM, which is associated with a single *line number* that is then used to access the three memory areas in parallel. Suppose the line extracted from the action's memory includes an action of a type forwarding decision. In that case, the pipeline propagates only such line to the next stage, eventually configuring the XDP Environment. Otherwise, each of the three lines extracted from the three memories is provided to the downstream pipeline. The content of the memory lines is set by the Warp Optimizer, which contains the full configuration for the Context Restoration Unit to build the stack and registers values. Both the TCAM entries and the memory lines can be re-configured at runtime.

5.2.7 Context Restoration Unit

The Context Restoration Unit comprises two parallel pipelines composed of modules closely resembling the Extractor modules of the Key Extractor. However, instead of reading just up to 2B from the packet chunk, the extractors of the Context Restoration Unit can read up to 8B each. This aligns with the eBPF VM's register size (64bit). Furthermore, they replace the lookup key buffer with larger buffers to host either the partially reconstructed stack or the reconstructed registers' state. Finally, instead of an offset, each extractor provides to its downstream extractor the line read from either the Registers Configuration

Memory or the Stack Configuration Memory, depending on which of the two parallel pipelines the extractor belongs to. Here, we point out that this approach was needed since the Context Restoration Unit has an additional complexity element when compared to the Key Extractor. The Key Extractor configuration is the same for any received packet, whereas the configuration for the Context Restoration Unit strictly depends on the content of the received packet. Since the entire system is organized in a pipeline, each stage of the Context Restoration Unit has to carry along the configuration to restore the context for the specific packet being processed in that stage.

These two parallel pipelines are fed by a second *splitter* that duplicates the packet chunk. The pipeline that restores the Stack has 10 stages and is connected to the Stack Configuration Memory. The line extracted from this memory provides the Extractors with the information needed to populate the stack, including constant values and values extracted from the packet chunk. This information includes: (i) the offset at which the packet chunk should be read; (ii) the operation to be performed on the read byte (and the constant value associated to that); (iii) the target address in the stack. The pipeline that restores the Registers has 10 stages, too, including 9 Extractors and a Delay element. This is the case since only 9 registers need restoration ($R1$ - $R9$) and the Warp Optimizer ensures that $R0$ is only read if its value is changed by the loaded XDP program after restoration.[1] The Delay element is required to synchronize the two context restoration pipelines.

5.2.8 Integration with hXDP

The Warp Engine pipeline ends in hXDP. Here, the Warp Engine waits for hXDP to be available to receive packets. Once that is the case, it first copies the packet data in the hXDP's Active Packet Selector (APS). The packet data transfer is also pipelined, and it

[1]Register $R10$ is read-only, and in hXDP, it has a constant value. $R0$ is used to store the return value of helper function calls, which are usually the first instruction run by the program after restoration.

happens in synch with the Warp Engine's pipeline. That is, multiple packets of data are moved through the pipeline at each clock cycle. Then, we have two possible behaviors. If the current action memory's line contains a forwarding action, then the Warp Engine sets $R0$ and instructs the APS to proceed with packet forwarding. The APS will then carry out forwarding according to the value provided in $R0$. Instead, if the action is a restore context action, then the Warp Engine sets the hXDP's program counter, registers $(R1 - R9)$, stack, and starts the hXDP's Sephirot eBPF Processor. Sephirot will then run the XDP program starting from the instruction pointed by the program counter, and using the provided registers and stack values.

5.2.9 Implementation

We implemented the Warp Engine design using the latest version of hXDP, which is integrated into Corundum [23], clocked at 250MHz, and targets a Xilinx Alveo U50 FPGA NIC [5]. The Warp Engine is clocked at 250MHz too, and its pipeline is 28 clock cycles long. Since at 250MHz, each clock cycle takes 4 nanoseconds, the Warp Engine introduces a fixed 112 nanoseconds of latency to each processed packet. This is a negligible overhead in the vast majority of cases, and it is the only runtime overhead introduced by the Warp Engine.

Our design has several parameters, e.g., the number of Key Extractor stages and the packet chunk size, which may be changed to meet different use case requirements. We summarize them in Appendix, along with the configuration we implemented in this dissertation, which is driven by the requirements of the 6 use cases we tested during evaluation (cf. Table 5.2).

The hardware implementation of this work is based on the one of hXDP [56]. This architecture is a streaming add-on module that intercepts packets on the AXI-Stream bus before they arrive at hXDP. This module is composed of the following sub-modules:

- AXI-Stream Splitter

- Programmable Input Parser
- Match-Action Table
- Programmable Output Parser

AXI-Stream Splitter. Since for all the use cases on which we have evaluated this work, no one required to access more than the first 128B of the packet, this module is responsible for taking the first two 64B (512 bits) AXI-Stream frames and combining them into a wider, 128B datapath. This module has a latency of 1 Clock Cycle.

Programmable Input Parser. This module is responsible for generating K_0 which will be used to index the Match-Action Table. It's composed by 16 programmable extraction micro-engines which are configured by the Host through the AXI4-Lite bus which is memory-mapped onto the PCIe Base Address Register #1 by Corundum. Each micro-engine receives the offset and the size of the field to extract. From the analysis of the use cases, the maximum size of the field to be extracted is 2 B and the maximum size of the key is given by Facebook's *kathara*, which is 16B. Each microengine takes exactly 1 Clock Cycle to extract the field and compute the partial key.

Match-Action Table. This module is the core of this contribution. The MAT is composed of a Ternary Content Addressable (TCAM) memory indexed by K_0. Since our compiler (?) generates a default rule by simply putting all `don't care` inside the key mask, the TCAM always matches. To cover all the use cases, the TCAM must support 64 entries, which can be easily fitted onto the FPGA's SRAM. The TCAM takes 2 clock cycles to output the matched line, which then is used by the RAMs containing the instructions to restore the context. The context is partitioned into

- Verdict & Program Counter

- Register Restoration Instructions
- Stack Restoration Instructions

If a Verdict is present for the matched line, all other information is disregarded and the Verdict gets passed to hXDP's Active Packet Manager which will implement the action. On the other hand, if the verdict is not present, the Register and Stack restoration instructions are passed to the Programmable Output Parser. Each memory takes 1 CC, thus the overall latency of the MAT is 3 CC.

Programmable Output Parser. Similarly to the PIP, the POP is organized in two parallel branches, each one composed of 8 micro-engines capable of extracting 8B in 1CC from the packet or inserting an immediate inside the context. The overall latency of this block is 8CC.

The total latency introduced by the add-on module is 28CC, that at 250MHz equates to 112ns. The throughput remains the same due to the pipelined nature of the architecture.

5.3 Evaluation

In this Section we present an evaluation of our system prototype, addressing program warping correctness, compiler optimizations impact, hardware resources requirements, and end-to-end system experimental evaluation.

5.3.1 Applications

We use 6 different applications to perform the evaluation, as detailed next. Table 5.2 summarizes them and reports relevant metrics, including their requirements in terms of

Application	*Instructions*		*TCAM*	*Match*	*Max Stack*
	eBPF	**hXDP**	*Entries*	*size [B]*	*size [B]*
L2 ACL	40	27	3	2	6
Router	119	95	9	4	8
Tunnel	283	155	7	4	24
DNAT	228	135	6	6	40
Suricata	138	65	49	12	40
Katran	1398	1013	20	16	88

Table 5.2: Tested applications and key metrics

Warp Engine's TCAM entries, lookup key size, and max Stack size.

L2 ACL (Running example). This is the application we used as a running example and described in the introduction to this chapter. It includes three branches: the main processing branch handles IPv4 packets and checks whether the source MAC address is present in the access list; the other two branches handle IPv6 packets, which are always dropped), and any packet that is not IP, which is passed to the networking stack.

Dynamic NAT. Network Address Translation (NAT) for flows coming from a LAN and destined to a public network, and reverse translation. The application has two main branches: (i) one for packets originating from the LAN, and (ii) the other for those coming from the public network. When a flow's first packet from the LAN is processed, the application selects a new *NATed* port and saves it in the NAT binding table using the 5-tuple as flow identifier. Then it performs address translation and forwards the packet. For any following flow's packet, the application retrieves the NATed port and performs address translation accordingly. In a similar way, packets from the public network are subject to a reverse NAT if there is a corresponding entry in the NAT binding table, or if they are dropped otherwise. The context restoration occurs in the lookup of the NAT binding table.

XDP Router. An implementation of an IPv4/IPv6 router, provided as an eBPF application example with the Linux Kernel. It performs parsing of L2 and L3 headers, and then a lookup in two tables to take a packet routing decision. The first table is an exact match table that looks up the entire IP destination address. If the lookup in the first table fails, the application performs a second lookup in a Longest Prefix Match (LPM) table.

XDP TX Tunnel. This is another eBPF application example provided by the Linux Kernel. It performs IPinIP encapsulation matching on the destination IP address and destination L4 port. The application works with both IPv4 and IPv6, with the two main processing branches handling these two cases to assign the proper IPv4 or IPv6 encapsulation. A lookup in a hashtable matches the destination virtual IP address to retrieve the tunneling information.

Suricata IDS. Suricata [55] is a software Intrusion Detection System (IDS). Among its multiple features, it provides an XDP program that works as a filter, to perform early dropping of undesired flows. The XDP program contains a large number of processing branches to handle all the combinations of stacked 802.1Q and 802.1AD VLAN headers,nand performs a lookup in a hashmap to take some of the filtering decisions.

Katran. Katran [20] is an XDP-based Layer 4 load balancer. It encapsulates packets with specific destination Virtual IP addresses and balances the connections toward the available servers. The first part of the processing includes L3 parsing and handling of ICMP/ICMPv6 protocols, for early response to echo request messages. Then, a first map lookup retrieves the virtual IP information. The application uses this information to query a Least Recently Used (LRU) map, in order to fetch the address of a connection table. A query to the connection table finally retries the real IP address of the destination server.

If a destination is not found, and the packet has the SYN flag set, then Katran installs a new forwarding rule in the connection table to ensure forwarding consistency for the following packets of that flow.

5.3.2 Functional Equivalence

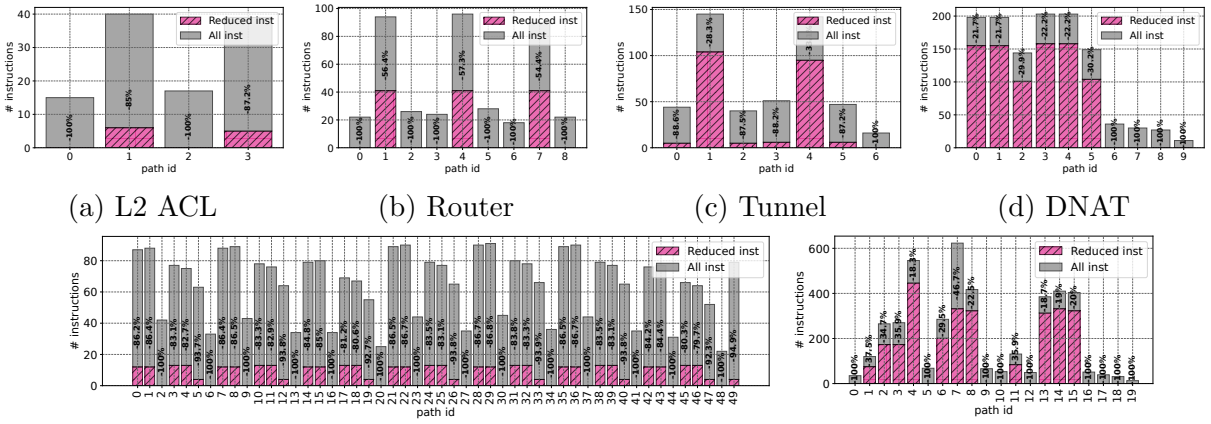
Program warping runs an XDP program on two executors that work together. Our first set of tests verifies that the resulting behavior is functionally equivalent to the behavior of a program running on a regular (single) eBPF executor.

Here, an important observation is that program warping deals only with the parts of a program that depend on the packet data, and in this sense the execution is stateless. Therefore, we can prove functional equivalence by comparing the output of the program running with program warping to the output of the program running on a regular eBPF processor. Since the Warp Optimizer does not modify the original eBPF bytecode, we could leverage `uBPF`[33], a userspace eBPF processor, extending it to implement a Warp Engine emulator in software, and use this implementation to check functional equivalence. In particular, for each of the tested applications, we: (i) enumerate all the program's control paths; (ii) generate input packets that trigger the execution of each of the listed paths; (iii) and finally verify that the output produced by our emulator is the same output produced by uBPF without program warping, for all the generated input packets.

We run this verification for the 6 applications described earlier, matching in all cases the expected output.

5.3.3 Warped instructions

We now evaluate the gain provided by the Warp Optimizer in terms of instructions being *warped*, i.e., instructions whose combined function is implemented by the Warp Engine,



(a) L2 ACL (b) Router (c) Tunnel (d) DNAT

(e) Suricata (f) Katran

Figure 5.4: Number of instructions per program's control flow path (background bar) vs number of instructions executed per path with program warping (foreground bar). Program warping reduces the instructions to be executed by 50-100%.

and which are therefore not executed anymore by the eBPF processor. This requires evaluating the instructions being actually executed at runtime. We re-use the `uBPF` emulator to perform this evaluation and compute the number of actually executed instructions for all the tested applications, and for all the control flow paths of each application. Since the control path at later stages of the program depends also on the stored state, e.g., entries in the maps, our testing strategy is adapted to test the multiple possible state conditions. For instance, in the case of the L2 ACL, after the map lookup there are two different paths: if the lookup returns an entry; or not (cf. Listing 5.1).

Figure 5.4 reports the results, showing the total number of eBPF instructions executed per path (background bar), and the number of instructions executed when program warping is in place (foreground bar). The results show that in many cases the number of instructions to be executed by the eBPF processor is reduced by over 50%, and that anyway in all cases it is reduced by at least 16.8%. More precisely, an application's control flow paths belong to one of two general categories, based on where their execution is going to be implemented: (i) *Mostly Warp Engine*; (ii) and *Mixed*.

Mostly Warp Engine. In all the applications there are at least a few control paths whose processing is mostly implemented by the Warp Engine. This is due to the practice of including in the programs several checks to take an early forwarding decision. Many of these early decisions are taken to *protect* programs from bogus or malicious traffic, which has an interesting implication: program warping may provide higher performance boosts when it is the most needed. For instance, some Denial-of-Service attacks' traffic may be entirely handled by the Warp Engine. Consistently, we can observe that all the control flow paths of Suricata fall in this category. In fact, Suricata generates XDP programs to filter network traffic as early as possible. As a result, program warping reduces the number of Suricata's eBPF instructions of up to 100%, i.e., the entire program is implemented by the Warp Engine. More generally, in Figure 5.4e we can see that this reduction depends on the specific path, and it is in the range 82%-100%.

Mixed. The second type of control flow path split their execution between the Warp Engine and the eBPF processor, either in equal parts or mostly using the latter. This is the case for e.g., Router, Tunnel, and DNAT. In such cases, the packet parsing and lookup key extraction are delegated to the Warp Engine, and the rest of the application logic, e.g., lookup in a single map and packet header mangling, is performed by the eBPF processor. In some programs, this second part is relatively simple. For instance, we see that program warping reduces the instructions of these paths by 54%-57% and 51% for Router (Paths 1, 4, 7) and DNAT (Paths 0, 1, 3, 4), respectively. In some applications, this second part is instead more complex. For instance, the Tunnel application's paths 1 and 4 are reduced by 28% and 33%, respectively. This is the case since these paths have a large number of instructions that deal with the packet encapsulation, which happens after a map lookup. In Katran's paths this is even more evident, due to the many lookups in maps that are performed in the program's paths. The reduction, in this case, ranges from

	Logic Res.		*Memory Res.*	
	LUTs	**Reg.**	**BRAM**	**URAM**
Corundum (C) + hXDP	10.7%	6.91 %	13.65%	2.34%
C + hXDP + Warp Engine	16.8%	9.86%	14.51%	8.44%
Increase in %pts	6.1%	2.95 %	0.86%	6.1%

Table 5.3: FPGA resources usage

16.8%-34.7%.

5.3.4 Warp Engine Hardware Requirements

We now evaluate the FPGA resources consumed by the Warp Engine. We compare the requirements with those of the latest hXDP version [5], which is integrated within the Corundum NIC [23] and targets a Xilinx Alveo U50. The U50 is equipped with a Xilinx Ultrascale+ FPGA, which offers 4 main types of resources that are of interest to us: (i) Lookup-Tables (LUTs); (ii) registers; (iii) block RAM (BRAM); and Ultra RAM (URAM). The LUTs and registers are the main building blocks to implement logic functions, whereas BRAM and URAM are two different memory blocks provided by the FPGA. BRAMs provide multi-port access, whereas URAM have a single read/write port but they are larger than BRAMs. Table 5.3 shows the resource requirements. For all the resource types, the Warp Engine is within our original requirement of increasing FPGA resource usage by no more than 5-10% of the overall available resources, when compared to hXDP. In fact, we are well below the target of using less than 20% of the resources for the combined design hXDP + Warp Engine.

5.3.5 End-to-end performance

We finally test the end-to-end system when processing traffic, measuring both packet throughout (in Million packets per second, Mpps) and end-to-end latency.

Testbed. We use two machines: a first machine is equipped with a 100Gbps Mellanox ConnectX-5 NIC, and it runs a DPDK-based traffic generator/receiver, capable of sending traffic at 100Gbps with 64B packets, i.e., ∼150Mpps; the second machine is equipped with a single port 100Gbps Xilinx Alveo U50, connected back-to-back with the first machine. We load our FPGA design on the U50, and configure it using the Warp Optimizer and hXDP toolchain. In all the tests, we measure packet forwarding that is handled entirely within the NIC, and drops. In this last case, we gain visibility by placing a dedicated drop counter within the FPGA design. Latency is always measured at the packet generator's machine, as the difference between the packet reception and packet sent (hw) timestamps. We do not measure the performance of packet processing that involves transferring data to the host system since Corundum's network driver can only forward a few Mpps, and it would therefore become the system's bottleneck [23]. However, we remark that in terms of Warp Engine+hXDP design, the transmission to a NIC's port or to the PCIe bus is implemented with the same hardware logic, therefore our system tests are representative of both cases.

Baseline. We perform a baseline test to measure throughput and latency when 100% of the program's instructions are implemented by the Warp Engine. Since the Warp Engine pipeline performs the same steps for all the applications and execution paths, the performance is the same in all cases. That is, we achieve ∼83Mpps when performing `DROP`, and 49.5Mpps with a 1∼ μs of per packet end-to-end latency when forwarding packets (`TX`). This is the same performance achieved by hXDP when running a program with a single instruction. In fact, the Warp Engine relies on hXDP to carry out the forwarding

action, and this performance matches the hXDP baseline performance. This confirms that the Warp Engine is never a throughput bottleneck in our design.

Applications. When considering the per-application performance, we have to take into account that each application has multiple execution paths, which are taken depending on the received traffic and the application's state. For paths that are 100% processed by the Warp Engine, the performance is the same as the baseline case, for all applications and paths. This often provides throughput improvements of over 10x for such paths (E.g., see last row of Table 5.4). For the remaining paths, in the interest of space, we report the performance for only a subset of them, focusing on those we identify as the most relevant cases. In particular, for L2 ACL, DNAT and Katran, we select the paths corresponding to successful lookups in the maps. For instance, this would be the path taken by established connections in both the DNAT and Katran cases. For Suricata, we see from Figure 5.4e a periodic pattern, which is due to the repetition of several different packet parsing combinations (e.g., including or not multiple levels of VLAN parsing). Among these, we select the worst case for program warping, i.e., the path corresponding to most instructions executed by hXDP. Finally, for Router and Tunnel, we analyze traffic traces to select the most common paths that would be triggered by processing such traffic. For the Router, we use a Datacenter trace [8], and the path handling IPv4 is triggered in over 80% of the cases. For Tunnel, we use a MAWI trace [39], and the case IPv4+TCP is triggered in 60% of the cases.

We summarize the results in Table 5.4. For the selected execution paths, program warping improves throughput by 1.23x-3.08x, and increases latency in the worst case by only 104 *nanoseconds*. We can make two important observations. First, program warping provides remarkable throughput improvements, nonetheless, compared to results from Figure 5.4, it seems that the tested paths provide a lower-than-expected speed-up. For instance, for the L2 ACL's path #1, Figure 5.4a shows that only 15% of the

instructions should be executed, which would suggest a potential throughput increase of over 6x. However, our test measures a 1.7x increase. This is the case since different hXDP instructions have different costs. For example, a `call` instruction may cost several clock cycles, and it also depends on variables such as the lookup key length. Therefore, the absolute number of instructions at compile time is not necessarily a good estimator for the achievable performance at runtime (unless an accurate per-instruction cost model is taken into account). Furthermore, it is important to notice that the Warp Optimizer works on the eBPF bytecode, which at a later stage is transformed by the hXDP compiler. The hXDP compiler may remove some instructions and parallelize others, therefore modifying the total program length (cf. Table 5.2). A side-effect of this is that the *warped* instructions may have been finally removed or parallelized by the hXDP compiler, which reduces the relative gain obtained by avoiding their execution. Second, in the case of Katran, we observe a reversed situation. Katran's throughput is improved to 2.3x, despite Figure 5.4f showing only an 18.7% reduction for the path #11's instructions. This is due to the relatively large number of (conditional) jumps in the first part of Katran's execution path. These jumps introduce bubbles in the hXDP's processor pipeline, lowering throughput and increased latency. In fact, Table 5.4 shows that in the case of Katran the Warp Engine significantly improves also forwarding latency, lowering it from 1.9$\sim$ μs to 1.5$\sim$ μs.

5.4 Discussion

Actual Performance. Program warping throughput improvement depends on: (i) the XDP program; and (ii) on which program's control path is executed. In our evaluations, we took a conservative approach and measured a set of paths that actually receives only a limited performance boost. In operational settings, other control paths are likely to be part of the workload. This may dramatically increase the performance gain. For instance, in Suricata, the last row of Table 5.4 shows the performance for one of the cases in which

Application **[Path ID]**	***Latency*** $[ns]$		***Tput*** $[Mpps]$		***Tput w/WE*** **vs w/o WE**
	w/ WE	**w/o WE**	**w/ WE**	**w/o WE**	
L2 ACL [#1]	1128	1024	9.26	5.43	170.37%
Router [#7]	1304	1212	3.47	2.66	130.58%
Tunnel [#4]	1368	1288	2.84	2.21	128.41%
DNAT [#2]	1444	1364	2.34	1.89	123.36%
Suricata [#46]	1124	1112	10.86	3.52	308.52%
Katran [#11]	1501	1942	2.08	0.90	231.08%
Performance for 100% instruction reduction scenarios					
Suricata [#23] (`DROP`)			82.87	4.31	1824,72%

Table 5.4: Warp Engine (WE) End-to-End Performance Results

the Warp Engine can entirely offload hXDP. In this case, the system provides an 18.2x throughput increase. Furthermore, it should be noted that these paths are not necessarily uncommon or rarely executed. On the contrary, often they may represent the commonly taken paths. For instance, CloudFlare defines XDP programs to perform early packet dropping for DDoS protection [11]. In such applications, these highly boosted paths are expected to be handling the majority of traffic.

Programming for Performance. This last observation suggests also another interesting property of program warping: programmers can describe processing *rules* using `if` statements and *hardcode* variables and constants when programming with the objective of optimizing performance. This is the same set of techniques used to optimize XDP programs running within the Linux kernel on x86 processors. Therefore, with program warping both the XDP programming models and best practices to write high-performance programs stay unchanged, matching XDP programmers' expectations.

FPGA vs ASIC. We designed program warping with re-programmability in mind. This gives us great flexibility in adapting the design parameters to accommodate dif-

ferent applications' requirements (e.g., changing the lookup key's size, the maximum number of TCAM entries, etc.). For instance, the design presented in this dissertation provides somewhat limited throughput for some of the use cases, because of the inherited hXDP limitations. That is, larger and complex programs, like Katran, can only achieve a throughput of few Mpps. Here, we point out that the entire FPGA design including Corundum+hXDP+Warp Engine accounts for less than 15% of the FPGA resources, leaving most of the FPGA for other (network-related or not) accelerators. Using Katran as an example, the resources occupation of hXDP+Warp Engine, and the per-flow state handling of Katran, would allow us to install multiple hXDP+Warp Engine modules and load balance packets among them in an RSS-like fashion. Alternatively, we notice that Katran requires roughly half of its hXDP processing time just to compute the packet's checksums for the newly introduced headers. This suggests that a small dedicated FPGA module may take over such common processing for all the packets, removing the corresponding instructions from the XDP program. Interestingly, these two approaches could be also combined.

A different take on this comes from the observation that program warping leverages a fixed design that does not necessarily rely by itself on FPGA re-programmability. Therefore, it should be possible to cast the very same design on an ASIC implementation, likely achieving a 5x-10x increase in the clock frequency (and performance). Indeed, for large volume use cases with more slowly changing requirements, we believe that building an ASIC-based hXDP+Warp Engine is a reasonable option, and we are considering exploring such a path in the future, e.g., integrating it in a switching chip design.

Chapter 6

Conclusions

Pure software approaches, such as SDN and NFV paradigms, are starting to expose deficiencies in jointly achieving performance and resource efficiency. Most come from the fact that commodity hardware is not growing as expected, posing a limit in how much this approach may result in sustainability in the future. In this Dissertation, we explored the new challenges and requirements brought out by processing packets at 100+ Gbps while exposing to the network's programmer a familiar interface called eBPF.

We tried to solve this issue by building a set of "building blocks": (i) Sephirot, an eBPF executor, (ii) hXDP, a complete eBPF offload system and (iii) Program Warping, which extends hXDP by offloading packet parsing to a dedicated hardware pipeline. By building Sephirot, we bridged the gap between the hardware and the software interface by implementing a "contract" between those known as the Instruction Set Architecture (ISA). In particular, we chose the eBPF ISA due to its popularity among network programmers and its capability to be implemented as an iterative executor (i.e., a processor). We chose a Very-Long Instruction Word Architecture (VLIW) for Sephirot to exploit the intrinsic Instruction Level Parallelism (ILP) contained in network programs. We synthesized the design on a NetFPGA SUME running at 156.25MHz, performing a design-space exploration informed by the workload profiling done on all Linux Kernel eBPF/XDP examples as well

as real-world use-cases, such as Meta's Katran. Being able to execute eBPF bytecode natively is only the first step. The second step is to support the entire software runtime environment in the hardware. This leads us to develop hXDP. This data plane contains Sephirot as the primary executor, interfacing it with *Maps* and *Helper Functions*. We also synthesized hXDP on a NetFPGA-SUME running at 156.25MHz, testing it head-to-head with a 3.7GHz Intel Xeon core. Our architecture provided significant performance benefits regarding throughput and latency, dramatically reducing power consumption for all the clusters of programs that live entirely on the NIC. With Program Warping, we slightly complicate the hardware architecture of hXDP by adding a Ternary Content Addressable Memory (TCAM) and a few programmable parsers to offload packet parsing, which is a standard first part of all eBPF programs we analyzed in our work. With Program Warping, we can make a forwarding decision on a packet in as much as 1 Clock Cycle, doing de-facto line rate packet processing at 100+ Gbps.

We hope that the combination of the works presented in this Dissertation can lay the foundation for a new model for both packet processing and application-level acceleration. We envision a new generation of hardware accelerators, tightly coupled with computing devices such as CPUs and GPUs, not necessarily made out of the same technological node, taking full advantage of the newcomer "chiplet" movement. By pushing fast and ultra-low power packet processing at the edge, we can deliver the promises of applications like real-time Artificial Intelligence (AI) inference at the edge and the Metaverse, as well as reduce the power footprint of cloud deployments.

References

[1] Cilium website. https://cilium.io.

[2] Hubble github repository. https://github.com/cilium/hubble.

[3] Linux socket filtering aka berkeley packet filter (bpf). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[4] P4-NetFPGA. https://github.com/NetFPGA/P4-NetFPGA-public/wiki.

[5] Pushing xdp into smartnics. https://fosdem.org/2021/schedule/event/sdn_hxdp_fpga/.

[6] Suricata Documentation. Using Capture Hardware: eBPF and XDP. https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html.

[7] AT&T, BT, CenturyLink, China Mobile, Colt, Deutusche Telekom, KDDI, NTT, Orange, Telefom Italia, Telefonica, Telstra, and Verizon. Network function virtualization - white paper. http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf.

[8] T. Benson. Data set for IMC 2010 data center measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.

[9] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[10] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[11] Gilberto Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, 2017.

[12] Marco Bonola, Giacomo Belocchi, Angelo Tulumello, **Marco Spaziani Brunella**, Giuseppe Siracusano, Giuseppe Bianchi, and Roberto Bifulco. Faster software packet processing on FPGA NICs with eBPF program warping. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 987–1004, Carlsbad, CA, July 2022. USENIX Association.

[13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[14] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM '13*, ACM SIGCOMM '13, pages 99–110. ACM, 2013.

[15] Marco Spaziani Brunella, Salvatore Pontarelli, Fabrizio Marrese, Marco Bonola, and Giuseppe Bianchi. Packet Manipulator Processor: A RISC-V VLIW core for networking applications. In *7th RISC-V Workshop*.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and*

Implementation (OSDI 18), pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[17] Derek Chiou. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 124–124. IEEE, 2017.

[18] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 1–14, New York, NY, USA, 2017. ACM.

[19] Mihai Valentin Dumitru, Dragos Dumitrescu, and Costin Raiciu. Can we exploit buggy p4 programs? In *Proceedings of the Symposium on SDN Research*, SOSR '20, page 62–68, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Facebook. Katran source code repository. `https://github.com/facebookincubator/katran`, 2018.

[21] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[22] FlowBlaze. Repository with FlowBlaze source code and additional material. `http://axbryd.com/FlowBlaze.html`.

[23] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020.

[24] Fungible, Inc. S1 DPU Product Brief. `https://www.fungible.com/wp-content/uploads/2021/01/PB0029.01.12020113-Fungible-S1-Data-Processing-Unit.pdf`.

[25] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[26] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design principles for packet parsers. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, page 13–24. IEEE Press, 2013.

[27] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: a datacenter infrastructure perspective. In *High Performance Computer Architecture (HPCA)*. IEEE, 2018.

[28] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2019.

[29] Oliver Hohlfeld, Johannes Krude, Jens Helge Reelfs, Jan Rüth, and Klaus Wehrle. Demystifying the performance of XDP BPF. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 208–212. IEEE, 2019.

[30] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Intel Corporation. Infrastructure Processing Units (IPUs). `https://www.intel.com/content/www/us/en/products/network-io/smartnic.html`.

[32] Intel Corporation. 5G Wireless. `https://www.intel.com/content/www/us/en/communications/products/programmable/applications/baseband.html`, 2020.

[33] IOVisor Project. uBPF repository. `https://github.com/iovisor/ubpf`.

[34] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):63, 2015.

[35] 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, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 243–259. USENIX Association, November 2020.

[37] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Marvell Technology, Inc. Data Processing Units. `https://www.marvell.com/products/data-processing-units.html`.

[39] MAWI. MAWILab traffic trace - samplepoint f - 2021-03-22. `https://mawi.wide.ad.jp/mawi/samplepoint-F/2021/202103221400.html`.

[40] 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.

[41] 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.

[42] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. Dynamic recompilation of software network services with morpheus, 2021.

[43] Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Comput. Surv.*, 54(4), may 2021.

[44] A. Nannarelli, M. Re, G. C. Cardarilli, L. Di Nunzio, **Marco Spaziani Brunella**, R. Fazzolari, and F. Carbonari. Robust throughput boosting for low latency dynamic partial reconfiguration. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, 2017.

[45] NEC. Building an Open vRAN Ecosystem White Paper. `https://www.nec.com/en/global/solutions/5g/index.html`, 2020.

[46] Netronome. AgilioTM CX 2x40GbE intelligent server adapter. `https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf`.

[47] NVIDIA Corporation. NVIDIA BlueField data processing unit (DPU). `https://www.nvidia.com/en-us/networking/products/data-processing-unit/`.

[48] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, 41(3):712–727, 2006.

[49] Pensando Systems. Pensando DSC-100 Product Brief. `https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-100-Product-Brief.pdf`.

[50] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[51] S. Pontarelli, M. Bonola, and G. Bianchi. Smashing sdn "built-in" actions: Programmable data plane packet manipulation in hardware. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–9, July 2017.

[52] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, **Marco Spaziani Brunella**, 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)*, Boston, MA, 2019. USENIX Association.

[53] Salvatore Pontarelli, Pedro Reviriego, and Juan Antonio Maestro. Parallel d-pipeline: A cuckoo hashing implementation for increased throughput. *IEEE Trans. Comput.*, 65(1):326–331, January 2016.

[54] 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 *ACM SIGCOMM '16*, ACM SIGCOMM '16, pages 15–28. ACM, 2016.

[55] Suricata. Suricata IDS Website. `https://suricata.io/`.

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

[57] **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 f. *Commun. ACM*, 65(8):92–100, jul 2022.

[58] **Marco Spaziani Brunella**, G. Bianchi, S. Turco, F. Quaglia, and N. Blefari-Melazzi. Foreshadow-vmm: Feasibility and network perspective. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 257–259, 2019.

[59] **Marco Spaziani Brunella**, Salvatore Pontarelli, Marco Bonola, and Giuseppe Bianchi. V-PMP: A VLIW packet manipulator processor. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–9. IEEE, 2018.

[60] **Marco Spaziani Brunella**, Sara Turco, Giuseppe Bianchi, and Nicola Blefari Melazzi. Foreshadow-VMM: on the practical feasibility of L1 cache Terminal Fault attacks. In *2019 ITASEC*.

[61] 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, page 122–135, New York, NY, USA, 2017. Association for Computing Machinery.

[62] Xilinx. *7 Series DSP48E1 Slice: User Guide*. 2018.

[63] Xilinx. 5G Wireless Solutions Powered by Xilinx. `https://www.xilinx.com/applications/megatrends/5g.html`, 2020.

[64] Xilinx, Inc. Alveo SN1000 SmartNIC. `https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html`.

[65] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 50–64, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communica-*

tion, SIGCOMM '20, page 283–295, New York, NY, USA, 2020. Association for Computing Machinery.

[67] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, 2014.