POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Fully Programming the Data Plane: a Hardware/Software Approach

JEFERSON SANTIAGO DA SILVA
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Avril 2020

© Jeferson Santiago da Silva, 2020.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Fully Programming the Data Plane: a Hardware/Software Approach

présentée par **Jeferson SANTIAGO DA SILVA**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Alejandro QUINTERO, président
Pierre LANGLOIS, membre et directeur de recherche
François-Raymond BOYER, membre et codirecteur de recherche
Guy BOIS, membre
Otmane AIT MOHAMED, membre externe

DEDICATION

To the ♡♡ of my life: Clara 🧍‍♀️ and Sophie 🧍‍♂️.

ACKNOWLEDGEMENTS

Premièrement, je souhaiterais remercier mon directeur de recherche, Pierre. Je pense sincèrement que c'est une personne dont tout le monde a besoin dans sa vie. Pierre a été un directeur exceptionnel, patient et calme, mais qui a su me motiver lorsque cela était nécessaire. Son optimisme est contiagieux! Ses remarques positives et bienveillantes m'ont toujours motivé et m'ont permis de me sentir bien même dans les moments où j'étais « down ». Merci Pierre!

Deuxièmement, je veux exprimer toute ma gratitude à mon codirecteur FRB, avec qui je m'estime chanceux d'avoir eu l'opportunuité de travailler. Je pense sincèrement qu'il est très fort techniquement, du matériel au logiciel. Par ailleurs, nos discussions portant sur le langage C++ et le « clean code »vont me manquer.

J'aimerais aussi remercier l'ensemble des professeurs et du personnel de Polytechnique avec qui j'ai eu l'honneur de travailler, notamment Yvon Savaria et Normand Bélanger.

I'd like to thank my labmates. A few have already left us, but you still remain in my thoughts. Merci Ahmed, Bachir, Imad. Les mots me manquent pour remercier Thomas et Thibaut, que je considère comme de vrais amis, et non simplement comme des collègues de laboratoire. Les gars, merci pour les super bons moments, pour les blagues, pour les discussions, pour les conseils...

My thanks also go to the folks of Kaloom who have been awesome during my internship there. At Kaloom I was able to link my research to real-world problems. Also, I thank Mitacs/Canada and CNPq/Brazil for providing me the funds to conduct this research.

Não poderia esquecer da minha família no Brasil: mãe, pai, irmã, sobrinhas, tios e avós. Sem o apoio de vocês, este desafio jamais se concretizaria. Obrigado por sempre me apoiarem, desde que decidi sair de casa para estudar engenharia em Porto Alegre!

Como sobremesa, o melhor vem no final: Clara e Sophie. Minha esposa Clara merece não só meus agradecimentos mas meu sincero pedido de desculpas. Obrigado por aceitar me acompanhar nessa jornada, por aguentar meu mal humor, por ouvir minhas lamentações, sempre me incentivando com o seu famoso jargão: "vai dar tudo certo". E perdão, principalmente pelo tempo juntos que este trabalho nos furtou.

Obrigado princesa Sophie! Obrigado pelas risadinhas, pelas sonequinhas juntinhos, pelos momentinhos que brincamos, que por vezes aliviaram a pressão que um doutorado impõe. Sophie, papai não é a doutora brinquedos, mas agora também é doutor!

RÉSUMÉ

Les réseaux définis par logiciel — en anglais *Software-Defined Networking* (SDN) — sont apparus ces dernières années comme un nouveau paradigme de réseau. SDN introduit une séparation entre les plans de gestion, de contrôle et de données, permettant à ceux-ci d'évoluer de manière indépendante, rompant ainsi avec la rigidité des réseaux traditionnels. En particulier, dans le plan de données, les avancées récentes ont porté sur la définition des langages de traitement de paquets, tel que P4, et sur la définition d'architectures de commutateurs programmables, par exemple la *Protocol Independent Switch Architecture* (PISA).

Dans cette thèse, nous nous intéressons a l'architecture PISA et évaluons comment exploiter les FPGA comme plateforme de traitement efficace de paquets. Cette problématique est étudiée a trois niveaux d'abstraction : microarchitectural, programmation et architectural.

Au niveau microarchitectural, nous avons proposé une architecture efficace d'un analyseur d'entêtes de paquets pour PISA. L'analyseur de paquets utilise une architecture pipelinée avec propagation en avant — en anglais *feed-forward*. La complexité de l'architecture est réduite par rapport à l'état de l'art grâce a l'utilisation d'optimisations algorithmiques. Finalement, l'architecture est générée par un compilateur P4 vers C++, combiné à un outil de synthèse de haut niveau. La solution proposée atteint un débit de 100 Gb/s avec une latence comparable à celle d'analyseurs d'entêtes de paquets écrits à la main.

Au niveau de la programmation, nous avons proposé une nouvelle méthodologie de conception de synthèse de haut niveau visant à améliorer conjointement la qualité logicielle et matérielle. Nous exploitons les fonctionnalités du C++ moderne pour améliorer à la fois la modularité et la lisibilité du code, tout en conservant (ou améliorant) les résultats du matériel généré. Des exemples de conception utilisant notre méthodologie, incluant pour l'analyseur d'entête de paquets, ont été rendus publics.

Au niveau architectural, nous avons proposé une méthode de cache pour une architecture de plan de données programmable hétérogène, pour laquelle le débit de traitement peut être inégal entre les différentes plateformes utilisées. Pour ce faire, nous avons caractérisé une trace provenant d'un centre de données afin d'identifier les propriétés des paquets pouvant être exploitées par la cache. Ces propriétés ont ensuite été utilisées pour concevoir des politiques d'éviction et de promotion. Une plateforme de simulation a été développée, et permet de démontrer qu'une politique de promotion aléatoire combinée à une politique de promotion heuristique basée sur la fréquence atteint un taux de succès élevé (∼90%) avec des tailles de caches relativement petites (8 k entrées).

ABSTRACT

Software-Defined Networking (SDN) has emerged in recent years as a new network paradigm to de-ossify communication networks. Indeed, by offering a clear separation of network concerns between the management, control, and data planes, SDN allows each of these planes to evolve independently, breaking the rigidity of traditional networks. However, while well spread in the control and management planes, this de-ossification has only recently reached the data plane with the advent of packet processing languages, e.g. P4, and novel programmable switch architectures, e.g. Protocol Independent Switch Architecture (PISA).

In this work, we focus on leveraging the PISA architecture by mainly exploiting the FPGA capabilities for efficient packet processing. In this way, we address this issue at different abstraction levels: i) microarchitectural; ii) programming; and, iii) architectural.

At the microarchitectural level, we have proposed an efficient FPGA-based packet parser architecture, which is a major PISA's component. The proposed packet parser follows a feed-forward pipeline architecture in which the internal microarchitectural has been meticulously optimized for FPGA implementation. The architecture is automatically generated by a P4-to-C++ compiler after several rounds of graph optimizations. The proposed solution achieves 100 Gb/s line rate with latency comparable to hand-written packet parsers. The throughput scales from 10 Gb/s to 160 Gb/s with moderate increase in resource consumption. Both the compiler and the packet parser codebase have been open-sourced to permit reproducibility.

At the programming level, we have proposed a novel High-Level Synthesis (HLS) design methodology aiming at improving software and hardware quality. We have employed this novel methodology when designing the packet parser. In our work, we have exploited features of modern C++ that improves at the same time code modularity and readability while keeping (or improving) the results of the generated hardware. Design examples using our methodology have been publicly released.

At the architectural level, we have proposed a heterogeneous match table caching scheme to alleviate the memory capacity/performance trade-off of current programmable dataplanes. To this end, we have characterized a real-world data center trace to derive caching premises. These premises have been the basis for novel network-aware cache eviction and promotion policies. Our work also includes an open-source simulation and implementation viability analysis of the proposed solution. The results indicate that a simple random promotion policy combined with a heuristic frequency-based promotion policy achieves a high hit ratio (∼90%) with relatively small cache sizes (8 k entries).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

ALU	Arithmetic Logical Unit
ASIC	Application-Specific Integrated Circuits
ASM	Abstract State Machine
BPF	Berkeley Packet Filter
CLI	Command Line Interface
CPU	Central Processing Unit
CRTP	Curiously Recurring Template Pattern
DAG	Directed Acyclic Graph
DC	Data Center
DPI	Deep Packet Inspection
DRY	Don't Repeat Yourself
DSA	Domain Specific Architecture
DSE	Design Space Exploration
DSL	Domain Specific Language
eBPF	extended Berkeley Packet Filter
eLOC	Equivalent Lines of Code
FE	Forwarding Element
FF	Flip-Flop
FIS	Flow Instruction Set
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
HDP	Heterogeneous Data Plane
HLL	High-Level Language
HLS	High-level Synthesis
HW	Hardware
IC	Integrated Circuit
LLVM	Low Level Virtual Machine
LOC	Lines of Code
LPM	Longest Prefix Match
LUT	Lookup Table
MAT	Match-Action Table
MMT	Multiple Match Tables
MpO	Module-per-Object

NCL	Network Classification Language
NP	Network Processor
NPL	Network Programming Language
ONF	Open Networking Foundation
OS	Operating System
PDD	Programmable Dataplane Device
PHV	Packet Header Vector
PISA	Protocol Independent Switch Architecture
QoR	Quality of Results
QoS	Quality of Service
RISC	Reduced Instruction Set Computer
RMT	Reconfigurable Match Tables
ROM	Read-Only Memory
RTL	Register-Transfer Level
SDN	Software-Defined Networking
SMT	Single Match Table
SPMD	Single Program Multiple Data
STL	Standard Template Library
SW	Software
TCAM	Ternary Content-Addressable Memory
TM	Traffic Manager
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
VM	Virtual Machine

CHAPTER 1 INTRODUCTION

1.1 Context and Motivation

Over the last decade, Software-Defined Networking (SDN) [1] has emerged to break the rigidity of computer networks by allowing network administrators to manage the network in a softwarized fashion. Indeed, SDN has decoupled the data and control planes, allowing both to evolve independently. Hence, complex network protocols and algorithms are implemented in software in a centralized controller while data plane devices only execute a series of configurable actions. The SDN paradigm has also changed key abstractions of the networking domain. Instead of a stack of protocols as in traditional network deployments, SDN is based on a clear separation of concerns. These concerns are straightforward:

- Users develop network applications by defining networking rules in a management plane based on the network status reported central entity.
- A centralized controller (control plane) collects network status (congestion status, connectivity). It also communicates with the data plane to install forwarding rules.
- The data plane forwards packets using predefined forwarding rules.

To consolidate the SDN paradigm, McKeown et al. [2] proposed OpenFlow in 2008. OpenFlow defines a standard interface between the controller and the OpenFlow switch[1] and it uses this interface to configure pre-determined actions in data plane devices. This standardization has thus made OpenFlow the *de facto* SDN protocol.

However, recent advances in programmable data plane devices and languages have the potential to undermine OpenFlow's hegemony. Since OpenFlow is standardized, new network applications and protocols cannot be deployed until a new standard revision takes place. Besides, when a new revision is released, switch vendors need to develop new chips, which increases not only the cost but time to market.

To deal with OpenFlow limitations, the P4 language was proposed in 2014 [3]. P4 is a Domain Specific Language (DSL) in which users can arbitrarily specify protocol-agnostic packet processing behavior. In P4, users can arbitrarily define the set of matching fields and the associated actions in case of a match. Also, network programmers can reformat packet headers in any fashion, allowing the development of custom and novel data plane algorithms.

The PISA [4] is a general and realistic abstraction to process P4 programs. PISA is made

[1] The term *switch* describes any network device that forwards data from an input to an output port.

of a programmable packet parser, a pipeline of Match-Action Tables (MATs), optional user-defined external functions and objects, a packet scheduler, and a deparser. By 2013, commercial programmable Application-Specific Integrated Circuits (ASIC) PISA switches had already far crossed the Tb/s barrier [5]. However, such high throughput comes with a programmability cost. Thus, P4 programmers are innovation limited due to resource scarcity (internal memory for stateful applications) or fixed-function modules (preset scheduling algorithms). Indeed, state-of-the-art PISA switches have no more than a few hundreds of megabits of internal memory shared between match tables and stateful user-defined memories.

Current Data Center (DC) applications, such as machine learning and load balancing, demand at the same time high throughput and programmability. These applications have historically run in DC server clusters. However, servers are also responsible for implementing soft-switches to isolate their tenants' Virtual Machines (VMs) and containers. To improve server efficiency, a recent trend has been to employ Field-Programmable Gate Arrays (FPGAs) for DC server offloading [6], [7].

Recent work has also proposed mapping P4 programs to FPGAs [8] as an alternative to inflexible and memory-bound ASIC switches and performance-limited soft-switches. While FPGA-based solutions have augmented the degree of programmability and the memory bandwidth of PISA switches, the performance of FPGA-based switches is far lower than its ASIC counterparts. This is mainly due to the intrinsics of the FPGA microarchitecture which does not include hardwired associative memories (required for implementing match-tables) and the lack of FPGA specific compiler optimizations.

In this work, we show that there is no silver bullet in programmable packet processing. Throughout the next chapters, we study the limitations of current packet processing solutions while proposing enhancements to the state-of-the-art. These enhancements include microarchitectural and compiler optimizations for a PISA module implemented in FPGAs. This is accompanied by a software-friendly FPGA design methodology that can be extended outside the network domain. Also, we propose coupling programmable heterogeneous devices to exploit the potential of each component aiming for efficient packet processing. To make this heterogeneous switch architecture more attractive, we study how a caching scheme split between the data and control plane can minimize reaction time and increase the hit ratio.

1.2 Problem Statement

SDN and OpenFlow have not directly dealt with the subject of data plane programming. Only recent efforts on packet processing languages, such as P4, have considered data plane devices are programmable platforms. Much of these efforts are due to state-of-the-art programmable switches that implement/emulate the PISA architecture.

However, current PISA switches limit the innovation potential brought by P4. Indeed, high-performance PISA switches trade-off programmability and throughput. In a computer architecture analogy, PISA is much like what a Reduced Instruction Set Computer (RISC) processor is: lean and fast. As for RISC, PISA fits the most of network applications, but a few ones suffer due to its limited instruction set.

As P4 is based on an explicit imperative match-action programming paradigm, P4-aware devices require fast associative memories to implement match-tables. Contrary to regular memory structures, associative memories, either Ternary Content-Addressable Memorys (TCAMs) or their algorithmic emulations, are expensive both in terms of silicon area and power. Thus, ASIC switches limit the amount of those memories. As a consequence, ASIC switches limit the number of active sessions which has a trend of increasing with the advent of next-generation mobile communications (>1 M).

This performance/memory trade-off, the infamous memory wall, has been observed since the beginnings of the computer era. Although not a new problem, solutions for it in the domain of programmable dataplanes have yet not been thoroughly studied.

More flexible architectures for packet processing have also been recently explored, such as memory-abundant soft-switches running on commodity servers [9] and FPGAs [8]. However, the benefits of soft-switches are overwhelmed by their low performance that is limited to a few dozens of Gb/s. For that reason, they are mainly used in data center servers for routing packets between the host Operating System (OS) and the tenants' VMs. FPGAs, on the other hand, are more limited than Central Processing Units (CPUs) in terms of programmability. However, FPGAs can outperform soft-switches by at least one order of magnitude in terms of throughput and many orders of magnitude for processing latency.

FPGAs seem a natural alternative for programmable ASIC switches. Nonetheless, the performance gap between FPGAs and ASIC switches (10× at best) needs to be reduced. This starts by investigating how packet processing blocks can be mapped into the FPGA fabric and how the intrinsic FPGA architecture can be leveraged to process packets.

However, widespread adoption of FPGAs is also limited because hardware designers have historically programmed FPGAs using Hardware Description Language (HDL), such as Ver-

ilog and VHDL. Such languages need verbose and explicit microarchitecture details, such as synchronization and pipelining. High-Level Synthesis (HLS) tools have simplified FPGA development by automatically generating HDL codes from behavioral C-based descriptions. Yet, no generalized design methodology and design patterns have been formally proposed to date targeting HLS design. As a consequence, the HLS adoption is mainly limited to hardware designers who understand in-depth methodologies for FPGA design, since one still needs to explicitly enforce microarchitectural details while using higher-abstracted languages.

FPGAs, however, will not scale in performance as modern programmable ASIC switches do. Also, the cost of state-of-the-art FPGAs limits their large scale deployment. Thus, heterogeneous switch architectures, comprising FPGAs and ASICs may be a solution to achieve high performance and programmability, besides their almost infinite pool of computing resources. Nonetheless, this coupling exposes a few challenges. First, the mismatched processing capabilities of these devices require careful mapping of applications to each device according to the processing requirements of each application. Second, a flow cache scheme is required to exploit the performance of programmable switches aiming at maximizing the matches on these devices. In this scenario, an FPGA is seen as the main memory to extend the limited amount of internal memory on programmable switches. Thus, a custom cache policer algorithm is required to coordinate flow migrations from one device to another.

1.3 Research Objectives and Contributions

In this work, we focus on leveraging FPGAs as part of a programmable packet processing system. We investigate the components of a packet processing pipeline and we identify those that are the best fit for FPGAs. In this research, we developed a few of these modules using a novel high-level HLS methodology targeting FPGAs. This methodology exploits high-level constructs of modern C++ to raise the design abstraction while improving performance and code modularity. To exploit the strengths of FPGAs for packet processing, we propose a heterogeneous packet processing pipeline using a programmable ASIC switch and an FPGA. To extract the maximum performance, a novel heterogeneous cache scheme was proposed.

In summary, this research led to three major published contributions:

1. Jeferson Santiago da Silva, François-Raymond Boyer, and J.M. Pierre Langlois. "P4-compatible high-level synthesis of low latency 100 Gb/s streaming packet parsers in FPGAs." *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018.
2. Jeferson Santiago da Silva, François-Raymond Boyer, and J.M. Pierre Langlois. "Module-

per-Object: a human-driven methodology for C++-based high-level synthesis design." *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019.

3. Jeferson Santiago da Silva, Thibaut Stimpfling, Thomas Luinaud, François-Raymond Boyer, and J.M. Pierre Langlois. "Virtually infinite match tables on programmable dataplanes." *Submitted to the ACM SIGCOMM Computer Communication Review*. 2020.

In our first contribution, we designed a programmable packet parser architecture directly derived from a P4 description (§3, [10]). We judiciously designed the parser's microarchitecture to improve performance and reduce FPGA resource usage. The microarchitectural components were described in C++ that were further synthesized using an off-the-shelf HLS compiler. Moreover, we developed part of a P4-to-C++ back-end compiler to automatically generate HLS-oriented C++ class templates. Our compiler also performs a series of graph transformations to improve pipeline efficiency. The proposed packet parser architecture achieves a line-rate greater than 100 Gb/s with latency comparable to hand-written packet parsers and resource utilization similar to the state-of-the-art.

As a second research contribution, we developed a novel high-level and human-driven HLS methodology for FPGAs (§4, [11]). This methodology aims at raising the HLS design abstraction by employing well-known software engineering techniques. The proposed methodology is anchored in five coding style premises: class templates, const variables, Standard Template Library (STL) usage, inheritance and polymorphism, and smart class constructors. In addition to the proposed methodology, our work identified limitations on current HLS tools while providing hints to programmers in how to overcome these limitations. We demonstrated that the achieved hardware Quality of Results (QoR) using our methodology approaches and is sometimes superior to traditional HLS while significantly improving software quality.

As a final contribution, we proposed a novel caching scheme aiming at leveraging heterogeneous programmable dataplanes (§5, [12]). As traditional caching systems may not be directly applied to programmable dataplanes, either due to resource constraints or low performance, we developed novel cache policies based on a real-world data center traces. From the trace analysis, we derived caching premises based on temporal traffic locality, flow duration times, and flow sizes. The proposed caching policies hit ratio evaluation approach the state-of-the-art. We also discussed the feasibility of the proposed caching scheme in the context of current programmable dataplanes.

1.4 Research not Included in this Thesis

Throughout this Ph.D. research, we have conducted several other research works that are not in the focus of this thesis. These supplementary works are summarized as follows:

- *Support for P4 externs* — [13]. In this work, we have studied the P4 language and compiler to support P4 externs. We have implemented a header compression engine as P4 extern objects. We have added to the open-source P4 backend compiler the support for generating arbitrary P4 externs.
- *Heterogenous Programmable Dataplanes* — [14]. We have proposed combining heterogeneous devices to emulate a single logical heterogeneous programmable dataplane. We have prototype such a dataplane made of an FPGA and a soft switch to evaluate its feasibility.
- *Evaluation of FPGAs as programmable switches* — [15]. We have thoroughly studied the microarchitectural aspects of mapping PISA components to FPGAs. Our analyses have found that match tables implementation are the main performance bottleneck. In addition, we have identified the network applications that are suitable for the current FPGA architecture.
- *Mapping P4 match tables to FPGAs* — [16]. In this work, we have exploited balanced binary trees to emulate P4-defined Longest Prefix Match (LPM) tables on FPGAs. We have also proposed a framework to automatically generate the LPM tables hardware which is optimized to improve post-implementation memory efficiency.

1.5 Thesis Outline

This paper-based thesis is organized into seven chapters. Chapter 2 presents a background and the literature review necessary to understand the context of this work. In this chapter, we review the history and the current status of programmable networks and programmable dataplanes. We also recap the DSLs used for packet processing and how data plane programs are compiled/mapped into real architectures. In Chapter 3, we present our first research contribution: an automatic packet parser generation from a P4 program. Chapter 4 describes the high-level HLS design methodology which the core of our second contribution. Chapter 5 presents the proposed caching scheme targeting high-speed heterogeneous programmable switches. In Chapter 6 we discuss the contributions of this thesis and its pertinence in the scope of programmable dataplanes. Finally, Chapter 7 concludes this work.

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW

In this chapter, we provide the necessary background and the literature review for this work. We start by providing the background on programmable networks, from their origins up to the latest developments. We then review the state-of-the-art on programmable dataplanes, including common hardware-software (micro)architectures for packet processing. Then, we present a review of data plane programming languages with the focus on the prominent P4 language. We close this chapter by presenting techniques for mapping packet processing programs to real hardware architectures.

2.1 Programmable Networks

This section review important concepts on programmable networks. We start by presenting a historical recap on the evolution of programmable networks. Then, we present the key factors that contributed to consolidate the SDN paradigm in recent years.

2.1.1 The Long Way to Programmable Networks

Technological leaps do not happen by turning a switch on. It rather takes several, sometimes imperceptible, small steps in between. Rigid computer networks have not turned into flexible fully programmable networks in the blink of an eye. To illustrate this, in 2014, Feamster et al. [17] conducted a historic study from how we transitioned from rigid computer networks to SDN. The authors reviewed the evolution of programmable networks over the past 25 years. Their work identified three eras in programmable network history:

i. The active networks era;
ii. The control and data path separation; and
iii. The OpenFlow era.

Active networks [18], [19] opened the first era of programmable networks. Active networks were precursors of network programmability by allowing users to run custom code in network appliances. To do so, programs executing in network equipment were encapsulated into packets and their execution was based on packet headers. As a consequence, active networks lowered the innovation barrier and paved the path to network virtualization and data plane programming. However, industry skepticism has almost life-sentenced active networks. This skepticism was well-grounded, especially with regards to security issues due to potentially malicious code running in network equipment in addition to the possible reduced performance.

The second era of programmable networks was marked by the *separation of the control and data planes* [20], [21]. Historically, network switches tightly coupled the control and data planes. However, the increasing demand for high throughput network devices pushed packet processing into dedicated hardware. Thus, complex control software could not directly interact with the packet processing logic. As a consequence, efforts for developing open interfaces between the control and data planes proliferated in the network community. Also, this decoupling led to pushing complex control software outside network switches in a centralized network-wise controller. Since no standardized control-data plane interface was defined at the time, a plethora of proprietary ones was developed, which were not necessarily compatible among different switch vendors.

To deal with the standardization issue, a group of researchers at Stanford's Computer Science department created the Ethane project [22]. The Ethane project was the basis for the development of *OpenFlow* [2], which started the third era in programmable networks. OpenFlow is based in a commodity Ethernet switch that uses flow tables for forwarding packets. OpenFlow also defines a standard user interface to provide means to configure the flow tables through a centralized controller. The OpenFlow specification [23], maintained by the Open Networking Foundation (ONF), defines the directives for OpenFlow-aware switches.

An OpenFlow switch has three parts: OpenFlow flow tables, OpenFlow channel, and OpenFlow protocol. Flow tables in OpenFlow switches are organized in a pipelined fashion divided in ingress and egress pipelines. A table matching triggers packet actions and statistics counters. The OpenFlow channel is an out-band link connecting OpenFlow switches to the controller. This channel is used to install forwarding rules on switches and to gather network statistics. The OpenFlow protocol defines the semantics of messages changed between an OpenFlow switch and the controller. The three main roles of an OpenFlow switch are selecting a forwarding rule from a flow table, updating flow statistics counters and applying actions over the packet according to the selected rule. Incoming packets that the switch is not able to process are eventually sent to the controller the slow path for further processing.

More recently, advances in programmable switch architectures (PISA §2.2.2) packet processing languages (P4 §2.3.2) can potentially start a new programmable networks era: the *programmable dataplanes* era (§2.2).

2.1.2 Consolidating the Software-defined Networking Paradigm

Historically, standard network hardware has defined network functionalities rather than user requirements. In today's fast-evolving Internet environment, traditional ASICs switches, which require a long design cycle, are no longer are a viable option. Users want to be able

to deploy new applications at any time using a software platform rather than ordering new switch chips as new requirements appear. Also, the development of new switch ASIC chips takes months to years while user applications change at a much fast pace.

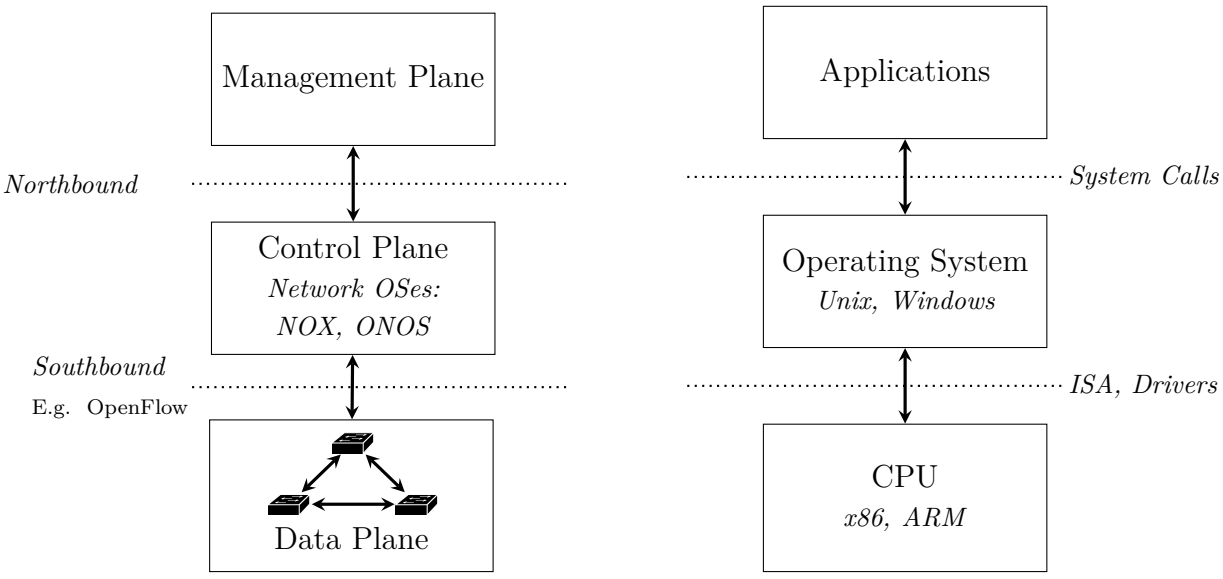
The SDN paradigm emerged from this need. SDN [1] completely decouples the control and data planes in network devices. Therefore, SDN allows network programmability because the lower-level hardware implementations are abstracted for the user. Therefore, users only need to be concerned about developing their desired network applications. Thus, switches are also simplified, becoming a set of actions applied to the packets based on pre-defined matching rules. In SDN, network intelligence is logically centralized in a controller that is responsible for configuring and managing a set of SDN switches. The controller communicates with the switches using a simple and well-defined interface. The SDN paradigm is vendor and protocol agnostic and aims for high scalability, from the enterprise to the carrier level.

While the control and data plane decoupling can be seen as the core of SDN, many other ancient aspects of network research were incorporated into the SDN paradigm. As discussed in §2.1.1, active networks, standardized interfaces, and logically centralized control played a major role to push SDN networks further. Besides, SDN networks provide a global view of the network by managing global network state, which fairly simplifies the development of network applications by offering a "single big switch abstraction" to network administrators.

As Kreutz et al. state in [24], a cornerstone principle in SDN is the clear separation of concerns. These concerns, in turn, can be split into three abstraction layers: management plane, control plane, and data plane. The management plane is responsible for implementing network policies. The control plane distributes these policies to data plane devices that forward packets according to strict network policies. Figure 2.1 presents a graphic view of the three-layer SDN separation alongside its analog in regular computing systems.

As in regular computing systems, the heart of the SDN paradigm is the network operating system, also known as SDN controller [26]–[28]. Historically, managing networks was a difficult and error-prone task since it mainly relied on vendor-specific assembly language or, at best, vendor-specific Command Line Interfaces (CLIs). In contrast, with network OSes, network application developers have at their disposal an abstract view of forwarding devices through well defined and open programming interfaces. As SDN research has mainly focused on SDN controllers and network OSes, a plethora of these was developed in recent years [26]–[29]. These works tackled several key aspects, such as centralized *vs* distributed control, network state distribution and consistency, control plane resilience and scalability, among others.

Applications running in the management plane communicate with the network OS through



(a) Three-layer SDN abstraction (b) Regular computing system

Figure 2.1 SDN *vs* regular computing systems. Adapted from [25]

a *northbound interface*. A myriad of network applications was developed in recent years for SDN networks. Interested readers should refer to a survey by Kreutz et al. [24] for more details. A standard northbound interface provides a common and vendor-agnostic abstraction for network applications accessing the network OS, similarly to what regular computer applications do using system calls. While several research efforts have proposed northbound interfaces [30], [31], to date, no standard northbound solution has been widely employed. In case of distributed controllers, *east/westbound interfaces* (not shown in Figure 2.1) are also required to guarantee network consistency among controllers. Similarly to the northbound, no consensus has been reached on standardized east/westbound interfaces.

On the other hand, OpenFlow [2] has recently been consolidated as the *de facto southbound interface*. Several other southbound interfaces have as well been proposed [32]–[34]. To date, however, none of them have gained the same attention as OpenFlow has. However, advances in programmable dataplanes [3] and data plane programming languages [4], [25] have the potential to changes this state of affairs (§2.2).

2.2 The Programmable Dataplanes Era

SDN has emerged as a technology to break the rigidity of traditional computer networks. Indeed, with the clear separation of control and data planes, more complex network appli-

cations are developed at a fast pace in a software-oriented fashion. However, traditional deployments of SDN have mainly provided flexibility for control plane development. OpenFlow, the *de facto* standard for SDN deployments, has considered data plane equipment as "dumb" forwarding devices, forwarding packets based on pre-defined matching rules against packet header fields. As a consequence, even OpenFlow switches were still rigid switches in their concept. Recent developments on programmable dataplanes (§2.2.2) and packet processing languages [4], [25] promise to unleash the full power of programmable networks.

2.2.1 The Origins

Network Processors (NPs) have dominated the domain of programmable packet processing from the second half of the 1990s to the first half of the 2000s. A common NP architecture is a massively multi-core/multi-thread application-specific instruction-set processor. Many NPs have been commercially developed by several companies worldwide [35], [36]. Packets are normally assigned to a specific processor core and are treated in a run-to-completion fashion. Since NPs are normally implemented on top of Von Neumann machines, complex packet processing, such as Deep Packet Inspection (DPI) and packet classification, is allowed. Nonetheless, such processing flexibility comes with an inherent performance cost. First, packets assigned to different processor cores may undergo different treatments. Thus, packet reordering is common in NPs. Second, NPs cannot guarantee a predictable processing latency because it is a function of the type of treatment that a packet undergoes. Another limiting aspect of NPs is programmability which requires low-level assembly programming or, at best, subsets of the C idiom.

Fixed-function ASICs switches have also been proposed aiming for fast packet processing. In contrast to NPs, ASIC switches offer very high performance with an upper-bounded predictable latency [37]. State-of-the-art ASIC chips have surpassed dozens of Tb/s of aggregated packet switching capacity. As a consequence, these fixed-function devices impose even lower programming abstractions compared to NPs, mainly relying on proprietary user-exposed CLIs for device configuration. However, the main drawback of fixed-function ASIC chips is that these devices come with a preset of supported protocols that are "burned" in silicon. Thus, new protocols cannot be deployed in such switches until a new tape-out revision is released, which compromises time-to-market due to the time-consuming ASIC design flow.

Versatile soft-switches [9], [38] have also found their place in the context of programmable dataplanes. Although, soft-switches have mainly been deployed in data center servers to multiplex network traffic between tenant's VMs or containers. Traditional soft-switch imple-

mentations are developed on top of fast I/O management frameworks, such as DPDK [39], fd.io [40], and netmap [41], running in user-space. Some state-of-the-art soft-switches, such as the Open vSwitch [9], are OpenFlow-aware implementations largely deployed in data center networks. However, since soft-switches are implemented in commodity servers, their performance is limited to a few dozens of Gb/s.

2.2.2 The Protocol Independent Switch Architecture

In 2008, Casado et al. put in place the current view of modern packet forwarding hardware [42]. The authors discuss the lack of a sweet-spot in the tradeoff between hardware simplicity and flexibility in packet forwarding devices. Their work states three clear premises for hardware implementations of network switches:

i. software (control-plane) and hardware (data-plane) must communicate through a clean interface;
ii. the hardware needs to be simple to keep up with the increasing performance requirements; and
iii. hardware and software should cooperate for flexible and efficient packet processing.

Casado's premises were consistent with the prominent OpenFlow development at the time. Besides, Casado et al. proposed the first sketch of a match-action packet processing approach that has dominated packet forwarding since then. Indeed, the OpenFlow's Single Match Table (SMT) switch architecture [2] is a generalization of Casado's work. In Casado's proposal, the hardware piece of the forwarding plane stores matching rules in a TCAM populated in software. Matched packets undergo actions (forward, drop, etc.) in hardware while the non-matched ones are treated in software.

However, fully reconfigurable protocol-agnostic packet forwarding was not proposed until 2013 when Bosshart et al. proposed and implemented the Reconfigurable Match Tables (RMT) architecture [4]. RMT, also known as PISA, is a RISC-like programmable hardware architecture for software-defined packet processing. As shown in Figure 2.2, PISA is made up of a programmable parser, a pipeline of match-action tables, a packet scheduler, and a deparser. In the figure, blue squares represent match stages while yellow trapeziums are action stages.

The RMT architecture is a generalization of the Multiple Match Tables (MMT) architecture. The MMT architecture decomposes a single match table into multiple tables, which reduces hardware complexity by allowing wide and large tables to be implemented with multiple narrow and shallow ones. However, in MMT, the matching values are fixed header fields. In

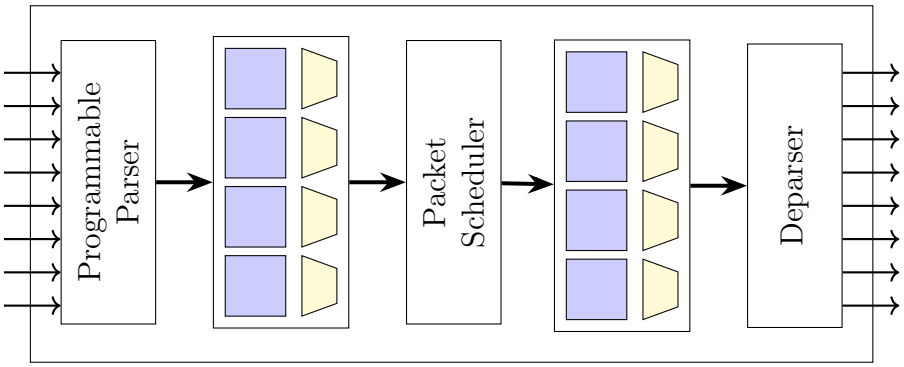


Figure 2.2 PISA internal architecture. Adapted from [4]

contrast, RMT allows arbitrary matching. This is achieved by unique RMT's characteristics:

i. the programmable parser for arbitrary header extraction;
ii. the flexible logical match tables topology; and
iii. configurable action engines for custom action definitions.

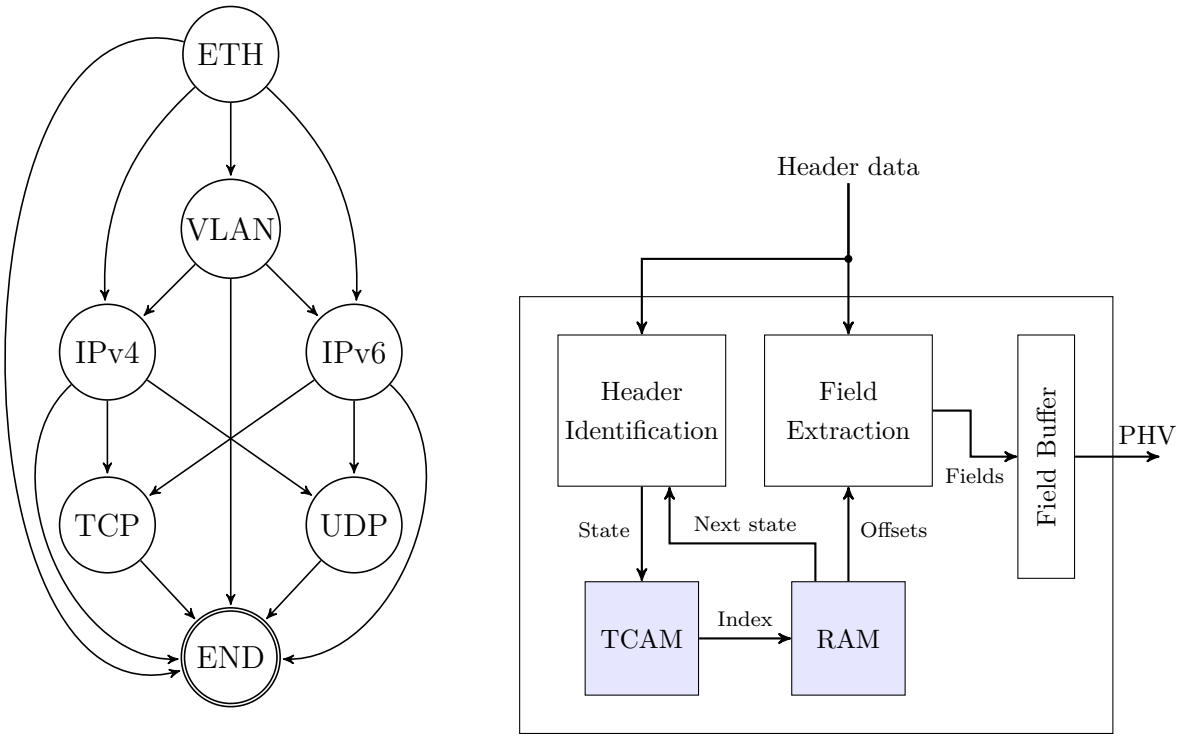
Throughout this section, the microarchitectural components of PISA are explained in details.

Packet Parser

A packet parser identifies the set of protocols supported in a programmable dataplane and extracts header fields for further processing in the match-action stages. A common abstraction to represent a packet parser is through an Abstract State Machine (ASM), where each state represents a protocol while state transitions represent the protocols stack supported by the data plane device. As a result, a packet parser ASM can be graphically represented by Directed Acyclic Graph (DAG), as shown in Figure 2.3a.

Gibb [43] proposed design principles for modern hardware-based programmable packet parsing. Indeed, Gibb 's work is the basis for the packet parser described in the original PISA paper [4]. Gibb proposed an abstract parser machine for both fixed and programmable parsers, as shown in Figure 2.3b.

Fixed and programmable parsers share common modules: header identification, field extraction, and field buffer. The *header identification* module implements the parser ASM identifies the correct headers sequence. The *field extraction* module extracts specific header fields and stores them into the *field buffer*. The output of the field buffer is the Packet Header Vector (PHV) to be used in match-action stages. The blocks in blue, the TCAM and RAM modules, are specific to programmable parsers as they allow in-field parser reconfiguration.



(a) Example of a parser graph

(b) Parser abstract machine. Adapted from [43]

Figure 2.3 Parser graph representation and parser abstract machine

Since Gibb 's proposal, a plethora of recent works have proposed hardware implementation of packet parsers for both FPGAs and ASICs [44]–[50].

Match Stage

The programmable match-action stages are the core of the PISA model. In the matching part, packet header fields (or metadata) are matched against matching rules stored in a match table. For instance, a simple set of matching rules can be:

- Allow a packet entering the network if it is within a specific range of IP addresses.
- Deny access otherwise.

Match operations include exact, range, longest-prefix, and ternary (wildcard) matching. Exact matching exactly compares a search key against the set of keys stored in a table. Range matching checks if a search key is within a specified range of keys. LPM searches for a stored key that matches the maximum number of bits in the searched key. Ternary or wildcard matching allows searching a stored key that matches a wildcardly masked search key. Finally, in PISA, matching operations are agnostic to the table contents.

However, implementing hardware-based match tables is expensive since it requires to implement associative arrays. However, advances in cuckoo hashing [51], [52] improved the memory efficiency for the exact match operation. Cuckoo hash tables implement a series of N independent hash tables HT using N unique hash functions hf. A match for a given key k is detected when $hf_j(k) \in HT_j, \forall\ j \in \{\ 0, \ldots, N-1\ \}$. Hash conflicts are resolved by recursively spreading keys throughout the tables. Bosshart et al. reported a 95% load factor for a cuckoo-based exact match table [4]. As a consequence, cuckoo hash tables became the *de facto* implementation for hardware-based exact match tables.

The other three search types (range, LPM, and ternary), however, still rely on TCAMs. TCAMs are known for being expensive and power-hungry devices. Indeed, TCAM memories exhaustively search entries in parallel in a table which increases power consumption. Moreover, because a search key may match multiple entries due to wildcard masking, a priority encoder is also required to resolve priority, which contributes to increasing power consumption and hardware resource usage. Although Arsovski et al. demonstrated a TCAM design with an overhead of only $\approx 2.7\times$ compared to an SRAM, real-world implementations report an overhead of $6\times \sim 7\times$ [4].

Therefore, due to power and area budget constraints, the amount of internal memory reserved for match operations is limited to a few hundreds of Mb. Algorithmic approaches for emulating content-agnostic lookup tables as well as rule caching schemes [54]–[57] have been (and are still being) proposed to minimize these limitations.

Action Stage

Actions are commonly executed following a match operation. In the case of a match, the match table returns an action (or action identifier) and its parameters (when required). If a miss occurs, the table returns a default action to be executed.

Actions are similar to procedures in traditional computer programs. Actions are made of basic arithmetic, logical, bit-shift, and conditional operations. Actions are straightforwardly mapped to hardware in a format of ALUs. Indeed, the PISA architecture implements these Arithmetic Logical Units (ALUs) using configurable Very Long Instruction Word (VLIW) processors [4].

Packet Scheduling

A packet scheduler decides at what times and in what order packets are sent. However, in PISA, scheduling algorithms cannot be programmable. Only a few preset scheduling

algorithms can be configured at compile time.

Thus, the push-in-first-out (PIFO) queue [58], [59] was proposed as an abstraction upon which a programmable packet scheduler can be built. More recently, the push-in-extract-out (PEIO) queue [60] extended PIFO to support more expressive packet scheduling algorithms, including non-conserving ones.

However, both PIFO and PEIO queues require using associative memories for selecting a packet candidate to be scheduled. As described in §2.2.2, the use of these scarce resources could jeopardize the practicality of programmable packet scheduling in high-speed switches.

Deparser

The deparser recombines the set of modified packet headers before sending a packet to an egress port. Little research efforts have focused on the design of packet deparsers. As a consequence, the deparser is commonly referred as the packet parser's counterpart. Thus, the design of packet deparsers normally follow the premises of parser's implementations.

2.3 Packet Processing Languages

As in regular computing systems, the need for good programming abstractions was also required for packet processing. As data-plane processing became more complex, and as a consequence the hardware architectures running these applications, the network community introduced several packet processing languages over the last two decades. In this section, we first recap the history on packet processing languages then we introduce the P4 language.

2.3.1 The Race for Abstractions

As discussed in §2.2.1, the dominant packet processing hardware at the beginnings of the SDN age was difficult to program, normally requiring low-level machine code.

Packet processing DSLs emerged to alleviate this programming burden. These languages contrast with regular general-purpose programming languages by exposing application-aware semantics that is normally associated with a particular Domain Specific Architecture (DSA).

The Network Classification Language (NCL) was one of the first DSLs designed for packet processing. NCL was a DSL designed by Intel for its IXA class of NPs. An NCL program defines a set of classification rules and the actions in case a match. In NCL, programmers specify packet headers and the parser state machine. Classification rules are defined as if-like statements. NCL supports table operations including match table creation and lookups.

Click [61] and its NP-Click [62] variation are other examples of packet processing DSLs. Click was initially intended to simplify router design. An IP router can be created by connecting basic blocks (elements) through a directed graph structure. Graph edges represent the packet flow throughout the router. Elements in Click are C++ objects, which are provided to the user as libraries. NP-Click is a Click variation tailored for NPs. NP-Click provides an intermediate programming model that exposes only the required underneath hardware details to the programmer. Both Click and NP-Click perform relatively well when compared to traditional lower-level implementations (less than 10% of performance overhead).

In 2009, Duncan and Jungck proposed packetC [63], a C-based DSL tailored for packet processing. PacketC keeps the semantics of the standard C99 while introducing specifics for packet processing, such as packet types, databases, and searchsets. packetC follows a Single Program Multiple Data (SPMD) parallel programming model in which a program is executed across multiple threads that share memory. Although packetC eases the task of programming a network device, its abstraction is too low and close to the hardware.

In 2013, Song introduced POF [25], a protocol-oblivious packet processing language targeting network processors. In POF, the packet parser is configured by a controller. Table lookups are defined by simple $\langle$ offset, length $\rangle$ tuples to guarantee protocol independence. Moreover, POF explores a generic Flow Instruction Set (FIS) [31] to compose complex actions from basic instructions. In 2015, Song et al. [64] proposed an abstract forwarding model to expand the POF supportability to different hardware architectures.

Brebner and Jiang developed the PX language in 2014 [65]. PX is an object-oriented packet processing language targeting Xilinx FPGA devices. PX syntax and semantics resemble C++, however, PX also includes packet specific built-in classes, such as parsers, classifiers, and search engines.

2.3.2 The P4 Language

By December of 2014, the OpenFlow standard version 1.5.0 [23] had already defined up to 44 header fields of several protocols that should be parsed in a network switch.

In this context, still in 2014, P4 [3] emerged as a protocol-agnostic alternative for describing packet processing. Today, the P4 consortium[1] is responsible for maintaining the P4 specification, the workgroups, and the open-source code base for the reference compiler.

P4 follows an imperative match-action-based programming model. In its first version, today known as $P4_{14}$, the P4 execution model was based on a fixed abstract switch model similar to

[1] https://p4.org

PISA. The main components of a $P4_{14}$ program are the header definitions, the packet parser, the action definitions, the match tables, and control blocks.

In P4, packet headers are defined in structures similar to C structs. Each of these structures is made of header fields, defined as an arbitrary stream of bits. Similarly, per-packet metadata can be described using similar constructs.

The packet parser is defined as a parser state machine. This state machine evaluates header fields and/or metadata to derive the set of supported protocols as well as extracts header fields to be used in match-action tables.

Match tables have particular semantics in P4. A P4 program can only read from a match table. Table entries are populated and modified in-field using a run-time program that interfaces with the control plane. The user defines the search type (exact, range, LPM, and ternary), the search key made of one or more header fields (or metadata), and the list of actions to be executed following a match/miss. Optional parameters include the table size and a default action in case of a table miss.

Actions are similar to procedures in general purpose programming languages. Actions are made of basic P4 primitives, such as additions, subtractions, header field modifications. Actions may have execution parameters, either fixed or as a result of a match table lookup operation. Moreover, $P4_{14}$ does not allow conditionals in actions.

The control block manages the program control flow using imperative statements. Tables and actions are applied inside control blocks. Conditionals are allowed within control blocks in a form of if-else statements. Packet modifications are also permitted in control blocks, including header (in)validation. $P4_{14}$ has no specific semantics for packet deparsing. The deparsing logic is inferred from the parser graph.

The P4 compiler compiles a P4 program into two DAGs: a parser graph and a table dependency graph. The parser graph is derived from the parser state machine and it indicates the dependency between headers. The table dependency graph is generated by analyzing the program flow in control blocks and it indicates the order in which tables need to be applied.

In 2016, P4 underwent a major language review [66]. The newest P4 version, known as $P4_{16}$, strengthens the language semantics (stronger types, valid type casts) and completely separates the language from the architecture it is compiled into. However, $P4_{16}$ broke backwards compatibility with $P4_{14}$. The enhancements brought by $P4_{16}$ can be summarized as follows:

- Language conciseness: the number of keywords was reduced by a factor 2×;
- Language-architecture separation;
- Explicit deparsing logic through emit statements;

- Formal support for externs;
- Reference compiler refactoring.

The P4 consortium encourages P4 programmers to use $P4_{16}$. $P4_{16}$ has better support for type checking, portability, and it has stronger semantics. The current compiler design includes several optimization passes, including life-time analysis and dead-code removal. As members of an open-source community, users are also encouraged to report bugs and propose modifications to the language/compiler.

2.4 Compiling Packet Processing Programs

This section cover recent research efforts on compiling packet processing programs. We start by presenting the mapping of these programs to CPUs and programmable switches. Then, we present recent works on mapping network applications to FPGAs.

2.4.1 CPUs and Programmable Switches

Soft-switches have been largely deployed in data centers for traffic forwarding between VMs in servers where OVS [9] has been consolidated as the OpenFlow-aware alternative. In 2016, Shahbaz et al. proposed PISCES [67], a protocol independent soft-switch. PISCES builds upon the classic OVS with specific back-end modifications to support P4. P4 programs are compiled into OVS-specific C code, replacing hardwired OVS software components (parser, match tables, and actions) by custom P4-derived code.

Berkeley Packet Filter (BPF) is a standard packet filtering mechanism employed in Unix-based systems. extended Berkeley Packet Filter (eBPF) is a BPF derivation for virtualized environments. Tu et al. proposed an eBPF back-end supporting P4 [68]. P4 programs are compiled into C code which is afterward compiled into eBPF programs using Low Level Virtual Machine (LLVM) back-end.

Mapping P4 programs to programmable PISA-like switches have mainly been done in the industry. Barefoot Tofino was the first proposed P4-aware programmable switch. P4 programs are compiled into a programmable parser and match-action pipelines. While the first Tofino™ version[2] achieves up to 6.4 Tb/s line rate and its second version achieves 12.8 Tb/s[3], it has several architecture limitations that may limit the innovation potential of P4 programmers. First, the PHV is limited to a few hundred bytes, which limits the number of parsed headers (e.g. deep header encapsulation). Second, mapping a P4 program into Tofino™ is

[2] https://www.barefootnetworks.com/products/brief-tofino/
[3] https://www.barefootnetworks.com/products/brief-tofino-2/

an ILP problem which means [69] that the program either compiles and runs at line rate or does not compile at all. Many factors may contribute to failing the compilation process. Table sizes and dependencies are such constraints directly exposed to programmers.

Cisco has recently released its new Cisco Silicon One™ Q100[4] switch which supports P4. Broadcom, on the other hand, has developed its own packet processing language called Network Programming Language (NPL)[5] for its new Trident 4 and Jericho 2 switch families[6]. At this moment, no much details regarding both Cisco or Broadcom architectures nor how packet processing programs are mapped into them are known. However, programmers are likely to face similar problems to ones observed in Tofino™.

2.4.2 FPGAs

Recent industrial and academic works proposed mapping data plane programs to FPGAs. Xilinx SDNet[7] is a proprietary tool that allows mapping PX programs [65] to Xilinx FPGAs. Recent versions of Xilinx SDNet [70] also support to P4 programs which are compiled into PX before being implemented on an FPGA. Ibanez et al. [71] used Xilinx SDNet to map P4 programs to the off-the-shelf NetFPGA board [72]. Netcope Technologies' P4-to-VHDL[8] is a similar commercial tool that automatically generates RTL descriptions from P4 programs.

P4FPGA [73] is an open-source P4-to-FPGA compiler. P4FPGA builds upon the open-source front-end compiler maintained by the P4 consortium while adding an FPGA-specific back-end. The back-end compiler generates BlueSpec System Verilog code which is further compiled into synthesizable Verilog. P4FPGA is vendor-independent and has been demonstrated in both Xilinx and Intel FPGAs. However, the performance of P4FPGA is limited to few dozens of Gb/s and relies on a proprietary BlueSpec compiler.

Recent works proposed specific microarchitectures for packet processing. Some of them were accompanied by frameworks for hardware generation from P4. Benácek et al. proposed P4-to-VHDL [44] for generating packet parsers from P4. P4-to-VHDL is an HLS-like tool that builds upon previous works on packet parser microarchitecture [74], [75]. Similarly, Benáček et al. extended P4-to-VHDL to also generate the packet deparser logic [76]. Also derived from [44], the work by Cabal et al. presented a packet parser architecture for terabit networks [77].

Mapping agnostic MATs on FPGAs were also proposed. Kekely et al. proposed a hybrid Cuckoo-tree approach for implementing exact and LPM tables. Pure exact match Cuckoo

[4] https://www.cisco.com/c/en/us/solutions/service-provider/innovation/silicon-one.html
[5] https://nplang.org/
[6] https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale
[7] https://www.xilinx.com/sdnet.html
[8] https://www.netcope.com/en/products/p4-to-vhdl

approaches were also proposed for high-speed networks achieving over terabit throughput [79], [80]. The generation of P4-based match-action tables were studied in [16], [81]. Pontarelli et al. proposed a RISC-like approach for implementing primitive actions on FPGAs [82].

Several other works have targeted data plane realization on FPGAs, although not P4 compatible. ReClick [83] is a Click-like language and compiler targeting network virtualization on FPGAs. Similarly, ClickNP [84] is also inspired in Click aiming at accelerating network functions. ClickNP is a C-like language and uses off-the-shelf HLS tools to generate FPGA-specific code. Emu [85] is a standard C# library for implementing network functions on FPGAs. Eran et al. [86] proposed a similar library for data plane acceleration on FPGAs but written in modern C++.

2.5 Chapter Conclusion

SDN beyond OpenFlow. Although OpenFlow has consolidated its place in SDN environments, advances in programmables dataplanes and languages, notably PISA and P4, may jeopardize its hegemony. The PISA+P4 combination is a powerful toolkit for describing agnostic packet processing at the data plane level, an aspect that OpenFlow had not yet covered. However, research regarding P4 is still maturing and many research avenues are yet to be explored.

P4 beyond PISA. PISA is indeed a realistic architecture for processing P4 programs at wire speed. However, the programming expressiveness allowed by P4 may be limited due to PISA's architectural constraints. Other devices, such as FPGAs and CPUs, have been recently explored for packet processing, however, a sweet spot for this has not yet been found. Heterogeneous data planes may be a potential solution to fill this gap. However, research regarding them is in preliminary research stages with many open questions, such as caching for heterogeneous match tables.

FPGAs beyond RTL. HLS tools and DSLs, such as P4, have made FPGAs accessible for a wider audience. Indeed, there is no more need for FPGA experts for developing and testing network applications. However, specialized FPGA microarchitectures, compilers, and good programming abstractions are still needed for more efficient P4-based packet processing.

CHAPTER 3 ARTICLE 1: P4-COMPATIBLE HIGH-LEVEL SYNTHESIS OF LOW LATENCY 100 Gb/s STREAMING PACKET PARSERS IN FPGAS

Authors: Jeferson Santiago da Silva, François-Raymond Boyer, and J.M. Pierre Langlois.
Published in: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.

Abstract

Packet parsing is a key step in SDN-aware devices. Packet parsers in SDN networks need to be both reconfigurable and fast, to support the evolving network protocols and the increasing multi-gigabit data rates. The combination of packet processing languages with FPGAs seems to be the perfect match for these requirements.

In this work, we develop an open-source FPGA-based configurable architecture for arbitrary packet parsing to be used in SDN networks. We generate low latency and high-speed streaming packet parsers directly from a packet processing program. Our architecture is pipelined and entirely modeled using templated `C++` classes. The pipeline layout is derived from a parser graph that corresponds to a P4 code after a series of graph transformation rounds. The RTL code is generated from the `C++` description using Xilinx Vivado HLS and synthesized with Xilinx Vivado. Our architecture achieves a 100 Gb/s data rate in a Xilinx Virtex-7 FPGA while reducing the latency by 45% and the LUT usage by 40% compared to the state-of-the-art.

3.1 Introduction

The emergence of recent network applications have opened new doors to FPGA devices. Dataplane realization in Software-Defined Networking (SDN) [87] is an example [8], [88] of such applications. In SDN networks, the data and control planes are decoupled, and they can evolve independently of each other. When new protocols are deployed in a centralized intelligent controller, new forwarding rules are compiled to the data plane element without changing the underlying hardware. FPGAs, therefore, offer the right degree of programmability expected by these networks, by offering fine grain programmability with sufficient and power-efficient performance.

A standard SDN Forwarding Element (FE) is normally implemented in a pipelined-fashion [4]. Incoming packets are parsed in order to extract header fields to be matched in the

processing pipelines. These pipelines are organized as a sequence of match-action tables. In SDN FEs, a packet parser is expected to be programmable, and it can be reconfigured at run time whenever new protocols are deployed.

Recent packet processing programming languages, such as POF [25] and P4 [3], allow describing agnostic data plane forwarding behavior. Using such languages, a network programmer can specify a packet parser to indicate which header fields are to be extracted. He can as well define which tables are to be applied, and the correct order in which they will be applied.

The main focus of this work is to propose a high-level and configurable approach for packet parser generation from P4 programs. Our design follows a configurable pipelined architecture described in `C++`. The pipeline layout and the header layout templates are generated by a script after the P4 compilation.

The contributions of this paper are classified into two classes: architectural and microarchitectural. The summary of the architectural contributions of this work is listed as follows:

- an open-source framework for generation of programmable packet parsers[1] described in a packet processing language;
- a modular and configurable hardware architecture for streaming packet parsing in FPGAs; and
- a graph transformation algorithm to improve the parser pipeline efficiency.

The contributions related to the microarchitectural improvements are as follows:

- a data-bus aligned pipelined architecture for reducing the complexity in the header analysis; and
- a lookup table approach for fast parallel barrel-shifter implementation.

The rest of this paper is organized as follows. Section 3.2 presents a review of the literature, Section 3.3 draws the methodology adopted in this work, Section 3.4 shows the experimental results, and Section 3.5 draws the conclusions.

3.2 Related Work

Packet processing languages. The SDN [87] paradigm has brought programmability to the network environment. OpenFlow [23] is the standard protocol to implement the SDN networks. However, the OpenFlow realization [89] is protocol-dependent, which limits the genericity expected in SDN.

Song [25] presents the POF language. POF is a protocol-agnostic packet processing language,

[1] Available at https://github.com/engjefersonsantiago/P4HLS

where the user can define the behavior of the network applications. A POF program is composed of a programmable parser and match-action tables.

P4 [3] is an emergent protocol-independent packet processing language. P4 provides a simple network dialect to describe the packet processing. The main components of a P4 program are the header declarations, packet parser state machine, match-action tables, actions, and the control program. Recently, P4 has gained adoption in both academia and industry, and this is why we have chosen P4 as the packet processing language in this work.

Packet parsers design. Gibb *et al.* present in [43] a methodology to design fixed and programmable high-speed packet parsers. However, this work did not show results for FPGA implementation.

Attig and Brebner [90] propose a 400 Gb/s programmable parser targeting a Xilinx Virtex-7 FPGA. Their methodology includes a domain specific language to describe packet parsers, a modular and pipelined hardware architecture, and a parser compiler. The deep pipeline of this architecture allows very high throughput at the expense of longer latencies.

Benácek *et al.* [44] present an automatic high-speed P4-to-VHDL packet parser generator targeting FPGA devices. The packet parser hardware architecture is composed of a set of configurable parser engines [74] in a pipelined-fashion. The generated parsers achieve 100 Gb/s for a fairly complex set of headers, however the results showed 100% overhead in terms of latency and resources consumption when compared to a hand-written VHDL implementation.

P4-to-FPGA mappers have been recently proposed [8], [70]. P4-SDNet translator is partially compatible with the $P4_{16}$ specification and it maps a P4 code to custom Xilinx FPGA logic. One limitation of P4-SDNet is the lack of support for variable-sized headers.

In this work, we deal with some of the pitfalls of previous works [44], [90], trading-off design effort, latency, performance, and resources usage. Our pipeline layout leads to lower latencies compared to the literature [44], [90]. Moreover, the FPGA resource consumption in terms of Lookup Tables (LUTs) is reduced compared to [44], since instead of generating each parser code we parametrize generic hand-written templated `C++` classes targeted to FPGA implementation.

3.3 Design Methodology

This section presents the methodology followed in this work. Section 3.3.1 draws the high-level architectural view. Section 3.3.2 deals with details on microarchitectural aspects. Sec-

tion 3.3.3 presents our method to generate the parser pipeline.

3.3.1 High-Level Architecture

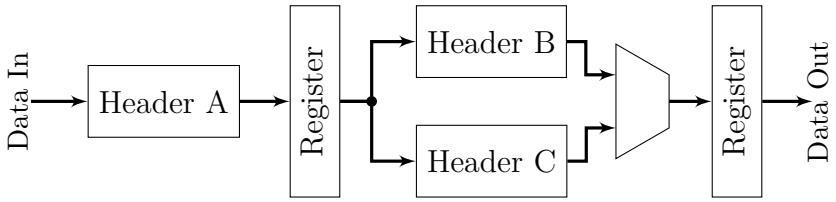
A packet parser can be seen at a high-level as a Directed Acyclic Graph (DAG), where nodes represent protocols and edges are protocol transitions. A parser is implemented as an Abstract State Machine (ASM), performing state transition evaluations at each parser state. States belonging to the path connecting the first state to the last state in the ASM compose the set of supported protocols of an FE.

Figure 3.1a depicts the high-level view of the packet parser realization proposed in this work. The proposed architecture is a streaming packet parser, requiring no packet storage. Header instances are organized in a pipelined-fashion. Headers that share the same previous states are processed in parallel. Throughout this work, we say that those headers belong to the same parser (graph) level. The depth of the parser pipeline is the length of the longest path in the parser graph. For sake of standardization, thick arrows in the figures throughout this work indicate buses, while thin arrows represent single signals.

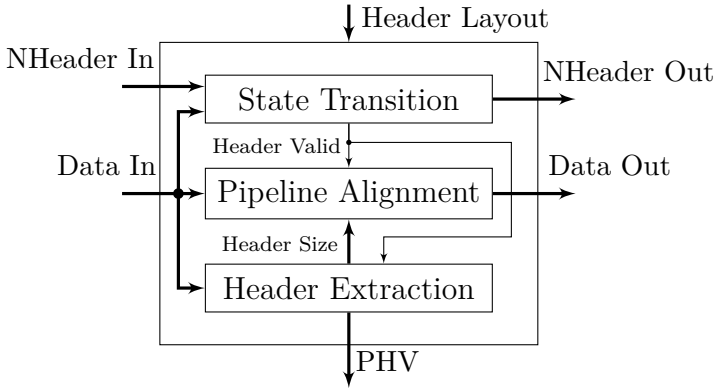
The internal header block architecture is shown in Figure 3.1b. This block was carefully described using templated `C++` classes to offer the right degree of configurability required by the most varied set of protocol headers this architecture is intended to support. This design choice was also taken to improve bit accuracy by accordingly setting arbitrary integer variables, reducing FPGA resources usage.

In Figure 3.1b, the *Header Layout* is a configuration parameter. It is a set of data structures required to initialize the processing objects. It includes key match offsets and sizes for protocol matching, lookup tables to determine data shift values, expressions to determine the header size, last header indication, and so forth. *Data In* is a data structure that contains the incoming data to be processed in a header instance. It is composed of the data bus to be analyzed and some metadata. These metadata include data start and finish information for a given packet and packet identifier. The packet identifier is used to keep track of the packet throughout the processing pipeline and to identify which headers belong to the same packet. *NHeader In* is assigned by the previous header instance indicating which is the next header to be processed. *PHV* is a data structure containing the extracted fields. It includes the extracted data, number of bits extracted, a data valid information, and header and packet identifier. Signals labelled with *In* and *Out* are mirrored, which means that *In* signals undergo modifications before being forwarded to *Out*.

Internal sub-blocks execute in parallel with minimum data dependency. In fact, only the



(a) High-level packet parser pipeline layout



(b) Internal header block architecture

Figure 3.1 High-level architecture

Header Valid information must propagate among the blocks within the same clock cycle and it is generated from a basic combinational logic. *Header Size* also transits from the *Header Extraction* to the *Pipeline Alignment* module. However, this information is only required in the next cycle, which does not constitute a true data hazard.

3.3.2 Microarchitectural Aspects

This subsection presents microarchitectural aspects of this work. We start by presenting the state transition block. Details of the header extraction module are drawn followed by the pipeline alignment block. Then, we present the case of variable-sized headers.

State Transition Block

Figure 3.2 shows the state transition block which implements part of the ASM that represents the whole parser. Each *state* (header) of this ASM performs state transition evaluations by observing a specific field in the header and matching against a table storing the supported next headers for a given state. In this work, this table is filled at compilation time and it is part of what we call *Header Layout*.

The state transition block uses only barrel-shifters, counters, and comparators to perform

state evaluations. Such operations can be easily done in an FPGA within a single clock cycle.

In Figure 3.2, *validHeader* is the result of a comparison between the *nextHeaderIn* and *thisHeader*. *thisHeader* is hardwired and it is part of the header layout. *validHeader* is used as an enable signal for all stateful components in the header instance. *ReceivedBits* is a counter that keeps track of the number of bits received in the same header. This information is used to check if the current data window belongs to the same window in which the *Key-Value* is placed in (*KeyLocation*). A barrel-shifter is used to shift the *DataIn* and to align it with the *KeyMask*. The bitwise AND (&) operation after the barrel-shifter guarantees this alignment. Finally, the *KeyMatch* compares the key aligned input data and the key table. If a match is found, the *NextHeader* is assigned to the value corresponding to the match and the *NextHeaderValid* is set. *HeaderException* is asserted otherwise.

Header Extraction Block

Figure 3.3 shows the header extraction block which retrieves the header information from a raw input data stream. Similarly to the state transition block, this module is implemented using barrel-shifters, comparators, and counters. Additionally, this module calculates header sizes derived from the raw input data in case of variable-sized headers. For fixed-sized headers, the header size information is hardwired at compile time.

In the header extraction module architecture, the counter *ReceivedWords* is used to delimit the header boundaries for comparison with the *HeaderSize*. It is also used to index a table that stores the shift amounts for the barrel-shifter. This table is fixed and it is filled at compile time. The bitwise OR (|) acts as an accumulator, receiving the current shifted and value accumulating it with the results from previous cycles. *HeaderDone* indicates that a header has been completely extracted.

The *SizeDetector* sub-block is hardwired for fixed-sized headers. For variable-sized headers, this sub-block has a behavior similar to the state transition module, returning the header size and the value of the field corresponding to the header size. More details regarding variable-sized headers are given in Section 3.3.2.

Pipeline Alignment Block

Unlike previous works, we opt for a bus-aligned pipeline architecture. That means that each stage in the parser pipeline aligns the incoming data stream before sending it to the next stage. This design choice reduces the complexity of the data offset calculation at the beginning of a stage. The bus alignment is done in parallel with other tasks within a stage and therefore

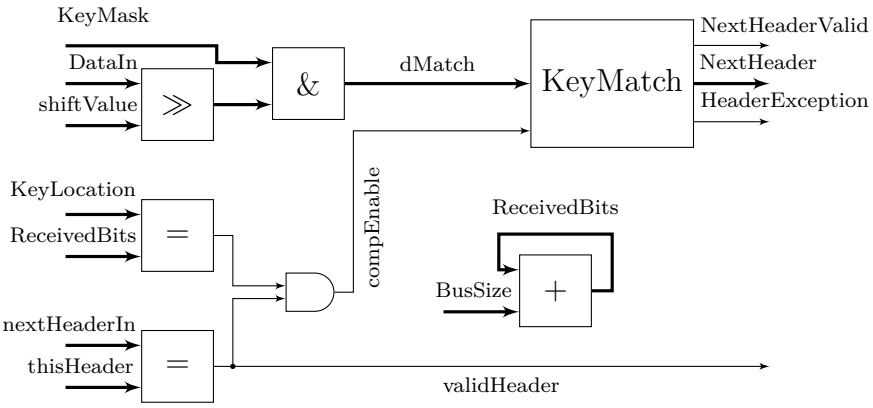


Figure 3.2 Station transition block

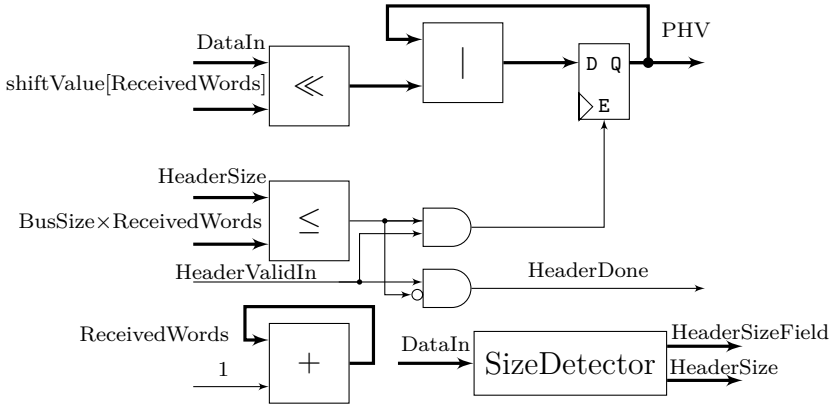


Figure 3.3 Header extraction block

has a low overall performance impact. The pipeline alignment block microarchitecture is depicted in Figure 3.4.

This block delays the input data and performs bit-shifts to remove the already extracted data at the same parser stage. Shift amounts are functions of the header size and the bus size. In the case of fixed-size headers, these shift amounts are hardwired. For variable-sized headers, they are calculated by the *ShiftAmount*, which is explained in more details in Section 3.3.2.

The output bus is then composed of data belonging to the current input data stream and from the previous cycle. When the current header instance is not to be processed, in the case where *HeaderValidIn* is not set, this block just passes the input data to the output bus, playing the role of a bypass unit.

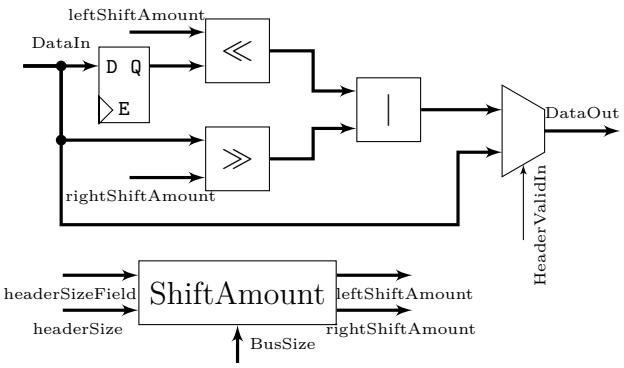


Figure 3.4 Pipeline alignment block

Handling Variable-sized Headers

It is not unusual to have a network protocol in which the header size is unknown until the packet arrives at a network equipment. The header size is inferred from a header field. IPv4 is such an example.

One approach to handle variable-sized headers would be to directly generate the required arithmetic circuit from the high-level packet processing program. However, this is an inefficient option based on our bus-aligned pipeline layout. In our architecture, supporting variable-sized headers would require dynamic barrel-shifters. Recall that a brute-force approach to design barrel-shifters uses a chain of multiplexers, resulting in $O(N\ log(N))$ and $O(log(N))$ space and time complexity respectively, which compromises both FPGA resources and performance.

To avoid dynamic barrel-shifters, we are inspired by a technique available in modern high-level programming languages known as template metaprogramming. Template metaprogramming uses the compiler capabilities to compute expressions at compilation time, improving the application performance. Based on this technique, during the P4 compilation in our framework, we calculate all valid results of arithmetic expressions and store them into Read-Only Memorys (ROMs). These expressions include header size calculation and shift amount taps for static barrel-shifters. The results for a variable-sized IPv4 header instance show 13% LUT and 15% Flip-Flop (FF) usage reduction when implementing these ROM memories rather than dynamic barrel-shifters.

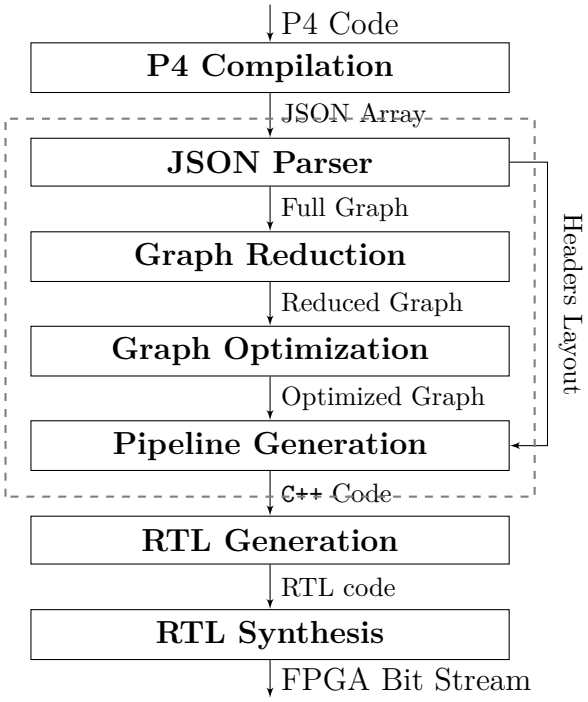


Figure 3.5 Parser pipeline generation

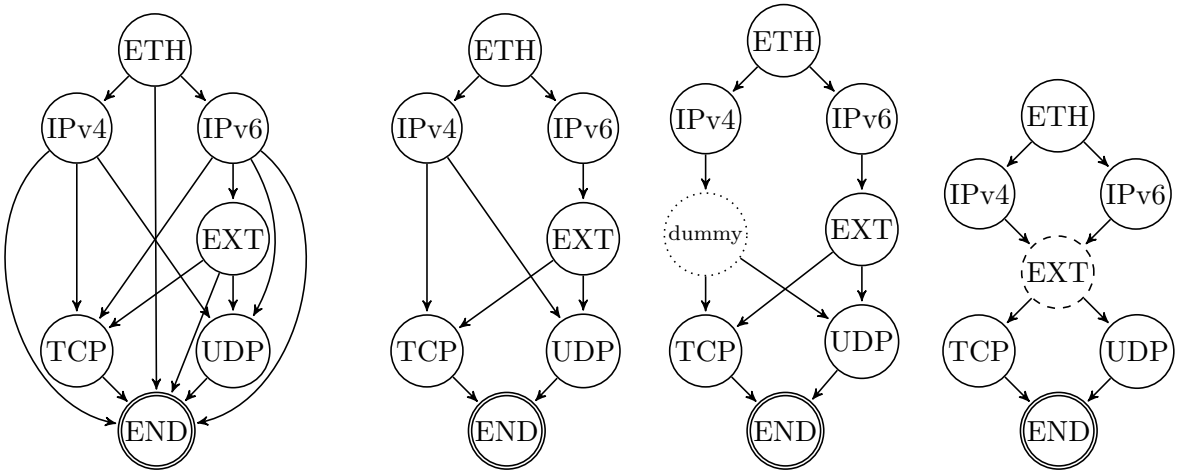
3.3.3 Pipeline Layout Generation

The procedure to generate the parser pipeline is depicted in Figure 3.5. The input P4 code is compiled using the P4C compiler [91] producing a JSON array. We have chosen to use the result of the P4 back-end compilation (p4c-bm2-ss driver) for sake of simplicity.

Our work is limited to what is enclosed by the dashed rectangle in Figure 3.5 and it is written in Python. It starts with the parsing of the JSON array file. While parsing, the script extracts the data structures necessary to initialize the multiple `C++` *Header* instances that compose the parser pipeline. The JSON parser also extracts the full parser graph. Figure 3.6a presents a full parser graph generated from a header stack comprising the following protocols: Ethernet, IPv4, IPv6, IPv6 extension header, UDP, and TCP.

For an efficient pipelined design, the graph illustrated in Figure 3.6a is not suitable. In that representation, almost all pipeline stages need bypass schemes to skip undesired state transitions, introducing combinational delays and increasing the resource usage due to the bypass multiplexers. We propose to simplify the original graph in order to have a more regular pipeline layout.

The graph simplification starts with the graph reduction phase that receives the full graph as input. This step performs a transitive reduction of the original graph in order to eliminate



(a) Original parser graph (b) Transitive reduced graph (c) Reduced graph with a spare node (d) Final transformed graph

Figure 3.6 Parser graph transformation

redundant graph edges. This phase also extracts the longest possible path of the parser graph. The result of this phase is shown in Figure 3.6b.

The graph presented in Figure 3.6c is an alternative representation for the reduced graph from Figure 3.6b. In this graph, a dummy node is introduced to offer the same reachability while balancing the graph. This dummy node only acts as a bypass element and therefore has no implementation cost, thus, it can be merged with existent nodes at the same graph level.

We propose a graph balancing algorithm in Algorithm 1 to optimize the reduced graph. It receives as parameters the transitive reduced parser graph and the longest path in the graph. As output, the algorithm returns a balanced graph tailored to our pipelined architecture. The first function call (line 2) in the algorithm executes the node level computation in relation to the root for all nodes. The first loop (lines 3 - 6) iterates over the nodes that are not in the longest path. It deletes the edges from these nodes to their children. The last loop (lines 7 - 9) iterates again over the nodes that are not part of the longest path and assigns a child to them. The chosen child is the first one belonging to the next graph level. Finally, the algorithm returns an optimized graph on line 10. An example of balanced graph is shown in Figure 3.6d.

The last step of the proposed approach illustrated in Figure 3.5 is the code generation. This phase receives as input a set of data structures representing the supported header layouts and the balanced graph. The header layouts are used to initialize both template and construction

Algorithm 1: Graph balancing algorithm

```
input : List of nodes representing a transitive reduced graph
input : Ordered list of nodes belonging to the longest path
output: Optimized balanced graph
Data: A node is a data structure that has pointers to successors/predecessors and methods to
      add/remove them. A node level represents the graph level and it is unassigned at the
      beginning.
1  Function graphBalance(tReducedGraph, longestPath)
       /* Compute the distance of all nodes to the root                    */
2      computeNodesLevel(tReducedGraph)
       /* Remove edges to successors from nodes not in the longest path    */
3      for node in tReducedGraph do
4          if node ∉ longestPath then
5              for sucNode in node.successors() do
6                  removeEdge(node, sucNode)
       /* Adding spare edges to balance the graph                          */
7      for node in tReducedGraph do
8          if node ∉ longestPath then
9              addEdge(node, longestPath[node.level + 1])
10     return tReducedGraph
```

parameters for the `C++` objects. The pipeline layout is drawn based on the balanced graph, with multiplexer insertion when required. The result of this phase is a synthesizable `C++` code.

The generated `C++` code is tailored for FPGA implementation. The next step in the processing chain is to generate RTL code for FPGA synthesis and place-and-route. Vivado HLS 2015.4 is used in this phase. Then, the generated RTL is synthesized under Vivado, which produces a bit stream file compatible with Xilinx FPGAs.

3.4 Experimental Results

To demonstrate and evaluate our proposed method, we conducted two classes of experiments, the same ones performed in [44], to simplify comparisons. These two classes are defined as follows:

- **Simple parser**: Ethernet, IPv4/IPv6 (with 2 extensions), UDP, TCP, and ICMP/ICMPv6; and
- **Full parser**: same as simple parser plus MPLS (with two nested headers) and VLAN (inner and outer).

We used Vivado HLS 2015.4 to generate synthesizable RTL code. The RTL code was afterwards synthesized under Vivado 2015.4. The target FPGA device of this work was a Xilinx Virtex-7 FPGA, part number XC7VX690TFFG1761-2.

Table 3.1 Parser results comparison

Work	Performance				Resources			Extracted Fields
	Data Bus [bits]	Frequency [MHz]	Throughput [Gb/s]	Latency [ns]	LUTs	FFs	Slice Logic (LUTs+FFs)	
Simple Parser								
[43]	256	184.1	47	N/A	14 906	2963	17 869	All fields
[43]	256	178.6	46	N/A	6865	1851	8716	TCP/IP 5-tuple
Golden [44]	512	195.3	100	15	N/A	N/A	5000	TCP/IP 5-tuple
[44]	512	195.3	100	29	N/A	N/A	12 000	TCP/IP 5-tuple
Hybrid [44] and this work	320	312.5	100	28.8	4699	7254	11 953	TCP/IP 5-tuple
This work	320	312.5	100	19.2	4270	6163	10 433	TCP/IP 5-tuple
This work	320	312.5	100	19.2	5888	10 448	16 336	All fields
Full Parser								
[43]	64	172.2	11	N/A	6946	2600	9546	All fields
[43]	64	172.2	11	N/A	3789	1425	5214	TCP/IP 5-tuple
Golden [44]	512	195.3	100	27	N/A	N/A	8000	TCP/IP 5-tuple
[44]	512	195.3	100	46.1	10 103	5537	15 640	TCP/IP 5-tuple
Hybrid [44] and this work	320	312.5	100	41.6	6450	10 308	16 758	TCP/IP 5-tuple
This work	320	312.5	100	25.6	6046	8900	14 946	TCP/IP 5-tuple
This work	320	312.5	100	25.6	7831	13 671	21 502	All fields

Table 3.1 shows a comparison against other works present in the literature [43], [44] that support fixed- and variable-sized headers. In the case of [43], because they do not provide FPGA results, we reproduced their results based on a framework provided by the authors [92]. For that, we developed a script that converts the P4 code to the data structures needed in the framework.

Analysing the data from Table 3.1, both this work and [44] outperform [43], which is expected since the framework proposed in that work for automatic parser generation was designed for ASIC implementation and not for FPGA.

We assume as a golden model, labelled as Golden [44] in Table 3.1, a hand-written VHDL implementation presented in [44], which the authors used to evaluate their method.

For the full parser, our work achieves the same throughput as [44], while not only reducing latency by 45% but also the LUT consumption by 40%. However, our architecture consumes more FFs, which is partially explained by the additional pipeline registers inferred by the Vivado HLS. Nonetheless, we can even have a lower overall slice utilization compared to [44], since in a Virtex-7 each slice has four LUTs and eight FFs, and our architecture does not double the number of used FFs.

Also, a notable resource consumption reduction is noticed when the number of extracted fields are reduced from all fields to 5-tuple, since a large amount of resources is destined to store the extracted fields, which matches with the findings reported in [43].

To compare the impact of our proposed pipelined layout, we implemented the pipeline organization proposed in [44] using the proposed header block architecture illustrated in Figure 3.1b since their source code was unavailable. This experiment is marked as "Hybrid [44] and this work" in Table 3.1. For the simple parser, our proposed architecture improves latency by more than 33%, while reducing by 16% and 10% the number of used FFs and LUTs, respectively. In the case of the full parser, the latency was reduced by 39%, while the resource consumption follows the results of the simple parser.

Moreover, this hybrid solution also outperforms the original work [44] in both latency and LUT consumption. It shows that our microarchitectural choices are more efficient in these aspects. In addition, these better results can also be related to the language chosen to describe each architecture. In [44], they generated VHDL code from a P4 description. Our design uses templated `C++` classes, which can fill the abstraction gap between the high-level packet processing program and the low-level RTL code.

When comparing to the golden model, the results obtained with our architecture are comparable in terms of latency. However, our design utilizes nearly twice the overall amount of

logic resources, following what has been reported in [44]. Such area overhead is explained by the hand-crafted low-level VHDL optimizations manually performed by the authors in [44].

As shown in Table 3.1, the present work achieves the best maximum frequency comparing to the state-of-the-art, which allows scaling to data rates higher than 100 Gb/s. Figure 3.7 presents the design scalability results for data rates ranging from 10 Gb/s up to 160 Gb/s. It is worth noting that the data rate scaling causes a non-expressive impact in terms of LUTs, corresponding to an increase of 35 $^{\text{LUTs}}/_{\text{Gbps}}$ in the case of the full 160 Gb/s parser. To achieve higher throughputs (> 160 Gb/s) in a single parser, a larger data bus (> 512 bits) is required. As a consequence, more than one minimum-sized Ethernet frame (64 bytes) could span over a single input data stream, requiring more complex hardware to detect frame boundaries. Therefore, multiple parser instances are required to support higher throughputs.

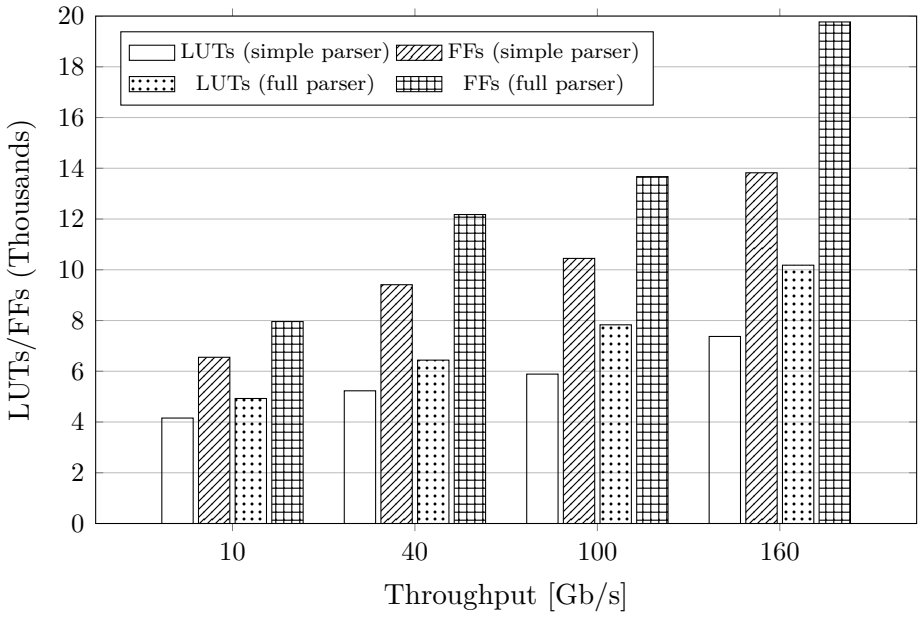


Figure 3.7 Synthesis results for multiple data rate parsers

3.5 Conclusion

FPGAs have increasingly gained importance in today's network equipment. FPGAs provide flexibility and programmability required in SDN-based networks. SDN-aware FEs need to be reconfigured to be able to parse new protocols that are constantly being deployed.

In this work, we proposed an FPGA-based architecture for high-speed packet parsing described in P4. Our architecture is completely described in `C++` to raise the development abstraction. Our methodology includes a framework for code generation, including a graph

reducing algorithm for pipeline simplification. From modern high-level languages, we borrowed the idea of metaprogramming to perform offline expressions calculation, reducing the burden of calculating them at run-time.

Our architecture performs as well as the state-of-the-art while reducing latency and LUT usage. The latency is reduced by 45% and the LUT consumption is reduced by 40%. Our proposed methodology allows a throughput scalability ranging from 10 Gb/s up to 160 Gb/s, with a moderate increase in logic resources usage.

Acknowledgments

The authors thank A. Abdelsalam, I. Benacer, M. D. Souza Dutra, T. Stimpfling, T. Luinaud, and the anonymous reviewers for their thoughtful comments. This work is supported by the CNPQ/Brazil.

CHAPTER 4 ARTICLE 2: MODULE-PER-OBJECT: A HUMAN-DRIVEN METHODOLOGY FOR C++-BASED HIGH-LEVEL SYNTHESIS DESIGN

Authors: Jeferson Santiago da Silva, François-Raymond Boyer, and J.M. Pierre Langlois.
Published in: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).

Abstract

High-Level Synthesis (HLS) brings FPGAs to audiences previously unfamiliar to hardware design. However, achieving the highest Quality of Results (QoR) with HLS is still unattainable for most programmers. This requires detailed knowledge of FPGA architecture and hardware design in order to produce FPGA-friendly codes. Moreover, these codes are normally in conflict with best coding practices, which favor code reuse, modularity, and conciseness.

To overcome these limitations, we propose Module-per-Object (MpO), a human-driven HLS design methodology intended for both hardware designers and software developers with limited FPGA expertise. MpO exploits modern C++ to raise the abstraction level while improving QoR, code readability and modularity. To guide HLS designers, we present the five characteristics of MpO classes. Each characteristic exploits the power of HLS-supported modern C++ features to build C++-based hardware modules. These characteristics lead to high-quality software descriptions and efficient hardware generation. We also present a use case of MpO, where we use C++ as the intermediate language for FPGA-targeted code generation from P4, a packet processing domain specific language. The MpO methodology is evaluated using three design experiments: a packet parser, a flow-based traffic manager, and a digital up-converter. Based on experiments, we show that MpO can be comparable to hand-written VHDL code while keeping a high abstraction level, human-readable coding style and modularity. Compared to traditional C-based HLS design, MpO leads to more efficient circuit generation, both in terms of performance and resource utilization. Also, the MpO approach notably improves software quality, augmenting parameterization while eliminating the incidence of code duplication.

This work was supported by the Brazilian National Council for Scientific and Technological Development - CNPq.

4.1 Introduction

High-Level Synthesis (HLS) has opened doors to an audience unfamiliar with FPGA hardware design methodology. Indeed, HLS tools can convert high-level and untimed C-based code into a synthesizable Register-Transfer Level (RTL) description, a task that once had to be manually done by Hardware (HW) designers. The RTL design flow is known to be much slower than its counterparts in Hardware (HW) [93], since it requires a detailed description of the desired micro-architecture, including synchronization schemes, pipelining, and parallelism. HLS tools, on the other hand, abstract away these micro-architecture aspects allowing a faster Design Space Exploration (DSE) through a SW development flow.

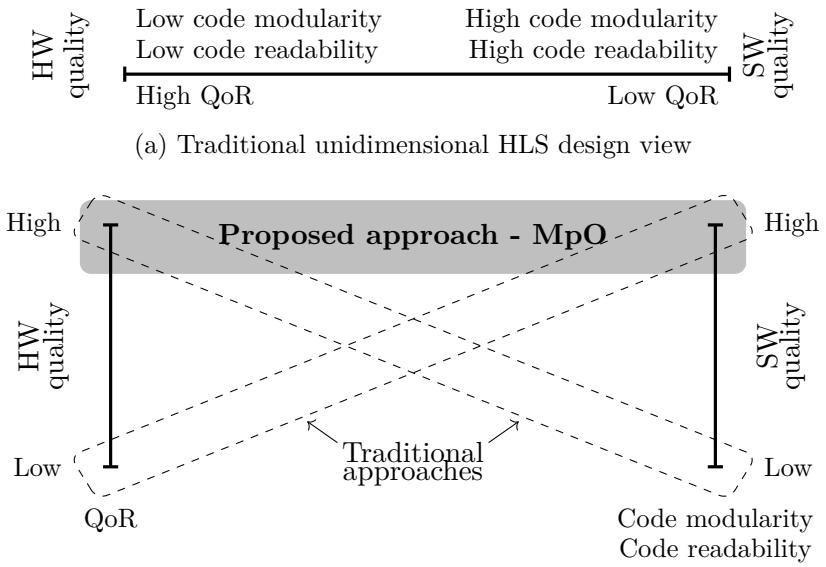
However, achieving good Quality of Results (QoR) in HLS environments is sometimes unintuitive and, in some cases, not straightforward at all. In the HW design context, the ratio between performance and design cost normally defines the QoR standard for a given circuit. In FPGA design, high performance is normally associated with throughput and latency, while design cost refers to circuit area, energy consumption, and development time.

Efforts have been made to improve QoR with HLS with source-to-source transformations and code restructuring [93], [94]. While improving QoR, such approaches lower abstraction and make code maintenance and reuse more difficult. The latter two aspects are well-known problems in HLS design and they have been subject of research as well [95], [96].

Satisfactory HW QoR with HLS-based design and good SW engineering practices are often seen as incompatible [97], [98]. Indeed, the majority of HLS users are HW developers who translate RTL codes into sometimes awkward HW-oriented C-based descriptions. They attempt to reproduce RTL-level microarchitectural expressiveness while still accelerating the FPGA design cycle through HLS design flow. Such HW-oriented C descriptions lead to incomprehensible codes difficult to reuse by other designers.

Although existing HLS approaches can sometimes deliver good code readability and modularity, and still produce good results, this is most often not the case. Normally, HLS development trades-off HW QoR and SW quality, following a sort of unidimensional view, as illustrated in Fig. 4.1a. However, a bi-dimensional HLS approach is required. Indeed, a bi-dimensional perspective highlights independence between HW QoR and SW quality. Fig. 4.1b shows the design space of this novel bi-dimensional HLS view. In fact, in the course of this work, we show that using our approach, it is possible to increase HW QoR and SW quality simultaneously by employing modern and high-quality C++ constructs, which leads to cleaner codes and reduces duplication.

In this context, we present design guidelines for C++-based HLS design targeting both HW



(a) Traditional unidimensional HLS design view

(b) Proposed bi-dimensional HLS design view

Figure 4.1 HLS design approaches

and SW designers. We present several C++ high-level constructs and, whenever possible, we show their correspondence in the generated HW. The HLS methodology we propose is called Module-per-Object (MpO). It is meant to be human-driven and used by ordinary programmers with limited HW expertise, not only by FPGA experts. We aim to close the gap between QoR and code modularity and readability. We use the results obtained by traditional HLS design as HW QoR metric. We focus on code modularity and readability as SW quality metrics. Code modularity is evaluated by the capability of reuse of a given module while code readability is related to the code expressiveness and conciseness.

As a final goal, we intend to widen FPGA usage by SW programmers by raising the FPGA development abstraction. Indeed, higher design abstractions allow programmers to use a single version of their code to run on an `x86` CPU or be synthesized for an FPGA device [99]. To do so, we propose to exploit high-level modern constructs and the Standard Template Library (STL). Such constructs are well known by SW developers to improve code readability [98]. We target QoR and code readability and modularity by extensively employing templated classes and structures that can tune the C++ objects according to design needs. In addition, we discuss the possibility of adopting templated C++ classes as an intermediate language to be used alongside a Domain Specific Language (DSL). The main contributions of this work are as follows:

- A methodology called Module-per-Object, a design pattern for HLS design that simul-

taneously achieves high modularity, readability, and QoR (§ 4.3);

- The extensive use of synthesizable templated C++ data structures and constructs to improve QoR and modularity with HLS (§ 4.3);
- A case-study on using C++ as an intermediate language for automatic code generation of a packet parser written in the P4 language targeting FPGAs (§ 4.4.1);
- Based on three specific use-cases, we have identified HLS tools deficiencies that prevent exploiting the full capabilities of high-level constructs, and we propose guidance for HLS designers and hints for future HLS tool releases (§ 4.4.2); and
- An evaluation of the benefits brought by the MpO approach on three design examples: a packet parser, a flow-based traffic manager, and a digital-up converter (§ 4.5).

4.2 Related Work

4.2.1 QoR Improvements in HLS-based Design

Liang *et al.* [100] conducted a study on how to restructure C codes in order to improve QoR with HLS for several different benchmarks. Their results showed up to 126× performance improvement over a pure software implementation, which were obtained after various rounds of code refactoring and `#pragma` insertions, which requires extensive HW expertise. In addition, when comparing to hand-crafted RTL design, their results are up to 20× worse. Also, the authors affirm that, in some cases, improving QoR conflicts with good SW engineering practices. Matai *et al.* [93] presented a methodology for code restructuring with HLS targeting FPGA devices. However, the transformed codes are unintuitive and not portable. Similar research was conducted by Homsirikamol and Gaj [101] and Liu *et al.* [102]. Zhou *et al.* have presented Rosetta [103]. Rosetta is a benchmark suite for HLS-driven FPGA design. The benchmarks have been meticulously coded and tuned for state-of-the-art HLS tools. While such practices improve performance and reduce FPGA area, in most cases, the source code is unreadable for a non-FPGA expert.

Source-to-source transformations have been explored by Winterstein *et al.* [94], [104]. The authors have proposed a framework that performs source-to-source transformations on the original C code in order to ensure synthesizability. The authors claim that the produced code is human-readable. Automated source-to-source transformations can result in descriptions that might not exactly match the original code. Gao *et al.* [105] and Cong *et al.* [106] have done similar research.

4.2.2 Raising the Abstraction Level in HLS

Cong *et al.* [99] have conducted a thorough study on HLS methods and tools. They have as well evaluated the performance of the former AutoESL's HLS tool. The authors have presented a design methodology for HLS-driven FPGA design, which includes code reusing practices through C++ templates.

Muck and Frohlich [95], [96] have exploited advanced and metaprogrammed C++ constructs to create compatible codes for both CPUs and FPGA devices. The authors present guidelines for FPGA-friendly pointer handling and static polymorphism implementation[107]. According to the authors, the resulting overhead in having reusable and modular unified C++ codes is worthwhile. The area and performance overhead are up to 30% and 50%, respectively, compared to HW-oriented C++ design. Our work leverages their ideas by employing several other C++11 constructs and by comparing the achievable results with RTL implementations.

Thomas [108] has presented a DSL library targeting recursion with C++ HLS tools described using C++11 constructs. The author has shown how compile-time metaprogramming and lambda expressions can leverage HLS-driven HW design. Indeed, in our work, we have confirmed that such constructs *can* be used by HLS designers, eventually leading to higher QoR, while raising the abstraction. Similar research was conducted by Richmond *et al.* [109]. Recently, Eran *et al.* [86] have proposed HLS-friendly design patterns for packet processing exploiting the capabilities of modern post C+11.

Zhao and Hoe [110] have assessed HLS-based flow in structural design. Their results for a network-on-chip implementation are comparable with a self-generated RTL approach. The area and performance results vary according to the network topology, ranging from $+1\%\sim+23\%$ in Lookup Tables (LUTs), $-71\%\sim-54\%$ in Flip-Flops (FFs), and $-14\%\sim+24\%$ in clock frequency. Their approach does not explore in depth the capabilities of C++ constructs supported by the HLS tool, which improves code modularity and readability.

Oezkan *et al.* [111] have also exploited templated C++ classes to build an image processing library targeting FPGA devices. The authors make extensive use of templates to generate highly parameterizable C++ classes. One of their final remarks is that the more the code is written in a "hardware design manner", the better its synthesis is. This "hardware manner" coding style lowers the abstraction, which could be alleviated by exploiting the potential of the available high-level constructs of the STL, augmenting thus code readability, avoiding code duplication, and improving code maintenance.

4.2.3 High-Level Languages (HLLs) as Intermediate Representation in FPGA Design

Other researchers have pointed to the use of DSLs for FPGA design [112]. Although increasing the development abstraction, such languages need to be converted into synthesizable RTL code, a process similar to what is done by HLS tools. Examples of such DSLs can be found in most varied domains, ranging from signal/image processing to network applications.

In the network domain, several works have used HLLs, such as P4 [3], for FPGA implementation. P4FPGA [8] is a framework for fast prototyping of network functions described in P4. P4FPGA uses BlueSpec Verilog as intermediate representation idiom, which requires a proprietary compiler to generate synthesizable RTL. The approach proposed by Khan [113] uses off-the-shelf HLS tools, however, it is difficult to evaluate the real impact of this work due to the lack of details provided. While Emu [85] is not used alongside a higher level network DSL, it could have been, since it comprises a set of standard network libraries written in C# in an object-oriented fashion that are compiled to Verilog using Kiwi [114]. These approach is similar to what Silva *et al.* [10] have done for a P4-compatible packet parser.

4.3 MpO HLS Methodology

4.3.1 Overview of the MpO Methodology

We propose the Module-per-Object (MpO) HLS methodology, in which we define the concept of *"module"* as a C++ object that logically represents a self-contained functional unit. To do so, this work exploits high-level constructs available in C++11, and that are supported by Xilinx Vivado HLS, to improve QoR while keeping a very high level of abstraction. Inspired by Cong *et al.* [99], Table 4.1 summarizes the synthesizable C++ constructs used in this work.

Table 4.1 Summary of C++ features used in this work

Constructs	**Benefits**	**Version**
Fixed-point types	Fixed-point arithmetic	C++98, vendor dependent
(Variadic) Templates	Parameterizable design	(C++11), C++98
Classes	OO paradigm, encapsulation, inheritance, polymorphism	C++98
Template metaprogramming	Compile-time calculation, performance improvement	C++98
STL	Modularity, code reuse, standardization	> C++98, in constant evolution
Data containers	Data storage and encapsulation	> C++98, in constant evolution
Algorithms	Standardization, code reuse	> C++98, in constant evolution
Iterators, range-based `for` loops	Syntax sugaring, easier container iteration	C++11
Lambda expressions	Function pointer properties	C++11
`constexpr` variables and functions	Compile-time calculation, performance improvement	C++11
`auto`, `decltype`	Automatic type inference	C++11

To increase code modularity and readability, our approach uses the concept of an MpO base class, which abstracts common functionalities between different modules. Consequently, this approach allows to reuse the same source code to describe functional modules with similar behavior. The five characteristics of an MpO class are: 1) Templates: class parameterization and code modularity (§ 4.3.3); 2) Systematic utilization of `const` and `constexpr` variables for static objects (§ 4.3.4); 3) STL constructs: zero-overhead abstraction, code reuse and modularization (§ 4.3.5); 4) Inheritance and static polymorphism (when appropriate): code reuse and modularization (§ 4.3.6); and, 5) Smart constructors: constant class member initialization (§ 4.3.7).

The main idea is to write generic code that is specialized at compile time. Generics codes, exploiting templates (1), STL constructs (3), and inheritance and static polymorphism (5), allow writing more compact and reusable code, reducing code duplication. Specialized objects also help reducing resource usage by allowing specific pieces of hardware to be precisely inferred. Indeed, `const` and `constexpr` variables (2) give hints to the compiler to perform constants propagation that can be used in conjunction with smart constructors (5) for class member initialization.

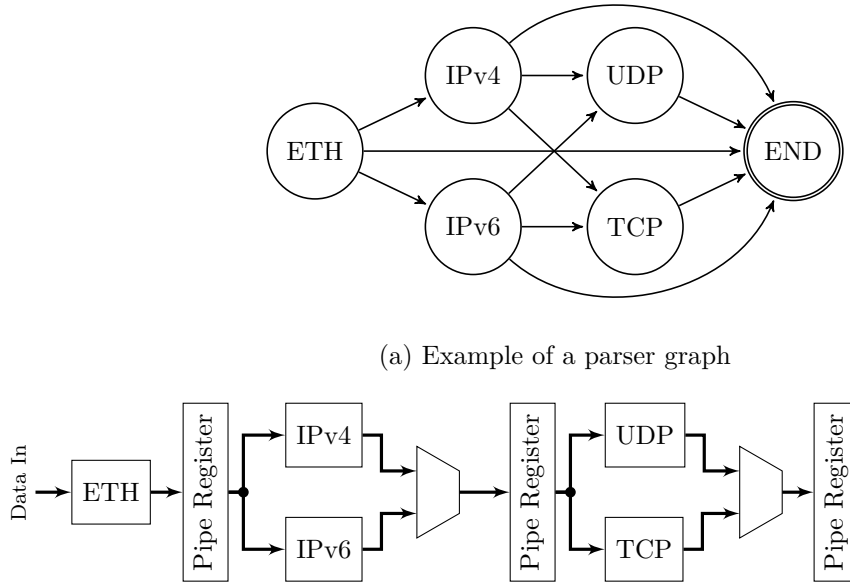
4.3.2 Illustrative Use Case: a Packet Parser

We demonstrate the viability of the proposed methodology with the design of a packet parser as a use case. A packet parser determines the set of valid protocols supported by a network device and extracts the required header fields that are to be matched in the packet processing pipeline.

A packet parser can be modeled at a high-level with a Directed Acyclic Graph (DAG), where nodes represent protocols and edges are protocol transitions [43]. A parser is implemented as an Abstract State Machine (ASM), performing state transition evaluations at each parser state. States belonging to the path connecting the first state to the last state in the ASM compose the set of supported protocols of a network equipment. A packet-processing language, such as P4 [3], can be used to describe such an ASM. Details on the implementation of a packet parser in FPGA can be found in [10]. Fig. 4.2a illustrates a parser graph for a layer-4 network device while Fig. 4.2b shows its possible hardware realization.

4.3.3 Specializing Classes with Templates

Templates are fundamental to correctly parameterize an MpO base class. Indeed, class templates allow generic code to be fine-tuned for different design instances, favoring code



(a) Example of a parser graph

(b) Pipeline organization for the example packet parser

Figure 4.2 Representation of a packet parser

reuse, reducing duplication while generating results comparable to hand-tuned codes.

Referring to Fig. 4.2a, the nodes of the parser graph share common properties and may share the same code, being a great starting point for an MpO base class. Listing 4.1 presents an example of an MpO base class that describes a node of the parser. For simplicity, only relevant code fragments are shown and cannot be compiled as is.

The class presented in Listing 4.1 is parameterized with four template parameters (line 1). The two first parameters, omitted in the listing, are integers and they are used to configure the arbitrary-sized integers. `T_HeaderLayout` is a struct type derived from a template. This type is used to declare the class member `HeaderLayout` on line 5, which represents the expected header layout to be processed. The last template parameter, `T_DHeader`, is also a type. However, this type is used to allow static polymorphism of methods of the `Header` class; therefore, it represents a type that is derived from the `Header` class itself [107].

Consequently, with the extensive use of templates, an MpO base class provides a high-degree of configurability to MpO class objects. Thus, MpO base classes contribute to more reusable and compact code. The graph described in Fig. 4.2a is an example where node is a different C++ object, sharing the same source code, described in Listing 4.1, using different template parameters.

Listing 4.1 The `Header` MpO C++ base class.

```
template<···, class T_HeaderLayout, class T_DHeader>
class Header {
   protected:
      typedef ap_uint<numbits(B2b(N_Size))> RXBitsType;
      const T_HeaderLayout HeaderLayout;
      const ShiftType stateTransShiftVal;
      const array<bool, ARR_SIZE> HeaderBusCompVal;
      RXBitsType rxBits;
   public:
      template<typename T, typename F>
      const T init_array(const F& func) const {
         typename remove_cv<T>::type arr {};
         for (auto i = 0; i < arr.size(); ++i)
            arr[i] = func(i);
         return arr;
      }
      Header (const headerIDType instance_id, const T_HeaderLayout& HLayout) :
         ···
         HeaderLayout(HLayout),
         stateTransShiftVal{shift_def(B2b(N_Size), N_BusSize,
            (HLayout.KeyLocation.first + HLayout.KeyLocation.second))
         },
         HeaderBusCompVal(
            init_array<decltype(HeaderBusCompVal)>(
               [HLayout](size_t i) {
                  return (HLayout.ArrLenLookup[i] >> numbits(N_BusSize)) > 0;
               }
            )
         ),
      { ··· } // end of constructor
      void StateTransition(const PktDataType& PktIn);
      void PipelineAdjust(···);
      void HeaderAnalysis(const PktDataType& PktIn,PHVDataType& PHV,PktDataType&
          PktOut);
};
```

4.3.4 Specializing Operands with `constexpr`

In MpO, we use `constexpr` functions and variables to set accurate bus sizes in a generic fashion, which leads to faster and more compact circuits while configurable yet synthesizable C++ descriptions are used. Also, `constexpr` functions are more comprehensible compared to the their equivalents using older C++ versions. Indeed, they allow template specialization and alleviate a task that before C++11 was only possible through template metaprogramming and partial template specialization.

The type `RXBitsType` in Listing 4.1 line 4 is such an example. The functions `numbits()`, `B2b()`, and `shift_def()` in Listing 4.1 are examples of `constexpr` functions. In [115], we present the implementation of the `numbits()` function along its verbose equivalent described in C++03. This function returns the size in bits to represent an arbitrary-sized integer.

One can benefit of compilers' ability to propagate constants by using `constexpr` functions to initialize class members in constructors. An example is the protected member `stateTransShiftVal` of the `Header` class in Listing 4.1 line 20, whose value is compile-time resolved when the class constructor is called (line 17), becoming a hardwired value in the HW implementation.

4.3.5 Exploiting STL Constructs

STL constructs raise the development abstraction and ease code readability and maintenance, characteristics favored by the MpO methodology.

Listing 4.2 shows how such constructs can be used to describe a possible implementation of a state evaluation function in a parser ASM. Its goal is to search the incoming packet stream to determine if there is a valid protocol transition for a given ASM state. To do so, the members `Key` and `KeyLocation` of the `HeaderLayout struct` are used. `Key` is an STL `array`

Listing 4.2 The `StateTransition()` method

```
1  template<···> void Header<···>::StateTransition(const PktDataType& PktIn){
2      typedef decltype(HeaderLayout.Key.front().KeyVal) KeyType;
3      const KeyType DataInMask = createMask(HeaderLayout.KeyLocation.second);
4      KeyType packetKeyVal = (PktIn.Data >> stateTransShiftVal) & DataInMask;
5      if (!NextHeaderValid && (rxBits > HeaderLayout.KeyLocation.first))
6          for (auto key : HeaderLayout.Key)
7              if (key.KeyVal == (packetKeyVal&key.KeyMask)) {
8                  NextHeader = key.NextHeader;
9                  NextHeaderValid = true;
10             }
11 }
```

container, composed of another data structure that holds information regarding the value to be matched and which is the next header transition in case of a match. `KeyLocation` is an STL `pair` type, where the first member is the key location in the incoming data stream and the second member is the key size in bits.

An `array<Type, Size> Array` container is a fixed-sized array similar to the array declaration `Type Array[Size]` in ISO C. However, since it belongs to the STL, it includes some useful built-in methods, such as `size()` and `front()`. These method calls can be resolved at compile time, and therefore they can be used to parameterize types and to set fixed loop bounds. One example of such utilization is shown in Listing 4.2 in the `KeyType` type definition on line 2. To define this new type, we use the `decltype` keyword. Again, one `constexpr` function is used, `createMask()`, to allow constant propagation on variable `DataInMask`.

Also, STL `arrays`, such as the `HeaderLayout.Key`, allow the use of iterators in a range-based `for` loop to iterate over the array. Such constructs lead to safer and more compact code since it is not required to calculate the iteration indexes or to specify loop bounds. Such an example is the `for` statement shown in Listing 4.2 line 6. In addition, automatic type resolution can be used with the `auto` keyword to determine the type of the loop iterator, simplifying the code as well.

According to our experiments, using STL constructs did not introduce overhead in terms of QoR. However, the increased code readability and modularity is noticeable, specially when dealing with data containers, such as `array`, by minimizing the need for raw pointer manipulation as required in C [116].

4.3.6 Inheritance and Static Polymorphism

The MpO methodology favors code reuse by employing inheritance whenever possible. Inheritance greatly improves code modularity and maintainability by reducing code replication.

Listing 4.3 Example of static polymorphism

```
1 template<···, class T_DHeaderFormat>class HeaderFormat {
2     ap_uint<HSIZE_BITS> getHeaderSize(const ap_uint<HSIZE_BITS>& expr_val) const
3         { return static_cast<T_DHeaderFormat*>(this)->getSpecHeaderSize(expr_val); }
4 };
5 template<···>
6 class varHeaderFormat : public HeaderFormat< ··· , const varHeaderFormat< ··· >> {
7     ap_uint<HSIZE_BITS> getSpecHeaderSize(const ap_uint<4>& ihl) const
8         { return ((0x4*ihl)*0x8); }
9 };
```

MpO exploits C++ to leverage the Don't Repeat Yourself (DRY) design guideline. Indeed, C++ offers adequate artefacts for improving inheritance, such as polymorphic methods and virtual classes.

Virtual classes are, to date, not supported by HLS vendors. However, inheritance and static polymorphism are allowed.

For the packet parser, it is of interest to keep the same method calls even if variable- and fixed-size headers are processed in a different manner. To do so, static polymorphism is a C++ mechanism that can be used with MpO.

To parse fixed-sized headers, all needed information is known at compilation time. When processing variable-sized headers, the header length must be retrieved from the header information itself. To do so, the `T_HeaderLayout` type in Listing 4.1 implements static polymorphism to retrieve both fixed- or variable-sized header length information using the same method call. The `T_HeaderLayout` definition is shown in Listing 4.3.

In Listing 4.3, the `HeaderFormat` is the base `struct`. The `struct varHeaderFormat` and `fixedHeaderFormat` (not shown in the code extract) are derived from `HeaderFormat`. Note that to allow static polymorphism, we use the Curiously Recurring Template Pattern (CRTP) technique [107] as in [95], [96], where the derived class is passed as a template parameter to the base class (lines 5-6). By doing so, the compiler is able to statically resolve pointer conversions, which results in a synthesizable description. In this example, the implementation of the `getHeaderSize()` method (line 2) is done in the derived struct (line 7).

The base class `Header` from Listing 4.1 also supports CRTP to implement static polymorphism. Two classes are derived from the `Header` class: the `FixedHeader` class and the `VariableHeader` class. Similarly to what is done with the `HeaderFormat` from Listing 4.3, the classes derived from the `Header` class have their own implementation for the method `PipelineAdjust()` (Listing 4.1 line 32). This method is responsible for keeping the output data bus aligned for the next processed header. To process fixed-sized headers, fixed bit-shift operations suffice for this alignment while barrel-shifters are required when dealing with variable-sized headers.

A naive barrel-shifter implementation in FPGAs is based on a chain of multiplexers, which results in $O(Nlog(N))$ area complexity and $O(log(N))$ delay. Contrary to ASIC design, implementing wide multiplexers can be costly in FPGAs, having normally the same complexity as an adder [99]. Thus, avoiding wide multiplexers is desired when designing efficient FPGA HW. In the parser, the number of bits to be shifted is a function of the current header size. Once we are dealing with wide data buses and the size of the processed headers is well-

constrained by a formula (Listing 4.3 line 8) in which only a few set of values are valid, then a natural choice is to use a small lookup table storing only the set of valid shift operands.

4.3.7 Smart Constructors

Class constructors can be used to initialize constant class members, which leads to more efficient circuits, as in the constant lookup table of shift values in the previous section.

An example of a smart constructor that makes use of a templated function is shown in Listing 4.1 line 10. The function is called in the constructor in line 23 to initialize the `const` class member `HeaderBusCompVal`. Note that the templated function uses a lambda expression as a callable parameter.

In C++, templated functions and objects allow callable objects to be passed as parameters to functions. Callable parameters allow functions to be reused, thus reducing code duplication. Such callable parameters can be function pointers, functors (function objects), or lambda expressions, which were introduced in C++11. Functors are objects with a single method, which once constructed can be called as a function. Modern compilers have the ability to optimize the object construction, inlining the code within the scope it is called. More interestingly, lambdas are local functions which are stored as variables, while allowing parameter passage and context capturing. In fact, lambdas are syntax sugaring for functors [108]. Indeed, for the same functionality, both the functor and the lambda implementation generate the same assembly (and LLVM) code [117].

Function pointers are unsynthesizable constructs by most HLS tools. Thus, functors and lambdas are alternative yet synthesizable ways to emulate function pointers. Besides being convenient and elegant, lambdas can contribute to more efficient HW generation by enabling constant propagation when initializing constant class members in class constructors.

4.4 Packet-parser Generation from P4

4.4.1 Top-Level Pipeline

Until now, we have described how a single HW module can be described using the proposed MpO approach. Several instances of the generic `Header` class from Listing 4.1 can be specialized to generate different HW modules. Therefore, the proposed MpO methodology from § 4.3 can be used to implement a complete packet parser.

Listing 4.4 shows a possible implementation for the packet parser illustrated in Fig. 4.2. The code in this listing is automatically generated from a P4 description [3]. Details on the

internal parser micro-architecture and the optimization steps for code generation are subject of previous work [10].

The generated HW architecture from Listing 4.4 is in accordance to the parser pipeline organization shown in Fig. 4.2b. This is ensured by the `static` declaration of the parser node objects (line 3 and 5), in a similar approach to what Zhao and Hoe have proposed [110]. The `static` keyword is used to declare stateful header objects. The pipeline is therefore inferred according to the data dependency graph. Conditional inputs in a given pipeline stage or in the output are resolved with the ternary (`?:`) C operator (line 17, 20, and 23), which generates a multiplexer in the final HW [118].

4.4.2 Adapting MpO to Current HLS tools

Vivado HLS supports C, C++, and SystemC for synthesis and simulation. The most recent C++ version supported by Vivado HLS dates from 2011. However, Vivado HLS does not fully support this C++ version, limiting the spectrum of standard high-level constructs that can be used to raise the development abstraction.

This work makes extensive use of the C++11 STL. While some constructs available in the library are expected to fail during synthesis, such as lists and maps, fixed-bounded constructs are well supported. These constructs, such as the standard `array` and `pair`, are described as classes in the STL and their operators are defined as functors in these classes. During synthesis, when facing each of these operators, Vivado HLS performs automatic function inlining for the method describing one operator, which leads to longer synthesis time and memory usage. The decision to use these STL constructs is, therefore, a trade-off between the synthesis time and the flexibility provided by these constructs.

During this work, we have struggled to correctly implement dynamic polymorphism with Vivado HLS. Static polymorphism through CRTP was the only found solution for polymorphism in this work. However, even static polymorphism is limited. Derived classes can access neither local members nor base class members. Such data accesses cause an invalid pointer reinterpretation error under synthesis. The detour for such errors is to pass the necessary operands as function parameters to the callee methods in the derived class. Accessing static members in the base class does not cause any error.

Modern compilers are able to devirtualize virtual methods of dynamically polymorphic classes at compile time and to inline the code in derived classes. Clang, for instance, is capable of devirtualizing with the compiler optimization flag set to `-O2` [119]. However, Vivado HLS does not support the compiler optimization flags. Since the optimization flag has no effect

on Vivado HLS, and its own synthesis pass is not able to infer the virtual type, dynamic polymorphism cannot be synthesized. Thus, as already concluded by other researchers [120], borrowing some front-end optimization techniques from modern compilers may be useful in the HLS world.

4.5 Experimental Results

4.5.1 Experimental Setup

In order to demonstrate the efficacy of the proposed MpO methodology, we conducted three design experiments: 1) a configurable packet parser [10]; 2) a flow-based Traffic Manager (TM) [121]; and, 3) a digital up-converter [122].

The first design experiment is a configurable packet parser briefly introduced in § 4.3.2. To enable reproducibility, the code of this experiment is open-source [123].

The second design experiment is a flow-based TM architecture proposed by Benacer *et al.* [121] in the context of SDN. The architecture is made up of a traffic policer, a packet scheduler, a systolic priority queue, and a traffic shaper. This source code is proprietary.

The third design experiment is a digital up-converter retrieved from an application note from Xilinx [122]. The up-converter design is composed of multi-stage FIR filters, a direct digital synthesizer, and a mixer. This implementation is open-source [124].

All experiments targeted a Xilinx Virtex-7 FPGA, part number XC7VX690TFFG1761-2. Vivado HLS 2015.4 was used to generate synthesizable RTL code. While we have tested more recent versions of Vivado HLS, according to our experiments, the version 2015.4 is the one that better supports modern C++ constructs. Xilinx Vivado 2015.4 was used for the synthesis and Place and Route (P&R). Code complexity is presented in terms of equivalent Equivalent Lines of Code (eLOC). The eLOC metric ignores blank and commented lines. Lines of Code (LOC), when presented, represents the actual number of lines of code. We measure code reuse with CCFinderX [125], an open-source tool based on the work by Kamiya *et al.* [126], to detect code clones.

4.5.2 Results

Configurable Packet Parser

Table 4.2 presents the results of the configurable packet parser experiment. In terms of throughput (omitted from the table) and latency, this work performs as well as the hand-

Listing 4.4 The Parser pipeline

```
1  void Parser(const PktDataType& PktIn, EthPHVDataType& eth_PHV, ··· , PktDataType&
       PktOut) {
2      array<PktDataType, 5> tmpPIn, tmpPOut;
3      static FixedHeader<···> eth (···);
4      static EthPHVDataType tmpEthPHV;
5      static VariableHeader<···> ipv4(···);
6      static Ipv4PHVDataType tmpIpv4PHV;
7      ···
8      tmpPIn[0] = PktIn;
9      eth.HeaderAnalysis(tmpPIn[0],tmpEthPHV,tmpPOut[0]);
10     eth_PHV = tmp_eth_PHV;
11     tmpPIn[1] = tmpPOut[0];
12     ipv4.HeaderAnalysis(tmpPIn[1],tmpIpv4PHV,tmpPOut[1]);
13     ipv4_PHV = tmpIpv4PHV;
14     tmpPIn[2] = tmpPOut[0];
15     ipv6.HeaderAnalysis(tmpPIn[2],tmpIpv6PHV,tmpPOut[2]);
16     ipv6_PHV = tmpIpv6PHV;
17     tmpPIn[3] = (tmpIpv4PHV.Valid)?tmpPOut[1]:tmpPOut[2];
18     udp.HeaderAnalysis(tmpPIn[3],tmpUdpPHV,tmpPOut[3]);
19     udp_PHV = tmpUdpPHV;
20     tmpPIn[4] = (tmpIpv4PHV.Valid)?tmpPOut[1]:tmpPOut[2];
21     tcp.HeaderAnalysis(tmpPIn[4], tmpTcpPHV, tmpPOut[4]);
22     tcp_PHV = tmpTcpPHV;
23     PktOut = (tmpUdpPHV.Valid)?tmpPOut[3]:tmpPOut[4];
24 }
```

Table 4.2 Packet parser results. Adapted from [10]

Work	**Freq. [MHz]**	**Lat. [ns]**	**LUTs**	**FFs**	**Slices**
VHDL [44]	195.3	27	N/A	N/A	8000
[44]	195.3	46.1	10 103	5537	15 640
[44] MpO	312.5	41.6	6450	10 308	16 758
MpO	312.5	25.6	6046	8900	14 946

crafted VHDL implementation reported in [44]. This work outperforms automatically generated VHDL code in all aspects except in the number of FFs. The LUTs reduction can be explained by the degree of parameterization that our specialized C++ classes offer. The operations are therefore fine-tuned for each header instance.

We have conducted a different experiment where we mimic Benacek's architecture using our MpO methodology. This experiment is labelled "[44] MpO" in Table 4.2. Architectural aspects aside, this hybrid implementation delivers better results than the original Benacek implementation, significantly reducing the latency (-10%) and the number of LUTs (-35%). One takeaway from this experiment is that VHDL lacks in abstraction to be used as a direct conversion language from a high-abstraction DSL, such as P4. On the other hand, C++ offers an adequate dialect to represent network semantics that can be described using P4.

Flow-based traffic manager

Table 4.3 presents the results of the TM implementation. This TM implementation can process 1024 different packet flows. The queue depth of this TM is 128. To provide a fair comparison for this experiment, we did not perform any algorithmic or architectural optimization in the original code. Also, we kept the same optimization directives of the original design.

Besides code modernization using C++11 constructs, we augmented the degree of parameterization of the TM design. The core component of this TM is a systolic implementation of a priority queue. In the original design, each systolic slice implemented a micro queue of two or three elements. The MpO implementation fully parameterizes these micro queues, not limiting to two or three. This can be seen in Table 4.3, in which we show the TM

Table 4.3 Flow-based TM results

Work	Freq. [MHz]	LUTs	FFs	Slices	eLOC
Systolic Slice Size = 3					
[121]	91.5	37 581	13 723	9833	784
[121] MpO	102.4	33 575	13 536	9182	1001
Systolic Slice Size = 4					
[121] MpO	74.5	55 625	13 891	14 669	1001
Systolic Slice Size = 8					
[121] MpO	44.9	116 666	13 585	31 884	1001
Systolic Slice Size = 16					
[121] MpO	31.0	200 450	13 930	57 876	1001

implementation results, with 4, 8, and 16 elements in each systolic slice.

The MpO version of the TM improved HW QoR. Noticeably, the circuit frequency was improved by more than 10%. The area consumption was improved as well, with a reduction of more than 10% in LUTs and 7% in occupied slices. No effects in latency, II, DSPs, and BRAMs were observed.

The MpO implementation augmented eLOC by 27%. Indeed, this was expected because we generalized a hardwired implementation of the systolic queue slice to support arbitrary systolic slices. Moreover, a significant contributor to the increased eLOC is a library that can be reused elsewhere. This library has roughly 10% of the total eLOC, in which we implemented type trait classes and generic helper functions. In both original and MpO-based implementation, CCFinderX did not find code clones.

Digital up-converter

Table 4.4 shows the results of the digital up-converter implementation. We did not perform optimizations on the original code. We only modified the code for the FIR filters. While HW QoR results consider the whole design, the SW quality analysis applies only to the filters.

As shown in Table 4.4, the MpO approach improves QoR metrics compared to the original digital up-converter implementation from Xilinx. There were improvements in the maximum frequency, latency, and area consumption, notably for FFs. The FFs reduction can be explained by the reduced latency, which means that a shorter pipeline was required in the MpO implementation. No effects on BRAM, DSP, and II were observed, thus, not reported in Table 4.4.

The MpO approach significantly improves software quality as presented in Table 4.5. The measure of eLOC in Table 4.5 shows how expressive the MpO is compared to the original design. The MpO-based code is 16% more concise than its original counterpart. Also, we evaluate code reuse by measuring code clone patterns as reported in Table 4.5. We observe that CCFinderX found 6 patterns of code clones in the original design while no clones were found in our implementation. Indeed, the MpO methodology favors code reuse and STL

Table 4.4 Digital up-converter HW QoR results

Work	**Freq. [MHz]**	**Lat. [cycles]**	**LUTs**	**FFs**	**Slices**
[122]	371.6	3394∼3395	3472	7388	1641
[122] MpO	404.0	3375∼3376	3010	5723	1568

usage, following a DRY methodology to avoid code duplication. In addition, in the original design, CCFinderX found an average of 2.33 replicated instances per clone pattern, with a maximum of 3. CCFinderX also reported an average of 83.5 LOC per clone, with a maximum of 115.

4.5.3 Analysis and Discussions

Zhao and Hoe [110] present quantitative results for the design of a network-on-chip. The authors compare their methodology to an auto-generated RTL implementation, while comparisons to hand-crafted RTL are not shown. On average, their results show an overhead of 11% and 8% for the LUTs consumption and the clock period. FF usage is reduced by 58%. Latency results are not presented. Their experiment is similar to the comparison between this work applied to the packet parser and [44]. Using our methodology, the maximum frequency is 1.6× higher, the latency is reduced by 45%, and LUTs by 40%, while increasing the number of FFs by 60%. These improvements in the LUTs consumption and the maximum frequency are due to our design's ability to specialize operations, leading to faster and more compact circuits.

Oezkan *et al.* [111] present comparative results between their HLS-based image processing library and the results of an image processing DSL that generates C++ code. In that comparison, their results outperform the auto-generated code, which is expected, since their library is directly hand-crafted in C++. Therefore, their results cannot be used as a baseline for a fair comparison against our proposed methodology.

While similar works have exploited modern C++ with HLS design [95], [96], [108], [109], no generalized methodology has been presented to date. Muck and Frohlich [95], [96] have focused on unified CPU-FPGA C++ code-base. Thomas [108] and Richmond *et al.* [109] have exploited the power of modern C++ to implement features not natively supported by HLS tools, such as recursion and high-order functions. None of these works have presented the benefits of using C++ in the generated HW, as we have shown. Also, no SW quality metrics have been presented in these works.

Table 4.5 Digital up-converter SW quality results

Work	**eLOC**	**Clones**	$\overline{\frac{\text{Instances}}{\text{Clone}}}$	$\overline{\frac{\text{LOC}}{\text{Clone}}}$
Original [122]	383	6	2.33	83.5
[122] MpO	323	0	∅	∅

4.6 Conclusion

HLS is a game changer to spread FPGA usage outside the HW world. However, achieving high QoR with HLS design still relies on detailed FPGA knowledge to generate FPGA-friendly low-level code, an uncommon skill for software developers. Such codes lower the design abstraction level making their comprehension and maintenance tedious even for experienced programmers. This HLS design approach follows a uni-dimensional design perspective, trading-off HW QoR and SW quality.

In this work, we introduced a bi-dimensional HLS design view by proposing the MpO methodology. The MpO methodology targets FPGA development with HLS exploiting standard C++11 constructs. The proposed MpO methodology builds on the concept of an MpO base class. The five presented characteristics of an MpO base class leverage HLS design, improving HW QoR, code readability and modularity while raising the abstraction development level. Through three design examples, we showed that using the MpO methodology, a C++ code can deliver results comparable to hand-crafted VHDL design. We as well showed that the code complexity can be reduced using the zero-overhead characteristic of C++.

CHAPTER 5 ARTICLE 3: VIRTUALLY INFINITE MATCH TABLES ON PROGRAMMABLE DATAPLANES

Authors: Jeferson Santiago da Silva, Thibaut Stimpfling, Thomas Luinaud, François-Raymond Boyer, and JM Pierre Langlois.
Submitted to: ACM SIGCOMM Computer Communication Review.

Abstract

The P4 language and modern programmable dataplanes have redrawn the networking landscape by allowing full data path programming in SDN environments. P4 offers an explicit imperative match-action-based programming model, which is the main processing abstraction in programmable dataplanes. However, modern programmable dataplanes lack the memory capacity to implement large match tables. Recent research has suggested to use heterogeneous programmable dataplanes (HDPs) to increase the memory capacity. Such an HDP is made of different programmable dataplane devices (PDDs). Each of these devices has its own memory capacity and processing capabilities, in a way that the most memory abundant device has the lowest performance and vice-versa. Hence, the bandwidth supported by an HDP is limited by the slowest PDD.

To address this issue, this work presents a cache hierarchy scheme for HPDs that allows to implement large match tables, while supporting a high packet throughput. We start our analysis by characterizing a recent data center trace. Then, following our observations, we derive the caching premises for match-action caching. We develop an open-source simulator to evaluate different caching schemes. Finally, our simulations suggest that a two-level cache hierarchy that employs a replacement policy combining random eviction with heuristic promotion can achieve a hit ratio that approaches the theoretical maximum, with a relatively small cache and low implementation costs.

5.1 Introduction

The Software-Defined Networking (SDN) paradigm has brought programmability to the once rigid network ecosystem. By allowing both control and data planes to evolve independently, SDN has opened new research avenues in networking, including data plane programming. Notably, the P4 language is a result of the SDN convergence [3]. P4 allows to configure how packets are processed by a programmable dataplane. Thanks to P4 and recent programmable

dataplanes, such as PISA [4], network admins can now deploy custom protocols by simply reprogramming the network switches according to their evolving needs, without the need to deploy expensive new hardware.

However, current requirements of data center networks are such that even state-of-the-art programmable ASICs switches cannot solely meet them. For instance, 5G mobile communications imply multi-million active sessions (>5 M) at terabit rates, stringently low end-to-end latency (<1 ms), and likely, P4-defined custom protocols.

We recently suggested using Heterogeneous Data Planes (HDPs) to alleviate data center network switch bottlenecks [14]. Indeed, using complementary and distinct packet forwarding devices increases the overall switch processing capabilities. However, research regarding HDPs is still in its infancy with many open questions, such as heterogeneous compilers, the issue of mismatched processing capabilities among devices, and distributed match-tables management.

In this work, we address the issue of distributed match-action table management in HDPs comprising two or more programmable dataplane devices (PDDs), as illustrated in Figure 5.1. As an example, an HDP could be made of a programmable ASIC for PDD_1, an FPGA for PDD_2, and a local CPU for PDD_3.

To that end, we borrow from the cache hierarchy concept of computer systems. In our proposed caching system, a first-level cache is a high-performance but memory-limited PDD. At every cache level, the performance metric P_k (throughput in our case) is decreased, such that $P_k > P_{k+1}$. Memory capacity is augmented to M_k, with $M_k < M_{k+1}$. The performance ratio between two successive cache levels is characterized by a processing slowdown factor SF defined as $SF_k = \frac{P_{k-1}}{P_k}$.

However, match-action caching is fairly different from CPU caches. First, temporal and spatial data locality is less predictable in network systems. Second, memory is scarce and the processing flexibility is limited in network switches, thus, complex cache policies may not be suitable. Third, due to dynamic traffic changes, a cache system needs to rapidly adapt to diversified workloads. As a consequence, traditional caching schemes may not be suitable in the context of heterogeneous match table caching systems.

To properly characterize the aforementioned issues, we evaluated the feasibility of such an HDP caching system by characterizing an recent data center network traffic to understand its temporal locality. Following the traffic analysis, we conducted cache simulations to evaluate and compare realistic cache policies to be implemented in HDPs. Finally, we evaluated which cache policies are tailored for current programmable dataplanes.

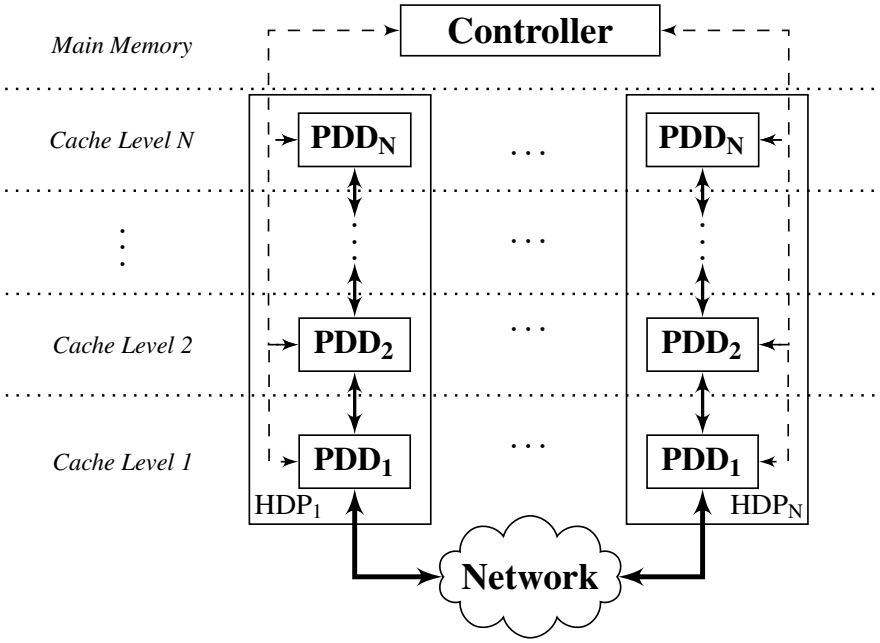


Figure 5.1 Reference caching system

To the best of our knowledge, this work is the first to consider match-action table management for HDPs. The contributions of this work are as follows:

- an open-source match-action cache policer for HDPs;
- a real-world network traffic analysis to derive match-action caching premises (§5.3);
- an evaluation of cache policies in the context of programmable dataplanes (§5.4); and
- a model to estimate the performance and a feasibility study of a match-action caching system in an HDP (§5.5).

5.2 Background

This works proposes "infinitely" extending memory for high performance heterogeneous programmable dataplanes. This is done by employing a cache hierarchy supported by network-aware cache policies. This section first recaps traditional cache policies. Then, we review the constraints of current programmable dataplanes.

5.2.1 Cache Replacement Algorithms

OPT — The optimal (OPT) cache replacement policy [127] is an oracle algorithm that relies on knowing the future traffic behavior. OPT uses this information to replace data that is furthest referenced in time. Due to that, OPT is not implementable and it is commonly used

to evaluate cache performance since it sets an upper performance bound for caching systems.

Random — Random or pseudo-random cache replacement policies use a stochastic random function to select a victim for eviction. The pseudo-random cache policy has been widely adopted in ARM-based processors. In random policies, no history on cache misses/hits nor cache access frequency is required. The efficiency of a random policy is directly related to the quality of its random generator function.

LRU — The LRU cache replacement policy evicts the most ancient cache entry in case of a cache miss. A possible LRU implementation uses a timestamp tag to sort cache entries by recency.

LFU — Contrarily to LRU, LFU uses hit frequency rather than the access time to evict entries. LFU uses a frequency counter to sort cache entries by frequency. In case of a hit, the frequency counter is incremented. Otherwise, the cache entry with the lowest frequency is chosen for eviction.

5.2.2 Constraints of Programmable Dataplanes

Single-chip homogeneous programmable dataplanes expose a clear trade-off on performance and memory resources.

State-of-the-art PISA-based programmable ASIC switches process packets at multi-terabit rates. However, these switches have no more than a few hundreds of MB of internal memory which is shared between lookup operations and user-defined stateful processing (e.g. metering, load-balancing). Moreover, as these devices need to guarantee a very low and fixed processing latency, programmers are not allowed to express loops or recursion that cannot be unrolled throughout physical pipeline stages. Indeed, neither loops nor recursion are part of the P4 semantics.

On current programmable dataplanes, forwarding rules are installed by an external host CPU as they do not yet support table updates in the data plane. Hence, no consensus has yet been reached w.r.t the maximum match table update rate as many factors can influence it. Jin et al. have reported an update rate in the order of 10 kUpdates/s [128]. A first impacting factor is the host CPU load in which we have little control of. The match type also contributes to the update rate. In our work, we are only interested in the exact match (EM) rule caching. EM tables are typically implemented as Cuckoo hash tables [3], [52]. Cuckoo hash tables recursively move entries across hash tables to solve hash collisions when inserting a new table entry. As the table load factor increases, several entries may be relocated which effectively decreases update rate. Finally, the key and action sizes also impact the update rate.

The aforementioned constraints limit the feasibility of online cache policy algorithms. Both LRU and LFU require to store extra information to select cache victims which increase memory usage. Also, these cache policies require sorting large volumes of data, which is difficult to implement (if possible) in programmable PISA switches. Finally, the match table update rate may sacrifice the reaction time of caching algorithms.

5.3 Learning from the Traffic

Network traffic has been observed to follow a Zipf distribution, with a few network flows accounting for most of the traffic [128], [129].

In our study, we conducted experiments to determine the traffic characteristics of a recent real-world data center trace. We replayed a CAIDA network trace extracted from an IXP in a New York City data center dated from January 2019 [130]. The analysed trace is 1 minute long monitoring ∼30 M packets in a full-duplex 40 Gb/s Ethernet link connecting New York and São Paulo/Brazil. A similar analysis was done by Spang and McKeown to estimate the number of TCP/IP flows [131].

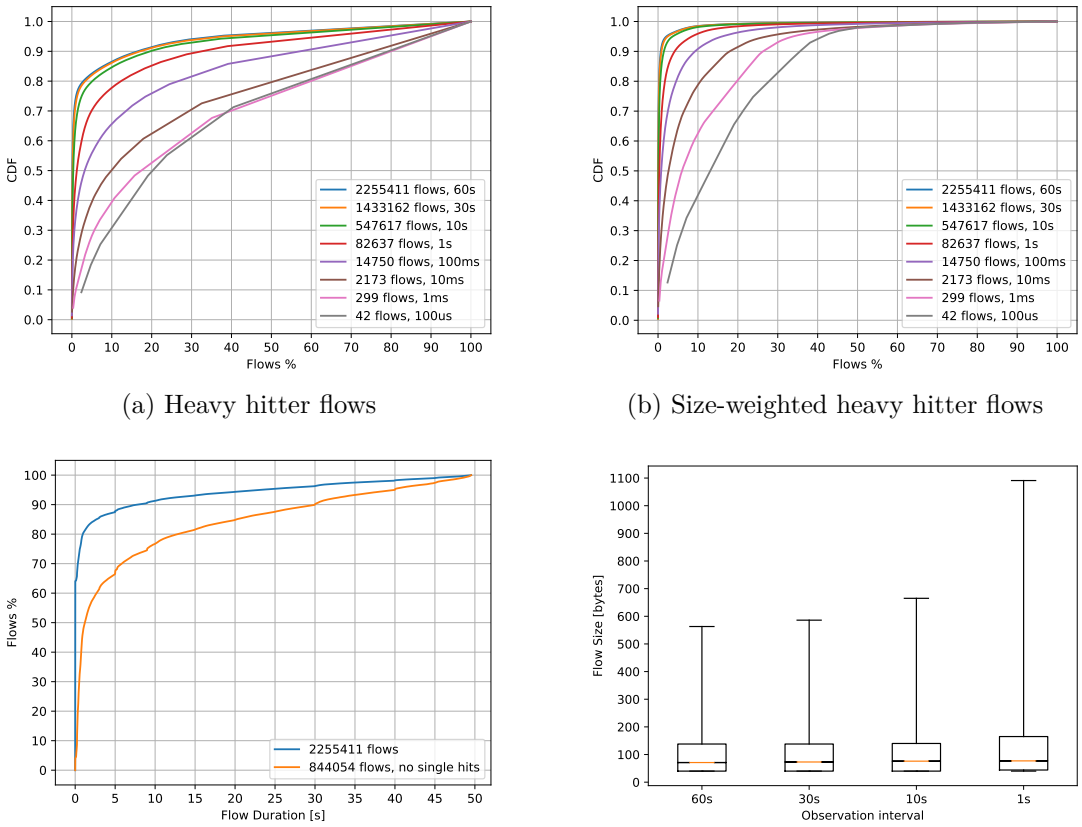
Figure 5.2 summarizes our observations. In our analysis, we defined a flow as being a unique five-tuple ⟨ SrcIP, DstIP, protocol, SrcPort, DstPort ⟩[1] connection.

Heavy hitters — As shown in Figure 5.2a and Figure 5.2b, the Zipf distribution characteristic is still present in current network traffic. Both figures present the CDF, in terms of packet hits and byte hits, for several observation intervals, ranging from 100 µs to 60 s. Although both curves are Zipf-like, the exponent that characterizes the distributions in Figure 5.2b is higher. For all observation intervals longer than 100 ms, fewer than 10% of the flows dominate more than 90% of the traffic. For shorter intervals, the Zipf dominance is still present although more skewed.

Flow duration — Figure 5.2c presents the flow duration time. The blue curve is dominated (60%) by single packet flows. Single packet flows are represented with a flow duration of zero seconds. Short-lived flows dominate the trace with ∼90% of all flows lasting less than 5 seconds. The orange curve in Figure 5.2c illustrates the duration of flows with multiple hits in the trace. Still, short-lived flows dominate the trace with ∼50% lasting less than 1 second.

Flow size — Figure 5.2d presents the average flow size for four statistically representative observation intervals. For all four measures, the first quartile, the median, and the third quartile are very similar. On average, small flows (∼100 bytes) dominate the trace with 75% of the flows being no larger than 150 bytes.

[1] We consider IPv4, IPv6, UDP, and TCP in our five-tuple definition.



(a) Heavy hitter flows

(b) Size-weighted heavy hitter flows

(c) Flow duration

(d) Average flow size

Figure 5.2 CAIDA trace summary

According to our experiments, the expected Zipf distribution characteristics of network traffic is still present in current data center traffic. Such characteristics favor flow caching in memory scarce programmable dataplanes. However, the heavy-hitter analyses show that transmitted byte-based heavy hitters are more dominant than packet-based ones. Thus, frequency-based cache policies (e.g. LFU) must consider the actual packet size in their frequency counters. In addition to that, we notice that short-lived flows dominate the trace. Therefore, the chosen cache policy algorithm needs a fast reaction time to quickly adapt to traffic changes. Besides, such temporal traffic characteristics possibly favor time-aware cache policies (e.g. LRU).

5.4 Traffic-aware Cache Policies

As traditional cache replacement positions may not be suitable in network scenarios, in this section we introduce network-aware cache policies. Based on the real traffic analysis, we first present cache eviction policies. Then, we present cache promotion policies aiming at

maximizing cache performance.

5.4.1 Cache Eviction

WLFU — *Vanilla* implementations of LFU perform poorly with real-world network traces [54]. *Vanilla* LFU considers all cache hits with equal weight, which is not realistic in network communications because larger packets result in greater network efficiency compared to small ones. Thus, Weighted LFU (WLFU) leverages LFU by considering the packet size in its frequency counters. A possible implementation of WLFU is illustrated in Algorithm 2. Note that as frequency counters are always increasing, periodic flushes (omitted in the pseudocode) are required.

Algorithm 2: WLFU policy

```
   input: Cache memory: list of ⟨entry, counter⟩ pairs
   input: Possible cache entry
   input: Packet Size
1  Procedure wlfuPolicy(cache, possible_entry, pkt_size)
2      if possible_entry in cache then                          // Cache hit
3          entry_found = findEntry(cache, possible_entry)       // Hit pointer
4          entry_found→counter += pkt_size                      // Increment size counter
5      else // Cache miss
6          victim = minFrequencyEntry(cache)                    // Victim pointer
7          *victim = ⟨possible_entry, pkt_size⟩
```

OLFU — Optmistic LFU (OLFU) is a proposition to overcome the limitations of *vanilla* LFU for flow caching. OLFU is an LFU derivation that gives a chance for a new entry to remain in cache regardless of its *actual* hit frequency. In OLFU, the cache policer behaves as the LFU in case of a cache hit. Otherwise, instead of re-initializing the frequency counter, the new cache entry takes control of the victim's counter, as shown in Algorithm 3 that omits the flushing logic.

Algorithm 3: OLFU policy

```
   input: Cache memory: list of ⟨entry, counter⟩ pairs
   input: Possible cache entry
1  Procedure olfuPolicy(cache, possible_entry)
2      if possible_entry in cache then                          // Cache hit
3          entry_found = findEntry(cache, possible_entry)       // Hit pointer
4          entry_found→counter += 1
5      else                                                     // Cache miss
6          victim = minFrequencyEntry(cache)                    // Victim pointer
7          *victim = ⟨possible_entry, victim→counter + 1⟩       // Replace reusing current counter
```

OWLFU — To optimize cache efficiency, OWLFU combines the OLFU and WLFU cache policies. In case of a hit, OWLFU behaves as WLFU. Otherwise, OWLFU modifies OLFU by incrementing the current packet size to the victim frequency counter, as in Algorithm 4.

Algorithm 4: OWLFU policy

```
   input: Cache memory: list of ⟨entry, counter⟩ pairs
   input: Possible cache entry
   input: Packet Size
1  Procedure owlfuPolicy(cache, possible_entry, pkt_size)
2      if possible_entry in cache then                                  // Cache hit
3          entry_found = findEntry(cache, possible_entry)               // Hit pointer
4          entry_found→counter += pkt_size                              // Increment size counter
5      else                                                             // Cache miss
6          victim = minFrequencyEntry(cache)                            // Victim pointer
7          *victim = ⟨possible_entry, victim→counter + pkt_size⟩        // Replace reusing current size counter
```

5.4.2 Cache Promotion

Due to traffic dynamics and programmable dataplane constraints, heuristic cache promotion policies are required for flow caching. Thus, we present two policies based on traffic observations.

WMFU — Heavy hitters are dominant in current data center traffic. As a consequence, selecting heavy hitters that generate cache misses are potential candidates for cache promotion. Thus, Weighted Most Frequently Used (WMFU) is a traffic-aware promotion policy derived from MFU. In classic MFU, frequent items are tracked for cache eviction. Hence, WMFU modifies MFU by considering the packet size in its frequency counters. Missed flows with higher counters are thus marked for cache promotion.

OWMFU — Optimistic WMFU is the OWLFU counterpart presented in §5.4.1. OWMFU optimistically speculates that frequent hitters will continue hitting thereafter. Similarly to WMFU, most frequently missed flows are selected for cache promotion.

5.5 Evaluating Cache Performance

In this section, we present the simulation results for an HDP caching system and we discuss its viability. Finally, we discuss the limitations of our approach.

5.5.1 Experimental Setup

Table 5.1 presents the simulation parameters used to evaluate a two-level caching system as in Figure 5.1. We simulated the cache performance by emulating different cache policer slowdown factor (SF). Note, that for a two-level caching scheme, SF also represents the cache policer reaction time. As performance metrics, we reported the cache hit ratio and the traffic size weighted hit ratio. The simulated traffic trace is the same as in §5.3. To enable

Table 5.1 Experimental parameters

Parameter	Value
Eviction policy	OPT, LRU, (O)(W)LFU, Random
Promotion policy	None, (O)WMFU
Cache size	64 to 8 k entries
Slowdown factor	1×, 10×, 100×

reproducibility, we open-sourced our codes[2].

5.5.2 Simulation Results

Figure 5.3 presents the results for our experiments when no promotion policy is implemented. To evaluate the efficacy of the proposed cache policies, we implemented the OPT algorithm [127] as a normalized theoretical upper-bound performance.

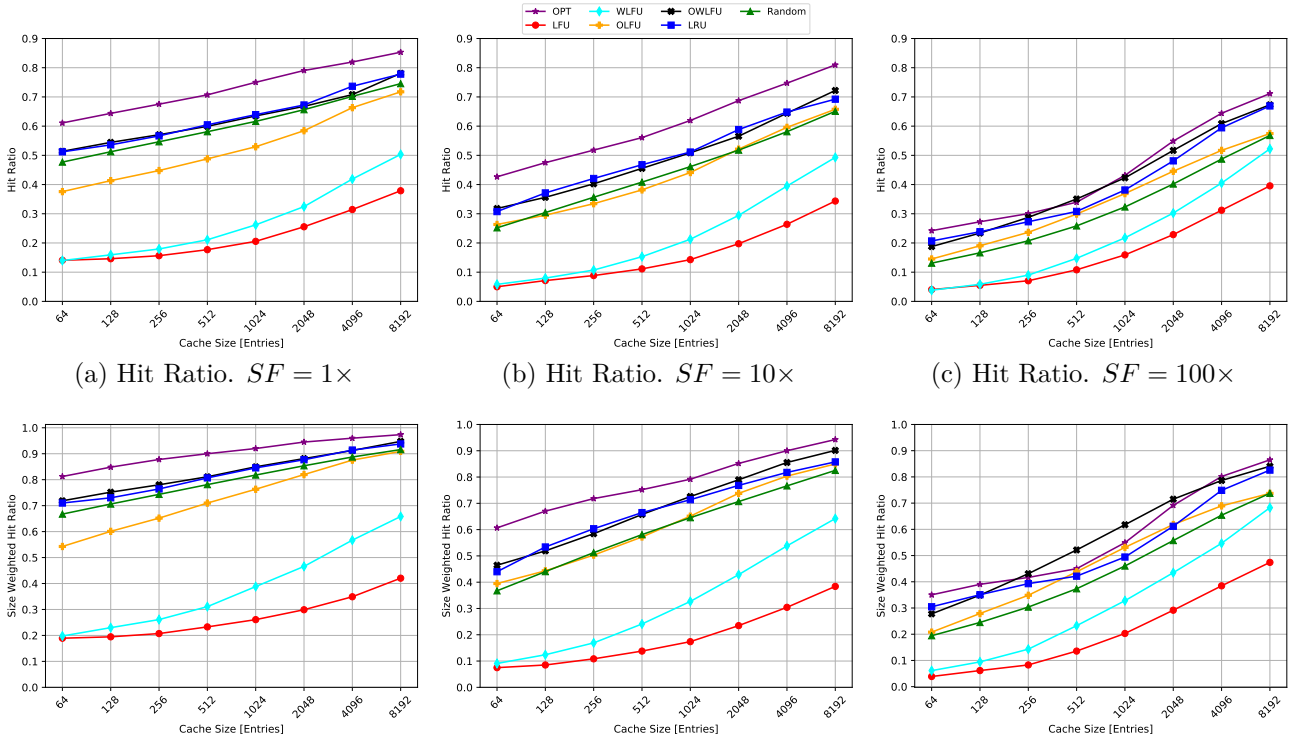
As already reported in [54], *vanilla* LFU performs poorly with real-world network traffic. Also, we confirmed the good performance of LRU-based policies with up-to-date data center traffic, which approaches to OPT as the cache size increases. The random policy achieves a relatively good cache performance considering the small overhead for implementing it. Indeed, as the traffic follows a Zipf distribution, the likelihood of randomly evicting a heavy-hitter is minimal. The random policy hit ratio lags behind the classic LRU and the OPT by no more than 10% and 15%, respectively.

All modified versions of LFU significantly improve the cache performance compared to *vanilla* LFU. WLFU achieves a sharp increase in its hit ratio as the cache size increases. OLFU has a steady performance approaching the random hit ratio. Indeed, OLFU introduces a pseudo-temporal variable to the LFU policy because it speculates that the promoted cache entry will be re-referenced thereafter. Last, OWLFU performs best in almost all scenarios since this policy combines the strengths of OLFU and WLFU.

As expected, the cache performance increases with the cache size for all policies. However, we observe that the OWLFU performance approaches to OPT for when comparing the size weighted hit ratio. Indeed, Belady's OPT algorithm [127] does not take into consideration a "data weight" when evicting; it evicts the entry which is furthest re-referenced in the future.

The *SF* scalability experiments attempt to mimic more realistic caching scenarios by considering a slower cache reaction time. For more realistic scenarios, as shown in Figure 5.3c and Figure 5.3f, the performance of the OWLFU is very close (or superior) to OPT. This is

[2] https://github.com/engjefersonsantiago/Infinite_MT.



(a) Hit Ratio. $SF = 1\times$ (b) Hit Ratio. $SF = 10\times$ (c) Hit Ratio. $SF = 100\times$

(d) Size Weighted Hit Ratio. $SF = 1\times$ (e) Size Weighted Hit Ratio. $SF = 10\times$ (f) Size Weighted Hit Ratio. $SF = 100\times$

Figure 5.3 Cache performance evaluation when no promotion policy implemented

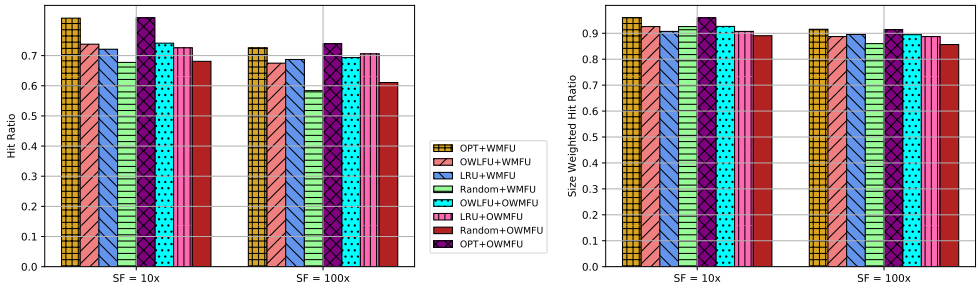
due to the fact OPT does not take into account that entries in cache are updated at a later time by the controller when $SF > 1$, and is thus not optimal as SF increases.

Figure 5.4 presents the impact of heuristic cache promotion. We evaluated two cache promotion policies: WMFU and OWMFU. These policies were combined with four cache eviction policies: OPT, random, LRU, and OWLFU. We fixed the cache size to 8 k entries and we ran simulations with $SF = 10\times$ and $SF = 100\times$ because heuristic cache promotion is only applicable when $SF > 1\times$.

Figure 5.4b shows a noticeable increase (>10%) in the size weighted hit ratio when a random-based eviction policy is implemented. Moreover, the results also suggest that there is no significant gain when LRU and LFU derivations are used.

5.5.3 Discussions

Although LRU and LFU derivations present simulation results approaching the theoretical maximum (OPT), these cache policies may be difficult to implement in high-performance programmable dataplanes. First, in terms of memory consumption, both replacement classes



(a) Hit Ratio. (b) Size Weighted Hit Ratio.

Figure 5.4 Cache performance evaluation when promotion policies are implemented

require $O(N)$ extra information to select cache victims. This is undesirable because memory is scarce in network switches and should be reserved for more profitable applications. Second, both algorithms require sorting data either by frequency or time, a costly and non-scalable operation as it normally requires $O(N\ log(N))$ comparisons with $O(log(N))$ time complexity. Moreover, such parallel sorting would require a N-port read memory, which is not available in current programmable dataplanes. An alternative would be sorting the data in software; however, the increasing data rates of current programmable dataplanes make this infeasible.

An alternative naive software approach for implementing LRU caches is using doubly-linked lists. This approach is also infeasible in current programmable dataplanes. The main reason is due to the feed-forward pipeline organization which prevents backpropagation of data from a stage S_i to S_{i-1}. Singly-linked lists could, however, be used for implementing an LRU cache. The most recent element would be placed at the head of the list while other elements would moved towards the tail. If the most recent element is already in the list, the data moving stops at its position, otherwise, it continues down to the tail. Such an implementation is possible in programmable dataplanes, however, it is still infeasible due to the limited number of pipeline stages.Multiple parallel singly-linked lists are possible but the scalability is also limited.

Not surprisingly, the performance of random-based policies applied to caches with Zipf access patterns approaches to classic (LRU) and novel (OWLFU) policies. These results follow what has been reported in the literature [132]. The simulation results show the cache hit ratio based on a random replacement policy is less than 10% lower than LRU and OWLFU. Moreover, implementing a random replacement policy in programmable dataplanes has no additional hardware cost as it may be implemented in software.

Contrary to LRU and LFU, implementing cache promotion policies in the data plane is viable. MFU and its presented derivations can be considered as a subclass of the classic

top-k hot items problem [133]. Detecting top-k hot items has already been demonstrated in P4 [134] and Domino [135] targeting current programmable dataplanes with sublinear memory consumption.

Although we are mainly interested in the data plane aspects of match table caching, the control plane component also plays an important role. Considering the constraints discussed in §5.2.2, the control plane ability in detecting and installing match table entries may expose a performance bottleneck, as reported by Miao et al. when designing a stateful load balancer [136]. Miao et al. found that the software overhead was related to the CPU load for hash calculations, not in the PCIe CPU-switch communication. However, a match table cache scheme has more stringent requirements in terms of match table update rate considering the observed flows lifetime. Thus, CPU-switch communications may still have a significant impact on performance for heterogeneous flow caching as the different HDP components are likely to have mismatched communication interfaces.

5.5.4 Limitations

Prefix shadowing — In this work, we are interested in detecting possible candidates for flow migration in the data plane. This is due to high-speed links in data center networks and the fast-changing nature of data center network traffic; therefore, a slow control plane interaction must be minimized. However, candidates detection for flow migration in the data plane can only be precisely detected for exact match rules due to the shadowing effect in ternary and LPM rules. For example, let us consider a case where a low priority LPM rule is frequently matched in a low cache level and would, therefore, be a candidate for flow migration. Using our method, this rule is moved to a higher cache level as expected. Now, a high priority rule belonging to the same prefix arriving at the HDP switch will match in the high cache level. However, a specific rule installed in a lower cache level for the same prefix will be shadowed, leading thus to a possibly wrong forwarding decision. Such cache ambiguity is a known problem and it has been studied in earlier works [56], [137].

Simulator Limits — The eviction and promotion cache policies evaluated in this work were fine-tuned based on an actual real-world data center network trace. This analysis showed that the traffic followed a Zipf distribution. Employing the caching strategy proposed in this work in other traffic scenarios has not yet been explored. As a consequence, our methods and propositions may not be viable in different traffic distributions.

The performance analysis in this paper has considered only a two-level cache. However, an N-level cache can be generalized as $(N - 1)$ independent two-level caches. Thus, the conducted two-level cache performance investigation is relevant for setting an upper bound hit ratio

analysis, even though we believe that a full caching system may expose other limitations, such as multi-level control plane interaction.

In our simulator, all table update metrics (communication, table insertion times, CPU load) are combined in a unified SF metric. Although SF attempts to mimic performance gaps, in real hardware it may not be realistic. For example, our simulator considers a perfect EM table disregarding *actual* hardware implementations in which a single table insertion may trigger many table modifications, which increases CPU load and communication overhead. Also, our simulator does not support batching for either table insertion nor when gathering eviction/promotion policy counters. Finally, the OPT algorithm implemented in the simulator may not be *the optimal* implementation in all tested scenarios as it does not consider packet sizes nor the SF impact for evicting entries.

5.6 Related Work

Flow caching has been studied since the early times in flow-based networking. Casado et al. [42] remarked in 2008 that a hardware-based SDN switch must achieve over 99% hit-ratio to avoid system bottlenecks due to software interaction.

Since then, flow caching has been explored for both hardware and software solutions. Kim et al. [54] revisited cache policies in the context of IP networks. Katta et al. [56], [57] addressed the issue of limited TCAM resources in hardware switches by proposing a hybrid hardware-software switch to exploit memory-abundant CPUs. The cache policy algorithm, however, is performed offline. From the software side, the Open vSwitch (OVS) has employed flow caching since its inception [9]. In OVS, the flow cache is split into two levels, microflow and megaflow. The microflow caches at a fine granularity for long-lasting connections while the megaflow, at coarser granularity, takes care of short-lived flows.

Grigoryan and Liu proposed a programmable FIB caching architecture [138]. They were inspired by the heavy-hitter implementation of Sivaraman et al. [134] to detect and evict infrequent TCAM entries. However, their approach requires data-plane based learning for cache replacement and it assumes that the switch can deal with variable lookup time, which compromises performance due to pipeline stalls.

Zhang et al. presented B-cache, a behavior-level cache for programmable dataplanes [139]. Similarly to Grigoryan and Liu [138], the authors exploit heavy-hitters to identify hot behavior in programmable dataplanes, which in turn could be cached. Similarly, this work is infeasible in current homogeneous high-performance switches since it breaks the streaming flow throughout the pipeline.

Kim et al. proposed extending the memory capacity in programmable switches by borrowing memory resources from RDMA-capable servers in data centers [55]. However, the achieved latency can be in the order of microseconds and the switch does not consider any cache policy mechanism.

5.7 Conclusion

P4 and the PISA architecture are bringing a new meaning to programmable networks as they promote data plane programming. Although current PISA-based programmable dataplanes offer high throughput, they lack memory resources for implementing P4 applications. Thus, recent research proposed heterogeneous dataplanes to balance the memory/performance trade-off. However, the problem heterogeneous match table caching has not yet been addressed.

In this work, we presented a cache hierarchy for HDPs. We analyzed real-world data center network traces to derive caching premises. Based on our observations, we proposed new traffic-aware cache eviction and promotion policies. These new policies, alongside classic ones, were evaluated in the context of HDPs. Simulation results show a size weighted hit ratio approaching 90% when HDP-realistic cache policies (Random+WMFU) are used. Moreover, the OWLFU eviction policy outperforms other policies and is very close to the theoretical optimal when no promotion policy is implemented. This may motivate future research on approximating OWLFU targeting data plane realization.

Acknowledgments

The authors thank the anonymous reviewers for their insightful comments. This work was supported by CNPq/Brazil, Mitacs/Canada, and Kaloom inc.

CHAPTER 6 GENERAL DISCUSSION

In recent years, we have observed a shift towards programmable dataplanes. Recent programming languages (e.g. P4) and state-of-the-art programmable switches (e.g. PISA) have strongly contributed to this shift. In this thesis, we identified open questions concerning current programmable dataplanes, including the lack of open-source compilers and optimized FPGA microarchitectures, and the management of scarce memory resources.

In this way, the main three contributions of this thesis aim at dealing with these issues. However, these three major contributions are only part of the whole investigation we have conducted during this Ph.D. research.

P4 is a recent yet fast-evolving language. By the time this Ph.D. research started in 2016, research regarding P4 was scarce. Thus, still in 2016, we started looking at P4 because we observed the language's potential impact on the SDN field. At that time, we studied missing features of the language, more specifically the lack of support for externally executed functions. Since P4 is not a Turing complete language, several applications could not be described only using P4. Thus, we proposed and integrated the required modifications to the back-end P4 compiler for supporting arbitrary externs. This work was published at NetSoft'18 [13].

This investigation on P4 externs allowed us to better understand P4 and its open-source compiler, which, eventually, led us to our first contribution. At the time, in mid-2017, we noticed the lack of open-source P4-to-FPGA compilers. Thus, we proposed a novel open-source programmable packet parser architecture targeting FPGA devices automatically generated from P4 (§3, [10]). This work, published at the FPGA'18 conference, uses high-level synthesis to automatically generate an FPGA-based packet parser hardware architecture. To that end, we developed a P4-to-C++ compiler that generates optimized HLS-based C++ classes. Furthermore, our compiler generates a fully pipelined data stream architecture after a series of graph transformations. The generated architecture is latency-optimized and achieves a 100 Gb/s throughput, which makes our work comparable to hand-crafted packet parsers while outperforming other P4-based proposals. Also, we demonstrated that our proposed parser architecture can scale up to 160 Gb/s with moderate consumption of hardware resources.

As a followup of our first contribution, we considered the problem of making FPGAs accessible to a wider audience outside the FPGA community. Thus, we presented a human-driven HLS methodology to leverage software engineering techniques in HLS design (§4, [11]). In this work, published at FCCM'19, we proposed five design premises when developing HLS-

based C++ codes. To that end, we exploited modern C++ constructs aiming at raising the abstraction level. These design premises significantly improve code modularity and readability. Besides, our methodology introduces no overhead on hardware QoR. Some QoR metrics are even improved using the proposed methodology. Our work has also identified some limitations of current HLS compilers. Following the observations, we gave hints for future HLS tool releases while presenting alternatives to deal with these current limitations. Finally, when possible, we open-sourced our codes for guiding non-expert HLS designers.

Research on compiling P4 codes to FPGAs started popping up as P4 grew in maturity. The commercial Xilinx SDNet [70] and the academic P4FPGA [73] are some examples. However, as our experience with programmable dataplanes grew, we observed that current FPGA implementations of P4 programs are performance-limited. In a joint work with other lab members, we thoroughly studied how PISA blocks can be mapped into FPGAs [15]. Also, we identified the main bottlenecks for packet processing in FPGAs and we proposed modifications to the microarchitecture of current FPGAs. One of these bottlenecks is the implementation of match tables. Thus, in another joint work, we proposed a framework to generate of P4-defined LPM tables on FPGAs that optimizes post-implementation memory efficiency [16].

From our experience, we noticed that a single processing device would not meet the current and future requirements of programmable dataplanes. Thus, in another joint study, we proposed a heterogeneous architecture for packet processing [14]. In this work, presented at P4EU'18, we presented a HDP made of an FPGA and a soft switch emulating a programmable ASIC PISA switch. The main idea of this work was to exploit the individual strengths of each device to emulate a single logical packet processing pipeline. Our preliminary results suggested that such an HDP can extend match table capacity while sustaining line-rate processing. However, important aspects, such as match table caching, were not covered in this work.

To bridge this gap, we proposed a caching scheme for heterogeneous programmable dataplanes, a work that is currently under review at SIGCOMM CCR (§5, [12]). Based on a data center trace analysis, we identified the limitations of classical cache policies and we devised a set of novel network-aware ones. We proposed three improvements for frequency-based eviction policies. These novel policies consider the packet size and speculate that frequent flows will be seen in the data plane in the near future. In addition, we considered the case for heuristic policies for cache promotion. Our simulations showed that the proposed cache schemes achieve a high hit ratio with moderate cache sizes. We also analyzed the implementation feasibility of the proposed caching system. Our findings suggested that combining a

random eviction policy with a heuristic promotion policy offers a viable performance-cost tradeoff, achieving ∼90% hit ratio for a cache storing 8 k entries.

To wrap up, in this thesis we explored the aspects of efficient programmable data plane processing. To that end, we exploited FPGAs to implement efficient data plane components. We leveraged the FPGA architecture to implement efficient packet parsers that were designed using a novel HLS methodology also proposed in this Ph.D. research. However, neither FPGAs nor programmable PISA switches are unable to solely meet the current requirements of programmable dataplanes. Thus, we finally proposed a novel caching scheme for heterogeneous programmable dataplanes, which allow us to virtually increase match table capacity without sacrificing performance.

CHAPTER 7 CONCLUSION

In this Ph.D. research, we were mainly interested in mapping network applications to FPGAs. To that end, we exploited the capabilities of the P4 language [3], a networking DSL, and PISA [4], a realistic DSA for high-speed programmable packet processing.

Thus, we proposed the optimized mapping of a PISA block, the packet parser, to FPGAs. In addition, we introduced a novel high-level methodology for designing HLS-based modules that we applied to the design of the packet parser. Finally, to overcome limitations of pure PISA switches, we presented a caching scheme for heterogeneous programmable switches.

These three works are summarized in §7.1. The limitations of each of our contributions are presented in §7.2. To conclude, §7.3 outlines some research hints for future works.

7.1 Summary of Works

Aiming at improving the performance of flexible packet parsing, we proposed an FPGA optimized packet parser architecture. We meticulously designed a feed-forward pipelined packet parser architecture that minimizes intra/interstage dependencies. In addition, our packet parser architecture is described using C++ and the hardware description is generated using off-the-shelf HLS tools. Moreover, we designed part of a P4-to-C++ backend compiler to automatically generate C++ templates from a P4 code. Our compiler also includes several rounds of graph optimizations to improve pipeline efficiency. Our results are comparable in terms of latency to hand-crafted packet parsers while outperforming auto-generated packet parser architectures in all metrics, except in flip-flop consumption. Our codes were open-sourced to permit reproducibility.

We have as well proposed a generalized methodology aiming at improving software and hardware QoR in C++-based HLS design. In our work, we leveraged a set of costless modern C++ constructs to improve code modularity and readability without compromising hardware QoR. We presented the five characteristics that HLS-aware C++ classes must have: i) class templates, ii) constants variables, iii) extensive STL usage, iv) inheritance and static polymorphism, and, v) smart class constructors. Combining these characteristics, we showed noticeable improvement in software quality while keeping or improving hardware results. We open-sourced our codes hoping to inspire other HLS designers.

As high-performance programmable switches expose an inherent memory-performance trade-off, we proposed a caching scheme targeting heterogeneous programmable switches. As tradi-

tional cache policies are either difficult to implement (LRU) or perform poorly in networking scenarios (LFU), we proposed novel cache policies based on real-world data center traces. According to our simulations, the proposed caching schemes achieve high hit ratios (>90%) with relatively small cache sizes (8 k entries). A feasibility evaluation showed that some cache policies, notably the combination of random eviction and heuristic promotion, are more interesting for networking applications w.r.t the performance-cost trade-off.

7.2 Limitations

Our packet parser architecture limits the throughput to 160 Gb/s. This is due to the internal pipeline data bus size. To achieve this throughput, the data bus width was limited to 512 bits. Achieving a higher throughput would require even wider data buses. However, a bus wider than 512 bits could encapsulate more than a single packet and our architecture is unable to treat this case. Also, it should be noted that due to paper length restrictions, the HLS pragmas used to generate the architecture have been omitted in §3. To maximize performance we have applied pipeline, loop unrolling, and function inline primitives while restricting the latency as much as possible, yet respecting the minimum pipeline depth. Finally, our packet parser does not support the `lookahead` functionality defined in the $P4_{16}$ specification.

Regarding our second contribution, although it is meant to be a generalized methodology, we only had access to three HLS compilers: the commercials Vivado HLS and Intel HLS compiler, and the academic LegUp [140]. Since C++ support for both LegUp and Intel compiler is severely limited, we only presented results for Vivado HLS. However, with Vivado HLS, we still found several limitations which limited the potential brought by our HLS methodology. The most important one relates to the difficulties of Vivado HLS in statically resolving polymorphic types at compile time. Specifically, derived polymorphic templated classes could not modify the state of base class members. This limitation requires these members to be passed as parameters to methods defined in derived classes. Finally, porting MpO to other tools, such as Cadence Stratus HLS[1] and Mentor Catapult HLS[2], would significantly enhance the evaluation of our methodology.

Managing match table cache entries in networking applications in the data plane is problematic. In our work, we only dealt with the exact table case since there is no ambiguity in cached entries. LPM/wildcard caching is therefore left for future research as they expose the match table hiding problem [56]. In addition, some of our ideas in terms of new cache policies may

[1] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[2] https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

be difficult to implement in current programmable dataplanes due to their resource scarcity. Finally, our simulation results are normalized against the OPT algorithm [127]. OPT sets an upper bound performance for blocking caches. However, for non-blocking caches, as in our proposed system, OPT is not optimal. This explains why in some scenarios, notably when $SF >> 1$, some of our proposed cache policies outperform OPT. Finding an optimal cache line replacement algorithm for non-blocking cache systems remains an open problem.

7.3 Future Research

As a possible avenue for future research, we believe that there is a salient need for an open-source P4-to-FPGA backend compiler. The reasons are twofold. First, P4FPGA [73], the only open-source P4-to-FPGA compiler, is no longer supported. Second, even if it were still maintained, the maximum throughput per port achieved with P4FPGA is limited to 10 Gb/s. Researchers interested in this subject could find inspiration in our packet parser P4-to-C++ compiler for designing a full P4-to-FPGA compilation chain.

A broader research path regards devising specialized microarchitectural components of FPGAs targeting network applications. As we have presented, current FPGAs are not adapted to some network-specific tasks. Notably, associative memories used for implementing match tables and packet schedulers are poorly mapped to FPGAs. Developing hardwired associative memories on FPGAs may be of interest to FPGA researchers and vendors.

A future industrial trend includes data plane hard virtualization. Network operators may allow tenants to implement custom data plane applications on programmable dataplanes in a multi-tenant fashion. However, such approach exposes security issues as in the early days of active networks. Research in this field includes tenant isolation, resource allocation, and the detection of potentially malicious code. Answering these research issues is a needy requirement to boost on-demand data plane in-network computing.

REFERENCES

[1] Open Networking Fundation, "Software-defined networking: The new norm for networks", *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks", *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors", *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890.

[4] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn", *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, Aug. 2013, ISSN: 0146-4833. DOI: 10.1145/2534169.2486011. [Online]. Available: http://doi.acm.org/10.1145/2534169.2486011.

[5] Barefoot Networks, *The World's Fastest & Most Programmable Networks*, 2013. [Online]. Available: https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[6] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services", *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 13–24, Jun. 2014, ISSN: 0163-5964. DOI: 10.1145/2678373.2665678. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665678.

[7] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture", in *The 49th Annual IEEE/ACM International Symposium on*

Microarchitecture, ser. MICRO-49, Taipei, Taiwan: IEEE Press, 2016, 7:1–7:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195647.

[8] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4", in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17, Santa Clara, CA, USA: ACM, 2017, pp. 122–135, ISBN: 978-1-4503-4947-5. DOI: 10.1145/3050220.3050234. [Online]. Available: http://doi.acm.org/10.1145/3050220.3050234.

[9] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch", in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, May 2015, pp. 117–130, ISBN: 978-1-931971-218. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff.

[10] J. Santiago da Silva, F.-R. Boyer, and J. M. P. Langlois, "P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs", in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, Monterey, CALIFORNIA, USA: ACM, 2018, pp. 147–152, ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174270. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174270.

[11] J. Santiago da Silva, F. Boyer, and J. M. P. Langlois, "Module-per-object: A human-driven methodology for c++-based high-level synthesis design", in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 218–226. DOI: 10.1109/FCCM.2019.00037.

[12] J. S. da Silva, T. Stimpfling, T. Luinaud, F. R Boyer, and J. P. Langlois, "Virtually infinite match tables on programmable dataplanes", in *Submitted to the ACM SIGCOMM Computer Communication Review*, 2020.

[13] J. S. da Silva, F. Boyer, L. Chiquette, and J. M. P. Langlois, "Extern objects in p4: An rohc header compression scheme case study", in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 517–522. DOI: 10.1109/NETSOFT.2018.8460108.

[14] J. Santiago da Silva, T. Stimpfling, T. Luinaud, B. Fradj, and B. Boughzala, "One for all, all for one: A heterogeneous data plane for flexible p4 processing", in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018, pp. 440–441. DOI: 10.1109/ICNP.2018.00063.

[15] T. Luinaud, T. Stimpfling, J. S. da Silva, Y. Savaria, and J. P. Langlois, "Bridging the gap: Fpgas as programmable switches", in *Accepted for publication at the 2020 IEEE 21th International Conference on High Performance Switching and Routing (HPSR)*, 2020.

[16] T. Stimpfling, J. Santiago da Silva, F.-R. Boyer, J. M. P. Langlois, and Y. Savaria, "Efficient Associative Memory for Longest Prefix Matching in Programmable Data Planes on FPGAs", *Submitted to the IEEE Transactions on Very Large Scale Integration Systems*, 2020.

[17] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks", *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014, ISSN: 0146-4833. DOI: 10.1145/2602204.2602219. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602219.

[18] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith, "The switchware active network architecture", *Netwrk. Mag. of Global Internetwkg.*, vol. 12, no. 3, pp. 29–36, May 1998, ISSN: 0890-8044. DOI: 10.1109/65.690959. [Online]. Available: https://doi.org/10.1109/65.690959.

[19] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse, "Ants: A toolkit for building and dynamically deploying network protocols", in *1998 IEEE Open Architectures and Network Programming*, 1998, pp. 117–129. DOI: 10.1109/OPNARC.1998.662048.

[20] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe, "The case for separating routing from routers", in *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, ser. FDNA '04, Portland, Oregon, USA: Association for Computing Machinery, 2004, pp. 5–12, ISBN: 158113942X. DOI: 10.1145/1016707.1016709. [Online]. Available: https://doi.org/10.1145/1016707.1016709.

[21] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform", in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05, USA: USENIX Association, 2005, pp. 15–28.

[22] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise", in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07, Kyoto, Japan: Association for Computing Machinery, 2007, pp. 1–

12, ISBN: 9781595937131. DOI: 10.1145/1282380.1282382. [Online]. Available: https://doi.org/10.1145/1282380.1282382.

[23] The Open Networking Foundation, *OpenFlow Switch Specification*, 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.pdf.

[24] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015, ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2371999.

[25] H. Song, "Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane", in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, Hong Kong, China: ACM, 2013, pp. 127–132, ISBN: 978-1-4503-2178-5. DOI: 10.1145/2491185.2491190. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491190.

[26] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[27] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al., "Onix: A distributed control platform for large-scale production networks.", in *OSDI*, vol. 10, 2010, pp. 1–6.

[28] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture", in *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, IEEE, 2014, pp. 1–6.

[29] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al., "Onos: Towards an open, distributed sdn os", in *Proceedings of the third workshop on Hot topics in software defined networking*, ACM, 2014, pp. 1–6.

[30] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic", *Technical Reprot of USENIX*, 2013.

[31] M. Yu, A. Wundsam, and M. Raju, "Nosix: A lightweight portability layer for the sdn os", *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 28–35, 2014.

[32] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch", *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[33] B. Pfaff and B. Davie, "The open vswitch database management protocol", 2013.

[34] B. Belter, D. Parniewicz, L. Ogrodowczyk, A. Binczewski, M. Stroiński, V. Fuentes, J. Matias, M. Huarte, and E. Jacob, "Hardware abstraction layer as an sdn-enabler for non-openflow network equipment.", in *EWSDN*, 2014, pp. 117–118.

[35] Intel Corporation, *Intel® IXP4XX Product Line of Network Processors*, 2010. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/ixp4xx-product-line-network-processors-spec-update.pdf.

[36] Mellanox Technologies, *NP-5™ Network Processor*, 2019. [Online]. Available: http://www.mellanox.com/related-docs/prod_npu/PB_NP-5.pdf.

[37] Broadcom Inc., *BCM56980 Series*, 2017. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56980-series.

[38] S. Choi, X. Long, M. Shahbaz, S. Booth, A. Keep, J. Marshall, and C. Kim, "Pvpp: A programmable vector packet processor", in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17, Santa Clara, CA, USA: ACM, 2017, pp. 197–198, ISBN: 978-1-4503-4947-5. DOI: 10.1145/3050220.3060609. [Online]. Available: http://doi.acm.org/10.1145/3050220.3060609.

[39] D. Intel, *Data plane development kit*, 2014. [Online]. Available: https://www.dpdk.org/.

[40] The Fast Data Project, *fd.io The Universal Dataplane*, 2019. [Online]. Available: https://fd.io/.

[41] L. Rizzo, "Netmap: A novel framework for fast packet i/o", in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12, Boston, MA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830.

[42] M. Casado, T. Koponen, D. Moon, and S. Shenker, "Rethinking packet forwarding hardware.", in *HotNets*, Citeseer, 2008, pp. 1–6.

[43] G. Gibb et al., "Design principles for packet parsers", in *Architectures for Networking and Communications Systems*, 2013, pp. 13–24. DOI: 10.1109/ANCS.2013.6665172.

[44] P. Benácek, V. Pu, and H. Kubátová, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers", in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 148–155. DOI: 10.1109/FCCM.2016.46.

[45] J. LI, B. HAN, Z. SUN, T. LI, and X. WANG, "Exploiting packet-level parallelism of packet parsing for fpga-based switches", *IEICE Transactions on Communications*, vol. advpub, 2019. DOI: 10.1587/transcom.2018EBP3333.

[46] H. Zolfaghari, D. Rossi, and J. Nurmi, "A custom processor for protocol-independent packet parsing", *Microprocessors and Microsystems*, vol. 72, p. 102 910, 2020, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2019.102910. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S014193311830560X.

[47] H. Zolfaghari, D. Rossi, and J. Nurmi, "An explicitly parallel architecture for packet parsing in software defined networks", in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–4. DOI: 10.1109/ASAP.2018.8445123.

[48] ——, "Low-latency packet parsing in software defined networks", in *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, 2018, pp. 1–6. DOI: 10.1109/NORCHIP.2018.8573461.

[49] Z. Cao, H. Zhang, J. Li, M. Wen, and C. Zhang, "A fast approach for generating efficient parsers on fpgas", *Symmetry*, vol. 11, no. 10, 2019, ISSN: 2073-8994. DOI: 10.3390/sym11101265. [Online]. Available: https://www.mdpi.com/2073-8994/11/10/1265.

[50] M. Lixin, L. Qingrang, and W. Xin, "Software-defined protocol independent parser based on fpga", in *Proceedings of the International Conference on Industrial Control Network and System Engineering Research*, ser. ICNSER2019, Shenyang, China: Association for Computing Machinery, 2019, pp. 42–46, ISBN: 9781450366274. DOI: 10.1145/3333581.3333591. [Online]. Available: https://doi.org/10.1145/3333581.3333591.

[51] R. Pagh and F. F. Rodler, "Cuckoo hashing", *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004, ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.002. [Online]. Available: https://doi.org/10.1016/j.jalgor.2003.12.002.

[52] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash", *SIAM J. Comput.*, vol. 39, no. 4, pp. 1543–1561, Dec. 2009, ISSN: 0097-5397.

[53] I. Arsovski, T. Chandler, and A. Sheikholeslami, "A ternary content-addressable memory (tcam) based on 4t static storage and including a current-race sensing scheme", *IEEE Journal of Solid-State Circuits*, vol. 38, no. 1, pp. 155–158, 2003, ISSN: 1558-173X. DOI: 10.1109/JSSC.2002.806264.

[54] C. Kim, M. Caesar, A. Gerber, and J. Rexford, "Revisiting route caching: The world should be flat", in *Passive and Active Network Measurement*, S. B. Moon, R. Teixeira, and S. Uhlig, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 3–12, ISBN: 978-3-642-00975-4.

[55] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes", in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, ser. HotNets '18, Redmond, WA, USA: ACM, 2018, pp. 1–7, ISBN: 978-1-4503-6120-0. DOI: 10.1145/3286062.3286063. [Online]. Available: http://doi.acm.org/10.1145/3286062.3286063.

[56] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks", in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, Chicago, Illinois, USA: ACM, 2014, pp. 175–180, ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620734. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620734.

[57] ——, "Cacheflow: Dependency-aware rule-caching for software-defined networks", in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16, Santa Clara, CA, USA: ACM, 2016, 6:1–6:12, ISBN: 978-1-4503-4211-7. DOI: 10.1145/2890955.2890969. [Online]. Available: http://doi.acm.org/10.1145/2890955.2890969.

[58] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan, "Towards programmable packet scheduling", in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV, Philadelphia, PA, USA: Association for Computing Machinery, 2015, ISBN: 9781450340472. DOI: 10.1145/2834050.2834106. [Online]. Available: https://doi.org/10.1145/2834050.2834106.

[59] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate", in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 44–57, ISBN: 9781450341936. DOI: 10.1145/2934872.2934899. [Online]. Available: https://doi.org/10.1145/2934872.2934899.

[60] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware", in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, Beijing, China: Association for Computing Machinery, 2019, pp. 367–379, ISBN: 9781450359566. DOI: 10.1145/3341302.3342090. [Online]. Available: https://doi.org/10.1145/3341302.3342090.

[61] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router", *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000, ISSN: 0734-2071. DOI: 10.1145/354871.354874. [Online]. Available: http://doi.acm.org/10.1145/354871.354874.

[62] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: a productive software development approach for network processors", *IEEE Micro*, vol. 24, no. 5, pp. 45–54, 2004, ISSN: 0272-1732. DOI: 10.1109/MM.2004.53.

[63] R. Duncan and P. Jungck, "packetC Language for High Performance Packet Processing", in *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, 2009, pp. 450–457. DOI: 10.1109/HPCC.2009.89.

[64] H. Song, J. Gong, H. Chen, and J. Dustzadeh, "Unified POF Programming for Diversified SDN Data Plane Devices", *ICNS 2015*, p. 106, 2015.

[65] G. Brebner and W. Jiang, "High-Speed Packet Processing using Reconfigurable Computing", *IEEE Micro*, vol. 34, no. 1, pp. 8–18, 2014, ISSN: 0272-1732. DOI: 10.1109/MM.2014.19.

[66] M. Budiu and C. Dodd, "The p416 programming language", *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, pp. 5–14, Sep. 2017, ISSN: 0163-5980. DOI: 10.1145/3139645.3139648. [Online]. Available: https://doi.org/10.1145/3139645.3139648.

[67] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "Pisces: A programmable, protocol-independent software switch", in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 525–538, ISBN: 9781450341936. DOI: 10.1145/2934872.2934886. [Online]. Available: https://doi.org/10.1145/2934872.2934886.

[68] W. Tu, F. Ruffy, and M. Budiu, "Linux network programming with p4", 2018.

[69] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches", in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, May 2015,

pp. 103–115, ISBN: 978-1-931971-218. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose.

[70] Xilinx Inc., *P4-SDNet Translator User Guide*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1252-p4-sdnet-translator.pdf, 2017.

[71] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The p4->netfpga workflow for line-rate packet processing", in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 1–9, ISBN: 9781450361378. DOI: 10.1145/3289602.3293924. [Online]. Available: https://doi.org/10.1145/3289602.3293924.

[72] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity", *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014, ISSN: 1937-4143. DOI: 10.1109/MM.2014.61.

[73] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA", in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: ACM, 2017, pp. 75–84, ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021745. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021745.

[74] V. Pus, L. Kekely, and J. Korenek, "Low-latency Modular Packet Header Parser for FPGA", in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12, Austin, Texas, USA: ACM, 2012, pp. 77–78, ISBN: 978-1-4503-1685-9. DOI: 10.1145/2396556.2396571. [Online]. Available: http://doi.acm.org/10.1145/2396556.2396571.

[75] V. Puš, L. Kekely, and J. Kořenek, "Design methodology of configurable high performance packet parser for fpga", in *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2014, pp. 189–194. DOI: 10.1109/DDECS.2014.6868788.

[76] P. Benáček, V. Puš, H. Kubátová, and T. Cejka, "P4-to-vhdl: Automatic generation of high-speed input and output network blocks", *Microprocessors and Microsystems*, vol. 56, pp. 22 –33, 2018, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2017.10.012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933117304787.

[77] J. Cabal, P. Benáček, L. Kekely, M. Kekely, V. Puš, and J. Kořenek, "Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput", in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, pp. 249–258, ISBN: 9781450356145. DOI: 10.1145/3174243.3174250. [Online]. Available: https://doi.org/10.1145/3174243.3174250.

[78] L. Kekely, M. Zádník, J. Matoušek, and J. Kořenek, "Fast lookup for dynamic packet filtering in fpga", in *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2014, pp. 219–222. DOI: 10.1109/DDECS.2014.6868793.

[79] M. Kekely, L. Kekely, and J. Korenek, "Memory aware packet matching architecture for high-speed networks", in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 1–8. DOI: 10.1109/DSD.2018.00017.

[80] M. Kekely, L. Kekely, and J. Kořenek, "General memory efficient packet matching fpga architecture for future high-speed networks", *Microprocessors and Microsystems*, vol. 73, p. 102 950, 2020, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2019.102950. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933119301334.

[81] M. Kekely and J. Korenek, "Mapping of p4 match action tables to fpga", in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–2. DOI: 10.23919/FPL.2017.8056768.

[82] S. Pontarelli, M. Bonola, and G. Bianchi, "Smashing sdn "built-in" actions: Programmable data plane packet manipulation in hardware", in *2017 IEEE Conference on Network Softwarization (NetSoft)*, 2017, pp. 1–9. DOI: 10.1109/NETSOFT.2017.8004106.

[83] D. Unnikrishnan, J. Lu, L. Gao, and R. Tessier, "Reclick - a modular dataplane design framework for fpga-based network virtualization", in *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, 2011, pp. 145–155. DOI: 10.1109/ANCS.2011.31.

[84] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware", in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 1–14, ISBN: 9781450341936. DOI: 10.1145/2934872.2934897. [Online]. Available: https://doi.org/10.1145/2934872.2934897.

[85] N. Sultana et al., "Emu: Rapid Prototyping of Networking Services", in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, 2017, pp. 459–471, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/sultana.

[86] H. Eran, L. Zeno, Z. Istvanz, and M. Silberstein, "Design Patterns for Code Reuse in HLS Packet Processing Pipelines", in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 1–10.

[87] The Open Networking Foundation, *Software-Defined Networking: The New Norm for Networks*, 2012.

[88] S. Zhou, W. Jiang, and V. K. Prasanna, "A flexible and scalable high-performance OpenFlow switch on heterogeneous SoC platforms", in *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, 2014, pp. 1–8. DOI: 10.1109/PCCC.2014.7017053.

[89] N. Gude et al., "NOX: towards an operating system for networks", *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, 2008.

[90] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA", in *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 12–23, ISBN: 978-0-7695-4521-9. DOI: 10.1109/ANCS.2011.12. [Online]. Available: http://dx.doi.org/10.1109/ANCS.2011.12.

[91] The P4 Language Consortium, *P4 Compiler*, https://github.com/p4lang/p4c, 2017.

[92] G. Gibb, *Network Packet Parser Generator*, https://github.com/grg/parser-gen, 2013.

[93] J. Matai et al., "Enabling FPGAs for the Masses", *CoRR*, vol. abs/1408.5870, 2014. arXiv: 1408.5870. [Online]. Available: http://arxiv.org/abs/1408.5870.

[94] F. Winterstein et al., "High-level synthesis of dynamic data structures: A case study using Vivado HLS", in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 362–365. DOI: 10.1109/FPT.2013.6718388.

[95] T. R. Muck and A. A. Frohlich, "Toward Unified Design of Hardware and Software Components Using C++", *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2880–2893, 2014, ISSN: 0018-9340. DOI: 10.1109/TC.2013.159.

[96] ——, ""A metaprogrammed C++ framework for hardware/software component integration and communication"", *Journal of Systems Architecture*, vol. 60, no. 10, pp. 816–827, 2014, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2014.09.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762114001143.

[97] C. Hoare, "The quality of software", *Software: Practice and Experience*, vol. 2, no. 2, pp. 103–105, 1972.

[98] B. Stroustrup and H. Sutter, *C++ Core Guidelines*, http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, 2018.

[99] J. Cong et al., "High-Level Synthesis for FPGAs: From Prototyping to Deployment", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011, ISSN: 0278-0070. DOI: 10.1109/TCAD.2011.2110592.

[100] Y. Liang et al., "High-level synthesis: productivity, performance, and software constraints", *Journal of Electrical and Computer Engineering*, vol. 2012, p. 1, 2012.

[101] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study", in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014, pp. 1–8. DOI: 10.1109/ReConFig.2014.7032504.

[102] X. Liu et al., "High Level Synthesis of Complex Applications: An H.264 Video Decoder", in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16, Monterey, California, USA: ACM, 2016, pp. 224–233, ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847274. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847274.

[103] Y. Zhou et al., "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs", in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, Monterey, CALIFORNIA, USA: ACM, 2018, pp. 269–278, ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174255. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174255.

[104] F. Winterstein et al., "Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis", in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 1–8. DOI: 10.1109/FCCM.2014.11.

[105] X. Gao et al., "Automatically Optimizing the Latency, Area, and Accuracy of C Programs for High-Level Synthesis", in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16, Monterey, California, USA: ACM, 2016, pp. 234–243, ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847282. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847282.

[106] J. Cong et al., "Bandwidth Optimization Through On-Chip Memory Restructuring for HLS", in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17, Austin, TX, USA: ACM, 2017, 43:1–43:6, ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062208. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062208.

[107] J. O. Coplien, "Curiously Recurring Template Patterns", *C++ Rep.*, vol. 7, no. 2, pp. 24–27, Feb. 1995, ISSN: 1040-6042. [Online]. Available: http://dl.acm.org/citation.cfm?id=229227.229229.

[108] D. B. Thomas, "Synthesisable recursion for C++ HLS tools", in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 91–98. DOI: 10.1109/ASAP.2016.7760777.

[109] D. Richmond, A. Althoff, and R. Kastner, "Synthesizable Higher-Order Functions for C++", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018, ISSN: 0278-0070. DOI: 10.1109/TCAD.2018.2857259.

[110] Z. Zhao and J. C. Hoe, ""Using Vivado-HLS for Structural Design: a NoC Case Study"", Carnegie Mellon University, ECE Department, Pittsburgh, PA USA, Tech. Rep., 2017. [Online]. Available: http://www.ece.cmu.edu/~coram/connect-hls/Tech_Report.pdf.

[111] M. A. Oezkan et al., "A Highly Efficient and Comprehensive Image Processing Library for C++-based High-Level Synthesis", in *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, 2017, pp. 1–10.

[112] N. Kapre and S. Bayliss, "Survey of domain-specific languages for FPGA computing", in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–12. DOI: 10.1109/FPL.2016.7577380.

[113] J. Khan and P. Athanas, "Creating Custom Network Packet Processing Pipelines on HMC-Enabled FPGAs",

[114] S. Singh and D. J. Greaves, "Kiwi: Synthesis of FPGA Circuits from Parallel Programs", in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, 2008, pp. 3–12. DOI: 10.1109/FCCM.2008.46.

[115] Jeferson Santiago da Silva, *constexpr Example*, https://godbolt.org/z/fpme-0, 2019.

[116] ——, *C++ Zero Abstraction Example*, https://godbolt.org/z/xkKXON, 2019.

[117] ——, *Lambda versus Functor Example*, https://godbolt.org/g/uQqU65, 2019.

[118] J. H. Kim et al., "FPGA-based CNN inference accelerator synthesized from multi-threaded C software", in *2017 30th IEEE International System-on-Chip Conference (SOCC)*, 2017, pp. 268–273. DOI: 10.1109/SOCC.2017.8226056.

[119] Jeferson Santiago da Silva, *Frontend Optimizations Example*, https://godbolt.org/g/dheE2Q, 2019.

[120] D. H. Noronha et al., "Rapid circuit-specific inlining tuning for FPGA high-level synthesis", in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–6. DOI: 10.1109/RECONFIG.2017.8279807.

[121] I. Benacer et al., "Design of a Low Latency 40 Gb/s Flow-Based Traffic Manager Using High-Level Synthesis", in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351332.

[122] A. Paek and J. Wu, *Designing a Digital Up-Converter using Modular C++ Classes in Vivado High Level Synthesis*, 2016.

[123] Jeferson Santiago da Silva, *Packet Parser Code*, https://github.com/engjefersonsantiago/P4HLS, 2019.

[124] ——, *Digital-up Converter Code*, https://github.com/engjefersonsantiago/MpO/tree/master/DUC, 2019.

[125] Peter Senna Tschudin, *CCFinderX Code*, https://github.com/petersenna/ccfinderx-core, 2019.

[126] T. Kamiya et al., "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002, ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1019480.

[127] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer", *IBM Systems Journal*, 1966.

[128] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching", in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[129] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging Zipf's Law for Traffic Offloading", *SIGCOMM Comput. Commun. Rev.*, 2012.

[130] The CAIDA UCSD Anonymized Internet Traces 2019 Dataset, https://data.caida.org/datasets/passive-2019/, 2019.

[131] B. Spang and N. McKeown, "On estimating the number of flows", in *Workshop on Buffer Sizing*, 2019.

[132] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy, "Performance evaluation of the random replacement policy for networks of caches", *Performance Evaluation*, 2014.

[133] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams", in *Database Theory - ICDT 2005*, T. Eiter and L. Libkin, Eds., 2005.

[134] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane", in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17, 2017.

[135] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches", in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[136] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics", in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[137] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups", *SIGCOMM Comput. Commun. Rev.*, 1997.

[138] G. Grigoryan and Y. Liu, "Pfca: A programmable fib caching architecture", in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, 2018.

[139] C. Zhang, J. Bi, Y. Zhou, K. Zhang, and Z. Ma, "B-Cache: A Behavior-Level Caching Framework for the Programmable Data Plane", in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018.

[140] A. Canis et al., "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems", in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11, Monterey, CA, USA: ACM, 2011, pp. 33–36, ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950423. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423.