Simplifying Offloading Applications to the Network

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Pietro Giuseppe Bressana

under the supervision of
Robert Soulé

July 2020

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Pietro Giuseppe Bressana
Lugano, 20 July 2020

Abstract

Recent advancements in the networking field have made the data plane fully programmable, thanks to the introduction of both programmable network architectures and network programming languages. P4 [1], the most prominent network programming language, provides software abstractions to the components of PISA [2], a programmable network architecture that is able to process huge amounts of network traffic, through a pipeline of match-action stages. P4 code can be compiled to several different targets, including programmable network switches [3] and smartNICs [4].

Leveraging network programmability, developers have started offloading a number of different applications to programmable network devices – the so called "In-Network Computing" –, and even hardware devices not intended for network processing have been made programmable through P4. However, programmable networks do not provide adequate tools for developing network applications and do not offer enough compute capabilities for accelerating demanding tasks.

Although several research projects have proposed solutions for simplifying the development of applications with network hardware [5, 6, 7, 8, 9, 10, 11, 12, 13], they are unsuitable for the vast majority of developers, which lack hardware skills. Recent works have explored offloading demanding tasks to programmable network switches [14, 15]. However, these solutions are constrained by the (limited) resources provided by network hardware.

This dissertation leverages programmable network devices and hardware-software co-design for providing developers with the tools they need for programming functions in the network as they do with software applications. It also describes a new network compute hierarchy that enables the acceleration of a wider class of applications in the network. We exploit programmable network devices because they offer enough flexibility and visibility into networked applications and we employ hardware-software co-design for overcoming the limitations of both software-only and hardware-only solutions.

Evaluation shows promising results, thus demonstrating the effectiveness of the proposed approach. To understand the benefits and the limitations of network acceleration, we offload Paxos consensus protocol to programmable network devices. Our prototype provides orders of magnitude improvement over software implementations, running at 100Gbps on a programmable switch ASIC, with less than 0.1μs latency. For simplifying offloading applications to network hardware, we develop a framework for network functions on FPGA-based devices, that achieves a much lower latency than tools based on the match-action paradigm and provides a higher predictability compared to software network functions. To understand the effect of the network on applications, we develop a toolset for network monitoring and emulation. Our prototype provided us with interesting insights about the performance of network applications when the latency and throughput of the underlying network infrastructure changes. We propose a network compute hierarchy that leverages FPGA-based accelerators to allow offloading intensive applications to the network. Although being limited by currently available hardware, our prototype demonstrates the feasibility of our design and provides a pioneering approach for enabling line-rate acceleration of intensive tasks, with a minimal increment on latency. We propose an architecture that enables testing functions running in the network by providing visibility into network hardware devices. Our implementation, running at line-rate on both smartNICs and programmable switches, allowed us to uncover bugs within different programs and architectures, with a resource overhead ranging from 9% to 15%.

The solutions proposed in this thesis would greatly help the whole research community both in offloading more complex functions to the network, and in designing more capable programmable network technologies. Our work makes porting software applications to network hardware easy for software developers and provides unprecedented visibility into both network hardware devices and in applications running in the network. It also launches an effort to overcome the limitations of currently available programmable hardware technology, thus providing a direction for future research projects in the field.

Preface

The result of this research appears in the following publications:

Pietro Bressana, Noa Zilberman, Dejan Vucinic, Robert Soulé. 2020.
Trading Latency for Compute in the Network.
ACM SIGCOMM 2020 Workshop on Network Application Integration/CoDesign (NAI 2020).

Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, Robert Soulé. 2020.
P4xos: Consensus as a Network Service.
Transactions on Networking (ToN): 10.1109/TNET.2020.2992106.

Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, Robert Soulé. 2019.
Partitioned Partitioned Paxos via the Network Data Plane.
Technical Report Series in Informatics, USI-INF-TR-2019-01.
arXiv:1901.08806 [cs.DC].

Pietro Bressana, Noa Zilberman, Robert Soulé. 2018.
A programmable framework for validating data planes (Poster).
ACM Special Interest Group on Data Communication (SIGCOMM18).

Nik Sultana, Salvator Galea, David Greaves, Marcin Wójcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, Paolo Costa, Peter Pietzuch, Jon Crowcroft, Andrew W Moore, Noa Zilberman. 2017.
Emu: Rapid Prototyping of Networking Services.
USENIX Annual Technical Conference (ATC '17).

Acknowledgements

I would like to express my gratitude to all the people that supported me in various ways.

First, I would like to thank my advisor, prof. Robert Soulé, for giving me the opportunity to work on a number of innovative research projects, using cutting-edge technologies, in collaboration with the most brilliant researchers in our field. Working with Robert made me grow both as an engineer and as a researcher.

I am grateful to prof. Noa Zilberman, for making me part of her research group in Cambridge and for her precious support. Her academic experience, combined with her solid industrial background, was invaluable.

I would like to thank my committee members prof. Fernando Pedone, Dr. Dejan Vucinic and prof. Antonio Carzaniga for supporting me during my PhD career. Being the leader of the distributed systems group, Fernando coordinated our activities and has the merit of building an interdisciplinary and cohesive research team. Dejan advised me during my internship in his research group at Western Digital. Working in Dejan's group was very fruitful for me, as I could experience innovative industrial research in a positive and stimulating environment. I am grateful to Antonio for his availability. As the dean of our faculty, he advised me and supported me, when needed.

I am grateful to Dr. Alberto Ferrante, which put me in contact with Robert and Fernando and supported me in starting my career at USI.

I am thankful to the researchers of the faculty of informatics, especially to the members of the distributed systems group. They made my daily work in the open-space pleasant and stimulating.

I would like to thank both USI and the Swiss National Science Foundation (SNSF) for financing my PhD position and for providing me with all the necessary support.

Finally, I am grateful to my family and to my friends for being a reference point, even in difficult time.

Contents

Figures

Tables

Chapter 1

Introduction

The recent introduction of programmable network devices has revolutionised the way we design and use networked systems. Historically, the network has been treated as a passive communication channel, that interconnects nodes in distributed systems. With the advent of Software-Defined Networking (SDN), networks have become configurable, through the separation of the control plane from the (fixed) data plane. Network programmability moves this paradigm a step forward, by making the data plane fully programmable, thanks to the introduction of both programmable network architectures and network programming languages.

Programmable network architectures provide target-independent hardware abstractions that can be programmed through domain-specific languages. The common architecture for programmable network hardware, called Protocol Independent Switch Architecture (PISA) [2], consists of a Very Long Instruction Word (VLIW) machine, that is able to process huge amounts of network traffic. The core component of PISA is a pipeline of "match-action" stages, each computing a function, called "action", based on the outcome of a match over one or more header fields.

Network programming languages offer tools to developers for programming high-level applications to network hardware. Although a few network programming languages have been proposed, the most prominent language, P4 [1], has been widely adopted both in the industry and in the academia. P4 provides software abstractions to the components of PISA architecture and can be compiled to a number of different targets, both software and hardware, including programmable network switches [3] and smartNICs [4].

Leveraging programmable network architectures and network programming languages, developers started offloading their applications to the network, thus launching a new research area, called "In-Network Computing". Beyond classic network functions, developers have demonstrated accelerating distributed system applications [16] and even tasks not related to the network. They also designed compilers targeting hardware devices not intended for network processing, such as GPUs [17].

However, programmable networks are still in their infancy. They do not provide adequate tools for developing network applications and for monitoring and testing network functions running in the hardware. Moreover, currently available programmable architectures offer limited compute capabilities, thus preventing developers from offloading intensive tasks to the network.

Although several research projects proposed solutions for developing applications with network hardware, they either make strong assumptions about the underlying hardware [18], or support only a subset of the network applications [19, 1, 20], and they often leverage Hardware Description Languages (HDLs) [5, 6, 7, 8, 9, 10, 11, 12, 13], thus being unsuitable for the vast majority of developers, which lack hardware skills.

Researchers have proposed software testing tools that, unfortunately, are not enough for testing network applications running in the hardware [21]. In contrast, hardware-based network testing [22, 23, 24, 25, 26] provides scarce visibility to the internal architecture of programmable network devices, with limited performance. Other approaches have leveraged P4 language for extracting information from inside the network data plane [27, 28]. The main weakness of these approaches consists of mixing the testing function with the network application to be tested.

Network monitoring tools are often limited either to specific network topologies, or to a specific network protocol [29, 30], and do not provide processing in the hardware, without sending data to the hosts [31]. Moreover, they usually focus on network performance, rather than on the performance on the applications running in the network [32, 33, 34].

Recent research projects have proposed offloading intensive applications to programmable network switches [14, 15]. All these approaches suffer of the same limitations: they are both target-dependent and application-dependent, meaning that they are not suitable for offloading other applications to different hardware devices, and their performance are constrained by the capabilities of programmable network switches.

Based on the limitations of currently available solutions, our main objective is to simplify the process of offloading applications to the network, by providing developers with the tools they need and by enriching programmable network architectures with the capabilities they still miss.

1.1 This Dissertation

This thesis addresses the issue of providing developers with the tools they need for programming functions in the network as they do with software applications. We also aim to extend the capabilities of programmable network architectures, for enabling the acceleration of a wider class of applications. We use programmable network devices, since they offer enough flexibility and visibility into networked applications. We leverage hardware-software co-design for maximising the efficiency of our approach, by overcoming the limitations of both software and hardware solutions.

The hypothesis of this work is that we can simplify offloading applications to the network, by leveraging the capabilities of programmable network devices.

There are three components in this thesis that support our hypothesis.

First, this thesis explores In-Network Computing, for understanding its benefits and its limitations, and provides tools for simplifying offloading applications to the network. We accelerate Paxos consensus protocol in the network and demonstrate that our approach can significantly improve the performance of consensus. Based on the experience gained in offloading Paxos, we contribute to two research projects, that address the issues in programming network hardware devices and in monitoring application performance in the network. We design a framework for network functions on FPGAs that helps developers in rapidly offloading network logic to FPGA-based programmable network hardware and we provide a toolset designed for profiling the performance of networked-applications through monitoring and emulation. Our solutions permit applications to run on different targets, enabling better debug capabilities and an easier transition of workloads among targets and provide an insight into the properties of applications, as they run over the network.

Second, we propose a solution for enhancing the capabilities of programmable network devices, thus enabling the acceleration of demanding applications. We design a programmable network hierarchy that extends PISA with FPGA-based accelerators. We leverage the flexibility and the scalability offered by programmable network switches and the programmability provided by PISA. We use hardware devices for accelerating those compute-intensive and memory-intensive functions that PISA alone is unable to process. Our prototype architecture, implemented using a Barefoot Tofino switch and two NetFPGA SUME cards, demonstrates the feasibility of the proposed solution and the potential benefits it could provide to network computing applications.

Finally, this thesis discusses the issue of testing programmable network devices, since network applications are affected by bugs, both at the software level, and at the hardware level and currently available tools are inadequate for this purpose. We present a taxonomy of bugs that affect the network data plane and, based on the taxonomy, we introduce a portable test architecture that leverages both P4 language and hardware design. Our approach provides automatic testing of network applications, both at the software level, and at the hardware level, through the integration with formal language verification tools. Our prototype implementation, targeting a FPGA-based device and a network switch ASIC, allowed us to detect numerous hard-to-find bugs, thus addressing the urgent need for improved tools and techniques for data plane testing and verification.

To verify the approach discussed in this dissertation, we first experiment network acceleration. Then, we design a library for simplifying offloading to network hardware. Next, we provide a toolset for monitoring the performance of network applications and we extend the compute capabilities of programmable network architectures with hardware acceleration. Finally, we propose an architecture for testing functions running in the network. Details of our work are provided below.

To experiment network acceleration, we designed a system, called P4xos, that offloads Paxos consensus protocol [35] to network devices. As a baseline, we compared P4xos with a software-based implementation, the open-source `libpaxos` library [36]. Evaluation showed that offloading Paxos to the network provides orders of magnitude improvement over software implementations.

To simplify offloading applications to network hardware devices, we developed a framework for network functions on FPGA-based devices, called Emu. We compared network functions written in C# and compiled using Kiwi [37], with their equivalent Verilog implementation and with a P4 specification compiled through P4FPGA [38]. We also compared network services implemented using Emu with their original software implementations, running on a host computer. Experimental results showed that Emu provides a minimal resource overhead, compared to both native Verilog implementations and P4 specifications compiled with P4FPGA.

To understand the effect of the network on applications, we developed NRG, a Network Research Gadget that leverages the capabilities of programmable network hardware for enabling reproducible networking experimentation through network emulation and monitoring. Experimental results showed that the introduction of a static latency lowers the performance of the applications under test, thus indicating how critical it is to allocate machines physically close to each other when running latency sensitive workloads.

To allow offloading intensive applications to the network, we proposed an heterogeneous network computer hierarchy, that leverages FPGA-based accelerators. We tested a prototype architecture using a Barefoot Tofino switch and two NetFPGA SUME cards. Another NetFPGA SUME was programmed with OSNT, for generating test packets. We evaluated the prototype by accelerating a storage function in the network. Although being limited both by the performance of the NetFPGA SUME card and by unsupported features in Barefoot's products, our prototype demonstrated the feasibility of our approach. Once implemented on more recent hardware, our solution would provide line-rate acceleration of intensive tasks, with a minimal increment on latency, and will be as scalable as network switches are.

Finally, to enable testing functions running in the network, we implemented an architecture that provides visibility into network hardware devices. We tested our portable test architecture (PTA) on both Barefoot Tofino and NetFPGA SUME. Our implementations of PTA enabled us to uncover a number of bugs within different programs and architectures. Evaluation showed that both the implementations run at line-rate, since both are based on PISA, which ensures deterministically high performance. Although PTA introduces two new modules to a device, the resource overhead on NetFPGA SUME never exceeded 15%. On Tofino, PTA shared no resources with the data plane under test, since it tested a data-plane program on one pipeline using other pipelines. We reported PTA's runtime, both on the NetFPGA SUME and on Tofino.

1.2 Research Contributions

Overall, this thesis makes the following contributions:

1. It demonstrates improvements in applications' performance, through network offloading.

2. It identifies issues in accelerating functions using programmable network hardware devices.

3. It proposes a taxonomy of bugs affecting programmable network hardware devices.

4. It describes a set of abstractions that programmable network hardware must expose for enabling testing various types of bugs.

5. It implements a new heterogeneous network compute hierarchy, that extends the capabilities of programmable network devices with FPGA-based

accelerators, for enabling offloading demanding applications to the network.

6. It presents an architecture for testing programmable network hardware that integrates with formal verification tools.

7. It extends the capabilities of a framework for writing network services in a high-level language and have them automatically compiled to run across a number of platforms including FPGA-based devices.

8. It adds new P4-based functionalities to a toolset designed for profiling the performance of networked-applications, that can advise users and operators on best resource allocation and placement.

The rest of this dissertation is organised as follows. We present the background for this thesis (§2) and we cover the related work (§3). Then, we describe three research projects that experiment In-Network Computing and provide tools for simplifying offloading applications to the network (§4). Following that, we propose a compute hierarchy for In-Network Computing, that enables a new class of applications to be accelerated in the network (§5). Then, we show a taxonomy of bugs and an architecture for testing programmable network devices, called PTA (§6). Finally, we conclude the thesis by outlining our main contributions and by presenting directions for future research (§7).

Chapter 2

Background

This chapter provides the necessary background for this thesis. First, we present programmable Application Specific Integrated Circuits (ASICs), the technology at the base of programmable network switches, and discuss the Protocol Independent Switch Architecture (PISA), the most common hardware architecture for programmable network ASICs. In particular, we focus on Barefoot Tofino, a innovative programmable network ASIC, that we extensively used for implementing our prototypes. Then, we discuss smarNICs and we present FPGA devices, as an introduction to the NetFPGA, a FPGA-based platform that we widely used for validating our network hardware architectures. We provide an overview of the P4 programming language, as the leading language for programmable network devices based on PISA. Many functions in our prototypes have been implemented using P4. Finally we present Kiwi compiler, as a background for one of the research works described in this thesis.

2.1 Programmable ASICs

Programmable ASICs are the basic components of programmable network switches. Modern programmable ASICs overcome the historical trade-off between programmability and performance, thus being able to run user-designed network applications at the speed of fixed-functions network devices. Beyond running custom applications in the network, programmable ASICs enable advanced network monitoring and testing. Additionally, programmable ASICs can effectively implement the functionality of many different network devices in a single data plane configuration, thus simplify complex network topologies.

Programmable switch ASICs are optimized for packet processing at very high rate. To program these devices, developers use high-level, domain-specific pro-

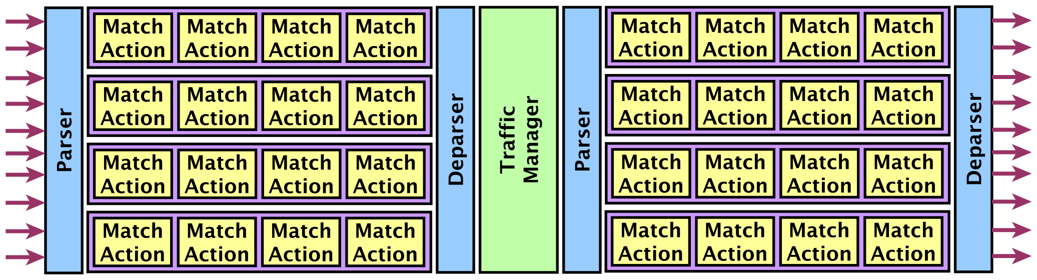


Figure 2.1. PISA architecture for programmable data planes.

gramming languages, in a way that is similar to software development. Compared to other hardware, ASICs provide the best performance and power efficiency, measured as number of network operations computed per watt [39]. On the other hand, meeting such strict performance requirements limits the flexibility of these devices and the amount of resources they offer. Programmable ASICs usually offer a small number of stages in their pipeline (i.e., tens of stages [3]), limited amount of memory per stage (in the form of network tables and SRAM banks), and simple functional blocks supporting a very limited instruction set.

Stateful operations and off-chip memory are limited in most programmable switches. Due to the fixed, vendor-specific architecture implemented by programmable network ASICs, developers cannot implement custom functions on a switch. However, they can leverage existing functions, as long as the silicon vendor has built those into the switch in advance.

2.1.1 PISA Architecture

The common hardware architecture for programmable high-performance packet-processing ASICs is often dubbed PISA [2], illustrated in Figure 2.1. In essence, PISA is a special kind of Very Long Instruction Word (VLIW) machine with a huge amount of I/O capacity and on-chip memory necessary to realize *match-action units* (i.e., SRAM or TCAM to match on packet header fields coupled with simple ALUs) and packet buffers, combined with generic packet parsing logic [40]. Unlike the conventional VLIW architecture used for CPUs, however, PISA does not have logic for heap, instruction memory, or stack pointers. Hence looping is not possible, and data dependencies between packets are resolved via a sequential pipeline of match-action units.

The amount of all physical resources on a PISA chip is determined and fixed at the chip design time and is in the order of tens of MB [2]. Important resource constraints include number of physical stages, number of match-action units in each stage, amount of on-chip memory per stage, memory for storing packet headers, and capabilities of match-action units and state-processing ALUs, etc.

CPUs can spend virtually as many resources (cycles and memory) as needed to process each packet. In contrast, PISA machines strictly limit the resources usage. This is necessary because a PISA machine must ensure deterministically high performance; the performance target of a PISA chip today is multi Tbps or billions of packets per second of throughput, and a sub-microsecond processing latency.

2.1.2 Barefoot Tofino

Barefoot Tofino [3] is a 6.5Tbps fully user-programmable network switch that implements a PISA architecture and can be programmed via the P4 programming language. The Tofino ASIC is available in different configurations, that can manage a variable number of network channels, with throughput ranging from 10Gbps to 100Gbps and sub-microsecond latency. Tofino-based programmable network switches implement a three-layer system, where the Tofino ASIC sits at the bottom layer, a Linux operating system above and a set of proprietary control plane applications run on top.

Use cases for Barefoot Tofino include rapidly prototyping and deploying new network protocols and implementing only the needed protocols, thus employing unused resource for increasing the capabilities of network switches. Developers can program monitoring features to Tofino switches either by reusing existing techniques, such as In-band Network Telemetry (INT) [27], or by implementing new functionalities, tailored to their applications. Tofino can integrate the functionality provided by network middleboxes, such as load-balancing, address translation and DDoS mitigation and can accelerate distributed applications in the network, including consensus protocols [41].

2.2 SmartNICs

SmartNICs are network interface cards with a programmable component. That component can be an ASIC, a network processor unit (NPU), a system-on-chip (SoC), or a field programmable gate array (FPGA). Some SmartNICs, but not all, implement a PISA-style architecture [42]. In this thesis, we focus on FPGAs, which has been prominently used by systems such as Catapult [11] to accelerate applications such as neural networks [43].

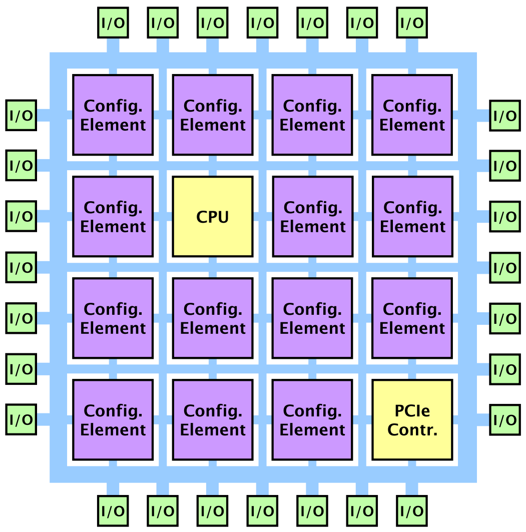


Figure 2.2. Schematic representation of a FPGA device.

2.2.1 FPGAs

FPGAs [44] are semiconductor devices that can be programmed and reprogrammed repeatedly, even after manufacturing. The internal architecture of a FPGA, shown in figure 2.2, is composed of a matrix of configurable elements, connected via a programmable interconnection network. Each configurable element includes a number of logic components, such as registers, multiplexers and function generators, coupled to a network of routing channels. The clock signal is propagated to all the components of the FPGA through a dedicated network. FPGAs can be interfaced with external components through a number of configurable input-output (I/O) pins. FPGAs may optionally include a library of configurable hardware blocks that implement widely used functions, such as PCIe interconnection.

FPGAs are similar to ASICs in the way both implement functionalities in the hardware, instead of running them in software, as CPUs do. However, generally speaking, FPGA devices are slower and less power-efficient than ASICs and they are more expensive. On the other hand, due to the intrinsic programmable nature of FPGA devices, developing new designs on FPGA is much cheaper than designing a new ASIC.

Although the most common way for developing FPGA functionalities consists in describing their hardware architecture through Verilog or VHDL language, alternative approaches, such as high-level synthesis (HLS), allow to translate a high-level functionality, e.g. a C program, into a low-level hardware specification.

2.2.2 NetFPGA Platform

NetFPGA [45] is an open-source platform for building high-performance networking systems in hardware. The aim of NetFPGA is to provide teachers, students and researchers with an affordable network development platform that is capable to process large volumes of traffic at the speed of bleeding-edge network hardware devices. The platform is composed of a FPGA-based card, a set of reference designs and a library of components.

The card is based on a large Xilinx FPGA and is equipped with additional hardware devices, including network ports, a PCIe interface and RAM memory. Based on its version, the card provides additional functions, such as interfacing with storage devices and with other cards and connecting expansion modules.

The reference designs implement a number of common network applications, leveraging the library of components. The library, coded in Verilog, provides controllers for the hardware components of the card, thus enabling connectivity, access to memory and interfacing with an host computer. Host interfacing is implemented through a Linux driver, included in the library, that provides read-write access to all the memory on the NetFPGA platform.

Since its introduction, the NetFPGA platform has been adopted by a number of academic and research institutions all over the world and an active community of developers has contributed the platform with several projects. Over the years, the NetFPGA platform has been enriched with a few different FPGA-based cards, each providing higher performances and newer hardware.

2.2.2.1 NetFPGA SUME

NetFPGA SUME [4] is the current incarnation of the NetFPGA platform, that provides a cost-effective open-source solution for implementing research projects in the field of datacenter networks, thus enabling the implementation of 100Gbps network applications, targeting bandwidth and throughput.

The NetFPGA SUME card is equipped with a large Xilinx Virtex 7 FPGA, four 10Gbps SFP+ network interfaces and a PCIe x8 Gen 3 connector. The FPGA is connected to two DDR3 RAM modules and three QDRII+ SRAM chips. An FMC connector provides ten 13Gbps serial links for interfacing with expansion modules. Additional NetFPGA SUME features support attaching MircoSD cards and external disks through Serial Advanced Technology Attachment (SATA) connectors.

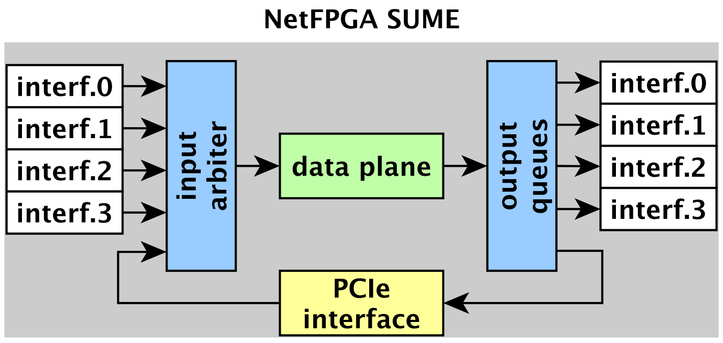


Figure 2.3. Block diagram of the NetFPGA SUME architecture.

The NetFPGA SUME architecture, shown in figure 2.3, provides controllers for both the network interfaces and the PCIe connector. All the traffic coming from both the network interfaces and the PCIe connector enters the input arbiter module, that assigns input packets to the internal data path, in a round-robin fashion. Traffic is then processed by the data plane application, taken from one of the NetFPGA SUME reference designs, or designed by the user. Processed packets are then sent to the output queues module, that forwards them either to the network interfaces, or to the host computer, through the PCIe interface.

The NetFPGA SUME platform can be configured to implement several different use cases. Using a soft-core processor, the card can operate as a stand-alone computing unit, for exploring reconfigurable systems and heterogeneous configurations. In a datacenter environment, the platform can implement a network management and measurement tool. Leveraging its PCIe connection, the NetFPGA SUME can implement a programmable Network Interface Card (NIC) and, thanks to the four network interfaces, it can also be used as a programmable switch. Using dedicated interfaces provided by the NetFPGA SUME, developers can explore novel complex architectures with line-rate performance. The NetFPGA SUME platform can be programmed using P4 language, through P4→NetFPGA [46], a workflow that allows developers to describe how packets are to be processed in the high-level P4 language and, then, compiles their P4 programs to run at line rate on the NetFPGA SUME board.

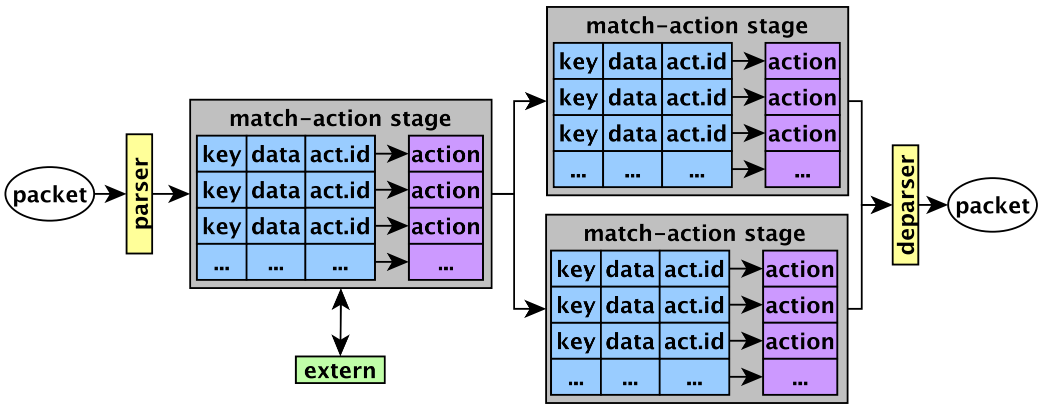


Figure 2.4. P4 abstract forwarding model.

2.3 P4 Language

P4 [1] is a domain-specific language for data plane programming. Its design is motivated by the need for customizable packet processing in network devices. Such customization could support both the evolving OpenFlow standard [47], and specialized data-center functionality, for example, to simplify network management or enable data-center specific packet encapsulations. The high-level abstractions of P4 directly follow from the PISA architecture design. A complete language specification is available online [48]. Here, we briefly highlight the main language features.

P4 language presents an abstract forwarding model, shown in figure 2.4 in which packets are processed by a pipeline of stages. Each stage is composed of a table, that matches packet header fields and performs an action, by providing optional parameters. Actions can forward, drop, or modify packets. P4 provides a specification for interfacing the pipeline with extern modules, that implement specific functions, such as checksum computation and access to stateful memory.

The P4 compiler is responsible for mapping the abstract representation onto a concrete realization in the particular target platform (e.g., SmartNICs, software switches, or reconfigurable hardware switches [2, 49]).

When writing a P4 program, developers use a few language abstractions.

Parsers. Parsers describe how to transform incoming packets to a parsed representation, from which header instances may be extracted. Parsers are specified as finite state machines that, at each state, extract headers from packets and, optionally, populate metadata fields. The execution of a finite state machine in a parser always start with a “start” state and ends either in a “accept” state, or in a “reject” state.

Deparsers. Deparsers transform the parsed representation, processed by the pipeline, back into the packet’s wire format.

Packet headers. Packet headers define collections of fixed-width fields. Headers are populated at the parsers, with data extracted from incoming packets. Header fields are then processed while traversing the pipeline of match-action stages. Header data is used by the deparsers for building outging packets.

Metadata. Metadata is a collection of fixed-width fields, used to pass information through different stages within the pipeline and as an interface to extern modules. The scope of the metadata is limited to the pipeline of match-action stages: it is processed in parallel to headers by the stages in between the parser and the deparser.

Tables. Tables specify which fields are examined from each packet, how those fields are matched, and actions performed as a consequence of the matching; Each table matches against a key and, based on the outcome of the match, triggers the execution of an action. A key is composed of one or more fields, taken either from the headers, or from the metadata, or from both.

Actions. Actions, which are invoked by tables, modify fields; add or remove headers; drop or forward packets; or perform stateful memory operations;

Control. Control blocks specify how stages are composed in the pipeline. If-else statements can be used for implementing alternative branches in the pipeline.

Registers. Registers provide persistent state organized into an array of cells. When declaring a register, developers specify the size of each cell, and the number of cells in the array.

Externs. Externs are roughly analogous to a foreign-function interface. Externs provide access to features that are not directly expressible in the P4 language. Such functions are implemented using either programming languages, such as C, or hardware description languages, such as Verilog, depending on the target platform.

2.4 Kiwi Compiler

Originally designed to support scientific computing applications, the Kiwi compiler [37] transforms the target language of .NET compilers (the Common Intermediate Language, or CIL) into a register-transfer level (RTL) description of

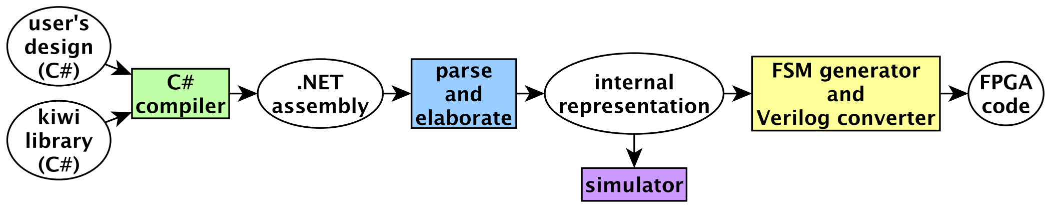


Figure 2.5. Kiwi's synthesis flow.

hardware in Verilog. The Verilog output can then be used to configure FPGAs. Kiwi aims to help in designing compute-intensive parallel applications and in deploying them to heterogeneous systems. Kiwi makes reconfigurable computing accessible to software engineers, by providing a unified programming flow for heterogeneous systems that extracts parallelism from software code and maps it to FPGA hardware.

Kiwi's synthesis flow, shown in figure 2.5, consists of a sequence of compilation steps that translate user's code to an FPGA hardware specification. Being designed for processing .NET programs, Kiwi takes a C# user's specification and provides a hardware-software library written in C#. The library provides two implementations: a software implementation supporting .NET concurrency mechanisms and a hardware semantics that is used for translating .NET code to a Verilog specification.

Kiwi's C# compiler translates user's program to a .NET assembly code that is parsed and elaborated for generating an internal representation, composed of both a sequence of assertions and pieces of imperative code. Users can simulate their code through a software simulation flow provided by Kiwi, before running their program on the FPGA.

Kiwi's FSM generator and converter translates the intermediate representation into a Verilog specification that can be programmed to a FPGA device. Parallel portions of the imperative code in the intermediate representation are mapped to parallel threads in the hardware. In order to provide a working hardware implementation, Kiwi's synthesis flow poses few constraints to resource allocation and function calling in user's code. This include knowing the size of variables and arrays at compile time and limiting recursive function calls to a run-time independent depth.

2.5 Chapter Summary

This chapter covered the necessary background for this thesis. We discussed programmable ASICs and the PISA architecture and we focused on Barefoot Tofino switch ASICs. We then introduced SmartNICs and FPGA devices, for discussing the NetFPGA platform. We also presented P4, a language for programming PISA-based network devices. Finally, we provided an overview of Kiwi compiler, as a background knowledge for one of the projects discussed in the fourth chapter of this thesis.

Chapter 3

Related Work

This chapter introduces research projects related to the work presented in this thesis and highlights the differences with our approach. We first review projects that optimise the performance of consensus protocols, including the ones exploring offloading consensus to network hardware. Second, we go over research works that focus on accelerating network applications with FPGAs, leveraging domain specific languages and high-level synthesis. Then, we describe projects for monitoring networking applications' performance, including hardware-based monitoring tools, with a focus on latency and reproducibility. We discuss solutions that leverage network hardware for accelerating demanding tasks. Finally, we present projects covering testing network devices using P4 language and formal verification and we discuss both testing fixed-function switches and network hardware in general.

3.1 Optimising Consensus' Performance

Consensus is a well-studied problem [35, 50, 51, 52]. Many have proposed consensus optimisations, including exploiting application semantics (e.g., EPaxos [53], Generalized Paxos [54], Generic Broadcast [55]), restricting the protocol (e.g., Zookeeper atomic broadcast [56]), or careful engineering (e.g., Gaios [57]).

Figure 3.1 compares several implementations of consensus along two axes. The y-axis plots the strength of the assumptions that a protocol makes about network behaviour (e.g., reliable delivery, ordered delivery). The x-axis plots the level of support that network devices need to provide (e.g., quality-of-service queues, support for adding sequence numbers, maintaining persistent state).

Lamport's basic Paxos protocol falls in the lower left quadrant, as it only assumes packet delivery in point-to-point fashion and election of a non-faulty

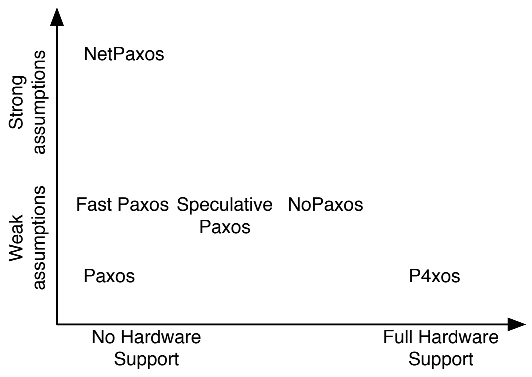


Figure 3.1. Design space for consensus/network interaction.

leader. It also requires no modification to network forwarding devices. Fast Paxos [58] optimises the protocol by optimistically assuming a spontaneous message ordering [58, 59, 60]. However, if that assumption is violated, Fast Paxos reverts to the basic Paxos protocol.

NetPaxos [61] does not require a specialized forwarding plane implementation. It assumes ordered delivery, without enforcing the assumption, which is likely unrealistic.

Speculative Paxos [62] and NoPaxos [63] uses programmable hardware to increase the likelihood of in-order delivery, and leverage that assumption to optimize consensus à la Fast Paxos [58]. In contrast, in this thesis, we make few assumptions about the network behavior, and use the programmable data plane to provide high-performance.

3.1.1 Accelerating Consensus with Network Hardware

Several recent projects have used network hardware to accelerate consensus. Notably, Consensus in a Box [9] and NetChain [16] accelerate Zookeeper atomic broadcast and Chain Replication, respectively. The work presented in this thesis differs from these approaches in that it separates the *execution* and *agreement* aspects of consensus, and focuses on accelerating only execution in the network. This separation of concerns allows the protocol to be optimized without tying it to a particular application. In other words, both Consensus in a Box and NetChain require that the application (i.e., the replicated key value store) also be implemented in the network hardware.

3.2 Accelerating Network Applications with FPGAs

As FPGAs are increasingly being deployed inside data centers and their performance is getting closer to specialised hardware [64], recently there has been a large body of work on how to offload critical network and application services to them [5, 6, 7, 8, 9, 10, 11, 12, 13]. All these proposals, however, leverage HDLs, thus making them unsuitable for the vast majority of developers, which lack hardware skills. This thesis addresses this issue by removing most of the challenges related to hardware programming and making FPGAs accessible to non-hardware experts too, while retaining high performance.

3.2.1 Domain-Specific-Languages

In the past there have been many efforts to make programming FPGAs easier, e.g., using a domain-specific language (DSL) [65, 66, 67, 68, 69], including network-specific ones [19, 1, 20]. These DSLs typically have a narrow scope and limit the performance or ability to implement certain network services. For example, P4 [1] is a popular DSL for packet processing that permits compilation to hardware including FPGAs. However, it is only applicable to tasks that can be processed by parse-match-action style systems. In contrast, the work presented in this thesis does not restrict the set of network services that can be implemented on top and offers a more familiar programming environment.

3.2.2 High-Level Synthesis Tools

High-Level Synthesis (HLS) tools [70] generate HDL from high-level languages such as Scala (using Chisel [71]), or Java (using Lime [72]) but they do not offer any specific support for network programming. One exception is the Maxeler's MPC-N system [18], which provides a "dataflow engine" to offload network computations into hardware. These engines run kernels that are programmed using a subset of Java, and tooling provided by Maxeler. This system, however, targets a proprietary hardware platform and lacks the ability of running seamlessly on both CPU and FPGAs. Conversely, our work makes very few assumptions about the underlying hardware and can be ported to different FPGA platforms.

3.3 Monitoring Networking Applications' Performance

Improving the performance of applications in the data center has been the focus of many works (e.g., [73, 74, 75, 76]), with significant focus on the contribution of the network to performance (e.g., [77, 78, 79, 11, 80]).

While previous work (e.g. [81, 82]) considered application performance with the latency between the user and the data center, we focus on latency within the data center, meaning latencies of orders of magnitude lower.

Yu *et al.* [29] have collected TCP statistics to profile network performance. A more recent work [30] collected TCP traces in the data center to devise a classification algorithm that identifies the root cause of failure. While our work can be used for such purposes, it is not limited to an end host nor to specific protocol. Furthermore, it ties directly to applications' performance metric.

3.3.1 Hardware-based Monitoring Tools

The goal of the work discussed in this thesis is research-focused rather than e.g., billing. This means that some loss of data is allowed for the sake of a longer observation period or efficient resource management. Consequently, solutions such as sketches [83] or counter sharing schemes (e.g. [84]) can, but are not inherently, be used. Contrary to other hardware-based tools [31], our approach does the processing in the hardware, without sending any packet to the host. This applies also to a wide range of capture solutions [85, 24, 86, 87].

3.3.2 Latency

Network latency has been identified long ago as a crucial factor for a good user experience [88]. Unlike bandwidth, latency is harder to control and improve [89, 90]. While our work focuses on the underlying traffic characteristics, previous work focused on latency variation as a result of queuing in switches, end-hosts, hypervisors or co-scheduling of CPU bound and latency-sensitive tasks [91, 92, 93, 94, 95]. Several works focused on providing network latency guarantees by controlling and reducing in-network queuing [32, 33, 34]. While these works show the impact of congestion on the performance, our work is extended to provide a methodology for exploring how other manifestations of network properties affect the application performance.

3.3.3 Reproducibility

Approaches to reproducibility range from simulation [96, 97], through emulation [98, 99, 100] and test beds to (relatively) small-scale full reproduction of a test environment. Well-known experimentation testbeds include PlanetLab [101], Emulab [102], Everlab [103], GENI [104], GpENI [105] and OFELIA [106]. Most of them are not typically useful for data center research. Grid5000 [107] is an experimental targeted at cloud research, but is limited in topologies and scale. DataMill [108] is part of the inspiration for our work and is an automated system for running a range of experiments on a distributed system and varying parameters in a controlled way. However, DataMill does not provide the ability to conduct experiments on networks.

3.4 Heterogeneous In-network Computing

The work presented in this thesis deals with programmable network switches and focuses on extending their capabilities with FPGA-based hardware accelerators. Since the recent introduction of programmable network switches, the academic community has provided a number of solutions for increasing the performance of applications that traditionally run on general purpose hardware, leveraging the capabilities of the new devices. However, little effort has been put so far in extending programmable network architectures with FPGAs, for enriching programmable switches with the processing power required by the most demanding applications.

In the following, we present previous works that combine network switches with accelerators and we discuss projects that leverage either programmable network switches, or FPGAs, for In-Network Computing.

Switch ASICs with FPGAs. Switch boxes that encompass switch-ASIC and FPGA have been a common practice in networking for a long time. In most cases, the FPGA is used for board control and management, and in some cases as a co-processor. The latter is used for offloading statistics collection, e.g., when the FPGA can connect to off-chip memory and help poll statics less often. However, these FPGA devices are used as termination point, and not as part of the processing pipeline.

Switch ASICs with other ASICs. Switch-ASIC also sometimes connect to other ASIC devices [109], used to extend on-chip tables. These devices are the evolution of off-chip TCAM, and do not offer the same level of flexibility as FPGA.

In-network Computing with Switch ASICs. The last few years we have witnessed the rise of In-Network Computing usage for caching [110], query processing [14], consensus [111], storage [15], and many other application. However, all these examples were limited by on-chip switch resources, or traded performance for functionality (e.g., through recirculation). Kim *et al.* [112] combined switch ASIC and remote memory on RDMA NIC as a means to overcome switch memory limitation. Our solution attends also to functionality limitations, and as there is no network traversal, offers the prospect of lower latency and lower network load.

In-network Computing with FPGAs. Using FPGA as the main computing component for in-network computation has attracted further research interest [9, 113, 114], but an FPGA device performance is 1-2 orders of magnitude less than a switch ASIC [39].

FPGA-based Fingerprint Computation. Our work has been partially inspired by Li *et al.* [115], that provided FPGA-based fingerprint computation for high-throughput data storages. However, our contribution focuses on the concept of network computing hierarchy, and the challenges of integrating ASIC and FPGA.

3.5 Testing Network Devices

Testing and validating network hardware is a well studied area. A range of software-based approaches have been proposed, including simulation or emulation. However, none of these offer a comprehensive solution, since simulators or emulators may not faithfully model actual deployments and cannot test scale-related bugs. In this thesis, we propose testing data planes using programmable network devices. Related work in this field includes testing using P4 programming language, formal verification of network data planes, testing fixed-function switches and, more in general, testing network hardware devices. Below, we briefly summarize the state-of-the-art in these areas.

3.5.1 Network Testing using P4

There are at least two prior efforts to use P4 and programmable network hardware for network testing. One approach, called in-band network telemetry (INT) [27] attaches meta-data to every packet as it flows through the network. This meta-data can be collected from the last-hop switch for analysis. The second approach, called postcards [28], also collects meta-data to every packet, but rather than modifying packets, each switch immediately sends the data off the device.

The work in this thesis differs from these prior efforts in several ways. First, both INT and postcards are reactive, in the sense that the information is received after the arrival of a packet. In contrast, our approach is *active*, since it generates specific packets to facilitate ad-hoc and exploratory analysis. Second, the functionality for INT and postcards are both embedded in the forwarding plane of the network device. If there is a problem with the forwarding plane, then INT and postcards may not be able to provide useful information. Our approach is deployed in parallel to the standard P4 pipeline. Finally, unlike the INT and postcard approaches, the design specified in this thesis has visibility to the internal part of the device that is different from what is available from externally generated packets.

3.5.2 Data Plane Verification

There are several recent projects on P4 program verification, including Vera [116], P4v [117], and Assert-P4 [118]. The details of the approaches differ, but essentially, all of these systems translate P4 programs and some control plane state into a logical formula, and use techniques such as symbolic execution [119] to check that correctness properties are not violated (e.g., a header field in a packet is not accessed if it has not been parsed). Our work complements these efforts by providing runtime testing.

For all P4 verification efforts, the major problem that needs to be addressed is how to cope with the lack of control plane state. In other words, P4 provides half a program, which makes it hard to check correctness. Vera [116] handles this problem by taking a snapshot of table entries, which results in limited coverage. In contrast, P4v and Assert-P4 both require that programmers provide annotations about assumptions on the control plane state, which increases coverage at the expense of the annotation burden. We believe the work presented in this thesis is complementary to verification, as it is used to catch a different class of bugs. Our work provides grey box testing, as often only partial information exists on the data plane under test. Grey box testing has been further motivated by prior works on router and network level, such as NetSonar [120].

3.5.3 Testing Fixed-function Switches

There has been significant prior work in both industry and academia on testing fixed-function switches. The most similar to our approach is the service activation test (SAT) [121]. SAT, used by carrier Ethernet service providers, is intended to ensure that network services are configured as specified and meet the predefined Service Acceptance Criteria (SAC). SAT uses packet injection as a mean to test the service, but it is closely defined and not programmable, covering only a limited set of the aspects enabled by our work. More low-level approaches, such as ATPG [122], focus on manufacturing faults rather than bugs. FPGA debug tools, such as ILA [123], enable limited functionality testing, far less than the tests described in this thesis, and are timing-affecting.

3.5.4 Hardware-Based Network Testing

Network operators often augment software-based techniques with hardware-based testing. For this purpose, equipment vendors such Ixia [22] and Spirent [23] sell highly-specialised platforms, which can generate and receive traffic at line rate. Unfortunately, the cost of these platforms is considerable, and therefore prohibitive. Moreover, they provide limited visibility because they function as external black-box tester. Research efforts, such as [24, 25, 26] offer lower cost solutions with similar intents, but they are limited, in terms of features, scale, and performance, e.g., OSNT [24] does not scale beyond 4 ports, and none of the proposals works with non-Ethernet packets.

3.6 Chapter Summary

This chapter introduced research projects related to this thesis. We presented works that optimise the performance of consensus protocols and projects that offload network applications to FPGA-based devices. Then, we discussed solutions covering monitoring network applications' performance. Finally, we reviewed research projects in the field of network hardware acceleration and present works covering network testing.

Chapter 4

Tools and Applications for In-Network Computing

In-Network Computing (INC) [111, 110, 124] is a new research area that implements computation in the network, leveraging programmable network technologies, such as programmable switch ASICs, smartNICs and network programming languages. Processing data as it traverses the network, as proposed by In-Network Computing, reduces network load, at a cost of a minimal overhead, since this requires no additional hardware resource for being implemented. In-Network Computing provides benefits to a number of use cases, ranging from classical network functions, to applications not traditionally related to networking and greatly improves the performance of network architectures, both latency and throughput, thanks to programmable network hardware. However, In-Network Computing poses a number of challenges, including abstracting hardware functions to software developers, extending the capabilities of available network hardware and providing tools and techniques for developing, testing and monitoring new networked applications.

This chapter presents three research projects that experiment accelerating applications in the network and provide tools for addressing the challenges of In-Network Computing. The first project explores offloading an application – an implementation of Paxos consensus protocol – to the network. Based on the experience gained in offloading Paxos to the network, we propose Emu, a framework that makes it simpler for software developers to accelerate their program using programmable network hardware and NRG, a toolset that leverages programmable network hardware for enabling network emulation and monitoring. For each project, we provide a brief description and discuss our main contributions.

4.1 In-Network Computing

In-Network Computing is a new research field focused on implementing computation in the network, through hardware devices that have been traditionally used for forwarding network traffic. In-Network Computing introduces only a minimal overhead on networked systems, since it requires no additional hardware resource for being implemented. Moreover, processing data as it traverses the network, as proposed by In-Network Computing, reduces the network load by terminating transactions in the network, instead of a the end hosts. In-Network Computing has been enabled by the introduction of programmable network hardware, both programmable network switch ASICs and smartNICs, and network programming languages, such as P4.

Use cases for In-Network Computing range from classical network functions, such as DNS servers [125] and load balancing [126], to applications not traditionally related to networking, such industrial automation and control [127]. Researchers have already demonstrated successful offloading of many applications, including caching [110], data reduction and aggregation [128] and query processing [14]. One of the projects presented in this chapter deals with implementing consensus in the network, an additional application of In-Network Computing.

Besides being suitable for a large number of use cases, In-Network Computing greatly improves the performance of network architectures, thanks to programmable network hardware. Currently available programmable switch ASICs provide sub-microsecond latency and tens of Terabit per second (Tbps) throughput, thus reducing the overhead of moving data in the network, while processing billions of operations per second. These performance benefits have a negligible impact over power consumption, since network hardware is already consuming power for forwarding traffic and the overhead introduced by In-Network Computing is small.

However, In-Network Computing poses a number of challenges to the research community. Network programmability requires abstracting network hardware resources for software programmers and even enabling network virtualization, for running many different applications over the same network hardware. Another challenge is porting designs in between network hardware targets and extending programmable network architectures with new hardware functionalities. Finally, the deployment of complex programmable network systems requires new tools and techniques for testing, verification, monitoring and emulation. The projects discussed in this chapter, and in the rest of this dissertation, address some of these challenges, for simplifying offloading applications to the network.

4.2 Research Projects

This section presents three research projects that explore the capabilities of In-Network Computing and provide innovative tools for simplifying the development of new applications in this field.

Although we were not the leaders of these projects, we actively collaborated to these works, through specific contributions. In the following, we provide a description of each project and we highlight our main contributions.

First, we worked on P4xos [41], a project that accelerates a consensus protocol using programmable network hardware. P4xos has already been described in a previous PhD thesis [129], defended at Università della Svizzera italiana in 2019. This work demonstrates in-network hardware acceleration of programs traditionally implemented in software. We contributed to the hardware acceleration of P4xos, mainly to its FPGA-based implementation.

Leveraging the experience gained by implementing P4xos, we collaborated to Emu [125], a framework for network functions on FPGAs, and we contributed to NRG, a project that leverages the capabilities of programmable network hardware for enabling reproducible networking experimentation through network emulation and monitoring. Our work focused on providing Emu with the FPGA hardware interfaces it needs for abstracting access to storage resources, and in implementing part of NRG's functionality on programmable network hardware, using P4 language.

4.2.1 P4xos

P4xos is an implementation of Paxos consensus protocol in the network, that runs directly on switch ASICs. We focused on Paxos [35] for two reasons. First, it makes very few assumptions about the network, making it widely applicable to a number of deployment scenarios. Second, it is deployed in numerous real-world systems, including Microsoft Azure Storage [130], Ceph [131], and Chubby [132]. P4xos improves latency by processing consensus messages in the forwarding plane as they pass through the network, reducing the number of hops, both through the network and through hosts, that messages must travel. Moreover, P4xos improves throughput, as ASICs are designed for high-performance message processing. We contributed in designing P4xos' programmable network hardware architecture, mainly targeting the NetFPGA SUME-based framework.

In the following, we discuss both the program flow and the P4 code that implement our FPGA-based design.

4.2.1.1 Paxos Consensus Protocol

Paxos consensus protocol [35] is used to solve a fundamental problem for distributed systems: getting a group of participants to reliably agree on some value (e.g., the next valid application state). Paxos distinguishes the following roles that a process can play: *proposers*, *acceptors* and *learners* (leaders are introduced later). Clients of a replicated service are typically proposers, and propose commands that need to be ordered by Paxos before they are learned and executed by the replicated state machines. Replicas typically play the roles of acceptors (i.e., the processes that actually agree on a value) and learners. Paxos is resilience-optimum in the sense that it tolerates the failure of up to f acceptors from a total of $2f+1$ acceptors, where a quorum of $f+1$ acceptors must be non-faulty [133]. In practice, replicated services run multiple executions of the Paxos protocol to achieve consensus on a sequence of values [134] (i.e., multi-Paxos). An execution of Paxos is called an instance.

An instance of Paxos proceeds in two phases. During Phase 1, a proposer that wants to submit a value selects a unique round number and sends a prepare request to at least a quorum of acceptors. Upon receiving a prepare request with a round number bigger than any previously received round number, the acceptor responds to the proposer promising that it will reject any future requests with smaller round numbers. If the acceptor already accepted a request for the current instance, it will return the accepted value to the proposer, together with the round number received when the request was accepted. When the proposer receives answers from a quorum of acceptors, the second phase begins.

In Phase 2, shown in figure 4.1, the proposer selects a value according to the following rule. If no value is returned in the responses, the proposer can select a new value for the instance; however, if any of the acceptors returned a value in the first phase, the proposer must select the value with the highest round number among the responses. The proposer then sends an accept request with the round number used in the first phase and the value selected to the same quorum of acceptors. When receiving such a request, the acceptors acknowledge it by sending the accepted value to the learners, unless the acceptors have already acknowledged another request with a higher round number. When a quorum of acceptors accepts a value, consensus is reached.

If multiple proposers simultaneously execute the procedure above for the same instance, then no proposer may be able to execute the two phases of the

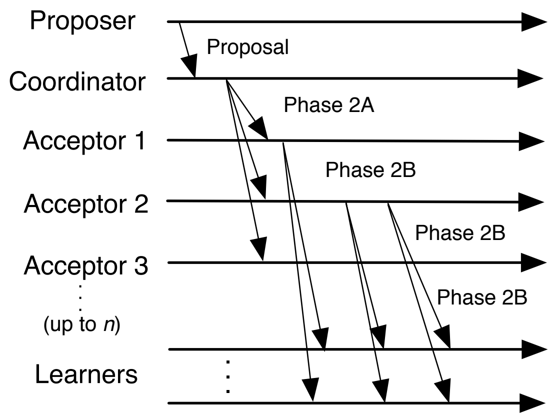


Figure 4.1. The Paxos Phase 2 communication pattern.

protocol and reach consensus. To avoid scenarios in which proposers compete indefinitely, a *leader* process can be elected. Proposers submit values to the leader, which executes the first and second phases of the protocol. If the leader fails, another process takes over its role. Paxos ensures (i) consistency despite concurrent leaders and (ii) progress in the presence of a single leader.

4.2.1.2 Paxos Performance Bottlenecks

Given the central role that Paxos plays in fault-tolerant, distributed systems, improving the performance of the protocol has been an intense area of study. From a high-level, there are performance obstacles that impact both latency and throughput.

Protocol Latency. The performance of Paxos is typically measured in "communication steps", where a communication step corresponds to a server-to-server communication in an abstract distributed system. Lamport proved that it takes at least 3 steps to order messages in a distributed setting [133]. This means that there is not much hope for significant performance improvements, unless one revisits the model (e.g., Charron-Bost and Schiper [135]) or assumptions (e.g., *spontaneous message ordering* [58, 59, 60]). These communication steps have become the dominant factor for Paxos latency overhead. Implementing Paxos in switch ASICs as "bumps-in-the-wire" processing allows consensus to be reached in sub-round-trip time (RTT).

Throughput Bottlenecks. Based on a test we have done using `libpaxos` library [36], we discovered the leader is the performance bottleneck in typical Paxos deployments, as it becomes CPU bound. This is as expected. The leader must process the most messages of any role in the protocol, and as a result, becomes the first bottleneck.

4.2.1.3 P4xos Design

P4xos is designed to address the two main obstacles for achieving high-performance: (i) it reduces end-to-end latency by executing consensus logic as messages pass through the network, and (ii) it avoids network I/O bottlenecks in software implementations by executing Paxos logic in the forwarding hardware. In a network implementation of Paxos, protocol messages are encoded in a custom packet header; the data plane executes the logic of *leaders*, *acceptors*, and *learners*.

Paxos Header. The header is encapsulated by a UDP packet header, allowing P4xos packets to co-exist with standard (non-programmable) network hardware. Moreover, we use the UDP checksum to ensure data integrity. Since current network hardware lacks the ability to generate packets, P4xos participants must respond to input messages by rewriting fields in the packet header. The P4xos packet header includes six fields. To keep the header small, the semantics of some of the fields change depending on which participant sends the message. The fields are as follows: (*i*) `msgtype` distinguishes the various Paxos messages (e.g., `REQUEST`, `PHASE1A`, `PHASE2A`, etc.) (*ii*) `inst` is the consensus instance number; (*iii*) `rnd` is either the round number computed by the proposer or the round number for which the acceptor has cast a vote; `vrnd` is the round number in which an acceptor has cast a vote; (*iv*) `swid` identifies the sender of the message; and (*v*) `value` contains the request from the proposer or the value for which an acceptor has cast a vote.

Proposer. A P4xos proposer mediates client requests, and encapsulates the request in a Paxos header. Ideally, this logic could be implemented by an operating system kernel network stack, allowing it to add Paxos headers in the same way that transport protocol headers are added today. As a proof-of-concept, we have implemented the proposer as a user-space library that exposes a small API to client applications.

Leader. A leader brokers requests on behalf of proposers. The leader ensures that only one process submits a message to the protocol for a particular instance (thus ensuring that the protocol terminates), and imposes an ordering of messages. When there is a single leader, a monotonically increasing sequence number can be used to order the messages. This sequence number is written to the `inst` field of the header. The leader receives `REQUEST` messages from the proposer. `REQUEST` messages only contain a value. The leader must perform the following: write the current instance number and an initial round number into the message header; increment the instance number for the next invocation; store the value of the new instance number; and broadcast the packet to acceptors.

Acceptor. Acceptors are responsible for choosing a single value for a particular instance. For each instance of consensus, each individual acceptor must "vote" for a value. Acceptors must maintain and access the history of proposals for which they have voted. This history ensures that only one value can be decided for a particular instance, and allows the protocol to tolerate lost or duplicate messages. Acceptors can receive either `PHASE1A` or `PHASE2A` messages. Phase 1A messages are used during initialization, and Phase 2A messages trigger a vote.

Learner. Learners are responsible for replicating a value for a given consensus instance. Learners receive votes from the acceptors, and "deliver" a value if a majority of votes are the same (i.e., there is a quorum). Learners should only receive `PHASE2B` messages. When a message arrives, each learner extracts the instance number, switch id, and value. The learner maintains a mapping from a pair of instance number and switch id to a value. Each time a new value arrives, the learner checks for a majority-quorum of acceptor votes. A majority is equal to $f + 1$ where f is the number of faulty acceptors that can be tolerated.

4.2.1.4 FPGA Implementation

We have implemented a prototype of P4xos in P4 [1] and compiled it using P4FPGA [38]. We have also written C implementations of the leader, acceptor, and learner using DPDK. The DPDK and P4 versions of the code are interchangeable, allowing, for example, a P4-based hardware deployment of a leader to use a DPDK implementation as a backup.

In this section we discuss our FPGA-based implementation. We present the program flow of P4xos and we explain each functionality provided in the program by commenting a few portions of its P4 code. As discussed in the following sections, we implemented P4xos using four FPGA-based cards, each running either the leader, or the acceptor functionality.

4.2.1.4.1 Program Flow

Figure 4.2 shows the program flow of the P4FPGA implementation, that includes both leader and acceptor roles. For simplicity, we do not provide a detailed description of the hardware logic that implements network interfacing and packet

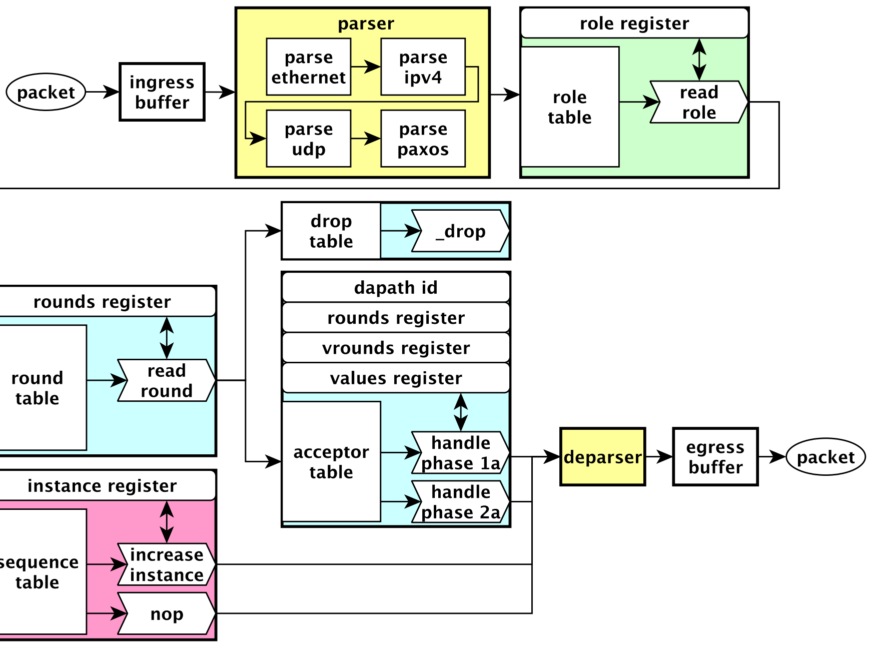


Figure 4.2. Program flow of P4FPGA implementation.

buffering and we omit the L2 forwarding functionality provided by the design. Instead, we focus on the logic implementing P4xos.

As shown in the figure, the architecture includes the hardware components that are necessary for interfacing Paxos' logic with the network and for buffering both incoming and outgoing packets. Parsing/deparsing logic, coloured yellow, process incoming/outgoing traffic and separate the headers used by P4xos from packet's payload. Our implementation works with standard UDP packets with the addition of a P4xos-specific Paxos header, trasmitted as UDP's payload. Packet's payload, not shown in the figure, is not processed by our design. Instead, it is temporarily stored into a shared buffer memory and attached to the processed headers at the deparsing stage, for generating an outgoing packet.

Before executing P4xos' logic, the program runs an self-configuring operation, for selecting which type of instance to execute, either the leader, or the acceptor. This operation is performed through a dedicated "role" stage, coloured green, that reads the configuration from a register set by the user. Once configured, the program implements either one of the two roles. In our prototype, one of the four FPGA-based cards is configured as the leader, the others implement the acceptor functionality. An alternative configuration consists in separating the code that implements each of the two roles, thus removing the initial configuration phase.

We chose to implement both the roles in the same program for making the system more flexible, through configuration. However, the impact of our choice on the FPGA resources is limited, since both the leader and the acceptor share many of their functionalities, including parsing, deparsing and L2 forwarding.

In case the "acceptor" role has been chosen, the program flow traverses the two stages coloured light-blue in the figure. The first stage retrieves the previous round number from a register associated to the instance carried by the Paxos header. Then, if the current round number, carried by Paxos header, is less than the previous round number, the packet is dropped, since the round associated with it has already been processed. Otherwise, the program runs the acceptor stage and checks the message type carried by the Paxos header. If the message is of type "PHASE1A", the acceptor stage creates a message of type "PHASE1B", by setting the fields of the Paxos header with the values retrieved from its registers. Similarly, if the message is of type "PHASE2A", the acceptor generates a message of type "PHASE2B" and stores the content of the Paxos header into its registers. In both cases, the unique ID of the acceptor is written to the header. If the message type is not recognised, the packet is dropped.

In case the "leader" role has been chosen, the program flow goes through the stage coloured purple in the figure, where the type of the message carried by the Paxos header is matched against the sequence table. If the message is of type "REQUEST", the leader retrieves the instance number stored into its register, increments it, and writes the incremented value both to the Paxos header and to the register. Otherwise, it performs no action and forwards the packet to the next stage.

Finally, processed UDP/Paxos headers reach the deparsing stage, where they are composed with packet's payload, for generating an outgoing packet.

4.2.1.4.2 P4 Code

Similarly to the program flow, our P4 code implements the basic P4xos functionalities, the acceptor and the leader in three separated sections, that we discuss in details in the following. We omit both the parsing/deparsing stages and L2 switching.

The common subset of P4xos' functionality, implementing the selection of the role, is shown in figure 4.3. The initial lines, (1 - 2) and (4 - 5), include the portions of the code implementing the acceptor and the leader and define convenient tags for selecting either one of the two roles. Line 9 instantiates the role register module, by providing a few configuration parameters, omitted in the figure, such as the register width and the number of registers provided

```
1  #include "leader.p4"
2  #include "acceptor.p4"
3
4  #define IS_LEADER 1
5  #define IS_ACCEPTOR 2
6
7  // Omitting parser ...
8
9  register role_register { ... }
10
11 action read_role() {
12    register_read(switch_metadata.role,
          role_register, 0);
13 }
14
15 table role_tbl {
16    actions { read_role; }
17 }
18
19 control paxos{
20
21    // Omitting L2 switching ...
22
23    if (valid(paxos)) {
24
25       apply(role_tbl);
26
27       if (switch_metadata.role == IS_LEADER) {
28
29          // run leader's functions
30
31       } else {
32          if (switch_metadata.role == IS_ACCEPTOR
                ) {
33
34             // run leader's functions
35
36          }
37       }
38    }
39 }
```

Figure 4.3. P4 code implementing Paxos.

by the module. Our code implements a single register, that is written by the user for setting-up the initial configuration of the system. The role register is read by a dedicated action, defined at lines (11 - 13), that stores its value to a pre-defined metadata field. The action is triggered by the role table, lines (15 - 17), that matches against any header that traverses the pipeline. Besides providing basic L2 switching (omitted), the control block at lines (19 - 39) checks if there is a valid Paxos header among the incoming parsed headers (line 23), before implementing the selection of the role. In case a valid Paxos header is not found, the headers are forwarded to the output network interfaces, without being processed. After retrieving the role configuration, at line 25, the program reads the configuration information and runs either the leader (line 27), or the acceptor (line 32) code.

Figure 4.4 shows the implementation of the acceptor. Lines (1 - 4) instantiate the register modules used for storing the state of the protocol, including the round numbers, the value and an ID specific to the acceptor instance. Round register is read by a dedicated action at lines (6 - 8), that stores its value to a metadata field. The action is executed by the table at lines (10 - 12), that matches against any incoming header. Lines (14 - 17) instantiate a table for dropping every matching packet, through a primitive P4 action called "_drop". The table at lines (35 - 38) provides the main functionality of the acceptor, by running either the first, or the second phase of the protocol, based on the type of message carried by the matched Paxos header. Each of the two phases of the

```
register datapath_id { ... }
register rounds_register { ... }
register vrounds_register { ... }
register values_register { ... }

action read_round() {
   register_read(paxos_packet_metadata.round,
        rounds_register, paxos.instance);
}

table round_tbl {
   actions { read_round; }
}

table drop_tbl {
   actions { _drop; }
   size : 1;
}

action handle_1a() {
   modify_field(paxos.msgtype, PAXOS_1B);
   register_read(paxos.vround, vrounds_register,
         paxos.instance);
   register_read(paxos.value, values_register,
        paxos.instance);
   register_read(paxos.acceptor, datapath_id, 0)
        ;
   register_write(rounds_register, paxos.
        instance, paxos.round);
}

action handle_2a() {
   modify_field(paxos.msgtype, PAXOS_2B);
   register_write(rounds_register, paxos.
        instance, paxos.round);
   register_write(vrounds_register, paxos.
        instance, paxos.round);
   register_write(values_register, paxos.
        instance, paxos.value);
   register_read(paxos.acceptor, datapath_id, 0)
        ;
}

table acceptor_tbl {
   reads { paxos.msgtype : exact; }
   actions { handle_1a; handle_2a; _drop; }
}

control acceptor {

   apply(round_tbl);

   if (paxos_packet_metadata.round <= paxos.
        round) {

      apply(acceptor_tbl);

    } else {

      apply(drop_tbl);

    }
}
```

Figure 4.4. P4 code implementing the acceptor.

```
register instance_register { ... }

action increase_instance() {
   register_read(paxos.instance, instance_register, 0);
   add_to_field(paxos.instance, 1);
   register_write(instance_register, 0, paxos.instance);
}

table sequence_tbl {
   reads { paxos.msgtype : exact; }
   actions { increase_instance; _nop; }
}

control leader {

   apply(sequence_tbl);

}
```

Figure 4.5. P4 code implementing the leader.

protocol are implemented by a dedicated action, lines (19 - 25) and lines (27 - 33), that sets both the ID of the acceptor and the type of message into the Paxos header. The action that implements PHASE1A updates the header with the current status of the acceptor. In contrast, the action implementing PHASE2A overwrites the status of the acceptor with the content of the header. The control block at lines (40 - 53) reads the round number from the register (line 42) and compares it with the value carried by Paxos' header. In case the round number of the header is greater than the one stored in the register, the program runs the acceptor table (line 46), thus executing either PHASE1A, or PHASE2A, based on the outcome of the match. Otherwise, the packet is dropped by applying the drop table (line 50).

The code implementing the leader is shown in figure 4.5. Since the only functionality of the leader consists in assigning an increasing instance number to each processed request, the code implements a single table, coupled to a register. The register at line 1 stores the last instance number assigned by the leader. It is accessed twice by the action at lines (3 - 7): the first access retrieves the value from the register, the second overvrites the value stored in the register with the new instance number. In between the two accesses, the action increments the instance number and stores it to the Paxos header. The action is triggered by the sequence table at lines (9 - 12), that matches against Paxos request messages. The control block (lines 14 - 18) implements only the sequence table.

4.2.1.5 Evaluation

Our evaluation of P4xos explores both the absolute performance of individual P4xos components and the end-to-end performance of P4xos. As a baseline, we compare P4xos with the software-based `libpaxos` library [36].

4.2.1.5.1 Absolute Performance

The first set of experiments evaluate the performance of individual P4xos components deployed on a programmable ASIC, an FPGA, DPDK and typical software processes on x86 CPUs.

For DPDK and FPGA targets, we used a hardware packet generator and capturer to send 102-byte consensus messages to each component, then captured and timestamped each message measuring maximum receiving rate. For Tofino, we used one 64-port switch configured to 40G per port. To generate traffic, we used a $2 \times 40Gb$ Ixia XGS12-H as packet sender and receiver, connected to the switch with 40G QSFP+ direct-attached copper cables.

Role	DPDK	P4xos (NetFPGA)	P4xos (Tofino)
Forwarding	n/a	0.370 μs	n/a
Leader	2.3 μs	0.520 μs	less than 0.1 μs
Acceptor	2.6 μs	0.550 μs	less than 0.1 μs
Learner	2.8 μs	0.540 μs	less than 0.1 μs

Table 4.1. P4xos latency.

To quantify the processing overhead added by executing Paxos logic, we measured the pipeline latency of forwarding with and without Paxos. Table 4.1 shows the latency for DPDK, P4xos running on NetFPGA SUME and on Tofino. The first row shows the results for forwarding without Paxos logic. The latency was measured from the beginning of the packet parser until the end of the packet deparser. Overall, the experiments show that P4xos adds little latency beyond simply forwarding packets, around 0.15 μs (38 clock cycles) on FPGA and less than 0.1 μs on ASIC. These experiments show that moving Paxos into the forwarding plane can substantially improve performance.

We measured the throughput for all Paxos roles on different hardware targets. The results show that on the FPGA, the acceptor, leader, and learner can all process close to 10 million consensus messages per second, an order of magnitude improvement over `libpaxos`, and almost double the best DPDK throughput. The ASIC deployment allows two additional order of magnitude improvement, processing over 2.5 billion consensus msgs/sec.

4.2.1.5.2 End-to-End Performance

We used two different network topologies for these experiments, shown in figure 4.6. In the `libpaxos`, FPGA, and DPDK experiments, all links are configured at 10GbE. The leader and acceptors where software processes for both the `libpaxos` experiments and the DPDK experiments. For the FPGA experiments, the leader and acceptors were NetFPGA SUME boards. The learners were the DPDK implementation for all the experiments. For experiments with Tofino, the leader and acceptors were Barefoot Tofino switches. The latency is measured at the client as the round-trip time for each message. Throughput is measured at the learner as the number of deliver invocations over time.

We measure the latency and predictability for P4xos as a system. We see that P4xos shows lower latency and exhibits better predictability than `libpaxos`: The median latencies are 22, 42 and 67 μs for Tofino, SUME and DPDK respectively, compared with 119 μs, and the difference between 25% and 75% quantiles is

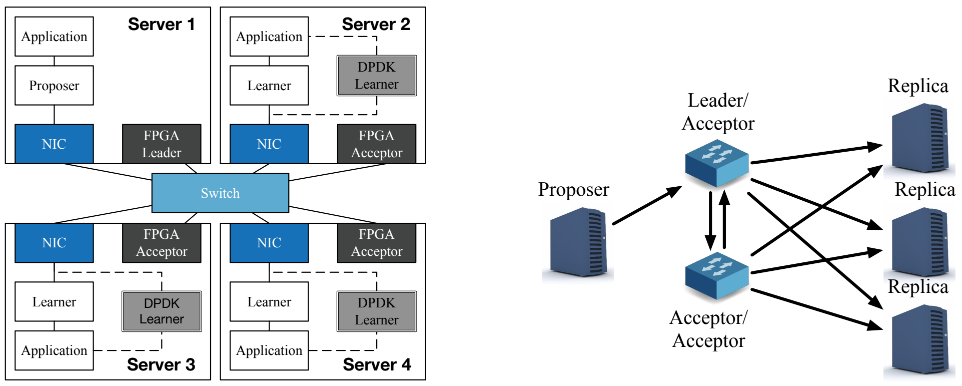


Figure 4.6. The two topologies used for the evaluation: FPGA testbed (left) and Tofino testbed (right).

less than 3 μs, compared with 30 μs in `libpaxos`.

P4xos results in significant improvements in latency and throughput. While `libpaxos` is only able to achieve a maximum throughput of 64K messages per second, P4xos reach 102K, 268K, and 448K messages per second for DPDK, SUME and Tofino respectively, at those points the server application becomes CPU-bound. The lowest 99th-ile latency for `libpaxos` occurs at the lowest throughput rate, and is 183μs. However, the latency increases significantly as the throughput increases, reaching 478μs. In contrast, the latency for P4xos starts at only 19μs, 52μs, 88μs and is 125μs, 147μs and 263μs at the maximum throughput for Tofino, SUME and DPDK correspondingly.

4.2.1.6 Discussion

Assumptions about the network. The central premise of our work is that you don't need to change a fundamental building block of distributed systems in order to gain performance. This thesis is quite different from the prevailing wisdom. There have been many optimizations proposed for consensus protocols, that typically rely on changes in the underlying assumptions about the network [58, 62, 63, 56]. Consensus protocols, in general, are easy to get wrong, as strong assumptions about network behavior may not hold in practice. In contrast, Paxos is widely considered to be the "gold standard". It has been proven safe under asynchronous assumptions, live under weak synchronous assumptions, and resilience-optimum [35].

Offload and network size. P4xos fits the changing conditions in data centers, where operators often increase the size of their network over time. As software and hardware components are interchangeable, P4xos allows starting with all components running on hosts, and gradually shifting load to the network as the data center grows and the consensus message rate increases.

Limitations of programmable network hardware. Despite the benefits provided by network offloading, we noticed that programmable networks are still affected by several limitations, mainly due to their recent introduction. When designing P4xos, we realised that we lacked most of the tools that developers use for coding software, such as libraries that we needed for implementing network applications in the hardware. After deploying P4xos in the network, we noticed we had no efficient and standardised tool for monitoring the performance of applications running in complex networked setups. Leveraging these experience, we collaborated to two projects, Emu and NRG, that provide tools for simplifying offloading applications to the network.

4.2.2 Emu

Emu is a framework for network functions on FPGAs that builds upon the Kiwi compiler [37]. Emu provides a powerful substrate for developers to rapidly offload network logic, specified through a high-level language, to FPGA hardware, with only modest effort. The main reasons for moving network services from the CPU to FPGAs are increased performance and save precious CPU cycles, which would otherwise be spent in polling the network card. Moreover, Emu provides automatic compilation of network services across different platforms, enabling better debug capabilities and an easier transition of workloads, without compromising on performance. Our contribution focuses on providing Emu with hardware interfaces for exploiting additional storage resources. We describe our Quad Data Rate II+ memory controller, that suppors Xilinx 7-series FPGA-based cards, including the NetFPGA SUME.

4.2.2.1 Overview

With Emu, developers can use a general purpose programming language to implement high performance network functions that run on FPGAs. Moreover, Emu provides an interface to IP (Intellectual Property) blocks, specialised modules taking account of hardware features. This further abstracts away the details of hardware development. The Emu runtime provides an abstract target environment, and a library of functionality that can execute on both CPU and FPGA, simplifying debugging and deployment.

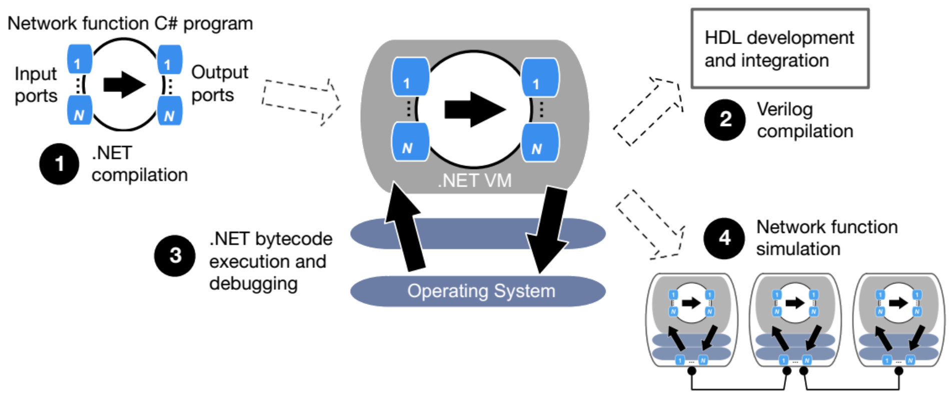

Network function C# program
Input ports
Output ports
1
N
1 .NET compilation
.NET VM
3 .NET bytecode execution and debugging
Operating System
HDL development and integration
2 Verilog compilation
4 Network function simulation

Figure 4.7. Emu development workflow

The major components of the Emu framework include: (i) a library tailored to network functions, (ii) runtime support for running C#-coded network programs on a CPU, and (iii) library support for developing and debugging such programs.

Emu extends Kiwi by making available library support customized to the networking domain. Kiwi also provides a "substrate" to support programs it compiles—the substrate serves as a runtime library for those programs and Emu extends this substrate.

The Emu development flow is shown in Figure 4.7. In ❶ the original C# program (e.g., processing packets from N input ports and forward results to N output ports) becomes .NET bytecode, which is then compiled to Verilog in ❷. This can be executed on hardware platforms such as NetFPGA. The bytecode can also be executed on CPUs and debugged using a provided software runtime in ❸. Layers of abstraction between the .NET VM and the OS provide virtual or physical network interfaces. By using virtual interfaces, we are therefore able to test network functions in a simulator shown in ❹. Creating parallel simulators enables us to achieve network-scale testing in a single machine.

Additional components provided by the Emu framework include protocol parsers, hardware IP blocks, concurrent threads and utility functions.

Parsers. Parsers for commonly-used packet formats are available for reuse and parsers that might be needed during runtime are instantiated on loading. The Emu library can handle TCP over IPv4 over Ethernet, as well as ARP over Ethernet and allows users to easily implement additional parsers for custom protocols.

IP blocks. While C# provides an easy development environment, to maximize the performance of a design it is sometime recommended to use specialized IP blocks that take advantage of the hardware capabilities. Such blocks are accessible through facilities provided by Kiwi. One such example is the QDRII+ memory interface controller, that we discuss in section 4.2.2.2.

Concurrent Threads. Kiwi reinterprets concurrency primitives that are used when programming software, to improve its hardware generation. Kiwi provides a thread-based concurrency library with two semantics: the software semantics reduces to concurrency primitives provided by .NET, while the hardware semantics forms logical circuits in which parallel threads may be wired into parallel logical subcircuits. Using these semantics, .NET programs may be executed on general-purpose processors such as x86 by using the software semantics, or in FPGAs by using the logical circuit semantics. In the latter case, Kiwi can produce descriptions with much finer parallelism than is possible by software platforms, whose parallelism is at most instruction-level. In Emu we take advantage of this, and further refine it, to achieve maximal pipelining of projects and increase throughput.

Utility functions. Utility functions form an important part of the support provided by Emu, above the compilation provided by Kiwi, and the target environment provided by FPGA. Utility functions consist of C# code and can help in abstracting various details, such as the functions for interacting with the FPGA target.

4.2.2.2 QDRII+ Memory Interface

Our main contribution to the Emu project is an open-source QDRII+ memory interface hardware module. The module can be leveraged in any Emu-based design that targets network devices equipped with Xilinx FPGAs.

4.2.2.2.1 QDRII+

Quad Data Rate II+ (QDRII+) is a type of static RAM (SRAM) designed for applications that require high speed, low latency memory. QDRII+ architecture consists of two separate ports: the read port and the write port to access the memory array. This allows to completely eliminate the need to “turn-around” the data bus that exists with common I/O memory devices. This feature, coupled to the lack of refresh, provides QDRII+ memory with constant, deterministic access latency.

The notion of “Quad” data rate comes from the ability to simultaneously read from a unidirectional read port and write to a unidirectional write port on both rising and falling clock edges. Therefore, QDRII+ can transfer four words of data on one clock cycle. Common applications of QDDRII+ memory include look up

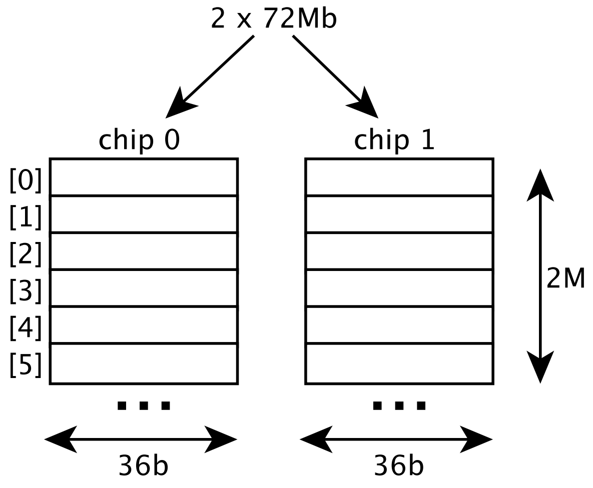


Figure 4.8. QDRII+ memory layout on NetFPGA SUME.

table for deep packet processing, packet forwarding queuing and buffering for high speed network switches and cache memory for high-end workstations. In the context of the Emu framework, the main application of QDRII+ memory is to implement bigger CAM tables, with little overhead on latency.

4.2.2.2.2 QDRII+ on NetFPGA SUME

We designed our memory interface for being used with the NetFPGA SUME framework. However, our design is configurable and can be easily ported to other FPGA-based network devices. Each NetFPGA SUME card is equipped with three Cypress QDRII+ memory chips [136, 137], that are directly connected to the I/O pins of its Xilinx FPGA. Unfortunately, only two out of the three chips can be used as the same time, due to a limitation of Xilinx's memory interface controller.

As shown in figure 4.8, each QDRII+ chip can store up to two millions 36b-wide words, thus providing a total capacity of 72Mb. The memory is organised as a an array of 36b-wide rows and each row is pointed by a 19b-wide address.

We decided to access two QDRII+ memory chips in parallel, for being able to store 72b-wide data words, instead of 36b-wide words. Although this configuration limits the size of the memory to two millions entries, it simplifies storing network header fields wider than 36b, such as ethernet addresses, to the QDRII+, when used as a CAM table. An alternative configuration consists in accessing the two QDRII+ chips sequentially, as a unique bigger memory module. Although

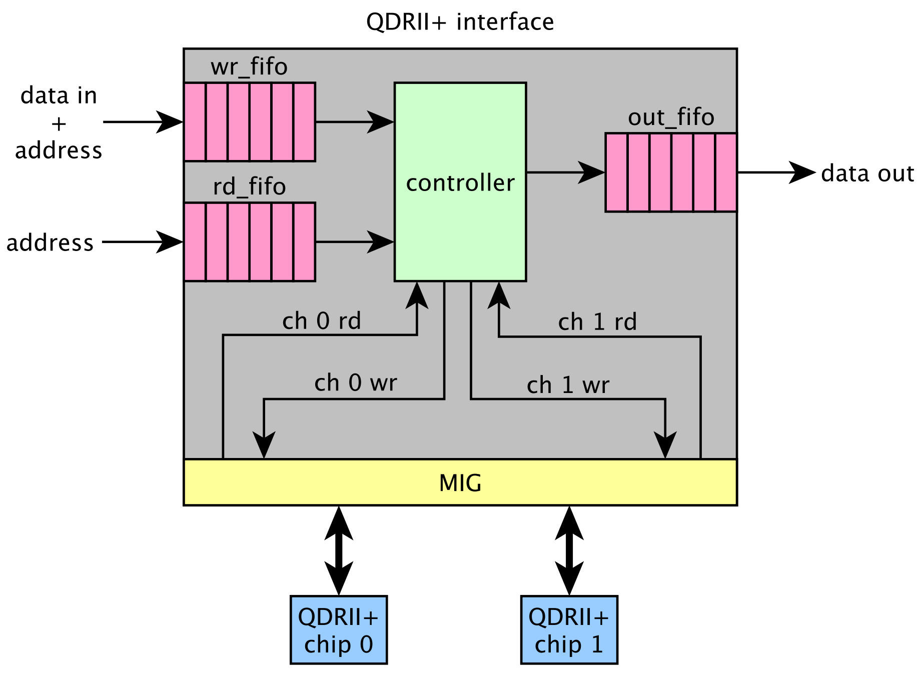


Figure 4.9. QDRII+ interface hardware module.

this configuration provides twice the rows, compared to the “parallel” configuration, it limits the size of each word to 36b, which is impractical for storing big network header fields.

Low-level access to the memory chips is provided by Xilinx’s Memory Interface Generator (MIG) [138], an FPGA hardware component that can be configure for interfacing a variety of memory devices, including QDRII+ and Double Data Rate (DDR) modules.

4.2.2.2.3 QDRII+ interface hardware module

Figure 4.9 shows a schematic representation of our QDRII+ memory interface module. The design, coded in Verilog, exploits Xilinx MIG, coloured yellow in the figure, as a low-level controller for accessing the native interface exposed by the two QDRII+ chips at the bottom.

We decided to implement a higher-level controller, coloured green in the figure, to be used on top of the MIG, since the interface exposed by the MIG to the user design does not implement any standard protocol and requires too many signals to be driven, thus making users’ design unnecessarily complex. The controller translates incoming instructions, in the form of FIFO signals, to the in-

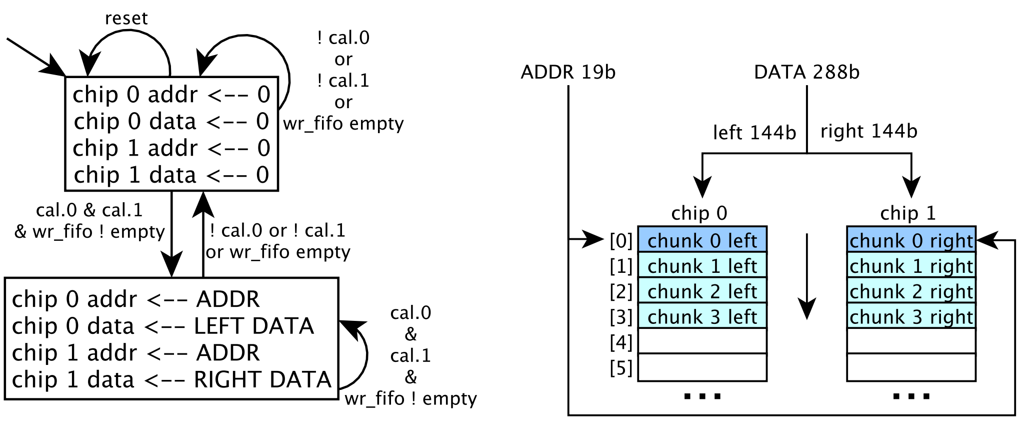


Figure 4.10. Write operation: FSM (left) and memory mapping (right).

terface exposed by the MIG. Although the architecture has been configured for accessing two QDRII+ chips in parallel, both the MIG and the controller can easily be reprogrammed for supporting different memory layouts, even including more than two chips.

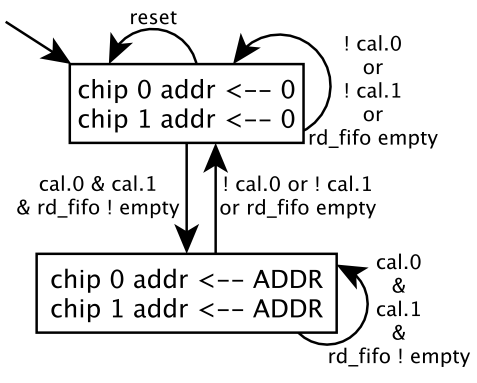
Since QDRII+ memory can sustain a single read operation and a single write operation at each clock edge, we provided each input/output channel with a configurable FIFO, coloured purple in the figure. This solution allows the architecture to tolerate bursts of identical operations, such as sequences of write operations, not interleaved with read operations.

The whole architecture, packed into a standard Xilinx hardware IP core, exposes three FIFO interfaces, two at the input and one at the output. The write interface takes both the data to be stored and a memory address and provides no feedback upon completion. On the contrary, the read interface requires only the address of the memory location to be read. The output interface provides the requested data.

4.2.2.2.4 Controller

Based on QDRII+ architecture, our controller implements two separate channels, one for reading, the other for writing data to the memory. The two channels are directly connected to the FIFOs. The read channel takes read requests from the read FIFO and provides read data through the output fifo. The write channel is attached only to the write FIFO. All the operations are synchronous: at each clock edge, two operations are computed, a read and write.

The controller drives the MIG interface through its native signals, thus translating both read and write operations to memory access instructions. This func-


reset
chip 0 addr <-- 0
chip 1 addr <-- 0
! cal.0 or ! cal.1 or rd_fifo empty
cal.0 & cal.1 & rd_fifo ! empty
! cal.0 or ! cal.1 or rd_fifo empty
chip 0 addr <-- ADDR
chip 1 addr <-- ADDR
cal.0 & cal.1 & rd_fifo ! empty

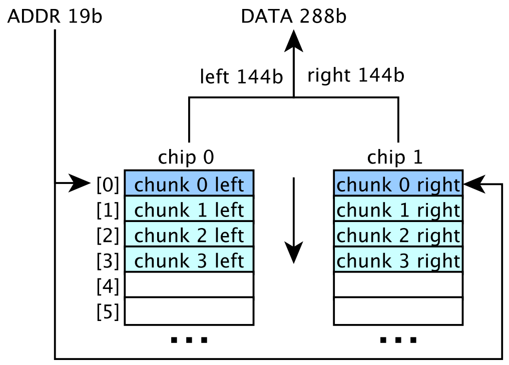


Figure 4.11. Read operation: FSM (left) and memory mapping (right).

tionality is implemented through two separate finite state machines (FSMs), running independently inside the controller.

The FSM that implements the write operations it shown in figure 4.10. It is composed of only two states: an initial “idle” state, that sets the memory control signals to zero and an “active” state that fills both the data bus and the address bus. No additional state is required, since the write instruction takes a single clock cycle for being issued to the controller. Besides the initial reset operation, the FSM keeps oscillating in between the two states, based on the status of the QDRII+ chips and on the occupation of the FIFO. Each of the two QDRII+ chips performs a low-level calibration of its signals through the MIG and exposes a flag to the controller, for signalling an ongoing calibration operation. Similarly, the write FIFO rises a flag every time it becomes empty. In case either one of the two chips is in the calibration process, or the write FIFO is empty, the controller transitions to its idle state, without processing new requests. Otherwise, it extracts a single write request form the FIFO and maps it to the lower-level signals that control the two QDRII+ chips in parallel.

A schematic representation of the write operation is shown in figure 4.10, in form of write accesses to the two QDRII+ memory chips. Since the two chips are accessed in parallel, as a single, wider, memory chip, the address provided in the write request points to two rows at the same time, one in each of the two chips. Although a single memory row is 36b-wide, the data transferred by the controller is 288b-wide, since a single write request is converted to two parallel bursts, each composed of four memory requests. This operation is performed both by the controller, and by the MIG. The controller splits a write request into two concurrent operations and divides the input data into two halves of 144b each. The MIG takes each of the two operations and translates it to a burst of four smaller write operations, each storing 36b-wide data chunks to four memory locations. The MIG automatically increments the address provided by the write request, for generating four subsequent addresses, coloured blue and light-blue in the figure. Therefore, the user's logic, connected to the memory interface, can address 500 memory rows ($2M/4$), each storing a 288b value ($2x(36bx4)$).

Figure 4.11 shows the FSM that implements the read channel. Similarly to the write channel, the read FSM includes only two states and state transitions are triggered by both the calibration signals and the empty flag of the FIFO. No additional states are required, since all read instructions are computed in a single clock cycle. However, the FSM drives only two signals, one for each QDRII+ memory chip, since only the address is needed for performing a read operation.

A schematic representation of the read operation is shown in figure 4.11. The mapping of the operation to the memory layout is similar to the write operation: the controller automatically issues eight accesses to subsequent memory rows, coloured blue and light-blue in the figure, in a way which is transparent to the user logic. The only difference with the write operation is the movement of the data that, in this case, is retrieved from the QDRII+ memory and aggregated in a single 288b-wide chunk, before being written to the output FIFO.

Thanks to its standard FIFO interfaces, the memory controller easily integrates with Emu-based logic, for providing access to high-speed, low-latency memory. The controller is part of Emu's IP blocks library and can be instantiated into Emu-based designs through dedicated facilities.

4.2.2.3 Use Cases

We implemented different networking services to demonstrate the advantages of Emu. These include forwarding, measurement and monitoring, performance sensitive applications and more complex applications such as NAT and caching.

Learning switch. This project implements the standard layer-2 learning switch, similar in functionality to the NetFPGA SUME Reference Switch [139]. Beyond header processing, which is a basic networking function, this project provides an example of how content addressable memory (CAM) can be implemented in Emu, and how a native FPGA IP CAM block can be used by Emu. While the first option reduces the knowledge requirements from the programmer, the latter provides better resource usage and timing performance.

ICMP echo. We implemented an ICMP echo server to obtain two important baselines: (i) a qualitative baseline on the difficulty of implementing a very simple network server, and (ii) a quantitative baseline figure on how much time is saved by avoiding the system bus, CPU, OS, and network stack in a simple working system.

TCP ping. ICMP traffic is sometimes handled differently by the network when compared to the protocols used by applications themselves, such as TCP and HTTP. For example, a faulty configuration of the network might create a black hole for some TCP ports on a machine, but without affecting the reachability of that machine through ICMP [140]. TCP ping involves a simple reachability test by using the first two steps of the three-way connection setup handshake. The TCP ping is a similar but more complex extension to ICMP echo.

DNS. We implemented a simple DNS server that supports non-recursive queries. Our prototype supports resolution queries from names (of length at most 26 bytes) to IPv4 addresses, but these constraints can be relaxed (to handle longer names, and IPv6). If the queried name is absent from the resolution table, then the server will inform the client that it cannot resolve the name.

Memcached. Memcached [141] is a well-known, distributed, in-memory key/-value store that caches read results in memory to quickly respond to queries. Memcached is very sensitive to latency, and even additional 20 ms are enough to lose 25% throughput [95]. Initially designing our platform with a latency performance goal in mind, our first memcached implementation supported a limited version of memcached supporting GET/SET/DELETE using the binary protocol over UDP, and supporting 6-byte keys and 8-byte values. We later experimented with different optimizations for this design, including supporting ASCII protocol, extending the key/value size, providing DRAM support and supporting multiple cores. These have different performance metrics in mind, including latency, throughput, and additional functionality.

4.2.2.4 Evaluation

The evaluation of Emu has the following aims: (a) Provide evidence that using Emu is beneficial in terms of resources and performance, compared with other solutions; (b) Provide evidence that Emu can be used to provide high performance network services; (c) Exemplify the advantages of using Emu.

The evaluation is conducted using a server with a single 3.5GHz Intel Xeon E5-2637 v4 CPU and with 64GB DDR4 Memory. The machine is running Ubuntu 14.04 LTS and it is equipped with a dual port 10GbE NICs. The machine also has a NetFPGA SUME board for the performance comparison. In addition, we use an Endace DAG 9.2X2 card for accurate latency measurements. For our throughput measurements, we use OSNT [24] as the traffic source.

	Emu	NetFPGA Reference
Logic Resources	5350	7413
Memory Resources	0	7
Module Latency	9 cycles	6 cycles

Table 4.2. Comparison between Emu Switch (written in C#) and NetFPGA Reference Switch (written in Verilog).

Network	Emu			Host		
Service	Latency μs	99th μs	Throughput (Q/s)	Latency μs	99th μs	Throughput (Q/s)
ICMP Echo	1.09	1.11	3226k	12.28	22.63	1068k
TCP Ping	1.27	1.29	2105k	21.79	65.00	1012k
DNS	1.82	1.86	1176k	126.46	138.33	226k
NAT	5.67	5.77	2439k	10500.18	26565.52	1037k
Memcached	1.85	2.03	952k	24.29	28.65	876k

Table 4.3. Performance Comparison Between Services Running on a Host and Emu Based Services (written in C#).

Performance Comparison. We evaluate the overhead of using Emu for programming an FPGA, and show that the resulting implementation is comparable with native HDL designs. We conduct this evaluation by comparing the Emu Learning Switch project, in C# and compiled in Kiwi, with NetFPGA SUME Reference Switch written directly in Verilog. Table 4.2 shows the resources consumed by the Main Logical Core in each design. These results confirm that the resources overhead is minimal, making Emu an attractive solution. Furthermore, out of the reported resources consumed Emu core, 85% are consumed by the CAM, which is an IP block, and only 15% by the C# generated logic. We note that in all our use-cases the FPGA resources were never exhausted, and consumed less than 32.7% of the logic resources. In terms of latency, Emu has only a minor overhead over the main logical core in the NetFPGA SUME Reference Switch design.

Absolute Performance. We compare the performance of different use-cases introduced in Section 4.2.2.3 with software-based solutions and present the results in Table 4.3. Further extending the above Emu use-cases can be done in different ways. For example, for memcached one way involves increasing the memory available to Emu, using either on-board or on-chip memory. On-board memory, e.g. using either the DDR3 DRAM memory modules, or the QDRII+ chips on NetFPGA SUME, has the advantage of memory size, but the disadvantage of increased latency. On-chip memory has the benefit of low, constant, latency, but is smaller in size. A different extension may seek to expand the throughput.

Possibly the easiest way to do so is by instantiating multiple Emu memcached cores. By instantiating four cores, one per port, the achievable throughput improvement is x3.7 for a setup that uses a mix of 90% GET and 10% SET queries. SET queries need to be applied to all instances, thus their relative ratio in performance can not be improved. The downside is that it requires changing the Main Logical Core wrapper in NetFPGA SUME.

4.2.2.5 Discussion.

Our evaluation demonstrates the advantages of Emu. First, hardware resource usage is significantly lower than other approaches, adding only modest overhead when compared with bespoke HDL-only designs. The latency overhead is minimal compared with HDL designs, and is comparable or better than the other baselines. Our evaluation also shows an important advantage of Emu over host-based solutions: while absolute performance will always depend on cores, memory bandwidth and frequency, FPGAs always enjoy the benefit of predictability: the median latency of our designs is both x10 lower than the median of the host based solutions, and has a very small variance. While the difference between the median and 99 percentile is less than 200ns for Emu (and typically less than 100ns), for host based designs the variance is in the order of microseconds to tens of microseconds. This does not only affects RTT and flow completion times, but can also allow users to better schedule resources, as they know when a reply is due.

4.2.3 NRG

NRG is a Network Research Gadget that leverages the capabilities of programmable network hardware for enabling reproducible networking experimentation through network emulation and monitoring. NRG can be used as a standalone device, or as an instantiated core within a network device. We use NRG to generate *Network Profiles*, the combination of the application's performance and the characteristics of the underlying network as the application runs, that can help attend to a multitude of application performance problems, and better provision others. We contributed to NRG by implementing two monitoring functionalities in hardware, using P4 language. In the following, we describe both the designs and present their NetFPGA SUME-based implementation. We also discuss the limitations we faced in our attempt to port the same code to a programmable network switch ASIC.

4.2.3.1 Network Profiles

We refer by Network Profiles to a collection of network-related characteristics and their relation to an application's performance. NRG enables users to run within their given local setup a set of experiments that actively change network conditions and passively collect network-related statistics. This collection of network conditions and statistics is the network profile of an application on a given setup.

A network profile of an application is setup dependent. Changes to system parameters such as number of machines or CPU type may change the network profile, for example by exhausting a server's resources or increasing the link utilization. However, the first thing that network profiles tell us is if an application is network sensitive or network insensitive.

The operational model of NRG is the substitution of some-part-or-all of an uncontrolled cloud-based black box experimentation model with a local user setup incorporating NRG. NRG can act as a bump in the wire or as a *transparent* module within a NIC or switch. Wherever NRG is instantiated, it can programmatically limit available bandwidth and increase latency for a given link.

4.2.3.2 Architecture

NRG assumes that a user has a controlled experimentation environment, possibly of limited size, which is used during the development process. NRG implements hardware-based network emulation: permitting high-performance, line-rate, experiments. When combined with a set of software tools, NRG allows a user to focus on exploring application and network variables in a reproducible manner.

An experiment begins by setting up and configuring a set of NRG-enabled devices within a networked-system, of networks, servers, and other computing equipment. Once the setup is ready, an experiment is triggered by NRG's orchestration module. At the end of the experiment, results are collected from the benchmarked application. Monitoring information is also collected from the system and from NRG-enabled devices. Last, the collected information is processed and network profiles are generated.

Control and Orchestration. NRG targets networked systems of multiple nodes (servers) and potentially multiple NRG devices. A single control and orchestration node is used to set up an experiment on all the participating nodes, by installing the application, setting up system configurations and experiment-specific configurations. The same node is also responsible to set up NRG-enabled devices, including configuring up the platform and all of its modules.

The orchestration module is the one to coordinate experiments, starting (and stopping) tasks on different nodes and NRG devices. The order and timing of orchestration is of an importance, as all NRG devices must be triggered and monitoring before the application is started, and as an uncoordinated application start can skew experimental results.

Network Emulation. Network emulation is an in-band component of NRG. It can be implemented as a stand-alone bump-in-the-wire device, or as part of a more complex device (e.g., switch), in which case we refer to the device as NRG-enabled. The in-band network emulation module is instantiated after a device's data plane modules, and provides latency control (through the delay module), and bandwidth and burstiness control (through the rate control module).

Network Monitoring. NRG uses hardware acceleration to provide flexible insights into the network, with no host processing overhead. NRG's monitoring is a passive module, instantiated in-parallel to the data plane, and is not traffic affecting. The module monitors aspects defined by the user, such as link utilization, burst size, inter packet gap, flow size, and user defined statistics. The module is triggered at the beginning of an experiment, and it logs statistics only when traffic is available, i.e., idle periods do not consume any limited module resource.

Data Collection and Processing. At the end of an experiment, a software module, in charge of performance reports, collects instrumentation information from all the nodes, including NRG, to create network profiles. The module gathers experimental results: performance results from the end hosts, processed statistics from NRG, etc.

4.2.3.3 Implementation

The software components of NRG are implemented in Python, with additional C-based code used to interact with the NRG platform. The prototype supports both a scripted environment for repeated experimentation, and a user interface for manual configuration and testing.

NRG enables adding drop-in user defined monitoring mini-blocks. These modules are not language bound. In addition to Verilog, we have also prototyped NRG's monitoring mini-blocks using .NET and P4. We used P4-NetFPGA [46] for a P4-based implementation. Here, the monitoring is done in-line with the data processing. We used Emu 4.2.2 for a .NET implementation of several monitoring mini-blocks. NRG's active path, the delay and rate control modules, are implemented in Verilog. Such modules, which implement queues and some glue logic, are less suitable for high level languages (e.g., P4), except as externs.

NRG's hardware is portable between different targets. We consider three types of targets: FPGA accelerators, smart NICs and switches. FPGA targets can operate as a bump-in-the-wire, as a NIC and as a limited-size switch. NRG can be ported to any smart NIC with an FPGA component, while for other types of NICs it would depend on the programmability of the device. Porting NRG to switch ASIC can be highly valuable, especially for monitoring, as switches located within the network fabric and can observe network dynamics not available to edge devices.

4.2.3.4 Use Cases

NRG was developed as a tool that enables exploring, understanding and reproducing networked-experiments. In this section we describe a few potential use cases of NRG and network profiles.

Resource Allocation. Understanding the performance of distributed applications can be a hard task. NRG's active path enables studying the sensitivities of different applications to bandwidth and latency through experimentation in a controlled environment, leading to better resource allocation. NRG's monitoring allows users to explore network utilization.

Understanding and Debugging Applications' Phenomena. Network Profiles provide an holistic approach to understanding applications' performance, providing insights that are beyond a single resource. In this sense, NRG and network profiles enable understanding why some applications are more sensitive than others to the allocation of system's resources. Similarly, NRG enables observing the effect of code-changes on network-level application behavior and to refine the code accordingly.

Reproducibility. NRG enables reproducible experimentation, while varying the underlying network conditions and by emulating a cloud network as a black box. There are several important aspects to such reproducible experimentation. First, from a developer's perspective, it provides a stable experimentation environment that provides the same network conditions time after time, and allows to explore to easily recreate failure conditions. NRG also allows customers to benchmark products in a non-production environment while emulating production conditions, extending the product's evaluation to tests applicable to their own operating environment. Last, as researchers, NRG provides an environment for reproducible benchmarking and comparison of different solutions.

```
1  bytes_counter = 0
2  packet_counter = 0
3  clock = 0
4  time_unit (value set by the user)
5
6  for every new packet:
7
8    bytes_counter += packet_size
9    packet_counter ++
10
11   if clock >= time_unit {
12
13     bandwidth_memory ← (bytes_counter, clock)
14     packet_memory ← (packet_counter, clock)
15     bytes_counter = 0
16     packet_counter = 0
17     clock = 0
18
19   }
```

Figure 4.12. Pseudo code of running statistics mini-block

4.2.3.5 Two Monitoring Mini-blocks

We present two monitoring mini-blocks that we implemented in P4 [1], through P4→NetFPGA [46], targeting the NetFPGA SUME platform [4]. Then, we discuss the limitations that prevented us from porting the two mini-blocks to a programmable network switch ASIC.

4.2.3.5.1 Running Statistics Mini-block

The first mini-block computes running statistics over a continuous flow of packets, on a configurable time interval. In particular, both the bandwidth and the packet rate are stored into two separate memory locations, that can be accessed by the users for collecting the results of the test.

The operations are illustrated by the pseudo code of running statistics mini-block, shown in figure 4.12. First, lines 1 - 3 initialise the counters to zero. Both the byte counter and the packet counter are then used for computing the final results of the test. The clock counter computes the duration of the test in clock cycles, up to the value of the threshold configured by the user through the time unit, at line 4.

The loop at line 6 runs an iteration of the test every time a packet enters the switch pipeline. For each incoming packet, the bytes counter is incremented with the packet size and the packet counter is incremented by one (lines 8 - 9). Then, in case the duration of the test has not reached the threshold yet, the execution

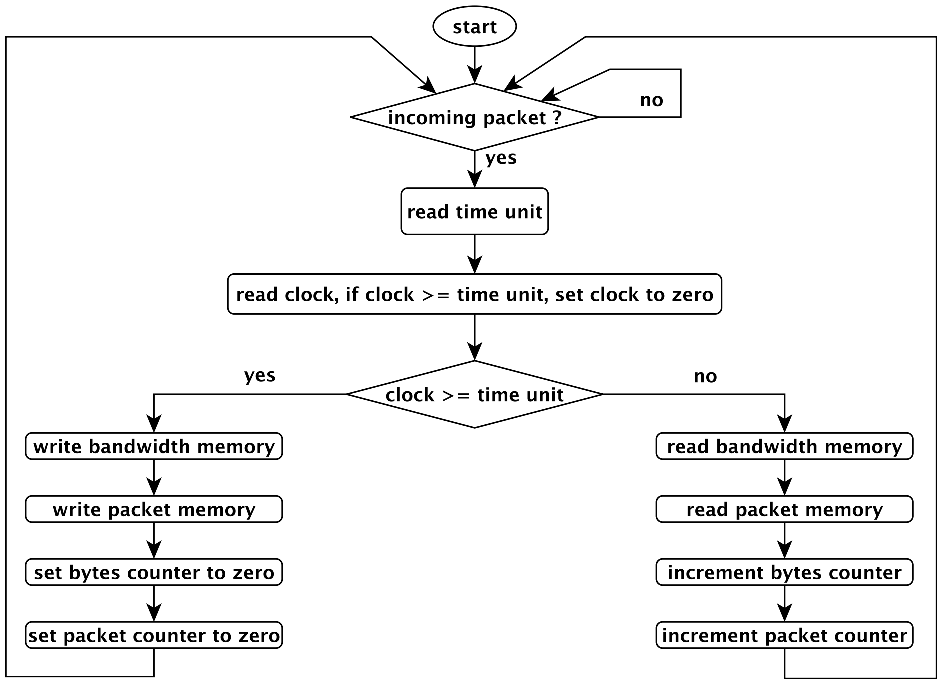


Figure 4.13. Program flow of running statistics mini-block.

goes back to line 6, waiting for a new incoming packet. Otherwise, both the bytes counter and the packet counter are stored into two memory locations, alongside the value reached by the clock counter at the end of the test (lines 13 - 14) and all the counters are reinitialised to zero, at lines 15 - 17, before starting a new test session (line 6).

The results of the test can be collected by reading the values stored in the bandwidth memory and in the packet memory. The bandwidth is computed by dividing the amount of bytes processed during the execution of the test, by the duration of the test. Similarly, the packet rate is computed by dividing the number of processed packets, by the duration of the test.

Figure 4.13 illustrates the program flow of the first mini-block, as mapped to the PISA architecture, using extern modules. Since all the operations in the data plane are triggered by an incoming packet, the program initialises its internal state after an incoming packet is detected, and not before, as specified by the pseudo code shown in figure 4.12. State initialisation consists in reading the user-provided time unit from a pre-defined register location. We assume both the counters and the clock are automatically initialised to zero, since it is possible to specify the initial value of a register in a P4 program.

```
#define READ_OP 8w0
#define WRITE_OP 8w1
#define RDINC_OP 8w2
#define RDZR_OP 8w3
#define ADD_OP 8w5
#define RGTZ_OP 8w6

extern void clock_ext(...);
extern void time_unit_ext(...);
extern void bw_memory_ext(...);
extern void pkt_memory_ext(...);
extern void packet_ctr_ext(...);
extern void btctr_ext(...);

// Omitting parser ...

control TopPipe(...) {
  bit<...> clock;
  bit<...> time_unit;
  bit<...> pktctr_out;
  bit<...> btsctr_out;
  bit<...> bwmem_op;
  bit<...> pktmem_op;
  bit<...> pktctr_op;
  bit<...> btsctr_op;

  apply {
    time_unit_ext(READ_OP, time_unit);
    clock_ext(time_unit, RGTZ_OP, clock);

    if (clock >= time_unit){
       bwmem_op = WRITE_OP;
       pktmem_op = WRITE_OP;
       btsctr_op = RDZR_OP;
       pktctr_op = RDZR_OP;
    }else{
       bwmem_op = READ_OP;
       pktmem_op = READ_OP;
       btsctr_op = ADD_OP;
       pktctr_op = RDINC_OP;
    }

    btctr_ext(metadata.pkt_len, btsctr_op, btsctr_out);
    packet_ctr_ext(pktctr_op, pktctr_out);
    bw_memory_ext((btsctr_out ++ clock), bwmem_op);
    pkt_memory_ext((pktctr_out ++ clock), pktmem_op);
  }
}

// Omitting deparser ...
```

Figure 4.14. P4 code implementing running statistics mini-block.

After initialisation, the program reads the clock value from an external module, programmed as a clock generator. It also compares the clock value to the time unit and, in case it is greater than or equal to the time unit, it resets the clock generator. This operation is necessary, since extern modules in PISA-based architectures can be accessed only once per packet. Therefore, we decided to condense two accesses to the clock generator into a single operation, by implementing the check into the logic of the extern module. For the same reason, we postpone all the remaining register operations as much as we can, for reducing the number of operations to be computed over the extern modules.

We also replicate all the operations in both the branches of the “if” conditional statement, since all the extern modules need to be accessed anyway at each execution of the program. In case the “if” condition is not met, the program increments both the bytes counter and the packet counter, as specified by the pseudo code. In this case, reading the bandwidth memory and the packet memory is equivalent to perform a “no-op”, since the values stored into the two memory locations remain unchanged. In case the “if” condition is met, the program stores the results into both the bandwidth memory and the packet memory and resets the bytes counter and the packet counter. Then, in both cases, the program goes back to its initial state, waiting for new incoming packet.

Figure 4.14 shows the P4 implementation of running statistics mini-block. For simplicity, we omit part of the code, including the parser and the deparser. First, lines 1 - 6 provide convenient definitions for the operations supported by the extern modules. Besides common operations, such as “read”, “write” and “add”, other operations have been implemented specifically for this mini-block.

Lines 8 - 13 instantiate all the extern modules used in the program. Once instantiated, the externs are exposed as functions in the P4 program. Although some externs, such as “bw_memory_ext” implement only a stateful memory, others provide more complex functionalities, including counters and clock generators.

Instructions at lines 18 - 25 declare few temporary variables used for operating with the extern modules. Variables at lines 18 - 21 are used for exchanging data with the externs. Variables at lines 22 - 25 specify which operation an extern has to perform.

The following instructions implement the program flow shown in figure 4.13. After initialising both the time unit and the clock value (lines 28 - 29), the program implements the “if-else” condition (lines 31 - 41). In order to access each extern module only once per packet, we decided to split each operation over the externs into two parts. The first part selects the operation to compute. The second part runs the extern module with the selected operation. Therefore, we map the first part to the two branches of the “if-else” condition and we call the externs with the selected operations after the “if-else” condition (lines 43 - 46). At line 43 we retrieve the size of the packet from the metadata bus of the NetFPGA SUME. At lines 45 and 46 we store the results by concatenating the clock value with the bytes counter and with the packet counter, since it is possible to write a single memory location at a time for each extern.

4.2.3.5.2 CDF Statistics Mini-block

The second mini-block collects the data for computing CDF statistics about the streams of packets entering the data plane during a configurable time interval. In particular, it counts how many packets of each different size have been processed and the number of packet bursts of various lengths. Both the results of the test are stored into two separate memory locations, that can be accessed by the users for further analysis.

Figure 4.15 shows the pseudo code of the second mini-block. First, lines 1 and 2, initialise both the burst size counter and the clock counter to zero. The burst size counter keeps track of the length of the current burst, by counting the number of packets entering the pipeline. The clock counter counts the time,

```
1  burst_size = 0
2  clock = 0
3  burst_gap (value set by the user)
4
5  for every new packet:
6
7    packet_size_memory[packet_size] ++
8
9    if clock < burst_gap {
10
11     burst_size ++
12
13   } else {
14
15     burst_gap_memory[burst_size] ++
16     burst_size=0
17
18   }
19
20   clock=0
```

Figure 4.15. Pseudo code of CDF statistics mini-block.

namely the gap, in between two subsequent packets. The user-provided burst gap, at line 3, indicates when a burst has to be considered as concluded, by setting an upper bound to the inter-packet gap.

The "for" loop at line 5 runs an iteration of the program every time a new packet enters the data plane and increments the packet size memory associated with the size of the incoming packet (line 7), thus storing the first result of the test.

Then, in case the clock counter has not reached the threshold yet, the size of the current burst is incremented (line 11). Otherwise, the program increments the burst gap memory associated with the size of the current burst (line 15), for storing the second result of the test, and the burst size counter is reset (line 16).

Finally, line 20 re-initialise the clock counter to zero and the program execution is restarted, for processing a new incoming packet.

The program flow, shown in figure 4.16, illustrates the mapping of the mini-block over a PISA architecture. Similarly to the first mini-block, most of the functionalities are implemented through extern modules and all the operations are triggered by the arrival of a new packet. Therefore, after a new packet has entered the pipeline, the program increments the packet size memory and reads both the clock and the burst gap, configured by the user, thus initialising its internal state. At the first iteration of the program, we assume both the burst size and the clock counters are automatically initialised to zero, though a specific function provided by the extern modules.

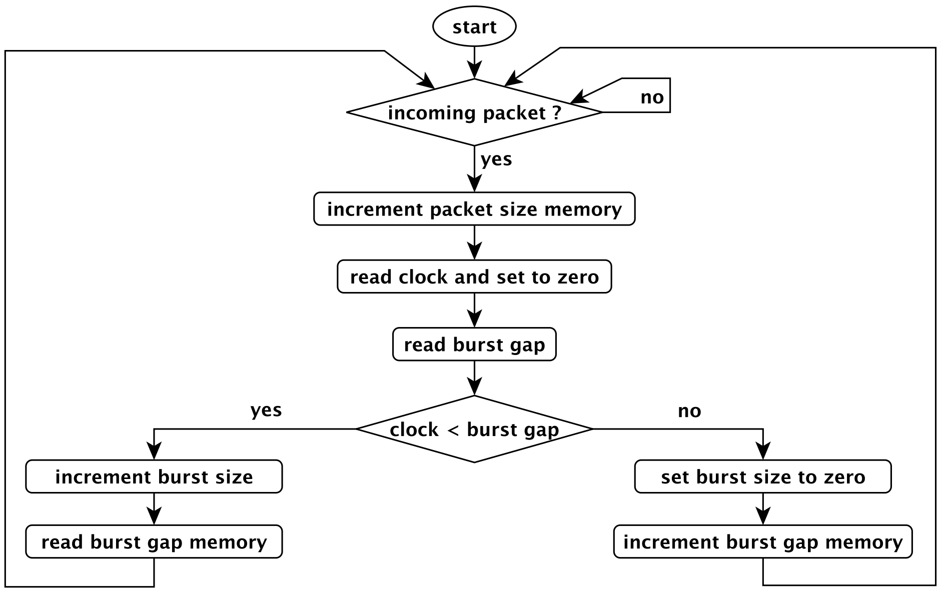


Figure 4.16. Program flow of CDF statistics mini-block.

Since it is possible to access the clock generator only once per packet, we decided to perform both the read operation and the reset operation at the same time. In case the value of the clock is less than the burst gap, the program increments the burst size and reads the burst gap memory, thus performing a "no-op". Otherwise, it resets the burst size and stores the result of the test, by incrementing the burst gap memory. In both cases, the execution goes back to its starting point, waiting for an incoming packet.

Figure 4.17 shows the P4 implementation of CDF statistics mini-block. For simplicity, we omit part of the code, including the parser and the deparser. Similarly to the first mini-block, lines 1 - 4 provide convenient definitions for the operations supported by the extern modules and lines 6 - 10 instantiate the extern modules used in the program.

Instructions at lines 15 - 20 declare few temporary variables used for operating with the extern modules. Variables at lines 15 - 18 are used for exchanging data with the externs. Variables at lines 19 - 20 specify which operation an extern has to perform.

The following instructions implement the program flow shown in figure 4.16. First, the program increments the packet size extern, using the length of the packet, provided by the metadata bus, as the index to the memory location (line 23). Then, it reads and resets the clock and retrieves the value of the burst gap, provided by the user through a pre-defined register (lines 24 - 25).

```
 1 #define READ_OP 8w0
 2 #define RDINC_OP 8w2
 3 #define RDZR_OP 8w3
 4 #define RDTS_OP 8w4
 5
 6 extern void packet_size_ext(...);
 7 extern void clock_ext(...);
 8 extern void burst_size_ext(...);
 9 extern void burst_gap_ext(...);
10 extern void burst_gap_memory_ext(...);
11
12 // Omitting parser ...
13
14 control TopPipe(...) {
15    bit<...> clock_out;
16    bit<...> bstgap_out;
17    bit<...> bstsize_out;
18    bit<...> bgm_index;
19    bit<...> bstsize_op;
20    bit<...> bgm_op;
21
22    apply {
23       packet_size_ext(metadata.pkt_len, RDINC_OP
             );
24       clock_ext(RDTS_OP, clock_out);
25       burst_gap_ext(READ_OP, bstgap_out);
26
27       if (clock_out < bstgap_out){
28          bstsize_op = RDINC_OP;
29          bgm_op = READ_OP;
30       }else{
31          bstsize_op = RDZR_OP;
32          bgm_op = RDINC_OP;
33       }
34
35       burst_size_ext(bstsize_op, bstsize_out);
36       bgm_index = (bstsize_out+1);
37       burst_gap_memory_ext(bgm_index, bgm_op);
38    }
39 }
40
41 // Omitting deparser ...
```

Figure 4.17. P4 code implementing CDF statistics mini-block.

At line 27 the program implements the "if-else" condition. Similarly to the first mini-block, we split each operation over the externs into two parts. The first part selects the operation to compute. The second part runs the extern module with the selected operation. Therefore, we map the first part to the two branches of the "if-else" condition and, then, we call the externs with the selected operations (lines 35 - 37).

Since PISA-based data planes are stateless, we decided to exploit the combination of a stateful extern memory with a stateless temporary variable for implementing a stateful memory location in the pipeline (lines 35 and 36). For each packet, we store the current burst size and we retrieve the previous size, through a single "read-increment" operation over a stateful extern module (burst_size_ext). The value read from the extern, that is used for incrementing the burst gap memory, is stored into a stateless temporary variable (bgm_index), after being incremented. The increment operation is necessary, since the value read from the extern refers to the previous iteration of the program and, in the current iteration, the length of the burst is one packet more.

The final instruction of the program either computes a "no-op" over the burst gap memory extern, or increments the memory location pointed by the bgm_index, in case the clock counter has reached the threshold configured by the user.

4.2.3.5.3 Porting to a Programmable Network Switch ASIC

We explored porting the two mini-blocks to a Barefoot Tofino switch [3]. Although this programmable switch ASIC provides much higher network performance than the NetFPGA SUME platform, it is much less flexible in implementing those functionalities not supported by its PISA architecture. These constraints prevented us from porting the two designs to the Tofino switch. In the following, we discuss the main limitations we faced, and we propose suitable mitigation strategies.

Custom Hardware Functions. Although Tofino provides a library of configurable extern modules, users cannot implement their own custom functions in the hardware. The only possible mitigation strategy is to use the logic provided by the registers, for computing elementary operations. However, registers introduce a number of limitations, that we discuss in the following. Complex functions that do not fit both the registers and the extern modules provided by the Tofino switch cannot be ported in any case.

Concurrent Register Operations. Although many elementary operations may be computed over the same register, only one can be executed for each packet. A valid mitigation strategy consists in replacing intermediate computations with operations over temporary variables, thus avoiding accessing a register more than once. Similarly, opcodes can be stored into temporary variables to separate the process of choosing which operation to perform from the one of running the chosen operation over a value into a register. Clearly identify which operation(s) needs to be performed over each register. Read-only register operations may be moved at the beginning of the pipeline, in order to have the results ready to be processed by subsequent operations. Write-only register operations may be moved at the end of the pipeline, for accessing the registers only once, by replacing the preceding operations over the same registers with operations over temporary variables. Exploit the programmability of the registers, for running a sequence of operations at each register access. Not portable designs include pipelines in which one (or more) register need to be accessed more than once per packet and the proposed design strategies are not sufficient for mitigating the limitation of the Tofino architecture.

Custom Register Operations. Although some degree of programmability is provided by the registers, only limited custom operations can be implemented. Possible mitigation strategies include exploiting temporary variables as much as possible, to perform more complex operations on temporary variables before accessing the registers, and scheduling complex operations over a sequence of registers, each processing a different piece of data. Very complex operations, that cannot be implemented using registers cannot be ported to currently available PISA-based architectures.

Clock Signal and Timestamp Counter. Hardware clock signal is not exposed to the data plane in PISA-based pipelines. Using timestamp, instead of hardware clock signal, could provide a workaround to this limitation, for programs that do not require time-stamping with fine granularity. Since P4 language provides no specific function for managing the timestamp counter, the counter cannot be reset from the data plane pipeline. A possibile mitigation strategy consists in computing the difference between the value of the counter and an offset value stored into a register. The difference is equivalent to a timestamp, for measuring the flow of time in the data plane pipeline. Not portable designs include pipelines in which the offset register needs to be accessed more than once.

4.2.3.6 Evaluation

We use NRG to explore how will different choices of network resources affect the user's application's performance. We use an experimental setup composed of 6 hosts, each with an Intel Xeon E5-2637 v4 CPU with four cores and with 64GB RAM. The hosts run Ubuntu Server 16.04 and are equipped with a 10Gbps Intel X520 NIC. Between one selected host and all other parts of the setup, we instantiate an NRG device acting as a bump-in-the-wire. In this manner, we have one server and five client machines, or one master and five worker machines.

We benchmark Apache webserver, and the performance metric is requests per second and we pick Tensorflow [142], a popular machine learning framework, and use the MNIST dataset [143] for training using a distributed learning functionality. The performance metric is training time. We study the effect of bandwidth and static latency on the performance of each application, varying both parameters on the link between one of the nodes (server) and the switch. Added latency ranges from zero, our performance baseline, and one millisecond. Bandwidth allocation ranges from 2Gbps to 10Gbps, with 1Gbps explored for some applications. Each experiment is run ten times, and we generate the network profile of each application, containing also the variance per experiment.

First, we compare the effect of static latency on the studied applications. Tensorflow loses 2.5% performance with 25μs added, and 8.3% with 100μs. At 500μs, all applications lose between 29% and 90% performance. This is an indication how critical it is to allocate machines physically close to each other when running latency sensitive workloads.

Next, we vary both bandwidth and latency and explore the effects on each application. Tensorflow is almost linearly sensitive both to latency and bandwidth. There is very little variability between experiments: less than 1.5% across all scenarios. The conclusion here is that using distributed learning with Tensorflow will work best when using closely placed VMs with very high bandwidth links, possibly even within the same physical server. Apache presents a more complex picture. The performance under bandwidth of 8Gbps to 10Gbps or up to 100μs RTT almost doesn't vary. The performance drops by 10% if the bandwidth is 8Gbps, combined with 100μs RTT, or if the RTT is 200μs. If bandwidth is further reduced, it becomes more dominant than latency in performance loss, meaning that it is better to have 10Gbps link with a millisecond latency, than to have a 2Gbps link with no added latency. This is not surprising, given the file transfer nature of Apache web server.

4.2.3.7 Discussion

NRG provides an insight into the network properties of applications, as they run over the network and can advise users and operators on best resource allocation and placement. NRG further provides the means to reproduce cloud-based experiments in a controlled environment, with latency and rate control. Using NRG, we have demonstrated the difference in network sensitivity of different cloud applications, informing user's resource selection and cloud provider's resource allocation. The network profiles NRG generates can provide valuable insights, such as buffer size, for switch vendors and cloud operators looking to reduce network congestion. NRG also indicates how often we see packet bursts, which is not available using watermarks or telemetry packets [87].

4.3 Chapter Summary

This chapter presented three projects that experiment offloading applications to the network and provide tools for addressing the challenges of In-Network Computing. For each of them, we provided a brief description and we discussed our main contributions. We experimented In-Network Computing, by offloading Paxos consensus protocol to programmable network devices, thus understanding both the benefits and the limitations of this new technology. Leveraging the experience gained in implementing P4xos, we collaborated to two projects, Emu and NRG, that provide tools for accelerating applications with network hardware and for monitoring their performance in the network.

Chapter 5

Trading Latency for Compute in the Network

In the previous chapter, we addressed some of the challenges of In-Network Computing, by proposing tools that simplify both offloading new applications to network hardware and monitoring programs running in the network. This chapter addresses another challenge of In-Network Computing: enabling new applications to be accelerated in the network, by balancing the trade-off between performance and expressiveness of network hardware.

The end of Moore's law and the limits of Dennard scaling have increased interest in the use of domain-specific processors as an alternative to general purpose compute devices. Domain-specific processors offer advantages not just because of the particular capabilities of the hardware, but also by virtue of *where* the hardware is placed. For example, computations might be performed in the network on data in-flight.

This has led to a blurring of the traditional division of labor between network and applications, as researchers have explored *In-Network Computing* (INC) [111, 110, 124] as a way to accelerate applications and services. Although In-Network Computing is still in its infancy, early research results are promising, often providing an order-of-magnitude (or more) increase in throughput and significant reductions in latency.

However, these performance improvements involve a trade-off. On the one hand, link speeds are fundamentally different between software, FPGA, and ASIC, with a gain of about 10× at each step. But, on the other hand, the expressiveness and flexibility of the hardware decreases inversely to the performance, strictly limiting the classes of applications that can be accelerated. So, while load-balancing [126] or consensus [111, 110] might be reasonable candidates for acceleration, In-Network Computing would unlikely be able to accelerate, for example, a Monte Carlo simulation.

It is therefore natural to ask: *is the exchange between performance and expressivity a characteristic attribute of In-Network Computing, or is there a way to navigate the trade-off?*

In this chapter, we argue for the latter, by proposing a “compute hierarchy” for In-Network Computing. By compute hierarchy, we mean that In-Network Computing should leverage a variety of compute hardware, and that computation should occur where it can best be performed. We identify benefits and limitations of both programmable network ASICs and FPGA-based network devices, and we discuss the key challenges in building a heterogeneous network architecture. We provide a concrete implementation and proof-of-concept data deduplication application and we validate the proposed approach in terms of performance, resource utilization, and power consumption.

5.1 Designing an Heterogeneous Architecture

We present a new heterogeneous approach to programmable architecture that extends the capabilities of programmable switch ASICs with FPGAs, thereby providing a path for accelerating a greater diversity of applications in the network.

Designing such an heterogeneous architecture is hard, due to the mismatch between different technologies. The mismatch is not only in expressiveness and flexibility, but also in performance and resources. Consequently, it is required not only that switch ASIC and an FPGA will *interoperate*, but also that they will *compensate* for the difference in performance, and *orchestrate* it in a manner that guarantees mandatory networking functionality remains unaffected.

There is likely no one-size-fits-all architecture that will work for all applications. It is therefore important that system designers understand the key design decisions that follow these challenges. In this chapter, we identify and elaborate these choices in depth. Moreover, we implement and evaluate one approach that we think will be broadly applicable.

As a proof-of-concept, we have implemented a fingerprint computation—i.e., mapping a large data item to a smaller bit string—on our architecture, realized with a Barefoot Tofino [3] ASIC and a NetFPGA SUME [4] FPGA. Our evaluation shows that the prototype runs at line-rate with a minimal impact on resources. Moreover, we demonstrate that using more recent FPGA technology and 100GE ports would scale up in throughput, while providing a reduction in latency.

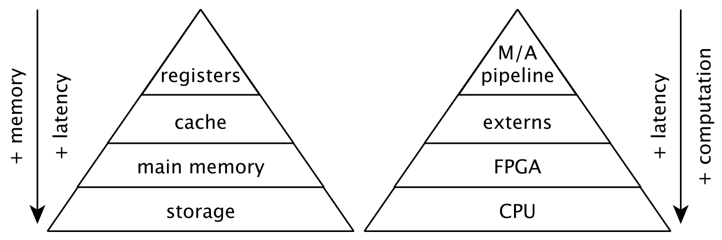


Figure 5.1. CPU hierarchy Vs network hierarchy.

5.2 Building Computing Hierarchies

Although several research prototypes have demonstrated the potential for In-Network Computing [111, 110, 124], we have yet to see wide spread adoption of switches as computing platforms. One reason is that In-Network Computing depends on platform-specific function modules (externs), meaning that a program can not be easily (or at all) ported between different types of devices. Another reason is that In-Network Computing consumes resources required for native networking functionality, meaning that a switch can not act as both a networking-device and an accelerator at the same time. Last, and possibly most importantly, switches do not have the right computing resources to run certain applications.

Benefits of Compute Hierarchy. To overcome the computation limitations of switches, we adopt a hierarchical approach, similar to CPU architecture. In CPUs, memory hierarchy allows one to trade memory space for latency. This means that access to registers is immediate, access to first level cache has a few clock cycles latency, and every additional level in the hierarchy provides more memory, for the price of higher access time. Our approach trades computation for latency; computation within the programmable switch-ASIC is almost immediate but limited in capabilities, computation within an FPGA provides more opportunities, but takes more time and so on. We note that by computation we refer also to computation-related services, such as access to storage and external memory. Figure 5.1 shows the equivalences between the two approaches.

In-network computing has been successfully adopted for caching use cases, but switches can cache only in the order of 64k [110]-128k entries, while applications such as memcached require millions to billions of unique keys [144]. Applications that require complex mathematical operations, ranging from multiplication to exponents and logarithms, are also divorced from switches. While look up tables can be used as an alternative to some operations [145], the resources required may be too extensive. For example, the multiplication of two 16bit operands will require a table of 2^{32} entries.

Using FPGAs as a second level of computation behind switches attends to many of these concerns. FPGAs have been extensively used as accelerators for scientific applications [146], meaning that their computations capabilities are not only more extensive than switches, but also more mature. Furthermore, FPGA's interfaces are configurable, allowing the FPGA to serve as a control and management layer between a switch and a memory or a storage device, providing access to additional resources. Moreover, many switch-boxes already include an FPGA, serving as a management device or, for example, for co-processing statistics.

Cost of Compute Hierarchy. There are, however, two prices to pay when using FPGA. First, the FPGA uses one (or more) of the switch ports. These ports achieve much higher data transfer rate than a PCI-express interface between the switch and a local CPU, but decrease the overall available port count. We argue that this is a better choice than dedicating switch I/O to memory or storage interfaces, as the number of pins required is smaller (per transfer rate), and the same switch-silicon flexibility supports a large number of configurations without design or manufacturing changes.

The second price, which can not be redeemed, is latency. While the latency through a switch is sub-microsecond, going through an FPGA and back, doubles or more the latency within a switch-box. A packet goes not only through the switch pipeline, but also to the FPGA and back, and potentially through part or all of the switch pipeline again, before going out to the user. We argue, however, that this trade-off of latency for computation, is beneficial.

Performance. Applications are offloaded to the network only when there is a potential performance benefit. For example, if a caching application can serve 100× requests per second running within the network. While unlikely, it may be, in certain designs, that going through the FPGA will end up with the same request-reply latency as sending the request to a host. We assert, however, that if for the same overall latency, the throughput gain is 100×, then there is a benefit.

Network Latency. Having an FPGA attached to a switch provides the lowest bound on the latency of a network-attached accelerator. Using a remote smart-NIC or FPGA means that any computation is further delayed by the latency of the connecting link or switches along the way. While this may be negligible in a ToR-to-NIC configuration, it is not if the processing ToR is located next to the user (e.g., edge termination), while the smartNIC is located next to the server, on the other side of the network. Furthermore, our scheme means that packets from the switch to the FPGA are not delayed by congestion within the network.

Network Load. Any computation sourced at a switch and offloaded to a remote FPGA or smartNIC increases the load on the network. Consider the case where a switch capable of processing ten billion requests a second has 1% cache miss ratio, and that this 1% is looked up remotely. That means a hundred million extra packets through the network every second, in each direction (query and result), potentially leading to congestion. Using an FPGA attached to the switch means that the network does not experience the extra load.

Power Efficiency. Tokusashi et al. [39] demonstrated that running an application on an FPGA is more power efficient than running on a host, and running on a switch is more power efficient than running on FPGA. Combining the switch and FPGA may increase the power consumption of a single switch box, but increases the overall end-to-end power efficiency of a data center, when considered holistically.

5.3 Challenges and Observations

Building a heterogeneous system with programmable network ASICs and FPGAs introduces a number of challenges. In this section we introduce some of the more expected and some of the less expected challenges.

Maintaining Performance. With 1-2 orders of magnitude gap in performance between switch-ASIC and FPGA, it must be guaranteed that switch throughput is not throttled by FPGA throughput. This means, for example, that a design where all incoming packets to the switch go through the FPGA is infeasible, even if there is sufficient I/O on both devices. Two potential mitigation techniques are i) compute on only a fraction of packets ii) send only a fraction of the data to the FPGA (e.g. 64B for every 8KB of data).

Matching Traffic Rates. Matching traffic rates is more than just ensuring that no performance is lost. The aim is find the maximum amount of traffic that can be processed by the FPGA, without leading to congestion. While in some cases this rate may be fixed, in others it may be workload dependant (e.g., with packet size distribution), and the switch needs to adapt to such dynamic conditions.

Blocking Architectures. Switch-ASICs achieve high performance through pipelining, moving the data all the time without stalling. In contrast, FPGA (often) achieve performance through parallelism, operating on multiple data-units at the same time, while each unit may be delayed. This conflict in approaches can lead both to performance loss and to congestion indications from the FPGA to the switch, which need to be avoided. An heterogeneous switch-FPGA architecture needs to be non-blocking, allowing the switch side not to stall on packets, while still allowing the FPGA to benefit from parallelism.

Merging Traffic. Consider a scenario where some packets are processed in the switch, and some in the FPGA. A combined design needs to ensure that the division of work is transparent to the user. This means, for example, that reordering of packets, where reordering is harmful, should not happen. This challenge becomes harder if there are also latency constraints, or if one tried to avoid buffering packets in the switch (e.g., due to memory limitations).

Encoding Information. It is not straight forward to send and receive packets between a switch and an FPGA. First, as it is desired to increase FPGA throughput by minimizing the processing required from it (e.g., header parsing) and as the interconnect between the switch and FPGA is a bottleneck. Second, as the switch needs to be able to distinguish between new incoming traffic and packets returning from the FPGA. Potential solutions include traffic encapsulation, bus adaptation and revised switch-traffic routing.

Multiple Switch Pipelines. Many switch-ASIC today use multiple pipelines to achieve target performance. This, however, creates a challenge connecting to FPGA, as two options exist. First, each pipeline can connect independently, through a dedicated port, to the FPGA, thus maintaining the context of a transaction, but wasting ports. Alternatively, all pipelines can connect to the FPGA through a single egress port, assuming the required bandwidth is sufficient, but the FPGA needs to know from which pipelinethe request arrived, and needs to return the reply to the right pipeline. None of this options is ideal.

Processing Results. Processing computations' results by the switch is not trivial. First, as the switch may need to associate the results with previous requests. Second, as we would like to avoid processing the same packet twice. Not only to save resources and maintain throughput, but also to correctly process data and avoid running the same computation over and over again.

Design and Engineering. A heterogeneous architecture also brings engineering challenges, such as the needs to integrate two parts of a program running on different types of devices. This means, for one, finding the optimal division of processing between switch and FPGA (which may not be trivial). It may also slow down the design cycles, as different languages and design tools are used by each component in the system.

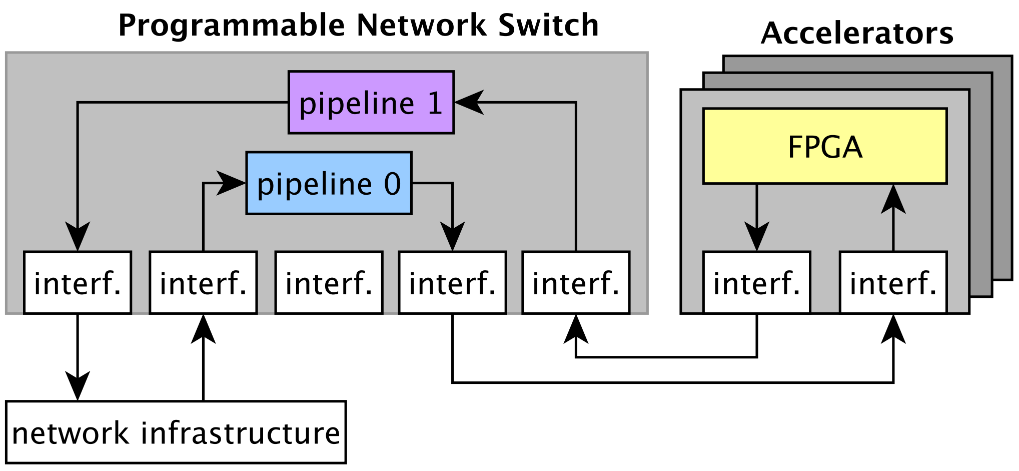


Figure 5.2. FPGA-accelerated network computing platform.

5.4 Architecture Design

Figure 5.2 illustrates our proposed architecture for a switch pipeline that combines a programmable ASIC with FPGAs. We assume a single switch is connected to a variable number of FPGAs. Below, we discuss key design decisions, along with trade-offs and possible alternatives. These decisions include: the connectivity between the ASIC and the FPGAs; the communication protocol between the ASIC and FPGAs; state management; the utilization of pipe; the allocation of data plane program logic; and potential automation.

Connectivity. The first, basic design question is how to connect the FPGAs to the switch. An FPGA will typically have a smaller number of network interfaces than a switch. For example, in our prototype, we used an NetFPGA SUME board with 4 network interfaces and a Barefoot Networks 32 port Tofino switch. The switch is divided into separate pipes. Each pipe may be programmed independently, and state is not shared between pipes.

One possible connectivity is to have a single FPGA per-pipe. This approach would logically partition the device into two independent pipes with separate FPGA accelerators. Switching would be done at a single point, the traffic manager. An alternative approach would have each FPGA connected to every pipe. This would allow the FPGA to receive input from any pipe and send it to any pipe.

Our design uses the latter approach, as it offers more flexibility. However, the former approach could potentially allow greater throughput.

Additional solutions include connecting chains of FPGA-based accelerators to the ASIC. Although such topologies increase the latency of the system, they provide speedup to applications that, requiring an excessive amount of resources, do not fit into a single FPGA.

Communication. The next design choice is to design the communication protocol between ASIC and FPGA. On the one hand, packets could be forwarded as-is from the ASIC to an FPGA. However, this would likely be inefficient, since the FPGA does not necessarily need all of that information. On the other hand, we could strip information from the header, by removing unused protocols. For example, we don't need IP header if we are not doing IP forwarding in the switch. Although that might allow us to process packets faster at the FPGA, we would then need to add those headers again after we return from the ASIC, which would introduce additional design choices. First, we need to store information about the stripped headers inside the ASIC. Ideally, programmable switch ASICs should provide a memory buffer, shared among all the pipelines. However, commercially available ASICs do not provide this feature. Therefore, we have to send processed packets back to the same pipeline that holds the data about the stripped headers. Second, assuming non-trivial topologies, we need to design a protocol for routing packets inside the platform. We also need to instruct ASIC's pipes for generating and managing such headers. Although introducing an internal protocol increases both the complexity and the latency of the system, it enables more flexible configurations and the support for more complex applications, through programmable internal routing. We believe a minimal set of internal header fields should include at least a source field, a destination field and a type field, similarly to the ethernet header. However, the format of the internal header depends both on the application and on the topology implemented in the system. Additional fields and header formats could be supported depending on the specific application.

State Management. Related to the topology of the connectivity between the ASICs and FPGAs is the question of how to manage state. There are at least two design decisions that must be made. The first question is how to divide state between the FPGAs and ASIC. For example, one might maintain all state in the ASIC, and treat the FPGAs as stateless computations. This would simplify the design, but under-utilizes resources. On the other hand, one could store some state on the ASIC and some on the FPGA. This leads to the second question. Assuming state is kept on FPGAs, should the state be partitioned (i.e., each FPGA has separate portions of the data) or replicated (i.e., each FPGA has the same state). The implications of these different design choices are mostly application-dependent. In case the state of the application can be fit to the ASIC, FPGAs could be programmed as stateless accelerators, for speeding-up the most compute-intensive sub-tasks of the application. Distributing the state among the ASIC and FPGAs would be necessary in case either the amount of memory provided by the ASIC is not enough, or the state cannot be fitted to the PISA-based organisation of the memory on the ASIC. In this case, replicating the data among the FPGAs would provide benefits to applications that require state sharing, coherency, and redundancy at the cost of an increased overhead on both communication and processing. On the contrary, state partitioning would enable task parallelisation over the various portions of the state and even running different applications on the same platform, at the same time, thus enabling function virtualisation. However, this would make the management of the system more complex.

In principle, our platform has been designed for supporting all the possible configurations discussed above, since the way the state is shared in the system depends only on the program running on the ASIC and on the architecture loaded on the FPGAs. However, we acknowledge the potential benefit in sharing the state among the ASIC and the FPGAs, since FPGAs do not impose any specific memory architecture, as ASICs do, and provide more memory resources. We believe the choice in between partitioning and replicating the state among the FPGAs is purely application-dependent.

Pipe Utilization. In our design, each pipeline of the programmable switch is reserved to process a separate stream of packets. In particular, considering a two-pipelines switch, a pipeline is assigned to the input network stream, the other processes the output stream. We adopted this solution, since it makes modularity and scalability easier and maximizes the throughput provided by each pipeline. However, we believe different strategies are possible, depending on the specific traffic to be processed and on the topology implemented by the system. Having different streams mapped to separate pipelines, enables assigning a separate subset of the available network interfaces to each pipeline. Therefore, we can implement different connection topologies, by simply changing the physical connections that interface our system to the network, to the accelerators and to other systems. Assigning input/output streams to separate pipelines, guarantees that each stream is processed at the maximum (deterministic) performance provided by that pipeline.

Logic Allocation. Allocating logic to our network computing platform consists in assigning specific functions to the most suitable components of the system. Although this process is strongly application-dependent, we provide some guidelines, based on our experience. We also discuss a concrete example in section 5.8. First, developers need to separate classical network tasks from the rest of the functions to be implemented in the hardware. Since the programmable switch has been designed for accelerating network functions, it is the ideal target for such tasks. In contrast, FPGA-based accelerators can implement all the functionalities that do not fit into the switch, including non-networking tasks and intensive computations. Moreover, the accelerators can implement many different computing architectures and can provide access to more capable memory resources. We also suggest exploiting the resources provided by the switch as much as possible, instead of concentrating all the compute-intensive tasks into the FPGAs.

Process Automation. Although automatic logic allocation would greatly reduce developers' efforts, we believe automation could be applied only to a portion of the process, due both to the heterogeneity of the architecture and to the wide variety of the algorithms it can accelerate. Taking inspiration from similar efforts targeting different architectures and based on our experience with FPGA design tools, we envision the implementation of an allocation process directed by the developers through pragmas (or alternative keywords) that identify specific tasks in the code, thus guiding the tool in allocating the various functional blocks to the most suitable processing elements in the hardware.

Our FPGA-accelerated network computing platform is both scalable and flexible by design. The system scales with the network, as programmable switches do, and each available network channel can be used either for interfacing the switch with the network infrastructure, or to connect the accelerators. Flexibility is provided by both the programmable switch and the FPGA-based accelerators. Accelerators can be rearranged in different configurations, according to the need, and traffic can be forwarded to specific network interfaces for selecting the sequence of operations to be computed by the accelerators connected to those interfaces.

5.5 Use Cases

The proposed network computing platform can leverage a number of use cases, ranging from well-known tasks, to future In-Network Computing applications. Although being able to efficiently run classical network functions, such as load-balancing, NAT and congestion control, our solution can speed-up both applications designed for being computed in the network and functions usually run on hardware accelerators. In-network applications that would benefit from our platform include both tasks requiring access to large amounts of memory, such as caching, coordination and distributed file system management, and compute-intensive functions, namely stream processing, graph processing, query processing, data analytics and, as discussed in this chapter, storage functions. Although researchers have already demonstrated efficient implementations of most of these use cases in the network, their approaches have been limited by the capabilities of programmable network ASICs. Therefore, we believe porting existing In-Network Computing applications to our platform, would provide them with the resources they need for achieving their best performance. Similarly, implementing classical hardware functions with our architecture, would provide them with network resources, thus enabling scalability, better interfacing and faster input/output communication. Intensive applications that could be accelerated by our network platform include operations for artificial intelligence and machine learning, computations for virtual/augmented/mixed reality, video processing/encoding/editing, digital signal processing and even industrial networking applications, that combine control and communication.

5.6 Prototype Implementation

The prototype architecture is composed of a Barefoot Tofino switch, connected to two NetFPGA SUME cards. The functionality implemented by the prototype architecture is composed of two main sub-functions: interfacing and forwarding and hardware acceleration.

Interfacing and Forwarding. Thanks to its programmable nature, the Barefoot Tofino switch implements both network interfacing and traffic forwarding among the various components of the prototype architecture. The switch, programmed in P4 [1], removes headers from incoming packets, forwards the payload to the accelerators, adds headers to the processed payload and forwards the processed packets to the Network Interface Card. Each of the two pipelines of the switch, programmed with different programs, implements half of the network interfacing functionality. Pipeline 0 elaborates incoming test traffic, pipeline 1 elaborates traffic processed by the accelerators. Due to limitations in the control plane infrastructure of the Barefoot Tofino switch, we were unable to implement more complex functionalities inside the switch. We believe the switch will have a more active role in accelerating specific parts of the computation once the control plane infrastructure would be able to support pipelines programmed with different functionalities.

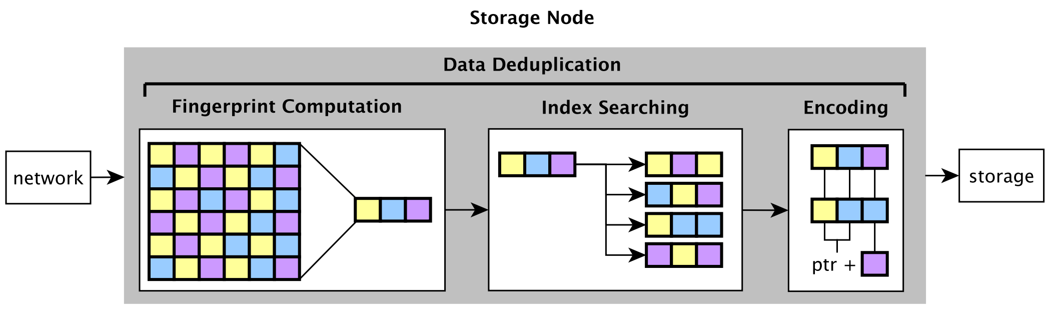


Figure 5.3. Similarity-based deduplication.

Hardware Acceleration. Hardware Acceleration is performed by four accelerator modules (AM), programmed on two NetFPGA SUME cards. Each module is connected to two of the four network interfaces of the NetFPGA SUME card and processes one of the four 10Gpbs packet streams coming from the Barefoot Tofino switch. Although being unable to saturate the bandwidth provided by the switch, this configuration can fully exploit the four 10Gbps channels of the NetFPGA SUME card that generates the test packets. Each of the four accelerator modules can be programmed independently through either hardware description languages or high-level synthesis flows. In order to keep our prototype architecture as simple and flexible as possible, we decided to use only two FPGA-based accelerators. However, the switch provides tens of network channels, that would allow us to connect many more accelerators of different types.

5.7 Evaluation

A class of applications that is becoming increasingly popular in the In-Network Computing field is storage functions acceleration: researchers have already started employing programmable network switches for accelerating functions such as fault-tolerant distributed systems [147] and erasure coding. Storage functions acceleration provides many interesting applications for evaluating our FPGA-accelerated network computing platform, since it leverages both network functions and hardware acceleration. Indeed, datacenter storage systems usually implement at least part of their functions in the hardware and exchange data with the rest of the infrastructure through the network. In order to evaluate the proposed solution, we implemented a prototype that demonstrates accelerating a storage function in the network.

5.7.1 Test Application

Data deduplication is a technique used for eliminating duplicate copies in a data set for saving storage resources. An implementation of data deduplication, called "similarity-based deduplication" [148], consists in a sequence of three phases, shown in Figure 5.3. The first phase computes a hash, called "fingerprint" for each block in the data stream to be deduplicated. Computed fingerprints are then compared to a set of reference fingerprints for identifying similarities between the new blocks and the ones already stored, through a process called "index searching". Finally, in case a match is found, only the difference in between the two matching blocks is stored, by encoding it with a pointer to the reference block. In case no match is found, the new block is entirely stored.

Data deduplication is typically computed at the storage node, just before the data is written to the storage devices. This approach, known as "target-based deduplication" is inefficient in two ways. First, it requires computing resources at the storage node to be reserved for data deduplication. Second, it increases the overall storage latency, by introducing an additional processing phase before accessing the storage medium. Leveraging network programmability, similarity-based deduplication can be improved by computing data deduplication while the data moves through the network, thus saving computing resources at the storage node and reducing the overall latency of the system. We call this new approach "network-based deduplication", since the network implements the deduplication functions, instead of the storage node.

Our proof-of-concept implementation only accelerate the first phase of similarity-based deduplication–the fingerprint computation. However, we believe that both the second and the third phases of similarity-based deduplication could be easily accelerated, without making substantial changes to the underlying prototype architecture.

5.7.2 Test Setup

The setup for the test, shown in figure 5.4, is composed of the prototype architecture, with the addition of a host computer that provides packet generation, packet collection and test management. In the following, we describe the main components of the test setup.

Fingerprint Module. The prototype architecture is configured for accelerating fingerprint computation, through specific fingerprint modules loaded to the NetFPGA SUME cards. Each fingerprint module, shown in Figure 5.5, includes four parallel Rabin_16 submodules for computing fingerprints over the incom-

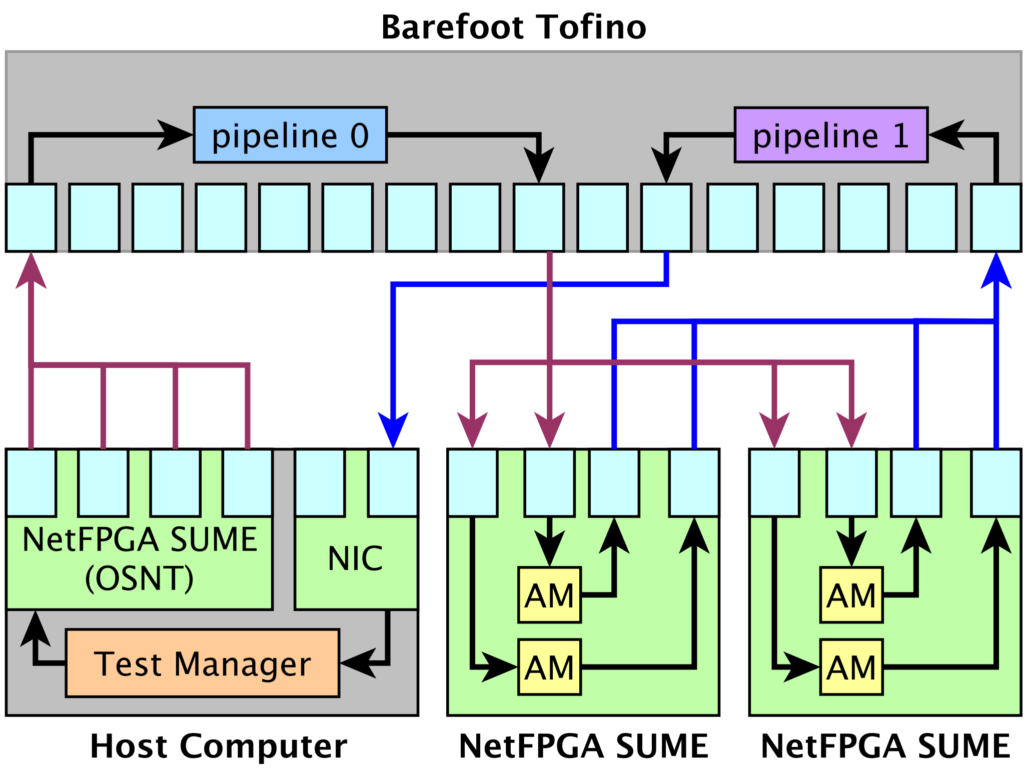


Figure 5.4. Test setup.

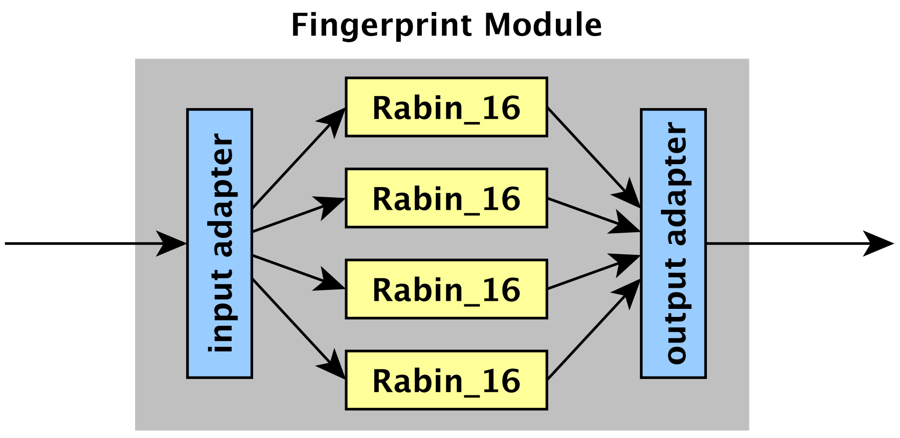


Figure 5.5. Architecture of the fingerprint module.

ing blocks of data. We took the submodules, that implement Rabin_16 algorithm [149], from a past project done at Western Digital Research [115] and we used them as basic blocks for building a fingerprint module. We implemented two adapter modules inside each fingerprint module, both for mapping the input traffic to the four submodules and for combining the generated fingerprints into a single output result. The whole fingerprint module has been implemented in Verilog, targeting the NetFPGA SUME platform [4].

Packet Generation. A NetFPGA SUME card, installed into the host computer and programmed with OSNT [24], is used for generating a stream of test packets that simulates traffic coming from a network infrastructure. Packets are injected into the Barefoot Tofino switch through four 10Gbps channels in parallel, by combining the four network interfaces of the NetFPGA SUME card. Using OSNT allows to generate test packets at line-rate, based on user-provided packet traces. Test packets can be populated with a customisable sequence of headers and with a test payload, that is processed by the FPGA-based accelerators.

Packet Collection. Processed traffic is forwarded to a NIC, installed into the host computer. The four 10Gbps streams of processed packets are aggregated into a single 40Gbps stream, before being forwarded by the programmable switch. Although being capable to process 100Gbps traffic, the NIC runs at 40Gbps in this implementation. Captured traffic is made available to the test management application, for further elaboration.

Test Management. The test manager, running on the host computer, supervises the test operations. First, it loads test packet traces, provided by the users, to the OSNT packet generator. Then, it triggers the generation of the test traffic through the NetFPGA SUME card and start listening on the NIC, waiting for the processed to come back. Generated packets go through the first pipeline of the switch and the extracted payloads are forwarded to the two NetFPGA SUME cards programmed with the fingerprint modules. Processed payloads are then sent back to the switch and traverse the second pipeline, where output packets are assembled and aggregated into a single output stream. Once the stream reaches the host computer, its is captured by the NIC and provided to the test manager for further elaboration. The test management could be programmed for performing elementary checks, such as comparing the fingerprints generated by the hardware with the result of an equivalent software implementation. More advanced functionalities include simulating a complete data-deduplication operation, by offloading some tasks, such as fingerprint computation, to our prototype architecture.

5.7.3 Test Results

We tested the prototype architecture with a sample stream of data, consisting in a sequence of packets, each including both a ethernet header, for interfacing with the switch over the network, and a payload, used for computing the fingerprints. Test data is generated at line-rate through the NetFPGA SUME card programmed with OSNT. A ethernet header is attached to the computed fingerprints before forwarding them to the host server, for further processing.

We asses the performance of the system through the most common metrics used for evaluating hardware architectures: latency, throughput, resource utilisation and power consumption. Although we were unable to collect accurate results for the Tofino switch, due to the closed-source nature of Barefoot's products, we provide publicly-available performance values, that set an upper bound to the performance of our prototype. In contrast, we were able to collect very precise performance results for the FPGA accelerators, since the NetFPGA SUME architecture is open-source and allows to perform cycle-accurate measurements.

Latency. Each Rabin_16 module introduces a latency of 512 clock cycles. Considering that the clock frequency of the FPGA-based accelerators is 200MHz, the latency introduced by each Rabin_16 module is 2.56μs. Assuming that the latency contribution of the network interfaces is $\sim$1μs (roundtrip) and considering the sub-microsecond latency added by the programmable switch, the total latency of the system is $\sim$5.5μs.

Throughput. Pushing the packet generator to its maximum throughput (40Gbps), each Rabin_16 module is able to generate a new fingerprint every 2.56μs. Therefore, it can sustain a throughput of 390.6 thousands fingerprints per second (Kfps). Since each fingerprint module includes four Rabin_16 modules in parallel, its throughput is equal to the throughput of the Rabin_16 module, multiplied by four, that is 1.56Mfps. In our prototype, we used two NetFPGA SUME cards, each implementing two fingerprint modules in parallel, thus providing a total throughput of 6.25Mfps. The overall performance of the system is limited by the maximum throughput that the FPGA-based accelerators can sustain. Indeed, each of the two NetFPGA SUME cards can process up to 10Gbps per network channel and we used two cards, for a total aggregated throughput of 40Gbps.

Resource Utilisation. As shown in table 5.1, the impact on FPGA resources is very limited. The fingerprint modules take less than 4% of the resources on each NetFPGA SUME card. If we include the network interfaces, the impact on the FPGA resources is less than 10% on average, thus leaving plenty of space for accelerating additional functionalities in the hardware. Only a fraction of

Resource	NetFPGA SUME	Xilinx VCU1525
LUTs	16.5K (3.8%)	33K (2.7%)
FFs	13.3K (1.5%)	26.7K (1.1%)
BRAMs	0 (0%)	0 (0%)

Table 5.1. Resource utilisation on NetFPGA SUME and Xilinx VCU1525 (one card, not including the network interfaces).

the available resources of the switch is used in this implementation: each of the two pipelines employs a table and two actions. All the remaining resources are available for implementing more complex functionalities on the switch.

Power Consumption. The power consumed by each of the two FPGA-based accelerators is ~6.5W. Due to limitations in Barefoot's software, we were unable to measure the power consumption of the Tofino chip alone. However, the overall power consumption of the switch, measured at the plug, is ~100W.

Alternative FPGAs. We port our design to the Xilinx VCU1525 platform, based on Ultrascale+ FPGA and equipped with two 100Gbps ports. To match incoming data rates, we use eight Rabin_16 modules (×2 increase), a data path width of 1024b and clock frequency of 322MHz. This leads to latency decrease of 62.5% per Rabin_16 module, to 1.6μs, and to end-to-end system latency of ~4.5μs. Each module's throughput is increased by 60%, to 625Kfps. As shown in Table 5.1, the impact on FPGA resources is negligible, below 3%, thus leaving plenty of space for implementing additional functionalities.

5.8 Discussion

Further Acceleration of Deduplication. The proposed solution is based on the trade-off between latency and computing power: attaching the FPGA-based accelerators to the programmable switch, increases the overall latency of the system. In contrast, having the switch implementing portions of the computation, instead of only forwarding packets, would mitigate the impact on latency. Unfortunately, we were unable to implement more complex functionalities into the switch, due to limitations in Barefoot's software.

However, we propose a mapping of the functionalities of the deduplication use case that will be feasible once the limitations in Barefoot's software are fixed.

Our approach consists in implementing both fingerprint computation and encoding in the accelerators, though keeping the two functions separated, while programming the index searching phase into the switch, in addition to all the network-specific functionalities.

Let's assume the switch sits at the interface among the network infrastructure, the accelerators and the storage node, similarly to our prototype architecture. Data to be stored is initially processed by the switch, that implements protocol conversion and load-balances the traffic going to the accelerators. Traffic coming from the switch is forwarded to a subset of the available accelerators, that implements fingerprint computation. The generated fingerprints are sent back to the switch, that aggregates the processed data and implements index searching through match-action stages. Depending on the outcome of the match, traffic is forwarded either to the storage node, or to the accelerators that implement encoding. In the latter case, encoded data is sent back to the switch, that provides protocol conversion and forwarding to the storage node.

Sharing Data Among Pipelines. Each pipeline in the Tofino switch is independent from the others and data cannot be shared among pipelines. Due to this constraint, we were unable to share header data between input packets and output packets. Therefore, we decided to populate the headers of the output packets with some pre-defined, fixed data. A workaround for this limitation needs to be found to make routing flexible, until data sharing among pipelines will be available in programmable network switches.

Bridging the Gap Between ASICs and FPGA. There are three important advantages to ASICs over FPGA: more high speed I/O, higher clock frequency and wider internal bus widths. These lead in turn to the higher ASIC throughput. However, for computation purposes, the FPGA does not need to match those. It only needs to match the computation throughput requirements of the data arriving on a given port. With a tailored architecture, this is feasible on FPGA. The requirement thus becomes being able in the switch ASIC to partition and load balance computations across FPGA ports, which is an addressable requirement.

Scalability. Although we were unable to test the scalability of the proposed architecture through more complex topologies, we believe scalability, that is granted by the programmable switch, will not be a limiting factor for our solution.

Additional Limitations. Not all the challenges described in Section 5.3 are applicable for all applications. In this chapter we did not attend to some of the challenges, including matching traffic rates, and design and engineering. We leave these challenges to future work, using different use cases. We also don't discuss management aspects of the system, such as master-slave relationship and bootstrapping. Our experience shows these will be system specific.

5.9 Chapter Summary

This chapter presented a flexible and scalable network computing platform that extends programmable switches with FPGA-based accelerators. It then presented a prototype architecture, implemented using a Barefoot Tofino switch and two NetFPGA SUME cards. Evaluation demonstrated the feasibility of the proposed approach and the potential benefits it could provide to network computing applications. We believe our pioneering effort in making new programmable networks heterogeneous will pave the way to many future research projects in this field.

Chapter 6

Finding Hard-to-Find Data Plane Bugs with a PTA

In the previous chapter we provided a solution for balancing the trade-off between performance and expressiveness of network hardware, thus enabling new classes of applications to be accelerated in the network. This chapter addresses the difficulties software developers face in finding and fixing bugs in such network-accelerated applications, by making testing programmable network hardware simpler. The research project described in this chapter is the lead contribution to this thesis work.

The increasing trend towards programmable data planes is having a profound impact on the field of networking, as adding features to the network can now be done at the speed of software development, as opposed to the costly and time consuming hardware design of new switching silicon. While increased programmability reduces development time, quick iterations on software can also result in bugs. Moreover, the diversity of applications for which programmable network hardware is being used is growing and researchers have begun to co-opt programmable NICs or ASICs for non-networking use-cases.

There has been also significant interest in the languages used to program these devices, as they provide higher-level abstractions, compared to traditional hardware design languages, and can be ported to several hardware platforms. One consequence of this portability is that developers can write programs in a single language that can run on heterogeneous devices.

In short, we see that developers are quickly iterating on software programs; that there is a great diversity of programs being implemented; and there are a large number of compilers being developed that are still in their infancy. Given the situation, it should come as no surprise that there are a lot of bugs.

Unfortunately, finding and fixing those bugs can be difficult, as testing hardware is significantly different from testing software and imposes a unique set of challenges. For addressing these challenges, this chapter provides a taxonomy of bugs that affect the network data plane and, based on the taxonomy, introduces PTA, a portable test architecture, that leverages both P4 language and hardware design. The key idea behind PTA is to dedicate a portion of the resources in programmable network hardware for testing. PTA's basic architecture is enriched with both a specialised version, that maximises flexibility through configuration, and a automated translation flow that implements the integration with a formal language verification tool. The chapter presents two prototype implementations of PTA, targeting a FPGA-based platform and a programmable network ASIC. Evaluation covers several bugs found using PTA and demonstrates the feasibility of our approach: PTA is able to find bugs in network devices at line rate, both in a manual and in an automated way, with a negligible impact on hardware resources.

6.1 Finding Bugs in Network Hardware

Bugs in network hardware can cause tremendous problems, resulting in great financial loss, security breaches, or significant downtime for essential services. Unfortunately, despite extensive testing, these bugs can be very difficult to find [150]. As an example, imagine that a device drops packets when the input traffic exceeds a certain rate. How would we find and diagnose this bug? This sounds like it would be a simple bug to catch. After all, we would certainly notice packet drops. In reality, the bug—which we found in the NetFPGA reference projects—was not discovered for more than a decade after the platform had been introduced.

The root cause of the bug was that the input arbiter in the design was not work-conserving, i.e., packets were held in an input queue even when the output was idle. However, the bug did not reveal itself on the NetFPGA 1G board with 4×1Gpbs interfaces, or on the SUME board with 4×10Gbps interfaces. It was only revealed when the design was ported to a 2×100Gbps Xilinx Alveo board. The bug in the design was passed from one generation to the next, and the capacity of the network interface masked the defect in the internal design.

So, how could we have found and fixed this bug sooner? There are a few immediate observations that we can make.

First, the bug only appears when the traffic rate exceeded a threshold, in this case, ∼40Gbps aggregate throughput on SUME. So, software-based approaches like simulation or emulation, which can slow the execution of a program by a factor of 10^6 [21], would not help. Instead, we need a test framework that can generate and receive traffic at line rate.

Second, finding this bug requires *internal* access to the data plane. Even if we could externally generate and send traffic to the device under test at the target rate (e.g., using an Ixia [22] or Spirent [23] platform), we need a way to distinguish a limitation of the network interface from the inefficient implementation of the input arbiter.

Third, we see that the same reference design was used on several hardware targets, including NetFPGA 1G, 10G, SUME, and Xilinx Alveo. Writing tests is time intensive, and having to repeat test-writing efforts for each target would be onerous. Just as the reference design can be ported across targets, we want the tests to be portable across hardware targets.

Although we have focused on an FPGA in this example, similar bugs and observations hold for programmable ASICs, although at greater scale. Consider trying to replicate the same test scenario on a Barefoot Networks' Tofino 2 ASIC, which has 128×100G ports.

6.2 Testing with Programmable Network Hardware

Using programmable network hardware as testing devices, rather than forwarding devices, presents a significant challenge, because testing and forwarding are fundamentally different. In particular, we identify three, high-level challenges: (i) active vs. reactive logic, (ii) dynamic processing behavior, and (iii) portability.

First, at the most basic level, forwarding devices are reactive, meaning that they execute logic only on the arrival of an incoming packet. In contrast, testing is an active process. A tester generates test stimuli in the form of test packets, and then checks a post-condition.

Second, testing devices require much more flexibility than forwarding devices. When used for forwarding, the data plane functionality of programmable NICs and switches only changes in limited ways, e.g., it might forward packets out a different port, depending on control plane configurations. But, the forwarding pipeline is not altered during operation.

In contrast, exhaustive testing often requires significant adaptation and permutation, dynamically changing the behavior depending on the needs of the test. As an example, imagine that we want to generate a variety of packets with different header sizes, similar to how Dumitru et al. [151] check for security exploits. Changing the data plane implementation of the test program for every permutation would result in significant overhead, in terms of compilation and installation, which can take hours on some platforms.

Third, the test architecture must be portable across a range of heterogeneous target devices. To provide portability, we need to identify a set of abstractions that are flexible and powerful enough to test for a variety of possible data plane bugs, but can be generally implemented on a range of devices.

6.3 Data Plane Bug Taxonomy

There is a wide range of bugs that can occur in network devices. These bugs can be due to incorrect program logic (i.e., functional bugs); or due to problems in compiler or target hardware architecture. Below, we provide a taxonomy of the types of bugs that a test framework must be able to detect. These error types provide requirements that motivate the design of PTA. Note that although PTA can be used to test both fixed-function and programmable hardware, our taxonomy highlights bugs that may be unique to programmable hardware (e.g., compiler bugs), and may not be comprehensive.

Functional Bugs. A functional bug is one in which the functionality provided by the network device is not the same as the functionality intended by the programmer. Functional bugs can occur in both the data plane and in the control plane. An example data plane bug would be not supporting IPv6 headers where such functionality was supposed to be supported. An example control plane bug would be not filling all the required entries in a given size table.

Performance Bugs. Performance bugs are related to aspects such as the maximum throughput or packet rate of a certain design, how certain packet sizes affect the throughput, whether congestion control is handled properly, and more. For performance testing, the user must be able to continuously fill the pipeline with packets of a certain size and check that no packets are dropped or lost at the output. Another performance aspect is the ability to mix packet sizes in explicit ways, which exercise different parts of a design (e.g., programmable data plane, schedulers, memory access).

Compiler Bugs. Although compilers are tested with scrutiny, there may be bugs. There are at least two classes of compiler bugs. The first class of errors regards functionality bugs, e.g., where a language feature is supported but the implementation is missing, or the functionality is implemented incorrectly. A second class of errors covers the compliance with the language specification.

Under-specification Bugs. The extent of a programming language definition, and the diversity between target platforms, lead to cases where language specification is not detailed, either intentionally [48] or not. This can lead to unexpected or unintended behaviours, for example, if the specification does not detail whether the initialization of a header should be to zero, or if can remain un-populated and random.

Architecture Bugs. Similar programs may target different data plane architectures, and even perfect programs may be susceptible to bugs in the underlying device architectures. One immediate class of such device architecture limitations is access hazards to tables, such as read-after-write. A second class of bugs uncovers limitations of the data plane architecture, such as a proprietary module (e.g., an extern) that is not responding within the expected time. A sub-class of bugs has to do with the integration of different modules in the architecture, such as caused by a mismatch in the connection of interfaces.

Security Vulnerabilities. Network devices can suffer from security vulnerabilities just like any other device, and programmable network devices introduce new threat vectors. A test framework should allow users to quickly and efficiently test a large number of such security threats. One such example would be looking for the "Meltdown" equivalent of a programmable data plane: can you craft a packet that would allow you to read the contents of previous packets, various tables, or memories? Another example is backdoors in the program, whether in the original users code or introduced as a by-product of the compilation process. The hardware test can reduce the security risk by testing the deployed program as it runs on the platform.

Comparing Designs. Programs can be written in several different ways, or in several different languages. The test framework should be language independent. Moreover, one should be able to use the same tests to benchmark the performance, functionality, and limitations of seemingly similar designs, and to pick the most suitable solution.

Coverage of Existing Solutions. There is likely no single tool that can catch all of these bugs. Rather, to have confidence in a network, operators need an array of solutions at their disposal. Recent work on formal verification [117, 116] leverages the fact that data plane programs function as specifications which can be checked. While these tools show great promise for catching functional bugs, without a formally verified compiler and additional hardware verification, they offer limited coverage. Simulation and emulation tools (e.g., [152, 153, 154]) vary in their fidelity to the hardware, providing only partial coverage for performance, monitoring and architecture tests. External network testers, as already mentioned, lack a "internal view" of the device under test, and can not accurately schedule internal events such as effects of cross-traffic between ports. PTA provides a unique set of features that complement this prior work: it can perform tests that depend on high input traffic rates, and offers an internal view of the device. Moreover, the novel data plane design provides developers with an extremely flexible framework that can be used for a wide variety of test cases.

6.4 Debug Abstractions

One of the main challenges in designing PTA is identifying the core set of abstractions to support debugging. We adopt a requirement driven design process. Based on the taxonomy in the previous section, we systematically explored the necessary abstractions for each of those classes of bugs. The set of abstractions is intended to be minimal, so that it can be readily supported by diverse hardware. At the same time, it is intended to encompass the set of functions needed for testing.

To illustrate the process, we first walk through the running example—i.e., the input arbiter bug from the NetFPGA reference project from Section 6.1—before summarizing the complete set of PTA debug abstractions.

6.4.1 Requirement Driven Design

So, how might a developer find and isolate the performance bug in the input arbiter? Because the module is (incorrectly) not work conserving, we clearly need to be able to generate packets at data-path rate, creating controlled back-to-back arrival events to the arbiter.

Many bugs (e.g., functional, compiler) would depend on a particular data plane program, suggesting that the debug framework needs a method to load a data plane image. However, in this case, the bug is in the architecture of our target device, and therefore independent from the user program that we would load. To test the architecture, we need access to the low-level abstractions offered by the hardware, including the metadata bus, stateful ALUs, and any externs provided by the architecture of additional hardware modules.

We need modules to initialize and check the values of stateful elements, e.g., counters and registers. This allows us to confirm the number of packets sent and processed by the pipeline, and more generally, and application specific logic.

Finally, to detect the presence of dropped packets (again at line rate), we need a way to collect and inspect output packets.

Abstraction	Description
Load_Image	Load the image file to the target
Init_Counters	Initialise the counters in the target
Init_Registers	Initialise the registers in the target
Generate_Packets	Generate test packets
Collect_Results	Collect raw results from target's registers

Table 6.1. PTA's user-facing abstractions.

Abstraction	Description
Metadata_Bus	Layout of the metadata bus
Stateful_ALU	Architecture of the Stateful ALUs
Extern	Architecture of the Extern modules
Register_Read	Interface for reading hardware registers
Register_Write	Interface for writing hardware registers

Table 6.2. PTA's back-end abstractions.

6.4.2 Core Abstractions

We identify two classes of abstractions: user-facing abstractions (Table 6.1) and back-end abstractions (Table 6.2). User facing abstractions are used to specify the functionality of a test. Back-end abstractions represent the architecture of the target device.

User-Facing Abstractions. Users writing tests will be using user facing abstractions, similar to functions. As these abstractions are not target-specific a test will be written only once.

The abstractions are used to load the program image (`Load_Image`), initialize registers (`Init_Registers`) and counters (`Init_Counters`), and to generate and collect packets (`Generate_Packets` and `Collect_Results`).

Note that although packet generation is exposed via user-facing abstractions, they are adapted (e.g. to vary packet size, contents, and transmission rate) using back-end abstractions.

Back-End Abstractions. Back-end abstractions are used to specify a network-device target, and are called by any tests using this target device. The back-end abstractions library includes both the layout of the metadata bus (`Metadata_Bus`), that is used by PTA as a configuration channel, and the architecture specification of both stateful ALUs (`Stateful_ALU`) and extern modules (`Extern`). Since the hardware components of the framework are usually accessed through a register interface, PTA provides two additional abstractions for reading and writing registers (`Register_Read` and `Register_Write`).

6.5 A New Data Plane Architecture

To address the challenges described in Sections 6.1 and 6.2, we propose a new *data plane architecture* for data plane testing. We use the term data plane architecture in the same way that it is used in the P4 programming language [1]. It identifies the P4-programmable blocks and their data plane interfaces. Essentially, it is the contract between the data plane program and the hardware target. As a point of clarification, we also use the term *device architecture*, which refers to the different hardware blocks in a device (e.g., ports, data plane, packet buffer, etc.) and the interfaces between connected blocks.

The P4 open source community has begun to standardize a few data plane architectures, including the Portable Switch Architecture (PSA) [155] for network switches, and the Portable NIC Architecture (PNA) [156] which models NICs.

We introduce the Portable Test Architecture (PTA), that provides the following contributions:

- The requirements for PTA are derived from a taxonomy of bug types in programmable network devices and we detail bugs that we have found in commercial and open-source software using the tool.
- Driven by the requirements of the bug taxonomy, PTA offers a small but powerful set of abstractions to support debugging.
- PTA has a novel data plane design that (i) separates target-specific from target-independent components, allowing for portability, and (ii) allows users to write a test program once at compile time, but dynamically alter the behavior via dynamic re-configuration.
- PTA complements prior work on automatic test packet generation [157], fuzz testing [158], and software validation [117], by providing a framework for running workloads generated by those tools on actual hardware.

To demonstrate how PTA can be used in conjunction with existing tools, we have developed a proof-of-concept integration with P4v [117]. Users can extract assumptions and assertions from an annotated P4 program, and map them to hardware test configuration.

- PTA uses programmable network hardware for testing, which differs from traditional forwarding in key ways. We present a set of lessons we've learned and assumptions that were challenged in the design of the framework.

6.5.1 Portable Test Architecture

Based on the taxonomy of bugs, presented in section 6.3 and on the core abstractions, discussed in section 6.4, PTA is designed to provide a comprehensive hardware data plane testing solution. PTA is *programmable*, meaning that the tool can be customized to the particular testing needs of the user for a diverse set of bugs. It is also *re-configurable*, meaning that new tests can be run via dynamic re-configuration (e.g., using register access), rather than re-programming (e.g., requiring a new image file). PTA allows for *integration with existing tools*, providing prior work on automatic test packet generation [157], fuzz testing [158], and software validation [117] with a path to run on hardware. When used to test devices with programmable data planes, PTA allows *access to internal state*, providing detailed fault localization. PTA allows users to test network devices in *real time* at full line rate, and test results are reproducible. We expect PTA to be deployed out of band, i.e., in parallel to live traffic. It does not incur additional latency or otherwise alter the traffic.

6.5.1.1 Overview

Imagine that a user wants to verify that a particular data plane runs at line rate for different packet sizes. To run this test, we need three components: a packet generator, which will inject packets into the data plane; an output checker, which will assert that post-conditions hold at the end of the test; and a management component, to run the test. Each of these components are illustrated in Figure 6.1, which shows the high-level design of PTA. Note that both the generation and checker modules are implemented in hardware, while the **management component** is a set of software programs.

Even for this simple example, there are a range of parameters and scenarios to be tested: How many packets should be sent? What sizes should the packets

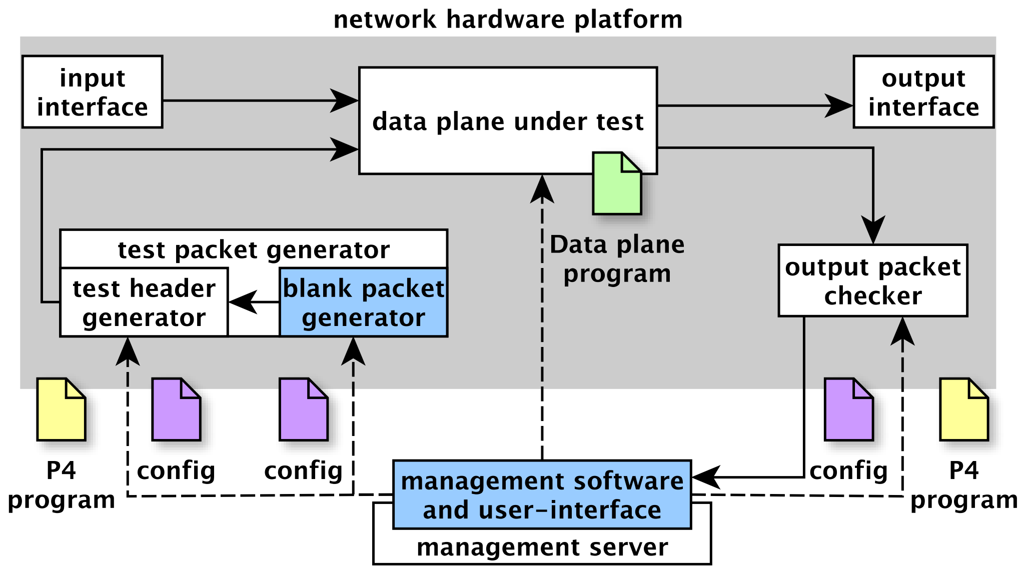


Figure 6.1. The proposed architecture: target specific infrastructure (light-blue), portable suite of P4 test programs (yellow) and test-specific configurations (purple).

be? At what rate should be packets sent? And, at what rate do we expect the output packets to arrive? What protocols are being used in the packet headers? Hand-writing tests for each of these scenarios would be tedious, and possibly error-prone.

To help reduce the burden, PTA separates tests into two parts: a programmable part and a re-configurable part. The programmable part can be thought-of as data plane specific. Users can, for example, write a program to generate packets with different protocols, and write a checker to assert that post-conditions of the test hold. The **re-configurable part** is test-specific. It is a control plane configuration of the programmable part, that allows users to change the packets sizes, sending rates, etc.

The programmable part of a test can be further divided into a target-dependent and a target-independent infrastructure. The **target-dependent** infrastructure is the device-specific implementation of key functionality (e.g., generation of blank packets). The **target-independent** infrastructure is written in P4, and controls, for example, the definition of protocol headers.

Note that although PTA's programmable parts are implemented in P4, the data plane under test need not be written in P4. PTA can be used to test data planes designed using a variety of different workflows and languages, including high level synthesis, C/C#, and HDLs (e.g., Verilog).

It is important to stress that all of the components of PTA are implemented *inside* the target network platform. This provides PTA with several important advantages. First, it allows PTA to test the data plane while avoiding the surrounding hardware, including the network interfaces. A failure of a test can guarantee that the cause is not in the interfaces but in the tested data plane. Second, it enables testing the device at line rate and at real time. Testing a device at line rate is challenging due to the cost of external traffic generators (e.g., Ixia [22]), making it outside the reach of many users. Thus, the internal data path may have a certain speed-up over the external interfaces, making it very hard to create and detect hazard scenarios such as read-after-write in two consecutive clock cycles, or certain cross-traffic scenarios that lead to consistency issues. Finally, PTA allows users to test and debug their data plane in the field, without additional equipment, and without changing the physical settings.

We discuss each of the major components of PTA in more detail below.

6.5.1.2 Target-Independent Test Infrastructure

The target-independent infrastructure of PTA is a set of P4 programs used for the packet header generation and the output checking. Both components are implemented as a sequence of match-action tables and a set of registers that change control flow. The entries in these tables are re-configurable, and provide the flexibility to support different tests.

PTA includes a set of default programs that developers may use to generate packets with standard protocols (e.g. Ethernet, IP, TCP, UDP, etc.) and to check for common conditions. Thus, for many test scenarios, users need not write any P4 code themselves. To support custom protocols and to check for data plane specific test scenarios (e.g., to generate a packet with a Paxos protocol header [111] and test for a specific post-condition), users can expand on these default programs for custom-protocols using P4.

Packet Header Generator. The packet header generator takes blank input packets, and turns them into stimulus packets injected to the data plane under test. The P4 program defines the protocols that need to populate the header and properties of the contents. Because tests are written in P4, developers can use any protocol that is implementable in P4, and can easily add custom headers, different fields, change the ordering of headers, and more. For example, an empty packet entering the test header generator will be emitted as a standard TCP/IP packet, with a certain sequence number and valid checksum. Combined with the blank packet generator, the test header generator will control packet size and contents, traffic pattern (e.g., inter-packet gap), and may even intentionally craft illegal stimulus packets. The output of the test header generator connects to the input of the data plane under test.

Output Packet Checker. The output of the data plane under test is connected to the output packet checker. The output packet checker, implemented in P4 and shown in Figure 6.1, can be programmed to expect specific values or sequences of values within the returned packets. It compares these values against the input traffic stream (e.g., to detect packet drop, reordering, or other points of failure). The stages within the checker's stages support different types of functionality, such as matching specific header fields, or comparing metadata bus values. The outcome of each check is stored in a memory. Typical types of stages include an ALU, that performs both logic and arithmetic operations over headers and metadata, and CAM and TCAM blocks that compute simple matches against header and metadata fields. The number of received packets is an example of a common functionality implemented using counters.

6.5.1.3 Target-Dependent Test Infrastructure

P4-based data planes are packet driven, and do not generate packets without a stimulus. For this reason, PTA uses a blank packets generator that creates empty packets, feeding the test packet generator's P4 pipeline. By a blank packet, we mean a packet with no header fields and no payload. The blank packet generator is target specific. For example, some ASIC switches (e.g., Barefoot Networks' Tofino) already have a built in-packet generator, while other devices (e.g., FPGA) require an dedicated implementation. Even if a packet generator already exists within the device, it varies in features and properties between devices, and is therefore target specific.

Additional target specific infrastructure is focused on the connectivity of PTA: connecting the output of the test packet generator to the data plane under test, and connecting the output of the data plane under test to the output packet checker. This connectivity depends on the hardware architecture of the device. For example, on NetFPGA, this will use 256-bit wide AXI-4 streaming buses, and the output of test packet generator is connected to NetFPGA's input arbiter. On other devices, the bus type and width will vary, as well as the connection points.

Finally, PTA includes 4 additional functions implemented in hardware that are useful for testing: random number generation, counters, time-stamping, and a method to swap fields. These are used as externs in the P4 programs.

6.5.1.4 Re-configuration

To fully explore the parameter space for a test, PTA allows users to re-configure tests. To re-configure a test, a user writes a simple test script. The script allows users to change features such as the packet rate burst length, the gap length between packet transmissions, the packet sizes, the payload sizes, etc. It also allows users to set initial meta data flags and fields.

All configurations updates are control plane changes that happen dynamically at runtime. This allows users of PTA to explore a wide variety of test scenarios without having to recompile the test programs and install a new image—which can take a long time for some targets.

6.5.1.5 Management Software and User-interface

While P4 programs define the type of packets that can be generated by PTA, the management software defines the properties of the test. It is responsible for configuring the control and data planes, triggering tests, collecting and processing results, and declaring Pass/Fail. For this purpose, PTA includes a Python test module to help manage the tests.

Test packet generation is determined by configuring the blank packet generator. This includes both information about the generated packet stream (e.g., number of packets, packet size, burst properties) as well as the input metadata accompanying the packet (e.g., path through the generation data plane, which headers to add).

Test results are specified using assertions. The management software reads from the output packet generator the results of the test (e.g. number of packets received), and compares them to the expected values set as Pass criteria.

The management software is the most frequently used component of PTA. For each target device, the target specific infrastructure will be defined only once. For each data plane under test, the P4 programs will be written only once, or will be modified only slightly. However, the actual tests of the data plane will be implemented and driven through the management software. Therefore, a user will create a large set of programs and use the software to run them. This programs tend to be platform independent, and can therefore be portable between different design, reducing design for testing load. Platform-dependent variables (e.g., number of ports, when broadcast is tested), can often be parameterised.

In our FPGA implementation, the definition and execution of a test is done through a single test script written by the user, building upon the functionality supported by the programmable hardware. The manual tests are implemented

```
1  # Import PTA test module
2  import test_mod
3
4  # Collect initial timestamp
5  start = time.time()
6
7  # ----- Packet generation: user code begin -----
8  # Program hardware
9  test_mod.load_image()
10 # Initialize hardware counters
11 test_mod.init_counters()
12 # Initialize hardware registers
13 test_mod.init_registers()
14 # Generate test packets
15 test_mod.gen_packets("100","I-0","D-100","128")
16 # Collect results#
17 test_mod.collect_results()
18 # ----- Packet generation: user code end -----
19
20 assertions_list = [
21 # ----- Assertions: user code begin
22 ("nf0_packets_out", "EQ", "100"),
23 ("reg05", "LE", "nf2_packets_out"),
24 ("oqs_dropped_nf0", "GT", "20")
25 # ----- Assertions: user code end
26 ]
27
28 # PARSE AND CHECK RESULTS
29 test_mod.parse_check(results, assertions_list)
30
31 # Collect final timestamp
32 end = time.time()
33
34 # Compute test execution time
35 print("Execution␣time:"+str(end-start)+"[s]\n")
```

Figure 6.2. Example test script.

through a Python script, that covers both the configuration of the test infrastructure, including the blank packet generation module, and the specification of the list of the assertions to be applied to the results of the test.

We briefly discuss the configuration of an example test script, shown in figure 6.2. Line 2 imports the test module, for enabling PTA-specific functions. Before starting the configuration of the test, line 5 takes the initial time-stamp, for computing the total execution time at the end of the test. Note that the initial time-stamp could be taken later, in case the test configuration phase would not be taken into account. Instruction at line 9 loads the compiled image file to the FPGA, initialises the PCIe interconnection between the hardware architecture and the management server, loads the software drivers needed to manage the network interfaces of the FPGA card and configures the control plane of the test infrastructure. Being time-consuming, this function is supposed to be called only a few times in a test script. Lines 11 and 13 clear the content of the stateful elements in the test infrastructure, by setting counters and registers to zero. This operation is necessary for two reasons: for initialising the stateful elements to a known value and for clearing intermediate test states. Line 15 configures the registers of the blank packet generator, based on a sequence of parameters specified by the user, and triggers the generation of the blank packets, thus starting the execution of the test. In the example test script, the specified parameters configure the generation of a single burst of 100 128Byte-wide packets, with a inter-packet gap of 100 clock cycles, without providing a value for the metadata fields. In case no value is specified by the user, the initial value of the metadata fields is set to zero by default. Test results are collected at line 17 and stored into a file on the management server for further elaboration. Each line of the file includes both the collected result and the address of the register at which it was stored. Lines 22- 24 consist in a list of assertions, that will be applied to the collected results. Assertions can compare counters with other counters, registers or expected integer values. Each assertion is composed of three parameters: the first and the last identify the two values to be compared and the parameter in the middle specifies which type of comparison to perform. In the example test script, the assertion at line 22 states that the number of packets forwarded to network interface “nf0” (first parameter) must be equal (second parameter) to 100 (third parameter). Similarly, the assertion at line 23 states that the value stored in register “reg05” must be less then or equal to the number of packets forwarded to network interface “nf2”. Line 24 asserts that more than 20 packets must be dropped at network interface “nf0”. Instruction at line 29 parses collected results and applies the assertions included in the assertion list, thus generating a text file that summarise the outcome of the test, stating pass/fail for each applied assertion. Line 32 takes the final time-stamp. Similarly to the initial time-stamp, the final time-stamp could be collected earlier, in case user do not want to take in to account the precessing of the assertions. The last instruction in the script, at line 35, computes and prints on the screen the total execution time of the test.

6.5.2 Flexible Architecture

While PTA is fully programmable, coding the programmable infrastructure can still be cumbersome, as one will still need to test and validate the code. We therefore propose a PTA flexible architecture, which is based on a one-time data plane program, many-times control-plane configuration.

Our flexible architecture is based on studying previous works on programmable data planes (e.g., [159, 41, 46, 160, 27]), and identifying common and different stages in their pipeline. This enables us to create a shared architecture, where some of the tables are used for “standard” headers, while others are dedicated to “per design” headers and metadata.

Figures 6.3 and 6.4 presents the concept of the flexible architecture. The architecture covers both the test header generator and the output packet checker. Each module is represented by a single P4 program, implementing all match-action tables. The values within these tables are configurable, and provide the flexibility to support different programs.

Flexible Test Header Generator. The flexible test header generator is implemented as a pipeline of match-action stages of three different types: “standard”

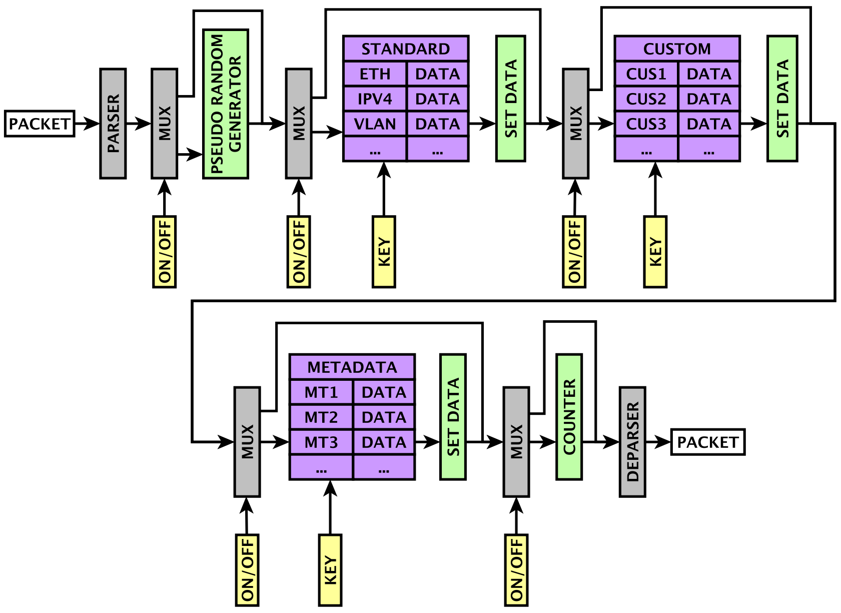


Figure 6.3. The flexible architecture: test header generator

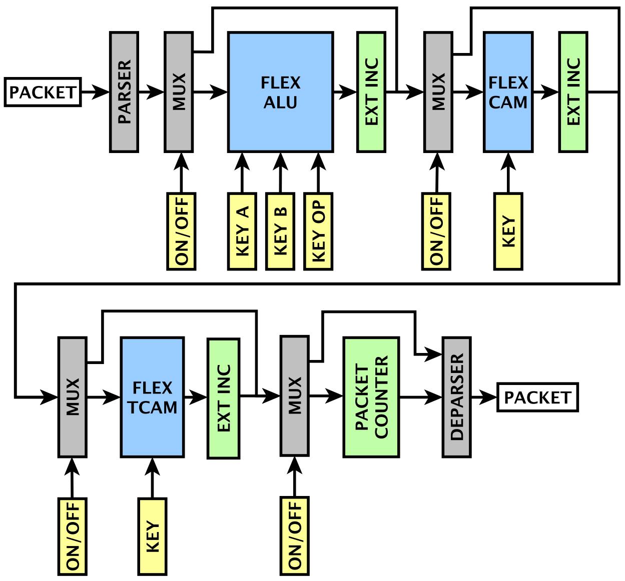


Figure 6.4. The flexible architecture: output packet checker

stages are used to generate standard headers, such as Ethernet and IPv4[1]; "custom" stages populate headers specific to the program under test; "metadata" stages set fields with the pipeline's metadata bus.

Match-action stages include a table, followed by a set of actions. In preparation of a test, the table is loaded with test data, used for populating the generated headers. The number of actions coupled to each table varies depending on the number of headers to be generated: each header requires a dedicated action that populates its fields with the test data provided by the preceding table.

The metadata bus plays a crucial role in P4-based designs. It not only provides information for different stages in the P4-pipeline, but it also serves as an interface to external modules. In the flexible architecture, the metadata bus carries signals for turning on and off each stage, as well as information for choosing which header is generated at each stage (based on values within the tables).

Additionally, a pseudo-random generator supports the generation of random header data (where desired), and a counter is used to track generated packets. Both are implemented as P4 externs, written in Verilog.

Flexibility Vs Resource Usage. The design of the flexible test header generator introduces a critical trade-off between the flexibility such a pipeline is expected to provide and the resources (number of match-action stages) it requires to be implemented. The general rule that guided our design efforts states that the degree of flexibility provided by the pipeline increases with the number of its stages. The effects of this rule are visible in our design: the higher the number of stages, the higher the number of headers the generator can put into a packet and the higher the number of combinations of headers it can generate.

Although this trade-off constraints the scalability of the generator, it keeps its design simple to implement and to configure. The proposed design works well for generating test headers, since typical test configurations require to generate only a fraction of the possible combinations of test headers and the number of headers to be put in a test packet is usually limited. A weaker assumption holds for the metadata bus, since it is usually small and only a subset of its fields is managed by the flexible test header generator, thus reducing the number of "metadata" stages in the pipeline. Most of the metadata bus is reserved for both the blank packet generator and the data plane under test.

Flexible Output Packet Checker. The assumptions that guided us through the design of the flexible test header generator do not hold for the flexible output packet checker. Indeed, the checker is required to process many more possible combinations of values, compared to the generator. First, checks are performed

[1]We expand on the need for multiple standard tables in Section 6.8.

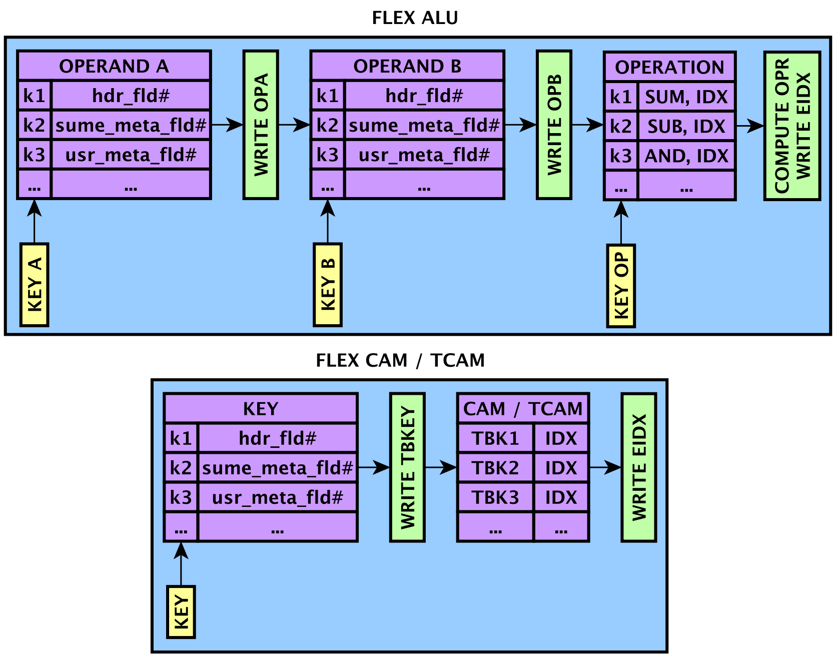


Figure 6.5. Components of the flexible output packet checker

at the granularity of the header/metadata fields, though in some cases even portions of fields, if not single bits, are checked. Instead, the generator processes data at the granularity of headers. Second, performed checks can be more complex than basic comparisons. As an example, consider a check that compares a header field with the result of a AND operation computed over two registers: in this case we need to put flexibility not only in the comparison, but also in the processing of the values to be compared.

In order to guarantee the scalability of the flexible output packet checker, we decided to overcome the trade-off between flexibility and resources, by making pipeline stages more flexible, though more complex, instead of increasing the number of stages in the pipeline. Although the flexible checker is similar to the flexible generator both in the way stages are turned ON/OFF and in the way extern modules are leveraged in the design, pipeline stages have been completely redesigned. Figure 6.5 shows the design of the three types of pipeline stages that can be used for building the flexible output packet checker: “ALU”, “CAM” and “TCAM”. In the figure, CAM and TCAM are depicted together, since there are minimal differences in the two designs.

ALU stages are used for preparing values to be checked. They compute arithmetic and logic operations over two configurable operands and write the result both to a dedicated metadata variable and to an extern register. Operands can be selected among both header fields and metadata fields. Supported operations include addition, subtraction, AND, OR and XOR. Each ALU stage is composed of a pipeline of three match-action stages. The first stage selects the first operand and stores its value into a dedicated metadata variable. Similarly, the second stage loads the second operand. Finally, the third stage selects the operation to be computed by the subsequent action and provides the index of the extern register to which store the result. ALU stages can be configured through three metadata fields, set by the blank packet generator. The three metadata fields are matched against the keys stored in the tables, for selecting the operands and the operation to compute.

CAM stages are used for implementing fast binary matching over a configurable input, chosen among header fields and metadata values, including the results of the operations computed by preceding ALU stages. TCAM stages are identical to CAM stages, except for implementing ternary matching, instead of binary matching. Each CAM/TCAM stage is composed of a pipeline of two match-action stages. The first stage selects the header/metadata field that is matched in the subsequent stage and writes its value to a dedicated metadata variable. This stage is needed for providing flexibility to the match operation, since P4 language requires the field to be matched to be known at compile-time. The second stage consists in a basic CAM/TCAM table that, first, matches the field selected in the previous stage and, then, runs an action for writing the result of the match to an extern register. Note that only the first stage needs to be configured through the metadata bus. The second stage is configured only through the data loaded by the control plane before the execution of a test.

Example Configuration. To illustrate the operation of the flexible architecture, assume an application is able to process packets with four headers: three standard headers (Layer 2 Ethernet, Layer 3 IPv4 and Layer 4 UDP), and one proprietary application header, that includes two fields. The application under test is supposed to XOR the first application header field with the metadata field indicating the network interface that received the packet. The result of the XOR operation is written to the second application header field. We want to test the application by injecting a stream of test packets and by checking that all the packets are correctly processed and forwarded. Assuming a flexible architecture has already been compiled and loaded to the network platform, we only need to provide a configuration for running the specific test we just described. The flexible generator, shown in figure 6.6, is composed of seven stages. The flexible checker, shown in figure 6.7, includes two ALU stages interleaved with two CAM

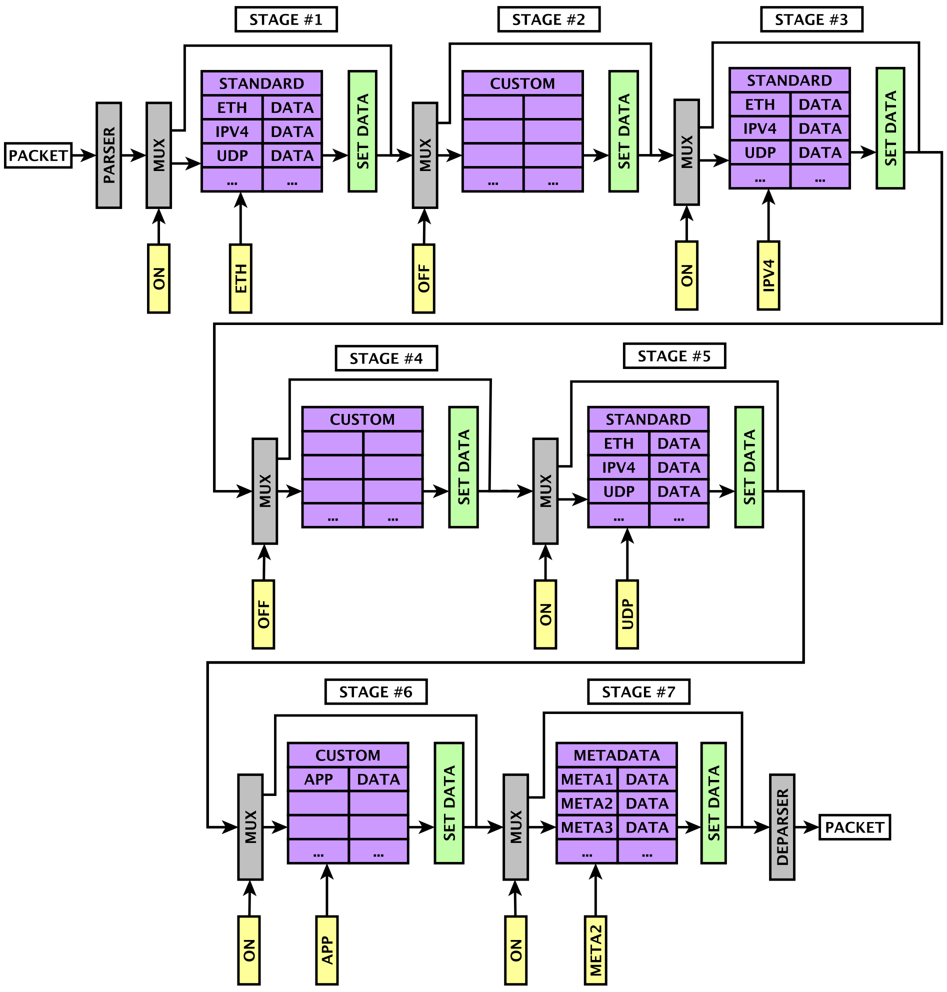


Figure 6.6. Example configuration of the flexible test header generator

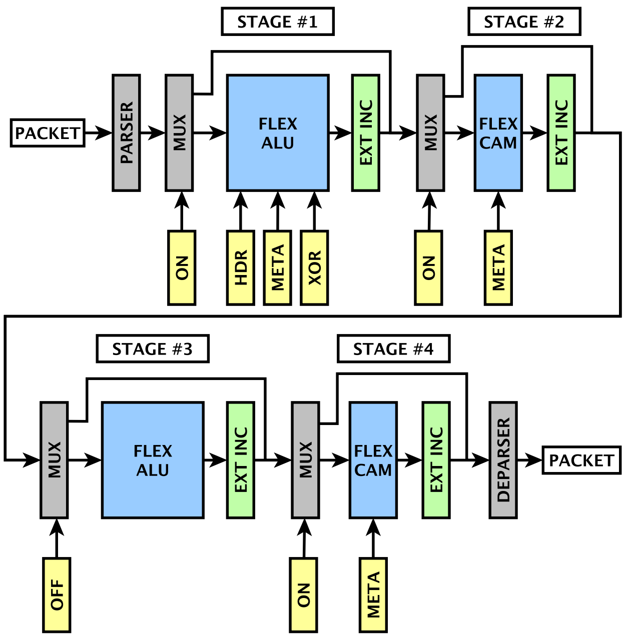


Figure 6.7. Example configuration of the flexible output packet checker

stages.

In order to generate the test headers, we configure the flexible generator as follows. The lookup tables for the four headers will be populated with entries: Tables 1, 3, and 5 will each be mapped to the standard headers, and table 6 will be mapped to the custom header. Tables 2 and 4 will not be used and be turned off[2]. A seventh table will be used to generate a metadata bus that feeds the data plane under test.

At the checker, we map the XOR operation to the first ALU (stage 1) and the related match to the first CAM (stage 2). The ALU at stage 3 is turned off. We use the CAM at stage 4 to check that packets are correctly forwarded to the output network interfaces. Therefore, we configure the flexible checker as follows. Using the control plane, we populate the tables in the three active stages with the values we need for running the tests, such as the addresses of the fields to match, the operations the ALU stage is supporting and the index of the registers to which the results will be stored. We note that, in many cases, the values programmed into the tables are program-specific, not test-specific. This means that a table configuration can be reused for running many different tests over the same program. Only the configuration of the metadata fields is test-specific. By

[2]the order of the tables corresponds to the order of the headers

setting dedicated header fields, in stage 1 we select the first application header field as the first operand and the metadata field indicating the network interface that received the packet as the second operand. Similarly, we indicate the ALU to compute a XOR operation. In stage 2 we select the metadata field containing the result of the XOR operation to be matched against the table. Finally, we select the metadata field indicating the network interface to which the processed packet has been forwarded to be match against the table at stage 4. The results of the checks compute at stages 2 and 4 will be written to dedicated extern registers.

For both the generator and the checker, the decision which tables should be used is embedded within a metadata bus internal to the packet generator, indicating how the tables need to be used. This metadata bus is user configurable, as well as the size of the blank packet generated at the beginning of the generation pipeline. Note that no P4 program needs to be written when the flexible architecture is used; by changing the metadata configuration and the entries in the tables, users can test many different data planes.

Implementation. We implemented the flexible pipeline architecture on NetFPGA SUME. However, currently available closed-source P4 to hardware compilers, including SDNet, come with firm restrictions when implementing tables and conditions. In SDNet, tables can not be smaller than a minimum fixed size, such as 64 entries, and can not share memory. In case many small tables, filled with the same data, are needed, as in our architecture, this limitation causes huge resource waste in the hardware. Moreover, the compiler performs little optimisation on conditions' implementation, thus leading to resource explosion. These limitations prevented us from fitting the flexible architecture within a single NetFPGA SUME card, due to the high resource requirement. The trade-off between flexibility and resources consumption forced us to revert back to less general architectures, adjusted per use case, that can be fitted into a single NetFPGA SUME, alongside the program under test. This does not eliminate the contribution of the flexible architecture: first, as NetFPGA SUME is implemented on Xilinx Virtex-7 FPGA, which is 3-generations behind the state of the art, and does not have as much resources as newer devices. Second, as Xilinx Labs indicated to us that their next SDNet release is attending to the resource-explosion limitation.

6.5.3 Integration with a Verifier

There has been significant prior work on workload and test case generation. This work covers a broad range of techniques, including automatic test packet generation [157], fuzz testing [158], and software validation [117]. P4v provides a path for these tools to run the tests that they generate on hardware. As a proof-of-concept about how such tools could integrate with PTA, we developed a prototype P4v-to-PTA translator.

Many software verification tools, including P4v [117] and Assert-P4 [118], are based on Hoare logic, which provides a formal system for reasoning about the correctness of computer programs. The central feature of Hoare logic is the Hoare triple. A Hoare Triple is of the form $\{P\}\ c\ \{Q\}$, where P is the precondition, Q is the postcondition, and c is the command, i.e., a piece of code that changes the state of the computation.

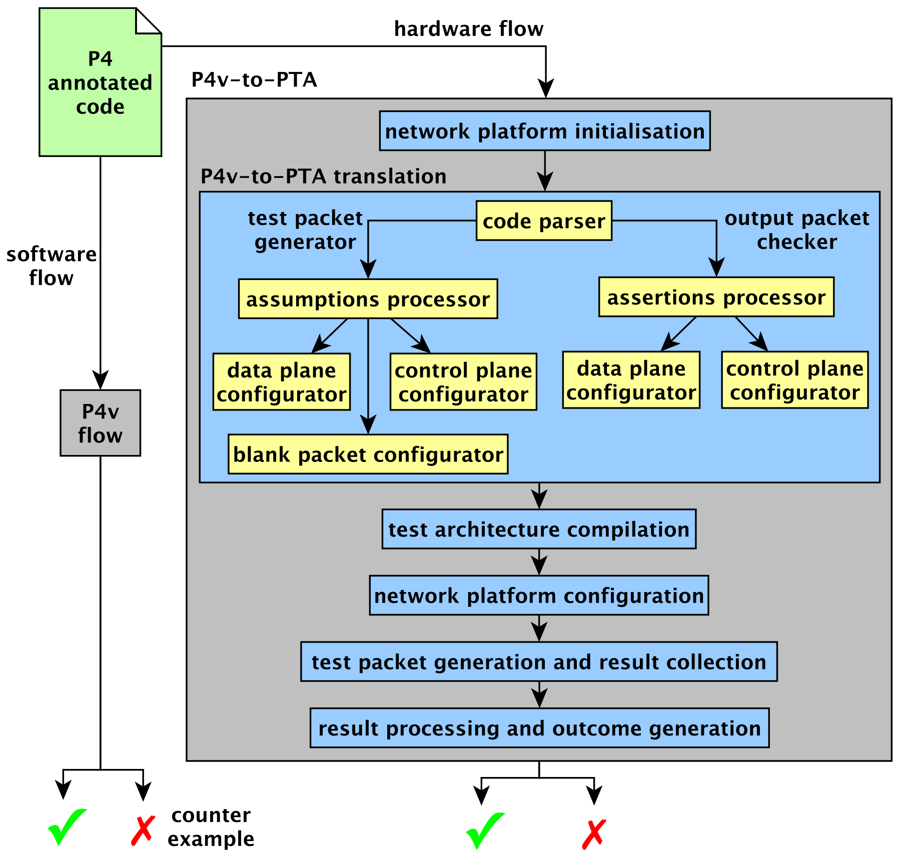
To verify correctness properties of a program, the verifier tool first translates the software into a set of logical formulas, combining them with user programmer-supplied annotations that specify assumptions (i.e., the preconditions) and assertions (i.e., the post-conditions). It can then use an automated theorem-prover to check if there is an initial state that leads to a violation. The theorem-prover can generate a counter example via weakest pre-condition analysis. In the context of network data planes, both P4v [117] and Assert-P4 [118] use this general approach. PTA can complement these verifier tools by acting as a run-time verifier.

P4v-to-PTA Flow. Figure 6.8 shows two complementary techniques for verifying programmable data planes. In order to verify a P4v-annotated code, users can use either a software flow, or a hardware flow, or both.

The software flow, based on P4v, employs software formal verification, as already discussed.

The P4v-to-PTA hardware flow provides verification at runtime, leveraging programmable network hardware. This process is fully automated. To test an annotated P4 program, a user simply needs to place the P4 code in a specified folder. Based on P4 annotations, P4v-to-PTA not only generates the test generator and checker data planes, along with the associated configuration, but also manages the whole test execution. The sequence of operations performed by P4v-to-PTA is shown in figure 6.8. First, the tool connects to the network infrastructure and resets its state. Then, it computes the translation from the input P4v annotations to a complete hardware test specification. The specification is compiled, programmed to the network infrastructure and configured for the test. Finally, the tool triggers the generation of test packets, collect test results and provides the user with the outcome of the test. The entire process is similar to using the P4v command line tool, but runs real, extensive hardware validation.

The core functionality of P4v-to-PTA is implemented by two main components: a data plane architecture template and a P4v-to-PTA translator. The data plane architecture template includes network hardware specifications for both



hardware flow
P4 annotated code
P4v-to-PTA
network platform initialisation
P4v-to-PTA translation
test packet generator
code parser
output packet checker
software flow
assumptions processor
assertions processor
data plane configurator
control plane configurator
data plane configurator
control plane configurator
P4v flow
blank packet configurator
test architecture compilation
network platform configuration
test packet generation and result collection
result processing and outcome generation
✓
✗
counter example
✓
✗

Figure 6.8. Two complementary techniques for verifying programmable data planes.

the test header generator and the output packet checker. The P4v-to-PTA translator consists in a set of python scripts that translate P4v annotations into a configuration for the hardware template. Both the components are discussed in detail in the following.

Data Plane Architecture Template. The data plane architecture template, shown in figure 6.9, provides a general specification for both the test header generator and the output packet checker. The configuration of the template is computed by the P4v-to-PTA translator, at each execution of a test.

Both the test header generator and the output packet checker are based on a linear pipeline architecture in which no test functionality is implemented by parsers and deparsers. The two pipelines have two types of stages in common: "set config" stage and "set dest port" stage.

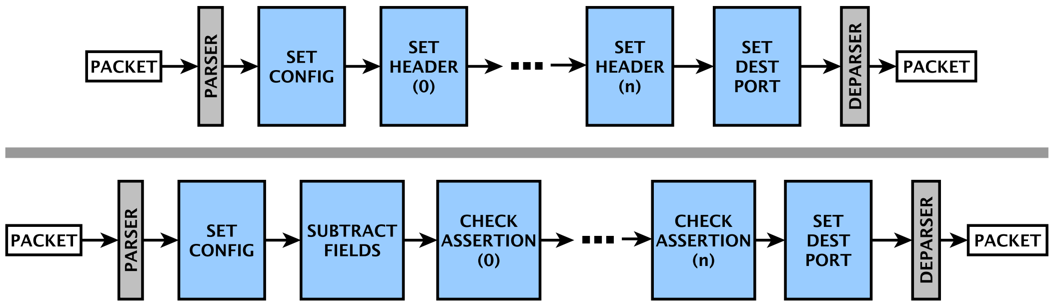


Figure 6.9. Data plane architecture template: test header generator (top), output packet checker (bottom).

"Set config" stages act as an interface for configuring each of the two pipelines with test-specific parameters. In this sense, they are equivalent to the metadata bus used for configuring the flexible architecture 6.5.2. Test-specific configuration is read and decoded as soon as a test packet enters the pipeline, and the following stages in the pipeline are traversed according to the configuration.

The final stage in both the pipelines, named "set dest port", sets the destination network port to which test packets are forwarded. This stage can be configured either statically, or dynamically at each execution of a test.

Additionally, the test header generator includes a variable number of "set header" stages for attaching header data to incoming blank packets. Each stage generates a specific test header, using the data provided by the P4v-to-PTA translator. The order in which the stages are traversed determines the sequence of test headers implemented in each generated test packet.

Similarly, the output packet checker includes a variable number of "check assertion" stages, each implementing a check, either over a single header field, or over two header fields. The number of "check assertion" stages implemented into the output packet checker is equal to the number of assertions in the input P4v-annotated code. Each "check assertion" stage is coupled to a stateful element, to which the result of the check is written. Stateful elements are then accessed through the control plane interface for collecting test results.

The sequence of "check assertion" stages is preceded by a single "subtract fields" stage that pre-processes all the header fields in parallel. The "subtract fields" stage subtracts from each header field the value it is compared against in its corresponding assertion. This operation is needed for keeping the implementation of the "check assertion" stages as simple as possible: instead of comparing a header field with any possible other field or constant value, each "check assertion" stage compares the difference computed by "subtract fields" stage with

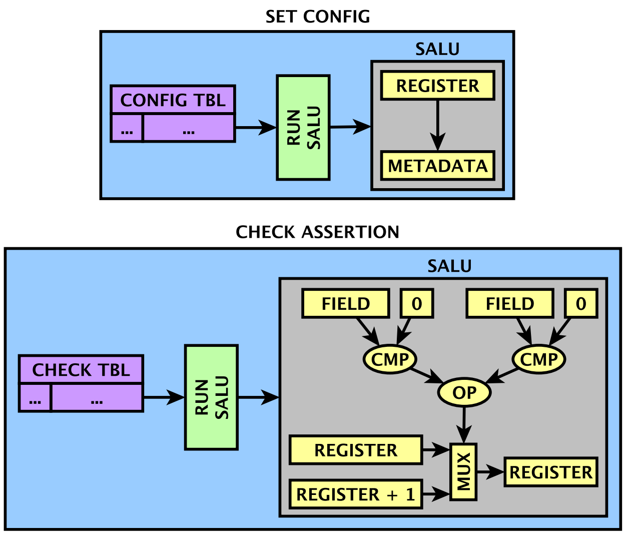


Figure 6.10. “set config” and “check assertion” stages, as implemented on Barefoot Networks’ Tofino ASIC.

zero. As an example, consider the assertion $F > 3$ and assume the value of header field F , set by the data plane under test, is 5. The assertion will be implemented as follows. First, the “subtract fields” stage computes the difference $F - 3 = 2$ and stores the result back to F . Then, the “check assertion” stage compares the value of F to zero, thus computing the operation $F > 0$. Since the value of F, that is 2, is greater than zero, the check is positive, meaning that the corresponding assertion is met.

Mapping Stages to Tofino’s Architecture. Figure 6.10 shows a schematic representation of “set config” and “check assertion” stages, as implemented in our prototype, running on the Barefoot Networks’ Tofino ASIC. Both the stages are composed of a sequence of three elements: a table, an action and a stateful ALU (SALU). Although the only operation performed by the table and the action is to trigger the execution of the SALU, these components are required to map the SALU to a specific stage in the pipeline.

The SALU is a specific component provided by Tofino’s architecture, that includes both a stateful element, in the form of an array of registers, and an ALU, that is capable to process very simple computations, such as comparisons, logic functions, and basic arithmetic operations. The functionality of a SALU is implemented in the data plane at compile time and cannot be changed at run time. Once triggered by a table-action couple, a SALU runs the pre-defined operations and, based on the result, changes the values stored into one of its registers. Optionally, the value stored into one of the registers can be copied to a header/metadata field.

"Set config" stages include a simple SALU, that implements only a copy from a register to a metadata field. Copying the configuration data from the register to the metadata field is necessary, since SALUs, due to their specific hardware implementation, can be accessed only once for each packet traversing the pipeline, through a table-action couple. Instead, accessing (and changing) the values stored in metadata fields is much less constrained. Therefore, we use metadata fields as "global variables" in the data plane and SALUs as configuration interfaces.

Compared to "set config" stages, "check assertion" stages include a more complex SALU, that leverages both the ALU and the stateful element. Figure 6.10 shows the most general configuration of a "check assertion" stage, that implements an assertion in the form (X CMP Y) OP (Z CMP W), where X and Z are header fields, Y and W can be either header fields or constants, CMP are comparison operators and OP is a logical operator. Since the "subtract fields" stage pre-processes X and Z before they reach the "check assertion" stage, X and Z are compared against zero in the SALU. The results of the two comparisons are then processed by the logical operation block. Finally, the value of the internal register is either incremented, or left unchanged, based on the outcome of the logical operation.

In case an assertion in the form (X CMP Y) is checked, the increment on the register is performed based on the outcome of the comparison. Both the second (unused) comparison and the logical operation are not implemented in the SALU.

Architecture Scalability. The design of the data plane architecture template is based on an innovative technique for mapping formal annotations to the network hardware. The technique consists in assigning the functions common to all the tests, such as configuration and forwarding, to fixed components of the pipeline, and in allocating a variable number of more specific resources, depending on the program to test.

Although this strategy helps in minimising the resource consumption in the pipeline, the proposed architecture is not immune to scalability limitations. The main limitations in the architecture are due both to the total number of available stages in the pipeline and to the maximum number of operations that can be computed in parallel by an action.

The limitation on the number of available stages in the pipeline can affect both the test header generator and the output packet checker. In the test packet generator, the number of stages used for generating headers is equal to the num-

Function Name	Used In	Description
extract_pragmas( )	code parser	extract P4v annotations from P4 code
parse_asm_asr( )	code parser	parse annotations and identify assumptions and assertions
identify_header_data( )	assumptions processor	parse headers used in P4 code under test
pop_hdr( )	assumptions processor	populate test headers, based on the assumptions
pop_subs( )	assertions processor	populate subtract stage, based on the assertions
pop_chks( )	assertions processor	populate check stages, based on the assertions
wr_dataplane( )	data plane configurator	configure data plane template with test-specific data
gen_cp_config( )	control plane configurator	generate control plane test-specific data
wr_control_plane( )	control plane configurator	configure control plane template with test-specific data
prep_pktgen_config( )	blank packet configurator	generate test-specific data for the blank header generator
wr_pktgen_config( )	blank packet configurator	configure blank header generator template with test-specific data

Table 6.3. Main functions implemented by the translation module.

ber of assumptions in the P4v-annotated code. In the output packet checker, the number of stages that implement the checks is equal to the number of assertions.

The number of operations that an action is able to compute in parallel can be a bottleneck for the output packet checker, since the number of header fields pre-processed by the "subtract fields" stage is equal to the number of assertions in the P4v-annotated code.

Although users may want to test programs with a very large number of annotations, we believe scalability will not reduce the effectiveness of the proposed architecture, for two reasons. First, testing programs with our prototype on Barefoot Networks' Tofino ASIC, we never ran out of resources, both in the test header generator, and in the output packet checker. This is due to the fact that different functions in a program are usually tested individually, thus reducing the amount of resources taken by each test. Second, we expect future programmable network devices to provide a much larger amount of resources, compared to currently available hardware, thus making scalability limitations negligible.

P4v-to-PTA Translator. The P4v-to-PTA translator maps P4v annotations to a configuration of the data plane architecture template. The translator, implemented as a Python script, imports a translator module, for enabling P4v-to-PTA-specific functions.

Table 6.3 lists the main functions provided by the translator module, along with a brief description. "Used In" column specifies which portion of the P4v-to-PTA translator calls the corresponding function.

Assuming an input P4v-annotated code, the P4v-to-PTA translation is implemented as follows. First, the code parser scans the P4v-annotated code line-by-line for identifying annotations in the form "@pragma assume/assert (...)".

Then, each annotation is parsed for detecting assumptions in the form X CMP Y and assertions specified either as X CMP Y, or as (X CMP Y) OP (Z CMP W). X and Z are header fields, Y and W can be either header fields or constants, CMP are comparison operators and OP is a logical operator. Other types of annotations are discarded, since the P4v-to-PTA translator is currently unable to process them. Finally, the code parser comments out all the annotations in the input P4 code, since they are not supported by the P4-to-hardware compilers. Assumptions and assertions are then processed separately, by two parallel blocks.

The assumptions processor generates the configuration of the test headers by populating each header field with a value derived from the assumptions. For understanding the structure of the headers to generate, the processor scans all the header definitions attached to the input P4v-annotated code and extracts all the headers that the program under test is able to process. Then, based on the parsed assumptions, the processor computes the value of the header fields in three phases. The first phase populates only those header fields to which assumptions in the form FIELD CMP CONSTANT apply. In the second phase, assumptions in the form FIELD CMP FIELD are processed. The third phase fills the unassigned header fields with default values.

As an example, consider the two assumptions Fa == 3 and Fb > Fa. The assumptions processor will assign value 3 to field Fa (first phase). Then, it will assign to Fb the minimum value that satisfies the second assumption, that is 4 (second phase). Finally, another header field, say Fc, not mentioned in the assumptions, will be populated with a default value, such as 1 (third phase).

As soon as all the header fields are populated, the assumptions processor generates the configuration data for both the test header generator pipeline and the blank packet generator. As it is currently implemented in our prototype, the blank packet generator is configured for generating a single packet for each test. However, users can easily tune the configuration parameters of the blank packet generator through the P4v-to-PTA translation script. Configuration data is then written to the template files. In particular, the generated sequence of "set header" stages is written to the test header generator template. Both the control plane of the test header generator and the blank packet generator are configured through specific Python scripts, that are based on the templates included in P4v-to-PTA.

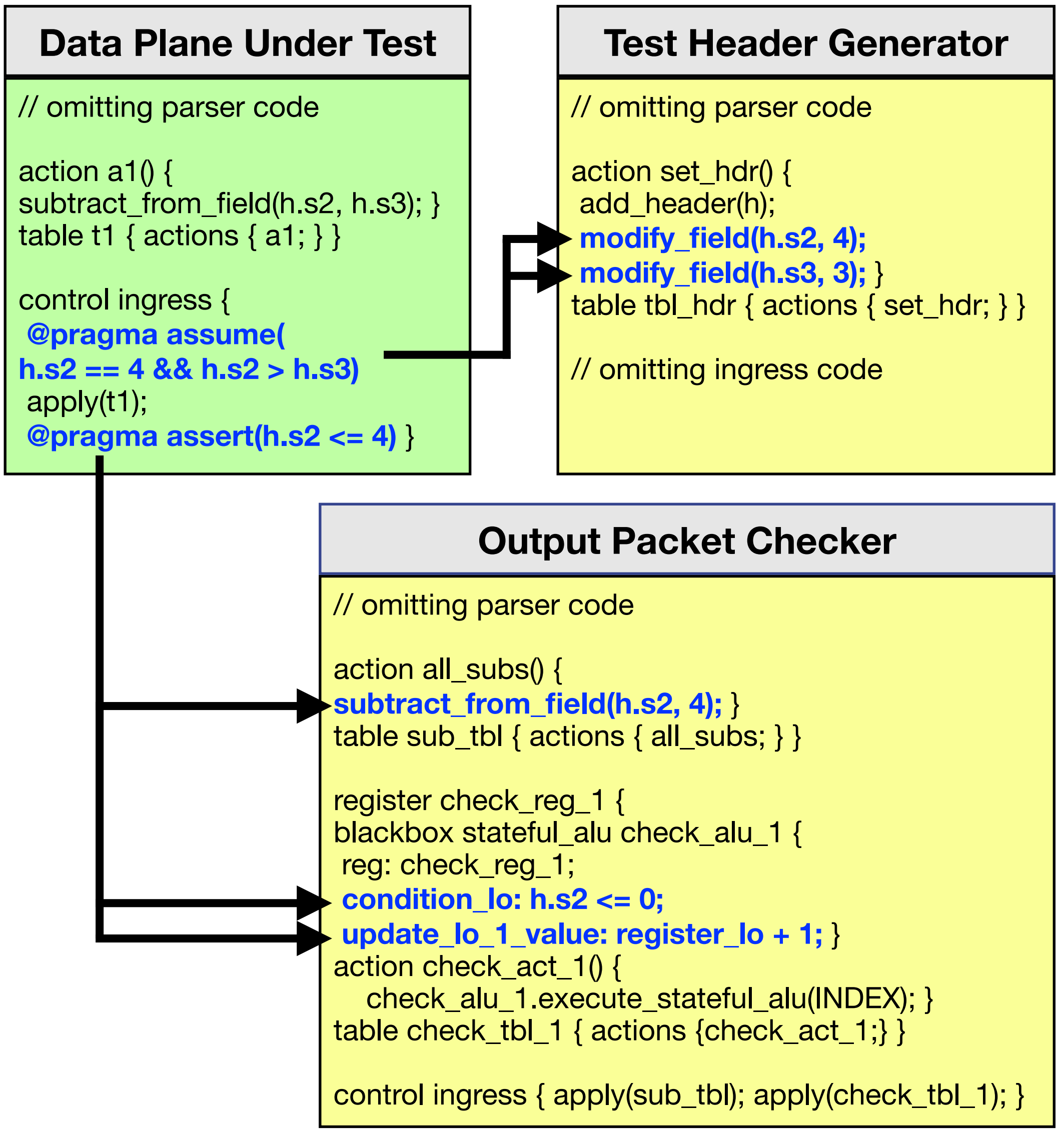
The process of writing the configuration data to the template files implements the connection in between the "software" portion of P4v-to-PTA and its "hardware" counterpart: the translation, implemented with software scripts, changes the specification of the test architecture, before it is compiled to the hardware.

The assertions processor maps each parsed assertion to a corresponding check in the hardware. It supports both assertions in the form (X CMP Y) and in the form (X CMP Y) OP (Z CMP W). First, the processor scans the parsed assertions for generating the list of subtract operations to be computed in the "subtract fields" stage. Each comparison, in the form A CMP B, found in the parsed assertions is translated to an operation that subtracts B from A. Then, the processor maps each assertion to a dedicated "check assertion" stage. As already discussed, the structure of each "check assertion" stage in the template can implement both assertions with a single comparison and assertions with two comparisons. Unused blocks in "check assertion" stages that implement assertions with a single comparison, are not instantiated by the assertions processor, thus saving resources in the pipeline.

The assertions processor generates the configuration data for the output packet checker pipeline, similarly to the generation process implemented in the assumptions processor. The output packet checker pipeline is populated with both the subtract operations, computed in the "subtract fields" stage, and the sequence of "check assertion" stages. The control plane of the output packet checker is configured through a specific Python script, based on the template included in P4v-to-PTA.

Example Data Plane Configuration. Figure6.11 shows an example configuration generated by the P4v-to-PTA translator, based on the P4v-annotated specification of the data plane under test. By inspecting the P4 specification, we note that the data plane under test processes only header "h", that includes fields "s2" and "s3". The only assumption found in the code compares field "s2" both to a constant value (4) and to field "s3". Since the first comparison is an equality operation, the translator generates an instruction for initialising field "s2" to 4. The second comparison states that the value of field "s2" must be greater than the value of field "s3". The P4v-to-PTA translator generates an instruction for initialising field "s3" to 3, since 3 is the closest value to "s2" that satisfies the assumption. Since the two comparisons are connected by an AND operator in the assumption, both the generated instructions are written to the test header generator template, as part of the "set header" stage that generates header "h".

The only assertion found in the code states that, after being processed by the data plane under test, the value of header field "s2" must be less or equal to 4. The P4v-to-PTA translator configures the output packet checker as follows. First, it populates the "subtract fields" stage with an instruction that subtracts 4 from field "s2". Then, it instantiates a "check assertion" stage with a single operation, that compares the computed difference to zero (h.s2 $\leq$ 0). Finally, the translator configures the SALU in the "check assertion" stage for incrementing the value stored to its register, based on the outcome of the comparison.



```
// omitting parser code

action a1() {
subtract_from_field(h.s2, h.s3); }
table t1 { actions { a1; } }

control ingress {
 @pragma assume(
h.s2 == 4 && h.s2 > h.s3)
 apply(t1);
 @pragma assert(h.s2 <= 4) }
```

```
// omitting parser code

action set_hdr() {
 add_header(h);
 modify_field(h.s2, 4);
 modify_field(h.s3, 3); }
table tbl_hdr { actions { set_hdr; } }

// omitting ingress code
```

```
// omitting parser code

action all_subs() {
subtract_from_field(h.s2, 4); }
table sub_tbl { actions { all_subs; } }

register check_reg_1 {
blackbox stateful_alu check_alu_1 {
 reg: check_reg_1;
 condition_lo: h.s2 <= 0;
 update_lo_1_value: register_lo + 1; }
action check_act_1() {
    check_alu_1.execute_stateful_alu(INDEX); }
table check_tbl_1 { actions {check_act_1;} }

control ingress { apply(sub_tbl); apply(check_tbl_1); }
```

Figure 6.11. Example data plane configuration.

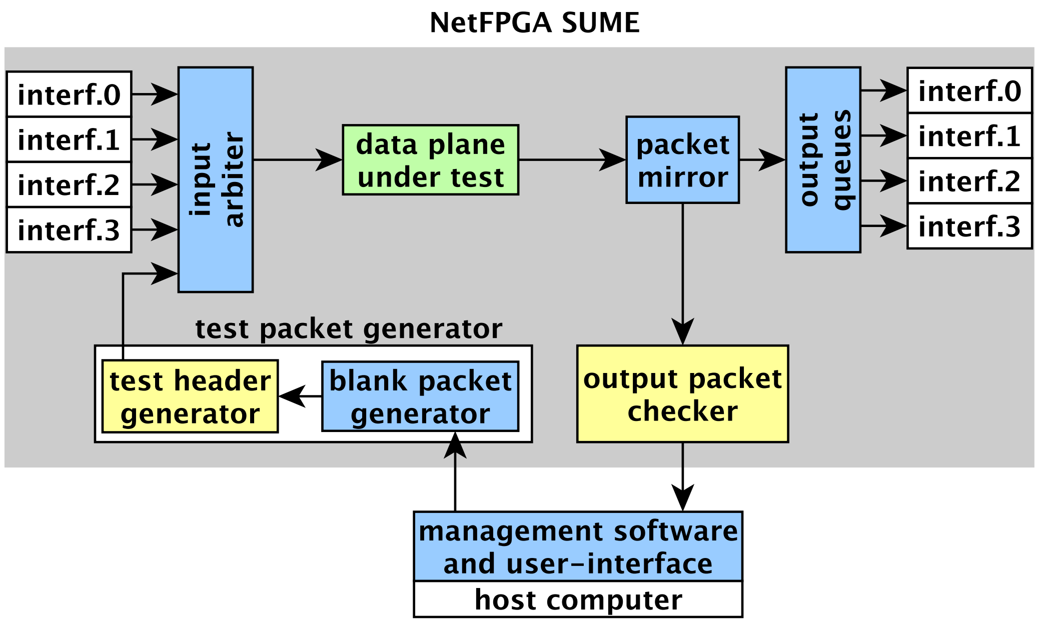


Figure 6.12. FPGA implementation.

Implementation. We implemented the integration with P4v on Barefoot Networks' Tofino ASIC, using P4_14 language. We were unable to implement the P4v-to-PTA flow on the NetFPGA SUME platform, since P4v can process only P4_14 code and the current release of the NetFPGA SUME framework supports P4_16 and not P4_14.

6.6 Prototype Implementation

We have implemented PTA on two target devices: the NetFPGA SUME platform and Barefoot Networks' Tofino ASIC.

6.6.1 FPGA Implementation

An FPGA-based prototype is implemented on the NetFPGA SUME platform[4] using the P4→NetFPGA workflow [46]. P4→NetFPGA uses Xilinx Labs' SDNet[69] compiler to compile P4 programs to the hardware. Although SDNet is a closed-source tool, it has been chosen because there is no open-source alternative for compiling P4 code to hardware. Indeed, SDNet offers a frontend for P4 language, which is able to translate a P4 specification into a FPGA hardware module for Xilinx FPGAs, including the one of the NetFPGA SUME card. While the NetFPGA platform is open source, the output of SDNet is encrypted, so we don't have visibility into the hardware-language implementation of the P4 program, nor can we add hooks from within the pipeline. All other modules are open, as well as the P4 code of the generator and checker.

The implemented architecture, shown in figure 6.12, builds upon the NetFPGA SUME reference architecture, which is composed of a data plane that processes traffic arriving from four independent network interfaces and a host (over PCIe). PTA taps to the architecture through a dedicated input interface and a hardware module, called "packet mirror", that provides a copy of the traffic processed by the data plane under test.

Test packet generation is triggered by a host computer, connected to the NetFPGA SUME card through its PCIe interface. The configurable blank packet generator module generates blank packets from inside the FPGA, serving as a stimulus to the P4 pipeline. In P4, the pipeline is driven only by packet events, and thus can not generate packets on its own, only modify existing packets. As the generated blank packets go through the P4-based test packet generator pipeline, packet headers are being populated with test data. Then, packets are injected into the primary data plane under test, thus being processed according to the functionality under test. At the output of the data plane under test, traffic is mirrored: one copy is forwarded to the output queues and interfaces of the card, while the other enters the output packet checker module, that performs checks on packet header fields, based on the P4 specification provided by the user. Finally, check results are sent to the host computer for further elaboration.

PTA is not limited to programmability of the P4 pipeline alone. In the FPGA implementation, users can instantiate additional hardware components, referred to in P4 as externs, such as registers, counters and hashing functions.

6.6.2 ASIC Implementation

We also implemented PTA on Barefoot Networks' Tofino, a programmable Ethernet switch ASIC that is able to process traffic at 6.5Tbps. The architecture of the Tofino switch, based on the PISA architecture [2], provides either two or four parallel network data planes, depending on the ASIC model, that can serve up to sixty-five network interfaces, each running at 100Gbps [3]. The data plane functionality implemented by the switch is programmable through P4 language.

The four pipelines of Tofino allow us to implement PTA using an architecture similar to the FPGA architecture, where each component is implemented

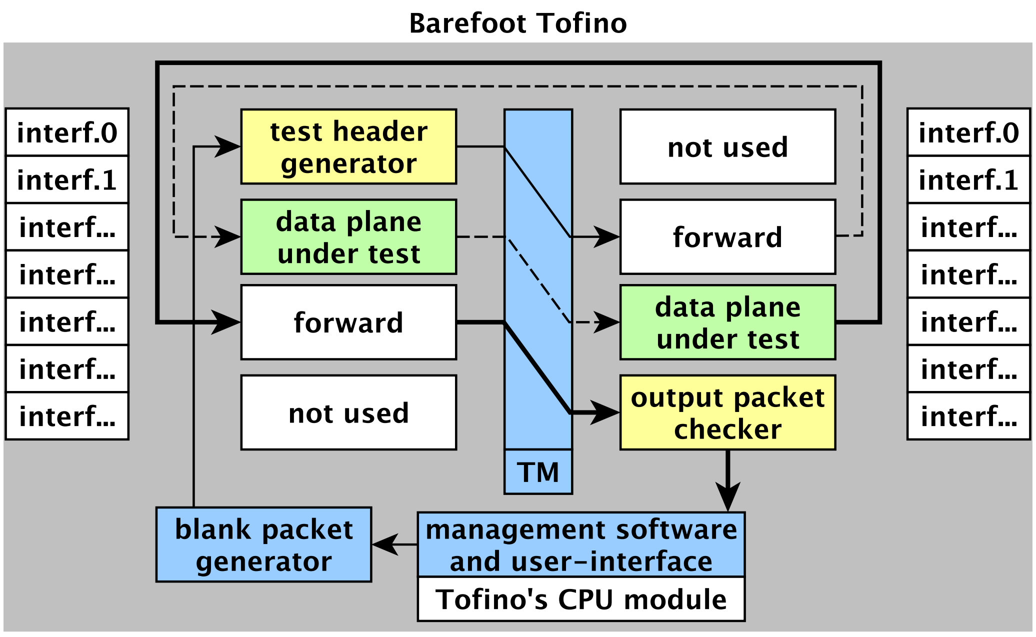


Figure 6.13. ASIC implementation.

within a separate part of the pipeline, and as a stand-alone program, without code dependencies. As the Tofino switch implements an ingress pipeline and an egress pipeline in every data path, the test header generator can be mapped to the ingress pipeline and the output packet checker to the egress pipeline. The data plane under test has a dedicate data path, using both ingress and egress pipelines.

An implementation of PTA within a single Tofino device is shown in figure 6.13. The traffic manager module, named “TM” in the figure, is in charge of moving packets either among different pipelines, or from a pipeline to one of the output network interfaces. Similar to PTA's FPGA-based implementation, a blank packet generator component is needed for injecting packets into the network pipelines. Tofino has such a configurable hardware module, named “pktgen”, that can generate batches of network packets of different sizes, and inject them to an ingress pipeline. This module is leveraged in our implementation.

While the Tofino architecture enables the above implementation, its control plane currently does not support the required functionality. In particular, it is unable to manage different programs, loaded on different data planes. Therefore, as a proof-of-concept, we implemented a prototype that uses three separate Tofino switches, each managed by a separate control-plane interface. In this con-

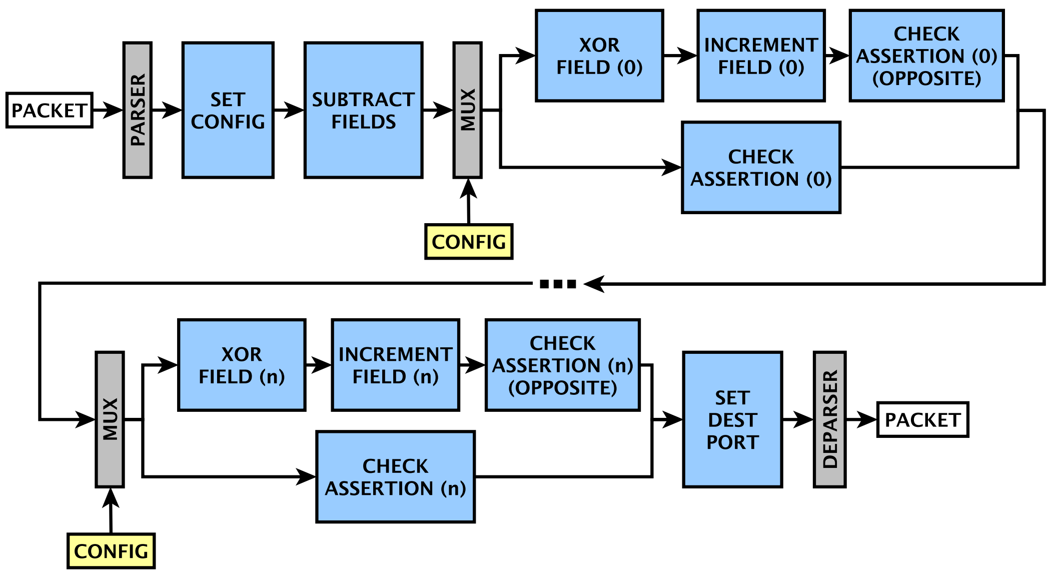


Figure 6.14. Workaround for signed fields bug.

figuration, one switch serves as a test packet generator, one as the device under test, and one as the output packet checker. The three switches are connected through 100Gbps QSFP+ network cables. We implemented the integration with the verifier, describe in section 6.5.3, using this prototype, thus demonstrating both the feasibility of PTA on ASIC and the capabilities of the P4v-to-PTA translation flow. Future releases of the Barefoot Networks SDE are expected to support the management of separate control planes. When that is released, PTA can be deployed on a single ASIC.

The ASIC prototype is managed by a host, implemented in the control plane, that configures the test packet generation and collects check results for further processing. Once the packet generation has been triggered, blank packets are injected into the test packet generation data plane, populating packet header fields with test data, based on the configuration specified by the user. Test packets are then forwarded to the data plane under test, programmed on the second Tofino switch. Processed packets, are sent to the third switch, where they go through the output packet checker data plane, and optionally get forwarded to one of the output interfaces of the switch for further inspection.

6.6.2.1 Workaround for Signed Fields Bug

Using PTA as a runtime tester, we were able to identify two bugs in the Barefoot Networks SDE compiler, related to saturating integers and signed fields. (Section 6.7.2.1). These bugs would not be caught by formal verification. The programs that suffered from these bugs were logically correct, but still exhibited unexpected behaviour.

Leveraging the modularity of the P4v-to-PTA architecture, we implemented a workaround for the bug affecting the signed header fields. As described in section 6.7.2.1, signed fields are treated as unsigned numbers in the hardware, although represented in two's complement form. This bug prevents our P4v-to-PTA prototype, running on Barefoot Networks' Tofino ASIC, to correctly manage negative header fields. More specifically, both negative fields and positive fields becoming negative after the subtract operation computed in the output packet checker, are treated as positive fields in the following "check assertion" stages, thus leading to incorrect results. We note that treating a negative number, expressed in two's complement form, as an unsigned number, is not equivalent to computing the absolute value of that negative number. As an example, consider the negative number -5, represented as 1011 in two's complement form. If 1011 is treated as an unsigned number, its value in decimal form is 11, which is not the absolute value of -5, that is 5.

The idea behind the design of the workaround implementation is that we need a technique for understanding whether a header field is becoming negative, or not, during the execution of a test. Assuming we know the sign of all the header fields before they are checked in the output packet checker, we need to make the pipeline flexible enough to be able to process the header fields both in case they are positive, and in case they are negative. The pipeline should be able to extract the correct absolute value from negative fields, even if they are represented in two's complement form.

Since it is impossible to understand whether a field is negative, or positive, in the data plane, we address this limitation by implementing new functions in the P4v-to-PTA translation module. The new functions implement a sort of "software simulator" that scans the input P4v-annotated code and keeps track of the status of all the header fields, by sequentially computing the operations found in the code. At the end of the simulation, the P4v-to-PTA translator knows the value that each header field will have after being processed by the data plane under test. Since in the PTA architecture 6.1 the output packet checker is directly attached to the data plane under test, P4v-to-PTA translator knows the value that the header fields will have when entering the output packet checker pipeline.

After computing the subtract operations over the header fields, thus simulating the “subtract fields” stage of the output packet checker, the P4v-to-PTA translator is able to identify the negative header fields that will be affected by the bug in the prototype architecture. The P4v-to-PTA translator includes this information into the configuration of the output packet checker.

For enabling processing both positive and negative header fields and for computing the absolute value of negative fields, we introduce a new data plane architecture of the output packet checker. The new version of the checker, shown in figure 6.14, provides a mechanism for computing the absolute value of a negative header field at line-rate. This operation is performed by two consecutive stages. The first stage, named “xor field”, computes a XOR operation over the negative field and a binary constant of the same size of the field, with all its bits set to 1. The result of the XOR operation overwrites the negative value in the field. The second stage, named “increment field”, increments the value of the field, thus obtaining the absolute value. As an example, assume a field F is populated with the negative value -5, that is 1011 in its, 4-bits wide, two’s complement binary form. First, we compute F XOR 1111 and write the result (0100), back to F. Then, we increment F by one, thus obtaining 0101, that is the absolute value of the field (5), in decimal form.

Leveraging the information provided by the P4v-to-PTA translator through the “set config” stage, it is possible to dynamically configure each check stage in the output packet checker pipeline for processing either a positive header field, or a negative header field. This is achieved by making check stages in the pipeline more flexible, though more complex. As shown in figure 6.14, each check stage is preceded by a multiplexer that routes incoming header fields, based on the configuration provided by the “set config” stage. In case the configuration states that an incoming header field is positive, the field is forwarded to a unmodified version of the “check assertion” stage, similar to the one shown in figure 6.10. If an incoming header field is negative, its absolute value is extracted, as already discussed. Since the sign of the field has changed, the check to be performed must be changed accordingly. Therefore, the header field goes through a modified “check assertion” stage that computes the same check performed by its unmodified counterpart, but with the opposite comparison. As an example, consider field F, with value -5. We want to check that $F > 0$. Therefore, we compute the absolute value of F, that is 5, and we configure the “check assertion” stage for implementing the comparison $F < 0$. Performing the comparison $5 < 0$ is equivalent to checking that $-5 > 0$.

The same technique is implemented for checking assertions that include two comparisons. This case leads to the implementation of four versions of each check. Two versions process the cases in which only one of the two comparisons is computed over a negative field. The third version covers the case in which both the comparisons are computed over negative fields. The fourth version processes the case in which both the comparisons are computed over positive fields.

Although the proposed workaround limits both the scalability and the performance of P4v-to-PTA, it enabled our prototype implementation on the Barefoot Networks' Tofino ASIC and allowed us to test programs that included signed header fields.

6.7 Evaluation

6.7.1 A Quantitative Evaluation

Performance. We have evaluated and confirmed that both the NetFPGA SUME and Tofino-based programs run at line rate. Beyond measurement, this is by design as both our FPGA and ASIC-based prototypes are implemented as Protocol Independent Switch Architecture (PISA) [161]-style architectures [46], in which the compiler strictly defines the resources used. This is necessary because a PISA machine must ensure deterministically high performance. Thus, the throughput and latency are characterized by a P4 program at compile time; at run time the machine's throughput and latency do not change, unless intentionally configured to run at a lower rate. No congestion is possible within PTA's modules.

Resource Consumption. PTA introduces two new modules to a device. On the generator side, NetFPGA programs required between 2 and 4 pipeline stages, using one table and 1-2 externs, and Tofino implementations required 2 tables. On the checker size, NetFPGA programs required between 5 and 7 pipeline stages, using two tables and 3-5 externs. On Tofino, 7 tables and 5 stateful ALUs were required in the checker.

We report the resources overhead introduced by PTA, but caution that it is difficult to quantify resources in a meaningful way, since the amount used depends on the program under test and the compiler. For example, on NetFPGA the compiler requires that all tables have least 64 entries, even if 16 entries would be sufficient. A newer version of SDNet (2019.1), not currently supported by NetFPGA, is more resource efficient.

On NetFPGA, representing an FPGA-based use case, the resource overhead of PTA (i.e, average of logic and memory use) never exceeded 15%, which was for the experiments with NDP [162], compiled with SDNet 2018.2. In many

Metric	**Device Property**	**Example**
Max # of headers in a single test	PHV size	4Kb
Max # of checks in a single test	# of Stateful ALUs	40
Max # of packets in a single test	Counter width	4 billion
Max test speed	Pipeline Bandwidth	1.6 Tbps
Max packet size	Max Transmission Unit	1514 B

Table 6.4. Additional Evaluation Metrics. Example values are indicative of the proof-of-concept implementations.

cases (e.g., INT, Learning Switch) this number drops to 9%. The blank packet generator required just 0.13% logic overhead, and no memory.

On Tofino, PTA tested a data-plane program on one pipeline using other pipelines. Since resources are not shared between pipelines, PTA does not “take away” resources from the data plane under test. ASIC resource are given, and PTA easily fits in, using the resources noted above.

Test Completion Time. PTA run time includes four components: platform setup (i.e., downloading an FPGA bit file), configuration, test execution time, and results collection and report. The test execution time is test-dependent, i.e., it depends on how long a user wants to send traffic, the number of parameters to explore, etc. For the tests that we ran on NetFPGA, the average overall time was $\sim$110s, including all four components, though for some tests this number was reduced to $\sim$70s. Out of that, the platform setup time, which is a one time process, is $\sim$20s, and test re-configuration, including populating tables, is in the order of seconds. An exhaustive performance test on NetFPGA SUME which tests throughput under each and every supported packet size, with a billion packets per packet size, was $\sim$3000s.

Additional Metrics. Many of PTA’s performance metrics, summarized in Table 6.4, are a property of the hardware target, not PTA. For example, the number of headers depends on the size of the packet header vector (PHV), and the number of verified aspects in a test (e.g., dropped packets, correct headers checks) depends on the number of stateful ALUs in the device.

6.7.2 Bugs Found

Our implementations of PTA enabled us to uncover bugs within different programs and architectures, while covering use cases discussed in section 6.3. Table 6.5 provides a partial list of tests run and bugs found using PTA. The table indicates the name of the program we used for the data plane under test, a brief description of the category of bug, the hardware platform, and whether or not the test passed. Note that when the program name is *Any*, it indicates that the bug was not tied to a particular program. We discuss these particular bugs as they highlight the diversity of test cases that PTA enables.

6.7.2.1 Compiler Checks

PTA was able to find or confirm three bugs in version 8.9.1 of the Barefoot SDE compiler. These bugs were found when integrating with P4v, discussed in Section 6.5.3. In the first bug, (test #01), the compiler generated incorrect byte swapping code (e.g., between big and little endian). This bug has been fixed in the 9.0.0 release of the SDE. The second bug is related to "saturating" an attribute in header fields (test #02); header fields marked as "saturating" always collapse to their minimum value after a "subtract" operation is computed on them. The third bug prevents the implemented data plane from correctly processing "signed" header fields (test #03). Although represented in two's complement form, "signed" fields are treated as unsigned numbers in the hardware, thus generating incorrect results. We reported these bugs to the developer, and they have since been fixed.

6.7.2.2 Functional Tests

We discovered several functional bugs in multiple designs implemented on NetFPGA SUME, including the Verilog and P4 Learning Switch designs, and NDP [162]. First, packets with invalid source MAC addresses pass through the data plane and reach the output network interfaces, even though they should have been dropped before traversing the pipeline (test #04). This bug differs from the Parse Reject bug (test #08), as the the value within the header should be banned, not the header itself. Furthermore, the issue is Ethernet compatibility, not compatibility with the P4 specification. Second, we find that, when the number of entries written to the MAC lookup table exceeds the size of the table, the write pointer will wrap around, and the first entry will be over-written (test #05).

When testing a P4 implementation of in-band network telemetry (INT) [27] on the NetFPGA platform, we detect some missing functionality (test #06). This

```
// Parse packet headers by specifying state
// machine transitions.
parser Parser(packet_in b,
          out Parsed_packet p,
          inout sume_metadata_t sume_metadata) {

   state start {
      b.extract(p.ethernet);
      transition select(p.ethernet.etherType) {
         IPV4_TYPE: parse_ipv4;
         default: reject;
      }
   } // Eliding IPv4 parser
}
```

Figure 6.15. Subset of a P4 program that reject non-IPv4 packets. The behaviour of the bold line is unspecified.

includes missing measurement of switch hop latency, egress port utilisation, or queue congestion status. The design also does not report which rules matched while traversing the data plane or provides information about other flows traversing the same network queues.

A more serious bug in the implementation of INT is the handling of packets with a large instructions count (test #07). The INT specification [27] states that "a device would cease processing an INT packet with an Instruction Count higher than the number of instructions that it is able to support". In our test, we find that if more than five instructions are requested, the program fails to set the Bottom-of Stack (BOS) flag to the last (fifth) INT header.

6.7.2.3 Under-specification Tests

Because different hardware targets have different capabilities and features, they may exhibit different behaviour. And, in some cases, it would be unreasonable to force all targets to have a uniform behaviour, because doing so would add unnecessary complexity to a design, or add additional performance overhead. For such situations, the language specification often leaves the implementation details as up to the compiler.

One example of such behaviour is illustrated in the snippet of code in Figure 6.15, which shows the implementation of a parser in P4 (test #08). It includes logic to extract the Ethernet header and examine the type field of the Ethernet header. If the type field indicates IPv4, the parser will transition to the parser state for extracting IPv4 headers. Otherwise, the program will drop the packet (i.e., `reject`).

The intention of this program is that any non-IPv4 packets should be dropped.

Test#	Program	Category	Description	HW Plat.	Pass/Fail
01	*Any*	Compiler	Byte swapping	Tofino	Fail
02	*Any*	Compiler	Saturating	Tofino	Fail
03	*Any*	Compiler	Signed fields	Tofino	Fail
04	NDP, Switch (P4, Verilog)	Functional	Invalid MAC	NetFPGA	Fail
05	NDP, Switch (P4, Verilog)	Functional	Table wrap	NetFPGA	Fail
06	INT	Functional	INT features	NetFPGA	Fail
07	INT	Functional	Instructions count	NetFPGA	Fail
08	INT	Underspecification	Parser reject	NetFPGA	Fail
09	NDP, Switch (P4, Verilog)	Performance	Write latency	NetFPGA	Fail
10	INT	Performance	Aligned packet sizes	NetFPGA	Fail
11	*Any*	Architecture	Input arbiter	NetFPGA	Fail
12	*Any*	Security	Meltdown	NetFPGA/Tofino	Pass/Pass
13	*Any*	Security	Read headers beyond	NetFPGA/Tofino	Fail/Pass
14	Switch (P4)	Designs	Port MAC	NetFPGA	Fail
15	Switch (Verilog)	Designs	Table wrap	NetFPGA	Fail

Table 6.5. A subset of the tests we ran and bugs found using PTA.

However, the behaviour of the program when compiled using P4→NetFPGA [46] might run counter to user expectations—the packet is forwarded through the programmable pipeline and out of the device.

The reason is because the P4 language spec leaves the choice of how to implement a parser reject state up to the architecture. The SDNet compiler [69] does not implement the reject state as drop and P4→NetFPGA [46] does not use the reject indicator provided by the Simple SUME Switch architecture used by SDNet.

Technically, forwarding the rejected packet is not a bug, since the implementation is not contrary to the specification. However, it does result in unintuitive behaviour that might surprise a developer. And, this behaviour would not necessarily be caught by verification tools like P4v [117] or Vera [116], depending on how they model `reject`.

6.7.2.4 Performance Tests

We evaluate the performance of several P4 and HDL based programmable designs built upon the NetFPGA infrastructure. We first discuss bugs that are specific to a given program. We then discuss bugs that are a property of the NetFPGA infrastructure.

In the NetFPGA Reference Switch design, we discover that the lookup table is not able to sustain subsequent entry updates with packet sizes of less than 385B, due to the write access latency (test #09). When the packet size is 385B or bigger, meaning 13 clock cycles or more between two updates, the design functions as expected.

Running a similar test on the P4-based learning switch resulted in a failure to support consecutive table updates at line rate, regardless of packet size. The root cause to this limitation is the separation of control and data planes, which means that updates to the lookup table must go through the host by design.

An evaluation of the P4-based INT design on NetFPGA yielded interesting performance results (test #10). We find that the data plane can sustain the full internal throughput (50Gbps) only with unaligned packet sizes (e.g., 65B, 97B), but not for data path aligned packet sizes (e.g., 64B, 96B). We expect that this issue is caused by the expansion of the packet within the encrypted data plane module, beyond the INT header added to the packet. This is also the explanation proposed to us by the P4→NetFPGA designers.

6.7.2.5 Architecture Tests

A few of the bugs uncovered by PTA had to do with the architecture of specific designs or with the underlying hardware infrastructure (test #11). For example, initial throughput testing of both HDL-based and P4-based learning switches resulted in a large number of packet drops. The cause was found in the arbiter at the input to the data plane, that turned out not to be work conserving. An additional architecture limitation was discovered at the output of the data plane, at the output queues. Full rate traffic through the data plane under test led to packet drops at the queues, which turned out to be an intentional design choice by the NetFPGA team. They designed the overall supported outputs queues throughput to be circa 40Gbps. We note that PTA found this bug after the platform had already been in use for more than 10 years. The fix has supported 2 more recent NetFPGA-based projects.

6.7.2.6 Security Checks

In the course of working on PTA, we have conducted several experiments, both on P4→NetFPGA and on Tofino, trying to uncover security vulnerabilities. In our exploration, we focused on one aspect of the P4 language, which is *Undefined behaviours* (Section G.2 of P4 Specification v1.1.0 [48]). This includes aspects such as uninitialized variables, accessing header fields of invalid headers, and accessing header stacks with an out of bounds index.

First, we tried to identify the networking-equivalent of a “Meltdown” bug by attempting to infer the contents of previous packets using malformed packets (test #12). In principle, we try to infer the contents of the memory by reading a value of a non-existing header in the packet, in an attempt to use previously stored header contents. This is one form of accessing header fields of invalid headers. Positively, we find that the SDNet compiler returns a zero value for such attempts, providing stateless operation between packets.

In another test, we attempted to read headers beyond the end of the packet, with a similar motivation (test #13). In this case the result was positive as well, with the Tofino switch dropping the “aggressor” packet. SDNet does not allow such operations either, invalidating all parsed bits of the offending packet. This case is interesting, as it touches on the delicate interface between compiler and architecture. Although SDNet guards against such operations, P4→NetFPGA did not handle the error indication from SDNet. Therefore, the unprocessed and partially corrupted packet may still be emitted.

6.7.2.7 Comparing Designs

By comparing seemingly identical designs, we do not identify bugs, as these are covered by previous scenarios. However, we do identify gaps in specification, behaviour, or performance. For example, we compare two implementations of a learning switch: one written in Verilog, and one written in P4. Both designs share the same NetFPGA infrastructure, and differ only in the data plane module. Despite the similarity, we find two differences in functionality. First (test #14), in the P4-based design, two ports cannot be assigned to the same MAC: once a Port-MAC binding has been learned, it cannot be overwritten by other packets. In contrast, in the Verilog-based design, after a Port-MAC binding has been learned, it can be overwritten by other packets. Second (test #15), in the Verilog-based design, overflow happens when exceeding the table size and the first entry is overwritten without any notice (as noted before). In the P4-based design, on the other hand, no overflow happens when exceeding the table size. In principle, such updates are expected to be silently dropped by the control plane. This is a property of the closed-source compiler, which we don’t have visibility to test.

6.7.2.8 Ethics and Corrective Actions

Ethical issues have been considered as part of this research. We have focused on the handling of vulnerabilities and weaknesses discovered in the different designs. Vulnerabilities have been disclosed and discussed with code and platform originators, which also helped us clarify what is considered a bug, a known design limitation, or an unsupported feature. We have further taken a positive approach and contributed code fixes to open source projects (e.g., NetFPGA), as a means to improve their quality based on our findings.

The reject limitation found in P4→NetFPGA was reported to the NetFPGA project and Xilinx Labs. Xilinx Labs have proposed a work-around that enables users to support functionality similar to reject in the pipeline, even though actions as a result of reject is not implemented in the compiler.

We discussed the architecture and performance issues in the NetFPGA Reference designs with the NetFPGA team. The NetFPGA team indicated to us that they were aware of a minimum-access latency limitation for table updates, but not to the discovered extent. We have also contributed a fix to the NetFPGA input arbiter module as part of this work, as well as the packet generation module of PTA.

Bugs in the Barefoot Networks SDE were reported by entering a ticket on the FASTER portal and via personal email communication. All bugs reported in this chapter have since been fixed by the developers.

6.8 Discussion

Our work on PTA introduced us to many of the challenges and limitations of testing network data planes, as well as to the limitations of programmable data planes as a platform for running tests. In this section we summarise these experiences, as we think they may be useful to the broader community.

6.8.1 P4 as a Language for Writing Tests

PTA allows users to write tests in the P4 language. It is natural to question if this is a good choice—after all, P4, by design, trades-off expressiveness for performance.

One reason for using P4 is opportunistic. There are P4 compilers targeting ASICs [1], NPUs [163], FPGAs [69, 38], GPUs [17], and CPUs [164, 165, 166]. Therefore, tests written in P4 should be portable to a wide variety of devices.

On the other hand, P4 is not completely target-independent. For PTA, we needed target-specific implementations of packet-generator code. Overall, though, we found that the target-specific code could be modularized, and that having a portable test written in a common language was attractive.

A second reason for using P4 is that we found that the match-action abstractions offered by the language were well-suited to writing test code. Of course, such a statement is a matter of taste. But, our experiences were that creating tests in P4 was relatively simple, and that they provided a nice high-level abstraction over hardware.

However, we are also using P4 for a task for which it was not intended. And, as a result, there are some language extensions which could be added to P4 to make it more amenable for use with testing. An obvious first step is to extend P4 with language features that allow users to generate specific types of packets. In other words, to provide a programmatic interface for packet generation.

PTA is designed to provide a programmable test framework with an internal view of a network device. However, this internal view is hampered by the closed-source nature of hardware solutions, e.g., modules generated by SDNet are encrypted. Testing would be improved if users could set hooks within the code or access the state or values of certain language constructs. A hook "breaks" the data plane structure, since it allows users to inspect status at a certain point within the design. Such extensions to the P4 language would allow testing in case of a failure, or when the pipeline is stuck.

Supporting watch-points and stepping through code are required future contributions in the field of programmable network devices. Some of the bugs introduced in this work, such as the performance limitation identified in the INT design, can not be easily tracked and tested today even by compiler vendors, with full access to the code.

6.8.2 Under-specification as a Source of Bugs

Because enforcing a uniform behavior on all hardware targets is impractical, some functionality is compiler specific (e.g., uninitialized values). Under-specification in the language may lead to bugs (Section 6.7.2) or security vulnerabilities [151].

Another form of under-specification, i.e., in the interface and division of responsibilities between the data plane and the rest of the device, can also cause errors. Integration bugs are not uncommon in hardware design, but the problem is exacerbated where different technologies interface. This ranges from the integration of a programmable pipeline within an otherwise fixed-function switch, as well as with the integration of externs within P4 programs.

Attending to under-specification requires a closer hardware/software co-design. This is a shift from the common paradigm focusing on the end-host (e.g., operating system and NIC) to a more holistic view that also includes network switches. While such an integrated design is likely to reduce bugs, the disadvantage is that portability may be restricted.

6.8.3 Writing a Test Suite

One of the challenges in testing a network device is creating a comprehensive corpus of tests. Considering the bugs detected by PTA, we identify three classes of tests.

The first class of tests is the “expected” list of tests. This list of tests will include test that were run during the design stage, and need to be validated on a newly produced target, such as tests written for P4v and later translated by PTA. It will also include traditional network tests, such as those using external test equipment, e.g., checking throughput under different stimulus scenarios (test #10).

A second class of tests is tests generated in a response to a bug discovered by a user, e.g., the byte swapping bug (test #01). The goal of such a test would be to (i) validate that the bug is fixed in a newer version of the program. (ii) be used as part of regression tests in future releases (iii) test deployment of bug fixes in the field. The last use case is a good example of the usefulness of PTA, as it enables testing devices in the field without physically connecting them to test equipment.

The third class of tests is tests targeting known potential points of failure that are typically hard to test. An example is testing saturation (test #02), which one would typically verify in a block-level simulation, but would be hard to trigger as part of traditional hardware validation, without carefully crafted targeted tests or additional built in self text (BIST) resources. PTA enables users to craft such tests without per-test resource overhead and while specifically targeting sensitive elements in the design.

PTA’s strength is in its ability to support this wide corpus of tests, from tests generated by other tools, through regression and in-field tests, to tests manually crafted for specific purposes. As tests are target independent in PTA, we envision building such an open source corpus of tests for community benefit.

6.8.4 Coverage and Test Case Generation

One of the advantages of network tests is that they can be run at line rate. On ASIC, that means exhaustive testing is feasible, since billions of packets with billions of header values can be tested every second. On the NetFPGA SUME platform, the supported packet rate is about sixty million packets per second. While these numbers are high, they are insufficient to fully cover all potential cases. For example, to test all combinations of 48bit source Ethernet MAC header, it would take about eight hours on a switch capable of processing ten billion packets per second. Testing the combination of both source and destination MAC header would take $\times 2^{48}$ longer. As switches often need to drop packets where the source and destination address are the same, or some forbidden MAC addresses, this is not a crazy scenario.

While PTA can be configured to run such tests, they are clearly inefficient. PTA supports using random field values (e.g., source and destination MAC address), as well as specific scenarios (identical source and destination MAC addresses). The advantage of PTA is that there is no need to write new P4 code for these different scenarios. One can simply re-configure the test via register access.

6.8.5 PTA and Hardware Targets

This work required intimate knowledge of hardware targets and their tools, despite the promise of P4 providing a higher-level abstraction over hardware.

One constraint that we had faced was the ability to share resources, such as tables, within the data plane. While the P4 specification does not restrict resource sharing, it was not possible on P4-programmable hardware. Therefore, identical tables had to be instantiated twice, taking twice the resources, where a single table would have sufficed. While not sharing resources simplifies the compiler, reduces hazards, and makes pipelining easier, it becomes a challenge on FPGAs, where available resources are the strongest constraint.

A challenge when fitting PTA on FPGA had been timing closure, meaning the ability to fit the design on the FPGA while matching a set of timing constraints. Timing limitations don't necessarily correlate with resource consumption, and are often a property of the program's design. For example, our attempt to test NDP [162] using PTA failed due to timing issues, which exposed a sensitivity in NDP's pipeline design. As engineers often push to maximize the performance and resource usage of their designs, this may remain a limitation.

We also learned that different hardware targets may affect the basic architecture of PTA. For example, when PTA was designed for FPGA targets, the metadata that describes how the packet should be processed was generated by the hardware module that generates the blank packets. On Tofino, in contrast, meta data can't be generated from the pktgen module and needs to be created within the P4 packet generation pipeline, in our case using registers. This insight inspires us to change the FPGA architecture, to support meta data generation within the programmable pipeline as well, yet this needs to be balanced with extra resource usage and less flexibility.

6.9 Chapter Summary

This chapter presented PTA, a portable test architecture for testing data planes. PTA leverages both the P4 language and hardware design to provide flexibility and visibility into programmable network devices. We proposed an enhanced version of PTA's architecture, for improving flexibility through configuration and we demonstrated integration with a formal language verification tool, by designing a P4v-to-PTA translation flow, based on a configurable architecture template. The prototype implementation of PTA, targeting both an FPGA-based device and a network switch ASIC, allowed us to detect numerous hard-to-find bugs. As In-Network Computing becomes increasingly popular, PTA addresses an urgent need for improved tools and techniques for data plane testing and verification.

Chapter 7

Conclusion

Developers have started accelerating a number of different functions in the network, thanks to the introduction of programmable network devices. However, offloading applications to the network is difficult, since programmable networks do not provide adequate development tools and do not offer enough resources for accelerating demanding tasks.

This thesis explores ways to simplify the process of offloading applications to the network, by leveraging the capabilities of programmable network devices. We have made the following contributions:

P4xos, Emu and NRG. We experiment the benefits and the limitations of network offloading by implementing P4xos, that accelerates Paxos consensus protocol in the network. Leveraging the experience gained with P4xos, we propose Emu, a framework that enables application developers to write network services in a high-level language and have them automatically compiled to network hardware, and NRG, a toolset that offers an insight into the network properties of applications, as they run over the network. We collaborate to P4xos, Emu and NRG through specific contributions in the field of hardware-software co-design and network programming.

Heterogeneous network compute hierarchy. We present a new heterogeneous approach to programmable architecture that extends programmable switches with FPGA-based accelerators, for accelerating a greater diversity of applications in the network. Our prototype architecture, implemented using a Barefoot Tofino switch and two NetFPGA SUME cards, demonstrates in-network acceleration of fingerprint computation, a function commonly implemented in datacenter storage appliances. Evaluation demonstrates the feasibility of the proposed approach and the potential benefits it could provide to network computing applications.

PTA. As our leading contribution to this thesis work, we discuss a taxonomy of bugs affecting programmable network devices and, based on the taxonomy, we introduce a portable test architecture for testing data planes that leverages both the P4 language and hardware design to provide flexibility and visibility into programmable network devices. PTA's prototype, targeting both an FPGA-based device and a network switch ASIC, implements automatic integration with a formal language verification tool. Evaluation shows that PTA is able to detect numerous hard-to-find bugs in programmable network devices, running at line-rate. As In-Network Computing becomes increasingly popular, PTA addresses an urgent need for improved tools and techniques for data plane testing and verification.

7.1 Limitations and Future Work

This thesis consists of three components: a set of tools for addressing the challenges of In-Network Computing; a heterogeneous compute hierarchy, for enabling a new class of network-accelerated applications; an architecture for testing programmable network devices. Together, those components have successfully made network offloading easier for software developers. However, this thesis does not provide a definitive solution for in-network acceleration. This section identifies some of the limitations and proposes directions for future research in this area.

Reconfigurability. PTA's flexible architecture demonstrates that, by combining programmability with reconfigurability, network devices can provide the same flexibility of software, at hardware speed. However, programmable network architectures and compilers have been designed for optimising programmability, and limit reconfigurability to reading/writing tables through the control plane. This dissertation shows this is not enough, and suggests that reconfigurability could be enhanced both by making compilers more efficient in allocating available hardware resources and by improving the way those resources are implemented in the hardware, thus breaking the trade-off between flexibility and resource consumption.

Portability. Although both P4 language and PISA aim to implement a portable network development paradigm, our work shows that, in reality, portability is limited. P4 and PISA specify a set of features that each program and each target must comply with, leaving the rest unspecified. This prevents designs using unspecified features from being easily ported across different hardware targets.

A common example is the usage of externs: P4 specifies the interface externs must expose, without forcing the way they are implemented in the hardware. Moreover, there is no minimal common set of externs that must be supported by all the targets. This thesis provides solutions for improving portability, both by exposing more functionalities at a higher level and by introducing general architecture templates to be implemented on different targets. However, we believe future specifications of P4 and PISA should address portability in a more effective way, as suggested by our work.

Reusability. Although the introduction of programmability has enabled offloading a number of applications to the network, new applications are often implemented from scratch. In other words, current network programming paradigm provides very limited reusability. This dissertation presents three solutions for enhancing reusability. First, it provides a library of network functions that can be easily reused by software developers. Second, it introduces new network constructs, that can be leveraged by different applications, running on different targets. Third, it implements tools that can be themselves reused and integrated with a variety of applications. However, similarly to portability, we believe future programmable networks should provide substantial contributions to reusability, in the form of software libraries, standard architecture components, design patterns and commonly used network functionalities.

Programming Languages. Network programming languages, such as P4, offer an abstraction to the underlying programmable network architectures, including PISA. However, this abstraction poses many constraints to developers that either need more visibility into the hardware, or would like to implement features not supported by the match-action paradigm. Although this dissertation provides many hardware-based solutions, great effort needs to be put in extending P4 language for meeting the needs of network applications developers. Possible extensions include exposing information about the internal status of the data plane, adding standard hardware metrics, integrating test and monitoring features at the software level and providing constructs for allocating available hardware resources to non-networking tasks.

Performance Gap. Besides being portable, programmable network technologies aim to be fast. Unfortunately, different programmable hardware targets provide very different performance, with even an order of magnitude gap among devices in the same network infrastructure. As an example, bleeding-edge programmable ASICs have already demonstrated 400Gbps connections, while most of the available smartNICs are still unable to support 100Gbps network interfaces. Part of the work presented in this thesis has been constrained by the performance gap between different hardware targets, mostly in implementing the integration among programmable network switches and FPGA-based accelerators. Although we tried to mitigate the performance gap as much as we could, we believe the whole network hardware ecosystem should put more effort not only in improving the capabilities of programmable network hardware devices, but also in increasing their performance.

7.2 Final Remarks

In summary, the introduction of programmable network devices is having a revolutionary impact on networks. However, being in their infancy, programmable networks are still difficult to use and offer limited compute capabilities. This is preventing developers from offloading more complex functions to the network. The work presented in this dissertation provides a set of solutions for simplifying network programming, testing and monitoring, that could greatly help the whole research community both in offloading more complex functions to the network, and in designing more capable programmable network technologies.

Bibliography

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. CCR, 2014.

[2] 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. SIGCOMM, 2013.

[3] Barefoot Networks. *Tofino*, 2016. `https://barefootnetworks.com/products/brief-tofino/`.

[4] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. IEEE MICRO, 2014.

[5] Shadi Atalla, Andrea Bianco, Robert Birke, and Luca Giraudo. Netfpga-based load balancer for a multi-stage router architecture. WCCAIS, 2014.

[6] Ken Eguro. Automated dynamic reconfiguration for high-performance regular expression searching. FPT, 2009.

[7] Ken Eguro and Ramarathnam Venkatesan. Fpgas for trusted cloud computing. FPL, 2012.

[8] Felix Engelmann, Thomas Lukaseder, Benjamin Erb, Rens van der Heijden, and Frank Kargl. Dynamic packet-filtering in high-speed networks using netfpgas. FGCT, 2014.

[9] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. NSDI, 2016.

[10] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. ASPLOS, 2016.

[11] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. ISCA, 2014.

[12] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Vissers Kees, and Raymond Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. FCCM, 2015.

[13] Louis Woods, Jens Teubner, and Gustavo Alonso. Complex event detection at wire speed with fpgas. VLDB Endowment, 2010.

[14] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. The case for network accelerated query processing. CIDR, 2019.

[15] Yi Qiao, Xiao Kong, Menghao Zhang, Yu Zhou, Mingwei Xu, and Jun Bi. Towards in-network acceleration of erasure coding. SOSR, 2020.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. NSDI, 2018.

[17] Peilong Li and Yan Luo. P4GPU: Accelerate Packet Processing of a P4 Program with a CPU-GPU Heterogeneous Architecture. ANCS, 2016.

[18] Maxeler. *MPC-N Series*, 2020. `https://www.maxeler.com/products/mpc-nseries/`.

[19] Michael Attig and Gordon Brebner. 400 Gb/s programmable packet parsing on a single fpga. ANCS, 2011.

[20] 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. SIGCOMM, 2016.

[21] Kevin Camera, Hayden Kwok-Hay So, and Robert W. Brodersen. An integrated debugging environment for reprogrammble hardware systems. AADEBUG, 2005.

[22] Ixia. *IxNetwork*, 2020. `https://www.ixiacom.com/products/ixnetwork`.

[23] Spirent. *Company's website*, 2020. `https://www.spirent.com/`.

[24] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, et al. OSNT: open source network tester. IEEE Network, 2014.

[25] Murali Ramanujam and Noa Zilberman. Towards a highly scalable network tester. ANCS, 2018.

[26] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. Hypertester: high-performance network testing driven by programmable switches. CONEXT, 2019.

[27] P4 language github reporsitory. *Inband Network Telemetry*, 2017. `https://github.com/p4lang/p4factory/tree/master/apps/int`.

[28] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. NSDI, 2014.

[29] Minlan Yu, Albert G Greenberg, David A Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling Network Performance for Multi-tier Data Center Applications. NSDI, 2011.

[30] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the Blame Game out of Data Centers Operations with NetPoirot. SIGCOMM '16, 2016.

[31] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. SIGCOMM, 2017.

[32] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter when You Can JUMP them! NSDI, 2015.

[33] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable Message Latency in the Cloud. SIGCOMM, 2015.

[34] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. SNC-Meister: Admitting More Tenants with Tail Latency SLOs . SoCC, 2016.

[35] Leslie Lamport. The part-time parliament. TOCS, 1998.

[36] Daniele Sciascia. *libpaxos*, 2013. `https://bitbucket.org/sciascid/libpaxos`.

[37] Satnam Singh and David J. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. FCCM, 2008.

[38] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. SOSR, 2017.

[39] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The case for in-network computing on demand. EuroSys, 2019.

[40] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design Principles for Packet Parsers. ANCS, 2013.

[41] Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. TON, 2020.

[42] Netronome. *NFP-6xxx*, 2013. `http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.60-Networking-epub/HC25.27.620-22nm-Flow-Proc-Stark-Netronome.pdf`.

[43] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Accelerating persistent neural networks at datacenter scale. HOTCHIPS, 2017.

[44] Xilinx. *What is an FPGA?*, 2018. `https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html`.

[45] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga–an open platform for gigabit-rate network switching and routing. MSE, 2007.

[46] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The P4->NetFPGA Workflow for Line-Rate Packet Processing. FPGA, 2019.

[47] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. CCR, 2008.

[48] P4 Language Consortium. *$P4_{16}$ Language Specification Version 1.1.0*, 2018. `https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html`.

[49] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. NSDI, 2015.

[50] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. PODC, 1988.

[51] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. USENIX ATC, 2014.

[52] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. JACM, 1996.

[53] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. SOSP, 2013.

[54] Leslie Lamport. Generalized Consensus and Paxos. Microsoft Research Tech Reports, 2004.

[55] Fernando Pedone and André Schiper. Generic broadcast. DISC, 1999.

[56] Benjamin Reed and Flavio P. Junqueira. A simple totally ordered broadcast protocol. LADIS, 2008.

[57] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. NSDI, 2011.

[58] Leslie Lamport. Fast paxos. DisCom, 2006.

[59] Fernando Pedone and André Schiper. Optimistic atomic broadcast: A pragmatic viewpoint. TCS, 2003.

[60] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. EDCC, 2002.

[61] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. SOSR, 2015.

[62] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. NSDI, 2015.

[63] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. NSDI, 2016.

[64] Noa Zilberman, Philip M. Watts, Charalampos Rotsos, and Andrew W. Moore. Reconfigurable network systems and software-defined networking. Proceedings of the IEEE, 2015.

[65] Gordon Brebner and Weirong Jiang. High-Speed Packet Processing using Reconfigurable Computing. MICRO, 2014.

[66] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.

[67] Eric Chung, John Davis, and Jaewon Lee. Linqits: Big data on little clients. ISCA, 2013.

[68] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. SOSP, 2013.

[69] Xilinx. *SDNet*, 2014. `http://www.xilinx.com/products/design-tools/software-zone/sdnet.html`.

[70] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. IEEE Design and Test of Computers, 2009.

[71] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: Constructing hardware in a scala embedded language. DAC, 2012.

[72] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. OOPSLA, 2010.

[73] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. SIGOPS OSR, 2010.

[74] Alan Shieh, Srikanth Kandula, Albert G Greenberg, and Changhoon Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. HotCloud, 2010.

[75] Angelos Chatzipapas, Dimosthenis Pediaditakis, Charalampos Rotsos, Vincenzo Mancuso, Jon Crowcroft, and Andrew Moore. Challenge: Resolving data center power bill disputes: The energy-performance trade-offs of consolidation. e-Energy, 2015.

[76] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. arXiv, 2017.

[77] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. CCR, 2011.

[78] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. CCR, 2011.

[79] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. CCR, 2011.

[80] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. OSDI, 2016.

[81] Adnan Faisal, Dorina Petriu, and Murray Woodside. Network Latency Impact on Performance of Software Deployed Across Multiple Clouds. CASCON, 2013.

[82] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The internet at the speed of light. HOTNETS, 2014.

[83] Xin Li, Fang Bian, Mark Crovella, Christophe Diot, Ramesh Govindan, Gianluca Iannaccone, and Anukool Lakhina. Detection and identification of network anomalies using sketch subspaces. IMC, 2006.

[84] Tao Li, Shigang Chen, and Yibei Ling. Per-flow traffic measurement through randomized counter sharing. ToN, 2012.

[85] Gianni Antichi, Stefano Giordano, David J. Miller, and Andrew W. Moore. Enabling open-source high speed network monitoring on NetFPGA. NOMS, 2012.

[86] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. DETER: Deterministic TCP replay for performance diagnosis. NSDI, 2019.

[87] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. SIGCOMM, 2015.

[88] Stuart Cheshire. *It's the Latency, Stupid*, 1996. `http://www.stuartcheshire.org/rants/Latency.html`.

[89] David A. Patterson. Latency Lags Bandwidth. Communications of the ACM, 2004.

[90] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. Communications of the ACM, 2017.

[91] Sean Kenneth Barker and Prashant Shenoy. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. MMSys, 2010.

[92] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. INFOCOM, 2010.

[93] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. NSDI, 2013.

[94] Jeffrey C. Mogul and Ramana Rao Kompella. Inferring the Network Latency Requirements of Cloud Tenants. HOTOS, 2015.

[95] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W. , Moore. Where Has My Time Gone? PAM, 2017.

[96] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. SIGCOMM, 2008.

[97] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. Direct code execution: revisiting library OS architecture for reproducible network experiments. CONEXT, 2013.

[98] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. CONEXT, 2012.

[99] Rogério Leão Santos De Oliveira, Ailton Akira Shinoda, Christiane Marie Schweitzer, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. COLCOM, 2014.

[100] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. SOSP, 2017.

[101] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. CCR, 2003.

[102] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. SIGOPS OSR, 2002.

[103] Elliot Jaffe, Danny Bickson, and Scott Kirkpatrick. Everlab: A Production Platform for Research in Network Experimentation and Computation. LISA, 2007.

[104] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. Computer Networks, 2014.

[105] James P. G. Sterbenz, Deep Medhi, Byrav Ramamurthy, Caterina M. Scoglio, David Hutchison, Bernhard Plattner, Tricha Anjali, Andrew Scott, Cort Buffington, and Gregory E. Monaco. The great plains environment for network innovation (GpENI): a programmable testbed for future internet architecture research. LNICST, 2010.

[106] M. Suñé, L. Bergesio, H. Woesner, T. Rothe, A. Köpsel, D. Colle, B. Puype, D. Simeonidou, R. Nejabati, M. Channegowda, M. Kind, T. Dietz, A. Autenrieth, V. Kotronis, E. Salvadori, S. Salsano, M. Körner, and S. Sharma. Design and implementation of the OFELIA FP7 facility: The European OpenFlow testbed. Computer Networks, 2014.

[107] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding Virtualization Capabilities to the Grid'5000 Testbed. CLOSER, 2013.

[108] Jean-Christophe Petkovich, Augusto Oliveira, Yuguang Zhang, Thomas Reidemeister, and Sebastian Fischmeister. DataMill: a distributed heterogeneous infrastructure for robust experimentation. Software Practice and Experience, 2016.

[109] Broadcom. *Knowledge-based processors*, 2020. `https://www.broadcom.com/products/embedded-and-networking-processors/knowledge-based/nla88650`.

[110] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. SOSP, 2017.

[111] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. CCR, 2016.

[112] Daehyeok Kim et al. Generic external memory for switch data planes. HOTNETS, 2018.

[113] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. Lowering the latency of data processing pipelines through FPGA based hardware acceleration. VLDB Endowment, 2019.

[114] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. IEEE MICRO, 2016.

[115] Dongyang Li, Qing Yang, Qingbo Wang, Cril Guyot, Ashwin Narasimha, Dejan Vucinic, and Zvonimir Bandic. A parallel and pipelined architecture for accelerating fingerprint computation in high throughput data storages. FCCM, 2015.

[116] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging p4 programs with vera. SIGCOMM, 2018.

[117] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. SIGCOMM, 2018.

[118] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in p4 programs with assertion-based verification. SOSR, 2018.

[119] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI, 2008.

[120] Hongyi Zeng, Ratul Mahajan, Nick McKeown, George Varghese, Lihua Yuan, and Ming Zhang. Measuring and troubleshooting large operational multipath networks with gray box testing. MSR-TR-2015-55, 2015.

[121] Metro Ethernet Forum. *Carrier Ethernet Service Activation Testing (SAT)*, 2014. `https://www.mef.net/Assets/Technical_Specifications/PDF/MEF_48.pdf`.

[122] Pierre Duhamel and J Rault. Automatic test generation techniques for analog circuits and systems: A review. IEEE transactions on Circuits and Systems, 1979.

[123] Charles Stroud, Eric Lee, Srinivasa Konala, and Miron Abramovici. Using ila testing for bist in fpgas. Test Conference, 1996.

[124] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. SOSR, 2018.

[125] Nik Sultana, Salvator Galea, David Greaves, Marcin Wojcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, Paolo Costa, Peter Pietzuch, Jon Crowcroft, Andrew W Moore, and Noa Zilberman. Emu: Rapid prototyping of networking services. ATC, 2017.

[126] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. SIGCOMM, 2017.

[127] Jan Rüth, René Glebke, Klaus Wehrle, Vedad Causevic, and Sandra Hirche. Towards in-network industrial feedback control. NetCompute, 2018.

[128] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. HOTNETS, 2017.

[129] Huynh Tu Dang. Consensus protocols exploiting network programmability. rérodoc Digital Library, 2019. `https://doc.rero.ch/record/324312?ln=en`.

[130] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. SOSP, 2011.

[131] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. OSDI, 2006.

[132] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. OSDI, 2006.

[133] Leslie Lamport. Lower bounds for asynchronous consensus. DisCom, 2003.

[134] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. PODC, 2007.

[135] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus. Journal of Algorithms, 2004.

[136] Digilent. *NetFPGA Sume Reference Manual*, 2016. `https://reference.digilentinc.com/_media/sume:netfpga-sume_rm.pdf?_ga=2.118020681.1577520214.1588238806-1012309890.1587140029`.

[137] Cypress Semiconductor Corporation. *CY7C25652KV18-500BZC*, 2017. `https://www.cypress.com/file/45151/download`.

[138] Xilinx. *Memory Interface Solutions*, 2010. `https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf`.

[139] NetFPGA. *NetFPGA SUME Reference Switch*, 2017. `https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-Reference-Learning-Switch`.

[140] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. SIGCOMM, 2015.

[141] Brad Fitzpatrick. Distributed caching with memcached. Linux Journal, 2004.

[142] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv, 2016.

[143] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 1998.

[144] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. SIGMETRICS, 2012.

[145] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. HOTNETS, 2019.

[146] Mariette Awad. Fpga supercomputing platforms: A survey. FPL, 2009.

[147] Huynh T Dang, Jaco Hofmann, Yang Liu, Marjan Radi, Dejan Vucinic, Robert Soulé, and Fernando Pedone. Consensus for non-volatile main memory. ICNP, 2018.

[148] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel Tomi Klein. The design of a similarity based deduplication system. SYSTOR, 2009.

[149] Michael O. Rabin. Fingerprinting by random polynomials. 1981.

[150] Peter Hoose. *Monitoring and Troubleshooting: One Engineer's rant*, 2011. `https://archive.nanog.org/meetings/nanog53/presentations/Monday/Hoose.pdf`.

[151] Mihai Dumitru, Dragos Dumitrescu, and Costin Raiciu. Can we exploit buggy p4 programs? SOSR, 2020.

[152] *Mininet*, 2019. `http://mininet.org`.

[153] Dimosthenis Pediaditakis, Charalampos Rotsos, and Andrew W Moore. Faithful reproduction of network experiments. ANCS, 2014.

[154] Mentor-Graphics. *ModelSim ASIC and FPGA Design*, 2019. `https://www.mentor.com/products/fv/modelsim/`.

[155] P4 Language Consortium. *$P4_{16}$ Portable Switch Architecture (PSA)*, 2019. `https://p4.org/p4-spec/docs/PSA.html`.

[156] Gordon Brebner. *Extending the range of P4 programmability*. P4EU, 2018.

[157] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. p4pktgen: Automated test case generation for P4 programs. SOSR, 2018.

[158] Andrei-Alexandru Agape and Madalin Claudiu Danceanu. P4fuzz: A compiler fuzzer for securing p4 programmable dataplanes. Aalborg University, 2018.

[159] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. NSDI, 2018.

[160] Theo Jepsen, Leandro Pacheco de Sousa, Huynh Tu Dang, Fernando Pedone, and Robert Soulé. Gotthard: Network support for transaction processing. SOSR, 2017.

[161] 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. SIGCOMM, 2016.

[162] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. SIGCOMM, 2017.

[163] Netronome. *NFP-6000 Flow Processor*, 2020. `https://www.netronome.com/m/documents/PB_NFP-6000_.pdf`.

[164] P4 Language Consortium. *The P4 Reference Switch*, 2015. `https://github.com/p4lang/behavioral-model`.

[165] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. SIGCOMM, 2016.

[166] Sándor Laki. High-Speed Forwarding: A P4 Compiler with a Hardware Abstraction Library for Intel DPDK. P4 Workshop, 2016.