Tampere University

HESAM ZOLFAGHARI

Flexible Low-Area Hardware Architectures for Packet Processing in Software-Defined Networks

Tampere University Dissertations 357

Tampere University Dissertations 357

HESAM ZOLFAGHARI

Flexible Low-Area Hardware Architectures for Packet Processing in Software-Defined Networks

ACADEMIC DISSERTATION
To be presented, with the permission of
the Faculty of Information Technology and Communication Sciences
of Tampere University,
for public discussion in Zoom
on 21 December 2020, at 12 o'clock.

ACADEMIC DISSERTATION
Tampere University, Faculty of Information Technology and Communication Sciences
Finland

Responsible supervisor and Custos	Professor Jari Nurmi Tampere University Finland	
Pre-examiners	Professor Guido Maier Politecnico di Milano Italy	Professor Seppo Virtanen University of Turku Finland
Opponents	Professor Guido Maier Politecnico di Milano Italy	Professor Peeter Ellervee Tallinn University of Technology Estonia

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

Cover design: Roihu Inc.

ISBN 978-952-03-1805-5 (print)
ISBN 978-952-03-1806-2 (pdf)
ISSN 2489-9860 (print)
ISSN 2490-0028 (pdf)
http://urn.fi/URN:ISBN:978-952-03-1806-2

PunaMusta Oy – Yliopistopaino
Vantaa 2020

ACKNOWLEDGEMENTS

This dissertation is based on the research carried out throughout the years 2017-2020 in the Electrical Engineering Unit of Tampere University (prior to 2019 by the name Department of Electronics and Communications Engineering at Tampere University of Technology). First and foremost, I express my deepest gratitude to my supervisor, professor Jari Nurmi for sharing with me his many-year experience in custom processor design as well as providing financial support and a peaceful environment for carrying out this research. I am also grateful to assistant professor Davide Rossi from University of Bologna for being the second author of all scientific papers included in this dissertation as well as for arranging a research visit to the Microelectronics Lab of University of Bologna.

I appreciate the time and effort of the respected reviewers, associate professors Guido Maier and Seppo Virtanen for providing constructive feedback on this work. Also, thanks to professor Peeter Ellervee and associate professor Guido Maier for accepting to be the opponents in my thesis defense.

This research was funded by The Pekka Ahonen Fund, Finnish Doctoral Training Network DELTA, HiPEAC, Nokia Foundation, 5G-FORCE project and TETRAMAX project. I hereby express my gratitude to all the above-mentioned funding bodies.

Finally, I wish to thank all members of my family for supporting me throughout the years and providing me with the energy required for achieving my academic goals.

Tampere, November 2020
Hesam Zolfaghari

ABSTRACT

Computer networks have changed radically in the last 10 years. Advances in computer networks and emergence of new network protocols require more flexibility and programmability in forwarding devices such as switches and routers. The main components of these devices are the control and data plane. The former instructs functionality and the latter just executes the dictated functionality. In the traditional philosophy for designing forwarding devices, the control and data plane were tightly coupled. With increase in the number and complexity of network protocols, this design principle proved to be inefficient. Software Defined Networking (SDN) breaks this tight coupling of the control and data plane. Under this network architecture, a central controller installs forwarding rules on the tables in forwarding devices. SDN-based forwarding devices only contain the data plane and the interface for communicating with the control plane. By matching the value of header fields against the installed rules, the data plane executes the corresponding actions. Research on SDN is done on the control and data planes as well as and the interface making their communication possible.

In this dissertation, the focus is on the programmable data plane. It is the enabling component for protocol-independent packet processing. The most notable hardware architecture for programmable data plane is Reconfigurable Match Tables (RMT). Despite its capabilities, there are a number of shortcomings associated with it that make it unnecessarily complex, limit its flexibility and use the memory resources inefficiently. In response to these shortcomings, a new architecture has been designed and implemented. The packet parser in this new architecture does not employ Ternary Content Addressable Memory (TCAM). As a result, it reduces the area of memories required for Match-Action packet parsing by 50%. The area saving is used for providing packet preprocessing functionality in the packet parser. The crossbar alternatives for search key generation and action input selection have been explored and the most area-efficient alternatives has been selected. Yet another packet parser is designed whose supported throughput is 10 times that of RMT parser whereas the area increase factor is less than 2. Finally, a packet processing pipeline has been designed with enhanced level of flexibility and functionality. Despite the enhancements, it has 31% less area compared to the RMT pipeline.

CONTENTS

List of Figures

List of Tables

ABBREVIATIONS

ALU	Arithmetic and Logic Unit
AOI	AND-OR-Invert
APCU	Advanced Program Control Unit
ASIC	Application-specific Integrated Circuit
BC	Branch Catalyst
BE	Best Effort
CRC	Cyclic Redundancy Check
DPDK	Data Plane Development Kit
DPI	Deep Packet Inspection
dRMT	Disaggregated Reconfigurable Match Tables
DSCP	Differentiated Services Code Point
DSL	Domain-specific Language
EPIC	Explicitly Parallel Instruction Computing
FCS	Frame Check Sequence
FD-SOI	Fully Depleted Silicon on Insulator
FE	Field Extractor
ForCES	Forwarding and Control Element Separation
FPGA	Field-Programmable Gate Array
GbE	Gigabit Ethernet
Gbps	Gigabit per second
GHz	Giga Hertz
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GRE	Generic Routing Encapsulation
HDL	Hardware Description Language
HLS	High-level Synthesis
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IHL	Internet Header Length
INT	In-band Network Telemetry

IoT	Internet of Things
IP	Internet Protocol
IPB	Incoming Packets' Buffer
IPC	Inter-packet Concurrency
ISA	Instruction Set Architecture
LoC	Lines of Code
LPM	Longest Prefix Match
MAC	Medium Access Control
MAU	Match-Action Unit
Mbps	Megabit per second
MPLS	Multiprotocol Label Switching
Mpps	Million packets per second
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NFV	Network Function Virtualization
NHRU	Next Header Resolve Unit
NIC	Network Interface Card
NOP	No Operation
NP	Network Processor
OSI	Open Systems Interconnection
PaCW	Parse Control Word
PC	Program Counter
PCIe	Peripheral Component Interconnect Express
PHV	Packet Header Vector
PiCW	Pipeline Configuration Word
PIEO	Push In Extract Out
PIFO	Push In First Out
PISA	Protocol Independent Switch Architecture
PLUG	Pipelined Lookup Grid
POF	Protocol-oblivious Forwarding
PPS	Packets Per Second
QoS	Quality of Service
RAM	Random Access Memory
RAN	Radio Access Network
RMT	Reconfigurable Match Tables
RTL	Register-transfer level

SDN	Software Defined Networking
SMT	Simultaneous Multithreading
SR	Segment Routing
SRAM	Static Random-access Memory
SRH	Segment Routing Header
Tbps	Terabit per second
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TLP	Thread-level Parallelism
TLV	Type-Length-Value
TM	Traffic Management
TPP	Tiny Packet Program
TTL	Time to Live
UADP	Unified Access Data Plane
UDP	User Datagram Protocol
VDP	Virtual Data Plane
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VLAN	Virtual Local Area Network
VLIW	Very Long Instruction Word
VM	Virtual Machine
VNF	Virtualized Network Function
WF^2Q	Worst-case Fair Weighted Fair Queuing

ORIGINAL PUBLICATIONS

P_I H. Zolfaghari, D. Rossi and J. Nurmi, "An Explicitly Parallel Architecture for Packet Parsing in Software Defined Networks," *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Milan, 2018, pp. 1-4, doi: 10.1109/ASAP.2018.8445123.

P_{II} H. Zolfaghari, D. Rossi and J. Nurmi, "Low-latency Packet Parsing in Software Defined Networks," *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, Tallinn, 2018, pp. 1-6, doi: 10.1109/NORCHIP.2018.8573461.

P_{III} H. Zolfaghari, D. Rossi and J. Nurmi, "A Custom Processor for Protocol-Independent Packet Parsing," *Microprocessors and Microsystems,* vol. 72, 2020, pp. 1-11, doi: 10.1016/j.micpro.2019.102910.

P_{IV} H. Zolfaghari, D. Rossi and J. Nurmi, "An Explicitly Parallel Architecture for Packet Processing in Software Defined Networks," *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, Helsinki, Finland, 2019, pp. 1-7, doi: 10.1109/NORCHIP.2019.8906959.

P_V H. Zolfaghari, D. Rossi and J. Nurmi, "Reducing Crossbar Costs in the Match-Action Pipeline," *2019 IEEE 20th International Conference on High Performance Switching and Routing (HPSR)*, Xi'An, China, 2019, pp. 1-6, doi: 10.1109/HPSR.2019.8808105.

P_{VI} H. Zolfaghari, D. Rossi, W. Cerroni, H. Okuhara, C. Raffaelli and J. Nurmi, "Flexible Software-Defined Packet Processing Using Low-Area Hardware," in *IEEE Access*, vol. 8, pp. 98929-98945, 2020, doi: 10.1109/ACCESS.2020.2996660.

1 INTRODUCTION

Computer networks have been subject to fundamental changes during the last decade. As a result of these changes, programmability and the role of software has become an indispensable part of computer networks. Today, computer networks operate based on the Software Defined Networking (SDN) concept. The main idea in SDN is the separation of the control plane from the data plane of forwarding devices such as switches and routers. The control and data plane are the two main logical entities within forwarding devices. They perform routing and forwarding respectively. Routing is the process of determining the routes that packets must traverse for reaching their destination. The outcome of routing is filling in the corresponding database for routing. Forwarding is the process of finding the right interface to which an incoming packet must be directed. As a result, routing is a wider problem which involves all the nodes within a network whereas forwarding is a problem to be solved within a forwarding device only. This tight coupling of the control and data plane was the dominant logical architecture of the forwarding devices. By mid 2000s, a router deployed by service providers was based on 100 million lines of source code [1]. Each new device had to add functionality on top of those of its predecessors. As a result, switches and routers were internally comprised of enormous logic to support all the network protocols that a potential customer may use. Obviously, not all of the functionalities of a commercial forwarding device could be utilized in a given deployment scenario.

Another shortcoming of the tightly coupled model was that it was counterproductive to innovation in the area of computer networks. If the research body had proposals for new network protocols, they had to start writing proposals and submit them to Internet Engineering Task Force (IETF) for standardization which was a lengthy process. Even if the idea was turned into a standard, switch and router vendors had to implement the new functionality into the devices, thus adding a few more years. A prime example is that of VxLAN. The first switch chip that supported VxLAN appeared 3 years after VxLAN was standardized [2].

As a result of these shortcomings, the idea of separating the control and data plane took off. In order for this separation to work, an interface must be made

between the control and data planes. One of the first efforts in development of such an interface was Forwarding and Control Element Separation (ForCES) [3]. A working group of the same name was formed at IETF and took the task of providing a standard interface between the control and data planes. Through this interface, the control plane installs forwarding rules in the data plane [4]. The next major step was Ethane [5]. In this architecture, flow management is handled by a centralized controller. Ethane-capable switches maintain a connection with the centralized controller that contains an overall view of the network. Ethane failed to convince commercial switch vendors for adoption.

A successful attempt was OpenFlow. First introduced in [6], it shared the main idea with Ethane. However, it was far more advanced. It provides a logical architecture for switches in which there are a number of tables containing forwarding rules. A match on a table results in the execution of actions associated with the matching entry. OpenFlow has been a commercial success and OpenFlow-based switches are available on the market. Although OpenFlow performs the task of interfacing between the control and data plane very well, it is not flexible because it is dependent on a number of protocols.

For the inception of truly SDN-based networks, further contributions were needed to support protocol-independent processing of packets. This required working on the data plane. The key to achieving this goal is support of programmability in the data plane. As a result of the clear need for the programmable data plane, efforts were made in both hardware and software.

On the programming language level, P4 was introduced in [7]. It is a target-independent language for describing packet processing behaviour in the data plane. It abstracts the underlying hardware as a series of Match and Action stages. Moving down to the Instruction Set Architecture (ISA) level, the term Protocol-oblivious Forwarding (POF) was first mentioned in [8]. It was continued in [9] and [10]. POF is a generic ISA for the processing of network packets. In a similar approach, NetASM was proposed as an intermediate representation in [11]. It is in the hardware-software interface of packet processing. On the hardware level, the Reconfigurable Match Tables (RMT) architecture appeared in 2013. It is a fully programmable protocol-independent architecture that sustains 640 Gigabits per second (Gbps) throughput. Clearly, RMT was not the first hardware architecture for packet processing, as Field-Programmable Gate Arrays (FPGAs) and Network Processors existed prior to RMT, but the innovation of RMT was maintaining programmability and performance.

Another development was the shift from middleboxes to commodity hardware for implementing network functions. Middleboxes are devices that perform non-forwarding functions. These devices were becoming costly, hard to manage and they increased the failure points within the network [12]. Network Function Virtualization (NFV) is the proposed solution for solving these issues. A network function, such as Network Address Translation (NAT) can be instantiated on a server. This class of network functions are referred to as Virtualized Network Functions (VNFs) [13]. The need for programmability manifested itself for implementing a wide range of network functions. However, packet processing on the general-purpose processor of a server has its own problems. The time between arrival of a packet at network interface card until being processed by the processor results in high latency. Moreover, even high-end processors can be overloaded with packets [14]. In order to solve these issues, SmartNICs appeared as a new class of Network Interface Cards (NICs) with enhanced functionality, performance and flexibility for offloading network functions and providing better performance [15]. SmartNICs come in a wide range of platforms such as Application-specific Integrated Circuit (ASIC), embedded processor, and FPGA for varying levels of flexibility and performance [16].

1.1 Objectives and scope

In this dissertation, the focus is on architectural aspects of Match-Action packet processing. The implementation target is ASIC. Specifically, the focus is on the problem of programmable packet parsing and packet processing. Issues such as packet scheduling and switch fabric are not within the scope of this dissertation. In the contributions made in this thesis, the key objectives are programmability, low hardware complexity and sustaining line rate throughput of 640 Gbps and above.

1.2 Research questions

There are research questions common to both packet parsing and packet processing as well as research questions specific to each of the two problems. One of the most recurring questions common to both packet parsing and packet processing is the question of which architecture is better, pipelined or run-to-completion. In the case of run-to-completion, is it better to use conditional execution or branches in order

to support the high-throughput nature of packet processing? Since increasing the frequency is not possible beyond a point, what architectural techniques are beneficial for enhancing performance?

1.2.1 Research questions specific to packet parser

Regarding the packet parser, the author investigates how programmability is achieved without expensive lookup entities such as Ternary Content Addressable Memories (TCAMs). Ways of enhancing the performance of the parser without increasing the operating frequency are also explored. With increase in line rates and complexity of network protocols, the question is, whether the parser is supposed to perform parsing only? Is there any performance benefit in processing the packets as they arrive?

1.2.2 Research questions specific to the packet processing subsystem

As for the packet processing subsystem, the first step in designing architectures with reduced area is to find out the major contributors to area. Since efficient use of lookup resources is a key goal, the author investigates and provides solution for program control mechanisms other than matching while still providing wire-speed performance. Support of advanced workloads is also a design goal. Simultaneous support of diverse set of protocols requires deep instruction memories which in turn cause noticeable increase in total area. The question is, how is it possible to support as many actions as possible while keeping the area overhead of instruction memories low.

Another research question relates to crossbars used for generating search keys, selecting operands to actions, and combining tables. Large crossbars occupy large area and make physical design challenging. Is it possible to use smaller crossbars in order to minimize the area while still maintaining programmability and performance? Is it feasible to combine as many match tables as required without large multiplexers? Minimizing recirculation is another research item addressed in this dissertation. Recirculation of packets increases packet processing latency and reduces throughput. What can be done in order to minimize the need for recirculating packets? Another question is whether the field referencing mechanisms in the latest programmable architectures are sufficient for supporting state of the art network protocols?

1.3 Research significance

Research on programmable data plane is mainly done in research and development departments of leading switch and router vendors. The amount of academic research on this topic is very small. Consequently, the outcome of research is not available to the public. The research based on which this dissertation is written provides substantial insight into the state-of-the-art packet processing hardware. Programmable architectures for protocol-independent packet processing are still in their infancy. Many SDN-related standards and contributions such as [7] and [17] describe the switch as a logical entity. The designer is free in making design choices as long as the desired functionality is achieved. The requirement analysis and architectural exploration in this thesis paves the way for further contributions and innovations for high-performance programmable packet processing hardware.

Performance in digital systems can be enhanced by increasing the operating frequency or replicating the functional units for providing parallelism. Upscaling the operating frequency is subject to physical limits. At 6.4 Terabits per Second (Tbps), there are 10 billion minimum-sized packets per second each of which requiring multiple cycles of processing. This means that even a processor with frequency of 10 Gigahertz (GHz) will not be able to keep pace with the rate of packet arrival. There are physical barriers that hinder scaling the frequency of digital systems beyond 5 GHz. Even within the range of feasible operating frequency values, lower frequencies are preferred to avoid excessive power and heat dissipation. The only solution for terabit-level packet processing is replication of functional units. The significance of low-area design is that the savings in area can be exploited for providing more on-chip match tables and/or more computational units without violating area constraints. Integrating more match tables increases the lookup capacity. Instantiating more functional units enhances functionality and/or throughput. An entire packet processing pipeline can be replicated so that the arriving packets are divided into the available pipelines.

The significance of supporting novel protocols by software means is obvious. Due to the time-consuming and costly nature of designing, implementing and verifying new hardware, it is best to have hardware that can be programmed for as many different purposes as possible.

1.4 Contributions and results

Table 1 outlines the contributions made in this dissertation.

Table 1. Contributions made in this dissertation

Contribution	Innovation	Original Publication
A low-area programmable packet parser	Use of program control instead of TCAM	P_I, P_{II}, P_{III}
Packet pre-processor	Enhancement of packet parser with packet processing functionality, processing packets on the fly	P_{IV}
Alternative crossbar architectures	Use of smaller crossbars while maintaining functionality	P_V
A pipelined parser for 6.4 Tbps parsing	Tenfold increase in throughput	P_{VI}
A flexible packet processing pipeline with advanced addressing mode and more efficient use of lookup tables	Custom action depth, advanced field referencing, unlimited table combination	P_{VI}

1.5 Author's contribution

The author of this thesis has been the first author of all papers included in this dissertation. The contribution includes coming up with the research idea, software implementation of selected network protocols, architecting the design, Register-Transfer Level (RTL) implementation, verification, and programming the implemented architecture. In addition, for P_{VI}, the ASIC synthesis has also been done by the author of this dissertation.

1.6 Thesis outline

This thesis is organized as follows. Chapter 2 provides an in-depth overview of packet processing solutions and justifies the need for custom hardware architectures. Chapter 3 contains the first contribution, which is a fully programmable packet parser. Chapter 4 provides enhancements to the packet parser for packet processing. Chapter 5 compares crossbar alternatives for the Match-Action pipeline. Chapter 6 provides an alternative packet parser for terabit-level packet parsing. Finally, chapter 7 provides a new packet processing pipeline with enhanced level of flexibility. Finally, chapter 8 concludes the work.

2 PACKET PROCESSING

Computer networks are the underlying means for communication of computer systems including servers, desktop computers, laptops, tablets, smart phones, and Internet of Things (IoT) devices. Internet is a prime example of a gigantic computer network. In computer networks, data traverses in the form of network packets. In order to simplify the design, operation, management, and troubleshooting of computer networks, networks are built of logical entities, each belonging to a layer. A reference model for this layered approach is the Open Systems Interconnection (OSI) model elaborated in [18]. Figure 1 illustrates the OSI model.

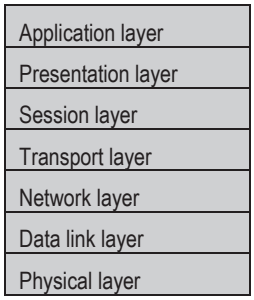

Figure 1. OSI Model

The lowest layer is the physical layer. It deals with electrical, optical, or wireless signals. As such, it has no knowledge of the contents of these signals. An instance of a system operating at the physical layer can be found in [19]. The next layer is the data link layer. It deals with accessing the transmission medium and addressing of nodes within a single network. The next layer is the network layer. This layer solves the problem of communication between independent networks which means how a packet destined to a node in another network must reach the target network. The next layer is the transport layer, which is in charge of transmission of variable-length data segments between two logical end points. The upper layers deal with more application-oriented matters. It is thanks to this layered model that when sending an email, it is not of significance whether the recipient of the email is using the Internet on a wired or wireless connection. Neither is it necessary to know what operating

system the recipient has. The message is created at the application layer and submitted to the lower layers in turn. Each layer is concerned only with its own specific issues. At the recipient's side, the flow of the corresponding packet(s) starts at the physical layer and moves upwards to the application layer.

The layered approach allows for interoperability. As far as a given implementation of a layer's functionality is fulfilled and the data is received and produced in the same format, different implementations can be swapped. Associated with each layer are a set of protocols each of which is a specific implementation of the tasks associated with the layer in question. For instance, the most dominant layer-2 protocol is Ethernet. The most dominant layer-3 protocol is the Internet Protocol (IP). Currently, IPv4 and IPv6 are being used on the Internet. At the transport layer, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are in common use. Each protocol has a header to wrap around the data it receives from the next higher layer protocol's data.

Packet processing refers to the operations performed on network packets. These operations are performed on the header(s) of network packets. It is also possible for the payload of the packet to undergo processing. For instance, in the case of packet fragmentation, the original packet is broken into multiple smaller packets each of which carries a fraction of the original payload. The payload may also be subject to encryption. Packet processing operations are executed in network switches, routers, Network Interface Cards (NIC), and in general-purpose processor as instructed by the operating system. This chapter provides an overview of packet processing operations and the packet processing solutions.

2.1 Packet processing operations

Packet processing operations can be classified based on different criteria. One such criterion is the direction of the packet. Processing on an incoming packet is called ingress processing while processing on an outgoing packet is called egress processing. Another classification is based on packet processing operations of which the most basic one is forwarding. Forwarding was discussed in the introductory section of chapter 1. Packet processing operations can be listed as follows:

-Parsing
-Integrity checking
-Header field manipulation
-Tunnelling

-State modification
-Lookup
-Classification
-Fragmentation and reassembly
-Traffic management

Each of them will be discussed in more detail.

2.1.1 Parsing

Parsing is the first step in processing of packets. In this chapter it is categorized as one of the packet processing operations. In the chapters to follow, parser is the prelude to packet processing. During parsing, the headers present in a packet are recognized and consequently, the kind of processing required for the packet is determined. According to [20], parsing can be done as the packets arrive or after the packet has been received in its entirety. Parsers operating based on these two models are referred to as streaming and non-streaming parsers respectively. Parsing must not be confused with Deep Packet Inspection (DPI) in which the payload of the packet is subject to inspection. Packet parser deals only with the headers.

2.1.2 Integrity checking

The contents of a packet might become corrupted during transmission as a result of noise or other defects. The purpose of integrity checking is to detect errors within the header. Ethernet frames contain a Frame Check Sequence (FCS) field that carries an error detection code. It is calculated using 32-bit Cyclic Redundancy Check (CRC). In IPv4, checksum of the header is calculated and then compared with the value contained in the Header Checksum field. After each header field manipulation, the checksum is recalculated and written to the Header Checksum field. Header checksum in IPv4 is calculated using one's complement addition [21].

2.1.3 Header field manipulation

Manipulation of header fields is the most obvious form of packet processing. One of the examples of header field manipulation is decrementing the value of Time-to-

Live (TTL) and Hop Limit fields within the IPv4 and IPv6 header respectively. Updating the value of checksum in IPv4 is another instance of header field manipulation.

2.1.4 Tunnelling

Tunnelling refers to the process of encapsulating a packet into another packet. It basically means adding a new header in front of the current header(s). One of the use cases for tunnelling is when a network cannot carry packets of a specific type. In this case, the packets have to be encapsulated in packets that can be transported by the network in question. For instance, if IPv6 packets need to traverse a network supporting only IPv4, IPv6 packets must be encapsulated into IPv4 as described in [22]. At the end of the so-called tunnel, the wrapping is removed.

2.1.5 State modification

Implementing the functionality of certain protocols requires maintaining state. A notable example is Transmission Control Protocol (TCP). Apart from such protocols, it is possible to associate some form of state with packets belonging to stateless protocols. For instance, a router can be configured to keep track of the payload length of IPv6 packets whose next header is UDP. With each IPv6 packet that fulfils this criterion, the router retrieves the state and adds the payload length of the packet to it. State modification may be used just for statistical or billing purposes and hence not affect the fate of packets. Alternatively, the value of the state may be used as basis for modifying header fields or even dropping packets for which a threshold value has been reached.

2.1.6 Lookup

Lookups are one of the most widely used operations in packet processing. The nature of packet processing requires that some fields be selected as the search key to be used for looking up a table and retrieving the associated data. For instance, when an Ethernet frame arrives in a switch, the destination address is used as a search key to look up into the forwarding table to find out the port to which the frame must be forwarded. When an Internet Control Message Protocol version 6 (ICMPv6) [23] is

encountered, the Type and Code fields must be used to obtain the correct instruction(s) for processing the ICMP message in question. Therefore, lookups are required both for retrieving data items and for program flow. Lookup can be regarded as a sub-operation of packet classification, which will be discussed shortly.

2.1.6.1 Exact Matching

Exact matching is a kind of matching in which an exact match for the search key in question is being searched. This kind of matching is encountered in forwarding of Ethernet frames. When presenting the Destination Media Access Control (MAC) address to the lookup table, an exact match is looked for. If a memory is used to host all possible MAC addresses, 2^{48} entries are required because MAC addresses are 48 bits wide. This amount of memory is gigantic. Instead of this naïve approach, hashing is used because a switch will deal with a far more limited range of MAC addresses rather than the whole address space. The major issue brought upon by hashing is that of collisions. It refers to the problem of distinct search keys being mapped to the same entry in the hash table. One of the most widely used solutions to this issue is cuckoo hashing [24].

2.1.6.2 Ternary Matching

Ternary matching allows a third state in addition to the default zero and one states to be stored in the match table. One use case of ternary matching is Longest Prefix Matching (LPM) in which the stored entries contain a non-ternary part called prefix followed by the ternary part. In LPM, matching entry with the longest prefix is searched. TCAMs can provide the means for ternary matching because they can store don't care bits as well. TCAMs have single-cycle latency [25] at the cost of area and power consumption. The area of a TCAM block is 6-7 times that of an Static Random Access Memory (SRAM) with equal size [26]. A TCAM consumes as much as 15 watts [27].

2.1.7 Classification

The purpose of classification is grouping packets into classes. Packets in a given class receive similar processing. The basis for classification is matching. Therefore,

classification is the outcome of an earlier match operation. The simplest form of classification is packet forwarding in which the basis for matching, and hence classification, is the destination address. All the packets having the same destination will be steered to the same port. More advanced packet classification involves using the value of multiple fields as the basis for matching. For instance, the *5-tuple* refers to source IP address, destination IP address, protocol/next header, source port and destination port fields from IPv4/IPv6 and TCP/UDP. It is used for identifying a transport-layer session. Use of 5-tuple as the basis for classification has use cases in NAT [28] and traffic management purposes [29].

2.1.8 Fragmentation and reassembly

If the size of a packet is larger than the Maximum Transmission Unit (MTU) of the network connected to the outgoing port, the packet has to be fragmented. This means that the payload of the packet must be broken into fragments and each sent as an independent packet. In the header of each fragment, there should be sufficient information to allow the reassembly of the fragments in the receiving host. In IPv4, the routers fragment a packet if its size is above the MTU of the path it must be forwarded to. In IPv6, fragmentation is done only by the source node. The minimum supported MTU as required by IPv6 is 1280 bytes [30].

Use of fragmentation and reassembly of packets is not limited to IPv4 and IPv6. Rather, it is sometimes done inside a network switch or router. Variable-length packets are fragmented into smaller fixed-size units called *cells* for better management of resources such as the internal switching fabric and packet buffers. The packet is then reassembled before being sent out through an egress port.

2.1.9 Traffic Management

Traffic Management (TM) deals with the problem of differentiating treatment of certain packets. Not all packet processing systems implement TM. In the absence of TM, all packets are treated equally and forwarded in *Best Effort* (BE) mode. Traffic management is a collective term for a wide range of operations such as marking, traffic policing, priority-based packet scheduling and traffic shaping. For each of these operations, there are various algorithms. The purpose of TM is providing Quality of Service (QoS) and/or preventing congestion. A notable instance of traffic management mechanism for IP traffic is Differentiated Services (DiffServ) [31].

2.2 Software-based packet processing solutions

In this section packet processing solutions employing software are discussed. Software switches and routers provide switching and routing functionality on general-purpose computers by means of software. The benefit of software routers is their flexibility and use of commodity hardware. In addition to general-purpose programming languages such as C, custom programming languages for packet processing have emerged to describe packet processing functionality. The operating system kernel performs packet processing services to applications sending and receiving packets. For enhanced performance, the operating system kernel can be bypassed. This concept is known as user-space packet processing. Each of the aforementioned subjects are elaborated in the subsections that follow.

2.2.1 Software Routers

Click [32] is a flexible and configurable software router. Its original implementation achieves forwarding rate of slightly over 170 Megabits per second (Mbps). At the time Click was presented, processors were running at sub 1.0 Giga Hertz (GHz) frequencies. With increase in processor speeds software routers achieved better performance. RouteBricks [33], builds a software router made of four servers connected through a mesh topology. It achieves throughput of 35 Gbps.

CuckooSwitch [34], is a software-based Ethernet switch. It has two underlying components: Intel Data Plane Development Kit (DPDK) and a scheme for ensuring consistency in spite of concurrent access of a writer and multiple readers. It achieves throughput of 92.22 Gbps.

Another development that pushed research for software routers was the rise of virtual machines (VM). Software routers steer packets towards or out of virtual machines. Open vSwitch [35] is a virtual switch that achieves 18.8 Gbps throughput when used as an Ethernet switch. With increase in the number of protocols and correspondingly increase in the complexity of software routers, the need to make them programmable became ever evident. PISCES [36] is a programmable software switch. It achieves throughput of slightly over 10 Gbps in a benchmark in which minimum-sized Ethernet frames arrive.

2.2.2 Programming Languages

As a general-purpose programming language, C can be used for implementing packet processing functionality. Linux kernel is implemented in C and contains components for processing of packets. C language does not natively support protocol-specific features such as variable-length fields and encapsulation. Apart from the limitations of C, the protocol format may be incompatible with the processing width and byte ordering of the computer that executes packet processing code written in a general-purpose programming language. Consequently, the applications need to perform the required adjustments before using the value of a header field. A Domain-specific Language (DSL) for describing the format of packets overcomes these shortcomings. PacketTypes [37] is a language specialized for packet specification. In this language, the layout of fields within a packet as well as constraints on their values can be defined as a type. It has native support for encapsulation, variable-length fields, and optional fields. The principle operation in PacketTypes is checking their membership of packets in a type. PacketTypes has been used for network monitoring, packet classification, and formal declaration of protocol formats.

P4 is a declarative domain-specific language for instructing the data plane on how the packets must be processed. P4 was first introduced in [7]. Many commercial switches today are P4-programmable. Currently, P4 has two releases, $P4_{14}$ and $P4_{16}$ that are described in detail in [38] and [39] respectively. P4 is based on an abstraction of the data plane in which the parser is followed by a Match-Action pipeline. Using P4 language, the headers can be described. The description of headers contains an ordered list of header fields and their size. The parse graph can also be described. Tables are described in terms of their size, the search key, the kind of lookup and the action that must be executed upon match. Associated with each packet is a set of metadata items called intrinsic metadata. It contains information such as the port on which the packet has arrived, the port to which it must be forwarded, whether the packet is a clone or recirculated packet, and other relevant information. P4 contains a number of primitive actions such as arithmetic and logical operations, header addition and removal, packet dropping, etc. More complex actions can be defined as a combination of primitive actions.

Domino [40] is a domain-specific imperative language with syntax similar to C. It is used to express data plane algorithms. The central concept in Domino is packet transaction which is an atomically executed code block separated from other blocks. Packet transactions allow the programmer to focus on the operations that must be performed on a packet, rather than concurrency issue brought upon by other

packets. In other words, packet transactions provide the illusion that a packet arrived at a switch is processed to completion and then processing of the next packet starts. When compiled for execution, packet transactions run at line rate in a guaranteed manner. It achieves this by imposing certain constraints. For instance, it does not allow loops nor unstructured control flow statements such as goto statements. When dealing with an array element within a transaction, only one element is allowed. What triggers the execution of a packet transaction is a guard that is a predicate. For instance, a guard could be defined as a header field having a specific value. Once this predicate evaluates to true, the packet transaction associated with it is executed. Domino has been evaluated in terms of its expressiveness when used to implement various data plane algorithms for traffic engineering, congestion control, active queue management, network security and measurement. The authors have compared the number of lines of code (LoC) for the data plane algorithms written in Domino and P4. The LoC value for Domino is considerably smaller than those of P4.

In chapter 1, ISA-level contributions such as POF and NetASM were mentioned. Describing desired packet processing functionality at ISA-level is cumbersome. However, this does not undermine the significance of ISA. A widely adopted ISA is of benefit to compiler development and hardware design. The compiler converts a given higher-level language to ISA-level representation. Hardware architects provide microarchitecture required for hardware implementation of the ISA in question.

2.2.3 User-space Packet Processing

Implementation of protocol stacks in operating systems has improved over the years. However, at high rates of packet arrival, these implementations lag behind. The survey in [41] has gathered the shortcomings of packet processing in OS from a number of research works. One of the notable shortcomings is the high cost of context switch to kernel and back to user space. Every time an application needs to receive a packet, it must make an OS system call. After the OS has taken control, another context switch is made back to the application. According to [42], as many as 1000 CPU cycles are consumed per packet in these context switches. The solution to this inefficiency is user-space packet processing in which the kernel is bypassed. This bypassing enables the packet buffers to be directly accessible from the user space. The other solutions required for mitigating inefficiencies of packet processing by kernel are sharing the packet buffer between user space and NIC, processing

packets in batches and supporting the multi-queue feature of modern NICs for load balancing [43].

Data Plane Development Kit (DPDK) is an open source set of libraries for fast packet processing in the user space. The purpose of DPDK is sending and receiving packets with minimum possible number of cycles. According to [44], the throughput of layer-3 forwarding of 64-byte packets with LPM as default lookup method using Intel NICs ranges from 29.76 to 74.4 Million packets per second (Mpps). For some packet processing functions, the throughput is hundreds of Mpps [45]. As of now, DPDK supports the dominant CPU architectures and NICs from different vendors. Instead of the interrupt-driven approach taken by the operating system's kernel, it uses polling because when the rate of packet arrival is high, interrupt-driven approach is inefficient. The other similar frameworks are netmap [46] and PFQ [47].

2.3 Hybrid packet processing solutions

In addition to the software-based solutions, there are solutions implemented on FPGAs. FPGAs are devices with a pool of hardware resources that can be interconnected in order to achieve the desired hardware architecture. The desired hardware architecture is provided in the form of Hardware Description Languages (HDLs) such as Verilog and Very High-Speed Integrated Circuit Hardware Description Language (VHDL). In recent years, it has become possible to describe the desired functionality in higher-level languages such as C/C++. The concept of using higher-level languages for obtaining the corresponding functionality in hardware is called High-level Synthesis (HLS). So, FPGAs are hardware solutions but since they allow reconfigurability, they have flexibility characteristics similar to software. For this reason, it is considered as a hybrid solution.

In 2010s, Graphics Processing Units (GPUs), received attention for use as the platform for execution of packet processing. GPUs are specialized processors for graphical operations such as high-performance rendering of images. The most notable architectural characteristic of GPUs is large pool of parallel resources for thread-level parallelism (TLP).

2.3.1 Solutions based on FPGAs

NetFPGA is an open-source FPGA-based platform for implementing network processing functionality. There are 1 Gbps, 10 Gbps and 100 Gbps NetFPGA variants [48]. NetFPGA SUME [49] is the latest in the line-up of NetFPGA devices. It is a Peripheral Component Interconnect Express (PCIe) board containing four 10 Gbps ports. The board hosts XILINX Virtex-7 690T device for custom logic realization. With more than 690K logic cells, 52,920 kb block Random Access Memory (RAM), and high-speed transceivers, it can be programmed for standalone, peripheral, and switch use cases.

SwitchBlade [50] is a platform for rapid deployment of custom protocols. It is designed with the aim of providing the right balance between flexibility of software and performance offered by hardware. It is implemented on NetFPGA board. In SwitchBlade, workloads pertaining to multiple protocols can run in parallel. Each corresponding data plane is called a Virtual Data Plane (VDP). Functional units in SwitchBlade are organized in pipelined fashion. The main operations in the pipeline are preprocessing in which fields for matching are selected, hashing, matching and post-processing. Both LPM and exact matching are supported by SwitchBlade. In the experimentation performed by the authors, it has been used for IPv4 and IPv6 forwarding, path slicing and OpenFlow switch. The forwarding rate of SwitchBlade is 1.5×10^6 packets per second (pps) for 64-byte packets. This translates to 732.42 Mbps throughput.

As mentioned earlier, the functionality of FPGAs is dependent on HDLs or languages such as C/C++. The fact is that many network innovators are not familiar with HDLs. Furthermore, even C/C++ languages are at a low abstraction layer for describing network processing functionality. For this reason, many FPGA-based platforms come with a toolchain that takes the desired functionality in a language close to networking. In [51], a complete solution is provided in which the functionality is described in high-level language called PX. The PX-specific compiler converts the code into the equivalent HDL and then to the bitcodes required for configuring the underlying FPGA platform. It achieves 100 Gbps throughput when dealing with minimum-sized Ethernet frames.

In a similar approach, [52] accelerates network functions for commodity servers. It is programmed in a language called ClickNP. When configured as a firewall, it can process 64 million packets per second (Mpps) with each packet being 64 bytes. Another work in which a complete solution is provided is P4FPGA [53]. It is P4 compiler with a custom backend for generating HDL code to be used as the input

to synthesis and place and route on FPGA. P4FPGA has been evaluated in terms of its capability to support different data plane applications. Match-Action processing for L2/L3 forwarding on P4FPGA takes 124 ns for packets whose size is up to 1024 bytes.

FPGAs have been extensively used for packet parsing. In these solutions, the header sequence is described in HDL or a higher abstraction layer. In [54], a domain-specific language called PP is used for describing headers. Based on this description, the FPGA is configured for providing the desired implementation. For a stack containing Virtual Local Area Network (VLAN), IPv4/IPv6 and TCP/UDP, it achieves throughput values of 302 and 578 Gbps using 1024- and 2048-bit datapaths respectively. The latency figures are above 300 ns. However, it is stated that these figures are raw throughput values obtained by multiplying datapath width and operating frequency. The effect of short packets and quantization over wide word must be taken into account. As a result, the actual packet parsing throughput is less than the provided values.

Since P4 language also describes headers, some solutions use it as the input to the tool chain that generates the HDL. This approach is used in [55] and [56] and the achieved throughput is 100 Gbps. The highest achieved parsing throughput using FPGAs is in [57] in which up to 1 Tbps throughput is achieved.

2.3.2 Solutions based on GPUs

PacketShader [58] uses the architectural features of GPUs to enhance the performance of software routers. In addition, I/O optimizations have been provided for implementing batch processing to eliminate the overhead caused by memory management on a per-packet basis. In IPv4 forwarding, PacketShader achieves throughput of almost 40 Gbps. For IPv6 forwarding, the throughput value is 38.2 Gbps.

The work in [59] considers both strong and weak points of GPUs. On the strong side, GPUs hide memory latency incurred by lookups by switching to another thread. General-Purpose Processors (GPPs) also support multithreading but they support 2 or 4 threads. In GPUs, tens of threads are supported and the switching between them occurs very fast. On the weak side, the high memory access latency of GPUs is undesirable for packet processing. In addition, the memory bandwidth degrades in packet processing applications because random memory locations are accessed. The main argument of their work is that performance brought by GPUs is not due

to their computational capacity, rather due to efficient context switching in hardware. In order to emulate such efficient context switching in CPUs, a technique called G-Opt is developed. It is based on group prefetching and fast context switching. It re-orders code for concurrent memory access. This access pattern allows software pipelining. Use of G-Opt on GPPs yields throughput similar to GPUs. For instance, using 4 cores, G-Opt achieves throughput of close to 50 Mpps while with GPU this figure is 40 Mpps.

APUNet [60] evaluates whether fast context switching in GPP can solve a wide range of network applications. The findings confirm that besides the fast context switching, the computational capacity of GPUs is indeed a key contributor to performance. However, a barrier to achieving the full performance gain of GPUs in packet processing is the transfer bottleneck of PCIe. Typical PCIe bandwidth is considerably smaller than that of GPU memory. In response, the authors suggest use of integrated GPU in which CPU and GPU share memory. As a result of this unified memory space, the data transfer overhead is eliminated.

2.4 ASIC-based packet processing solutions

So far, the relevant software and hybrid solutions have been reviewed. Software and virtual routers achieve throughputs in the range of tens of Gbps. FPGA-based solutions provide throughputs in the range of hundreds of Gbps. But as discussed earlier, FPGA-based solutions are based on a high- or low-level description of the workload. As a result of this, FPGA solutions contain hardware specific to a known set of protocols. This is in contrast with the protocol-independence principle of SDN. Solutions compliant with SDN do not contain any protocol-specific state. By removing this dependence on specific protocols from FPGAs, their performance will degrade. In addition, TCAMs are required in high-throughput environments because of their parallel search capability. In FPGAs, it is possible to achieve TCAM functionality by emulation. However, this is inefficient in terms of resource usage. Another issue with FPGAs is that they run at considerably lower frequencies when compared with ASICs. In Terabit-scale packet processing, the minimum required operating frequency is 1.0 GHz. Because of their low operating frequency, FPGAs rely on ultrawide datapaths. Multiplying datapath width by operating frequency gives a raw throughput value which is not achievable for small packets. This issue is discussed in [54]. The latency figures for parsing alone is in the range of hundreds of ns while commercial routers and switches perform entire processing in such a time

window [61]. This is confirmed by the line-up of high-end commercial products. As will be seen in sections 2.4.1 and 2.4.2, hardly any such device is built upon FPGAs. In this segment, the ASICs have no rivals. This is the motivation for using ASIC-based solutions.

2.4.1 Network Processors

Network Processors (NPs) gained popularity in early to mid-2000s. They were basically processors with functional units optimized for processing packets. The focus, at that point, was not protocol-independence. Instead they contained the logic for implementing and accelerating the most commonly used network protocols. In [62], some of the shortcomings of NPs are presented. The main challenge in reaching high performance with NPs is the large gap between the processor and the memory. As opposed to GPPs, use of caching is of little help because locality of reference is missing in network processing. Instead, NPs mitigated this issue by using multithreading. When an NP core requests an item from memory, it switches to another thread.

One of the most notable network processors was the Intel IXP2800 and IXP2850, the latter of which has integrated cryptographic units. The store-and-forward packet processing is performed by 16 32-bit micro-engines, each of which can run 8 threads. The maximum operating frequency is 1.4 GHz. Each micro-engine has an 8K instruction store. The micro-engines cooperate with each other for solving packet processing problems. The complete datasheet is available in [63].

Today network processors are not as widely used as early 2000s. However, there are a few of them in use. Cisco has a 400 Gbps multicore network processor which is comprised of 672 general-purpose processors [64]. Each of the processors has an 8-stage pipeline. The instruction set contains network-specific instructions. It can be programmed in C and assembly language. Another notable network processor is Nokia NP4. It is a 3 Tbps network processor that supports deep packet lookups and real-time telemetry [65].

2.4.2 Programmable Switch Chips

In the post-NP era, Pipelined Lookup Grid (PLUG) [66] was one of the first architectures providing flexibility with the aim of supporting new protocols. It consists of a grid of tiles that can be combined for implementing different protocols.

In [20], a programmable packet parser was presented. It operates based on the Match-Action principle. Based on the action determined by its current state, it extracts a specific field of the arrived header to determine its next state. Actions associated with a given state determine what fields of the arrived header must be extracted and what fields must be written to the field buffer. This parser achieves throughput of 40 Gbps.

RMT [26] contains 16 instances of the parsers presented in [20] and a 32-stage pipeline of Match and Action Units (MAU). The parsers write header fields to a 4096-bit register called Packet Header Vector (PHV) which traverses through the pipeline. The PHV has 224 entries. The physical architecture of RMT closely resembles the switch abstraction made by P4. Inside each MAU, there are 16 TCAMs, each being a 2K×40-bit unit. In addition, there are 106 SRAM blocks each of which is 1K×112 bits. These units can be flexibly assigned for exact match, action memory and statistics. What is meant by action memory is the parameters required for modification of header fields. Match crossbars generate the search key from the fields in PHV and present it to the ternary and exact match tables. The outcome of the match determines the actions to be executed. The actions are executed by action engines. Each action engine is an Arithmetic Logic Unit (ALU) for modifying PHV entries. There is an action engine associated with each PHV entry. Figure 2 provides a high-level view into a MAU. Only one of the 224 ALUs is illustrated. The output of the ALU is modified header field.

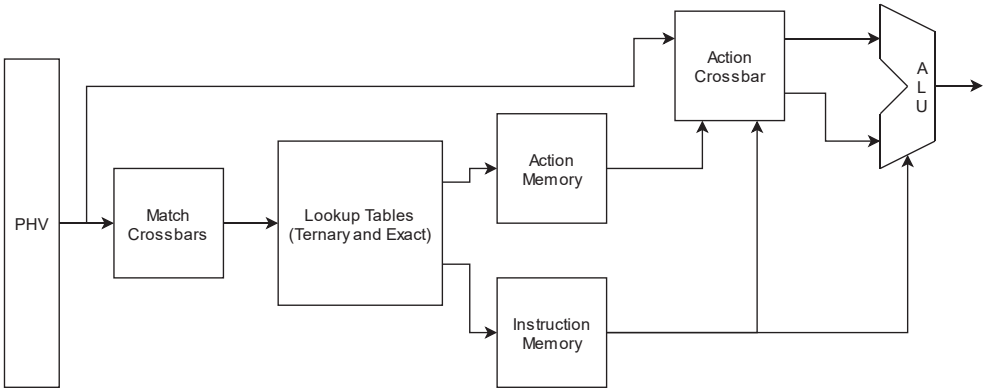

Figure 2. High-level view of the internal components of a MAU (adapted from [26])

If the dependencies in the program allow, it is possible to overlap the operation of MAU instances. In other words, it is not mandatory for MAU_i to start match operation after action execution in MAU_{i-1} has been done. However, this

overlapping is not always possible due to dependencies. There are match dependencies and action dependencies. Match dependencies occur when field(s) modified by a MAU must be used as the search key in the subsequent MAU. In this case, the matching in subsequent MAU starts only after the action execution in current MAU is over. Action dependencies occur when field(s) modified by a MAU must be used as input(s) to action execution in the subsequent MAU. In this scenario, matching in subsequent MAU can overlap with action execution in current MAU. Figure 3 illustrates the two dependencies with respect to time.

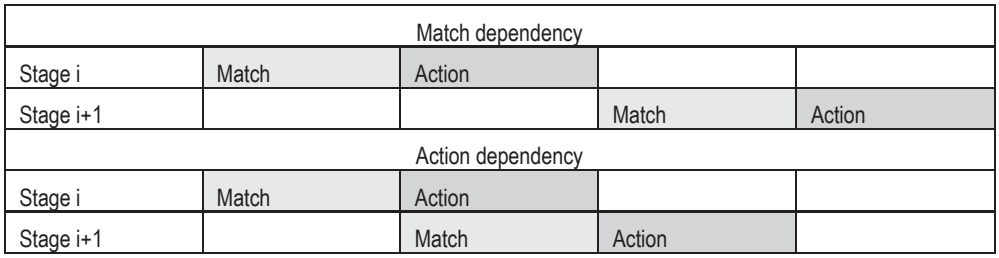

Figure 3. Match and Action dependencies in a Match-Action packet processing pipeline (adapted from [26])

The inter-MAU latency caused by match and action dependencies are 12 and 3 cycles respectively [67]. The delays are configured statically according to the dependencies present in the program. The parsers in RMT occupy a total area of 5.6 M gates after synthesis using an industry-standard 28-nm library. The area of the entire architecture has not been provided, though. Later on, RMT evolved into Protocol-independent Switch Architecture (PISA), which is a P4-programmable architecture [68].

The work in [40] provides a machine model for programmable switches. Banzai is the compiler target for the Domino language discussed earlier. Banzai models the Action part of the Match-Action pipeline. As such, it does not model the matching operations. Banzai's pipeline does not stall and always sustains the line rate. The functional units at each stage of Banzai are called Atoms. They modify header fields and state in a single-cycle manner. In order to provide consistency when it comes to modifying state, atoms perform read, modify and write operations in a single cycle. After analysing various data plane algorithms, 8 different Atoms with different levels of complexity have been designed and synthesized using 32 nm process technology for running at 1.0 GHz frequency. Stateless Atoms, with an area of 1384 μm^2, perform arithmetic and logic operations on header fields. Stateful atoms modify state variables. The basic stateful Atom type simply modifies state and has area of 431

μm^2. More advanced stateful atoms contain logic for predicated assignment, assignment for both outcomes of a conditional evaluation, and nested conditional evaluation statements. The area of these atoms varies between 791 to 5997 μm^2.

Disaggregated RMT (dRMT) [69] is a Match-Action architecture similar to RMT, except that instead of a pipeline with 32 stages, it comprises 32 processors. Therefore, instead of a pipeline model, it is a run-to-completion model. Once a packet is assigned to a dRMT processor, it remains there until all the processing has been done. In addition, the tables are not attached to the processors. The processors are connected to table clusters via a crossbar. The processors contain 32 ALUs, as opposed to 224 in RMT. The motivation for disaggregating memory from action stages, is that if the memory resources within a stage are not used in one stage, they cannot be assigned to another stage. By providing a crossbar, the authors of dRMT provide this added flexibility to RMT.

In addition to programmable packet parsing and processing, support for flexible packet scheduling is required as well. In [70], a hardware primitive for programmable packet scheduling has been designed. The primitive is called Push In First Out (PIFO). It is a priority queue to which items can be pushed to arbitrary positions according to their rank. However, dequeuing operation is applied only to the head. PIFO can be programmed to provide the functionality of different scheduling algorithms. It supports hierarchical scheduling of up to 5 levels with programmable scheduling for each level. It has been synthesized using 16 nm standard cell library. The synthesis results confirm that it can run at 1.0 GHz which makes it suitable for a switch with 64×10 Gbps ports. The results on area indicate that PIFO incurs only a 4% increase in chip area. A generalization of PIFO, referred to as Push In Extract Out (PIEO) is proposed in [71]. It improves the expressiveness of PIFO by allowing dequeuing to be applied not only to the head but to arbitrary positions as well. PIFO cannot be used for more general scheduling algorithms, such as Worst-case Fair Weighted Fair Queuing (WF^2Q), in which the eligible item with highest rank must be scheduled. This is because in scheduling implemented by PIFO, the item with highest rank is always assumed to be eligible. Such scheduling algorithms require a primitive that can dynamically filter a subset of items and select the one with highest rank.

One of the notable commercial products was Intel FM5000/FM6000 Ethernet switch chip that use a microcode-programmable packet processing pipeline called FlexPipe [72]. Although it is an Ethernet switch chip, it provides features such as a number of lookup tables that can be combined for flexible frame classification. In

addition, the Arithmetic and Logic Units (ALUs) provide a means for implementing custom actions.

Netronom NFP-6000 is a programmable multi-core processor for data plane processing [73]. It consists of 120 programmable flow processing cores and 96 packet processing cores. The amount of on-chip memory is 31 MB. It provides 200 Gbps throughput for L2-L7 processing. It supports both exact and ternary matches. It is used in Netronome Agilio LX series of SmartNICs.

Barefoot Tofino is a programmable Ethernet switch chip. Its internal architecture is called PISA, which is based on the RMT architecture. It is programmed using the P4 language. It sustains 6.4 Tbps aggregate throughput by integrating 4 pipelines into the chip [74]. Each of the four pipelines is very similar to RMT. In addition to packet forwarding, its programmability allows it to be used for telemetry and load balancing as well. Barefoot's successor to Tofino is Tofino 2. Built using 7 nm processing technology, it sustains a doubled aggregate throughput of 12.8 Tbps by doubling the number of transistors to 21 billion [75].

The Unified Access Data Plane (UADP) is the ASIC in Cisco Catalyst 9000 switches [76]. It consists of a flexible parser and pipeline. For some more advanced network functions such as fragmentation and encryption, it uses on-chip micro-engines instead of the pipeline. The latest architectural variant is UADP 3.0. It sustains 1.6 Tbps aggregate throughput. Built on 16 nm process technology, it contains 19.2 billion transistors. It contains 36 MB memory for buffering. The width of its lookup tables allows storage of IPv6 address in their entirety.

The most high-performance chip is currently the Broadcom Tomahawk 4. Designed for datacentre and cloud computing environments, it sustains aggregate throughput of 25.6 Tbps. It is built using 7 nm process technology [77].

As can be seen, very little architectural information is available for commercial products. It is notable that the major vendors offer products with similar capabilities. A unifying characteristic of all in recent years has been emphasis on flexibility and programmability.

2.5 Summary of Packet Processing Solutions

Table 2 summarizes and compares characteristics of the packet processing solutions reviewed in this chapter.

Table 2. Summary and comparison of packet processing solutions

Spectrum	Solution	Characteristics						
		Maximum throughput	Programmability	Latency	Commercial deployment	Primary use case	Power efficiency	Market segment
Software	Software routers	100 Gbps	High	Variable	Yes	Steering packets to and from virtual machines	Low	Entry-level
	Programming languages	Depends on the hardware target	High	Can be fixed or variable	Yes	Programming the packet processing system	Depends on the target platform	Entry-level to high-end
	User-space packet processing	A few tens of gigabits per second	High	Variable	Yes	Bypassing the OS	Low	Entry-level to mid-range
Hybrid	FPGA	Tens of gigabits per second	High but requires knowledge of HDLs	Can be fixed or variable	Limited	Prototyping, acceleration	Low	Mid-range
	GPU	Tens of gigabits per second	High	Variable	Not for packet processing	Using massive thread-level parallelism	High	Mid-range
Hardware	Network Processor	A few terabits per second	High but may require assembly- or microcode-level programming	Variable	Limited	Providing programmability into networking hardware	High	Mid-range to high-end
	Programmable switching chips	Tens of terabits per second	High	Fixed	Yes	Providing programmability and guaranteed performance into networking hardware	High	High-end

2.6 Applications of Programmable Data Plane

Programmable data plane enables innovation by providing the flexibility to implement different network protocols. In addition to this, they enable enhanced visibility into the network. This provides for more effective troubleshooting and diagnostic operations. As programmable data plane supports any protocol for which the corresponding functionality is written in software, a protocol could be designed

whose messages contain internal information of switches and routers. This information is updated as the packet belonging to this protocol traverses the network. This technique is referred to as In-band Network Telemetry (INT). A similar approach is taken in [78], in which sources of packets embed tiny packet programs (TPP) into packets for querying and manipulating network state. As the packet traverses the network, the containing TPP is executed on the switches and routers. This technique has become one of the most important use cases for programmable switch chips [79].

Another novel application is load balancing which is critical in datacentres. Modern flexible switch chips contain a vast amount of memory that can be flexibly used for storing per-connection state. In [80], a layer-4 load balancer is implemented in P4 for execution on a modern switch ASIC. Using their approach, up to hundreds of load balancing servers can be replaced by one switch, thereby reducing the cost of load balancing by two orders of magnitude. This solution has also become an industry solution [81].

One of the latest and most innovative use cases is offloading program execution to the switches. This concept is called in-network computing. Under this concept, the switches not only forward packets, they contribute to the processing of data contained in the packets as well. This is motivated by the fact that networking hardware are becoming more and more programmable. Furthermore, when computation is offered as a service, the packets must traverse the network until they reach the service provider's server(s). If the forwarding devices on the path to the service provider can also take part in the processing, more processing power is achievable. In [82], the problem of convolutional neural networks has been analysed in terms of its suitability for in-network computing. Their analysis reveals that current network hardware can be used to accelerate neural network inference workload of datacenters. In [83], similar analysis is performed for implementing a line-following algorithm on a P4-programmable NIC. By carefully dividing the required calculations among the Match-Action stages, they can achieve 19 decisions per second on 640×480 greyscale images. Some principles are provided in [84] for in-network computations. The authors suggest offloading primitive calculations rather than the whole application because the computational resources on switches are limited. Care should be taken so that failure of switches does not make computation by the server impossible. In addition, it should be possible to recover any lost data as a result of failure in switches.

The programmable data plane has also become an enabler for 5G radio networks offering ubiquitous connectivity using technologies from radio, transport and cloud

domains [85]. There is growing interest in using packet-based networks such as Ethernet in the transport network. The flexible Radio Access Network (RAN), in collaboration with SDN and NFV allows to configure the network with different functional splits in transport network nodes [86]. This dynamic solution requires virtual resource instantiation needs, referred to as network slices. In addition, different standards such as [87], [88], [89] each require different packet format. A programmable packet processing system is required for implementing packet forwarding and performing the reconfigurations when a different functional split is required for changing slice requirements.

3 A NEW PROGRAMMABLE PACKET PARSER

Packet parsing is the first step in processing of packets. It involves recognizing the headers present in a packet and extracting them for processing. In this chapter, a new programmable packet parser is presented. It uses program control instead of relying on TCAMs for maintaining state. The contents of this chapter are based on publications P_I, P_{II}, and P_{III}.

3.1 A Closer Look at Packet Parsing

In order for a packet parser to operate correctly, it must know the first header. When multiple headers must be parsed, the next header must be determined. Some headers have an indication of the header that follows. Examples of fields that contain this kind of indication are EtherType in Ethernet, Protocol in IPv4 and Next Header in IPv6, and Protocol Type in Generic Routing Encapsulation (GRE). For the headers that do not contain this indication, there must be a default header associated with them. Alternatively, the next header can be considered as the payload that is not subject to parsing. As far as a packet parser is concerned, the payload is the part of the packet that does not need parsing. For instance, a layer-2 switch is only concerned with layer-2 header such as Ethernet and the rest is treated as payload while it may indeed contain higher-layer headers. Another issue that the parser must be concerned with is determining the size of the header being parsed so that it knows the starting boundary of the next header or the payload. Some headers such as IPv4 and GRE have variable length because they contain optional fields. On the other hand, headers such as IPv6 and Multiprotocol Label Switching (MPLS) have a fixed size. The extension headers of IPv6 protocol are independent headers and not part of the fixed IPv6 header.

In addition to the points mentioned above, the packet parser must keep track of its progress within the stack of headers. Headers have different sizes. For instance, MPLS header has size of 32 bits while IPv6 has size of 320 bits. If the processing width is chosen to be 32 bits, the entire MPLS header fits into the data unit, while for IPv6 10 such data units are required. Conversely, if the data unit is chosen to be

320 bits, multiple headers of smaller size fit into the data unit. Due to this variety of header sizes, for all practical choices of processing width, there will be both smaller and larger headers. As a result, it is important for the packet parser to keep track of their progress. This means that packet parsing is a stateful operation.

3.2 TCAM-based State Machine

As mentioned earlier, packet parsing is a stateful operation. As such, the parser must maintain state. The packet parser in [20] and [26] operates in Match-Action mode. For matching, it generates 32-bit search keys from the arrived header. Assuming that the arrived header data contains EtherType field of the Ethernet header, it is extracted and used as a search key to determine the next header. The value of EtherType alone is not sufficient for correct state transition. For this reason, each field extracted from the header is appended with an 8-bit value that represents current state. The 40-bit search key is then presented to a 256 × 40-bit TCAM. With this addition, a given value of EtherType is distinguished from the same 16-bit value in another header and state transitions work correctly. Associated with each state is an action. The actions are stored in a 256 × 128-bit SRAM. Figure 4 illustrates the entities present in the parser.

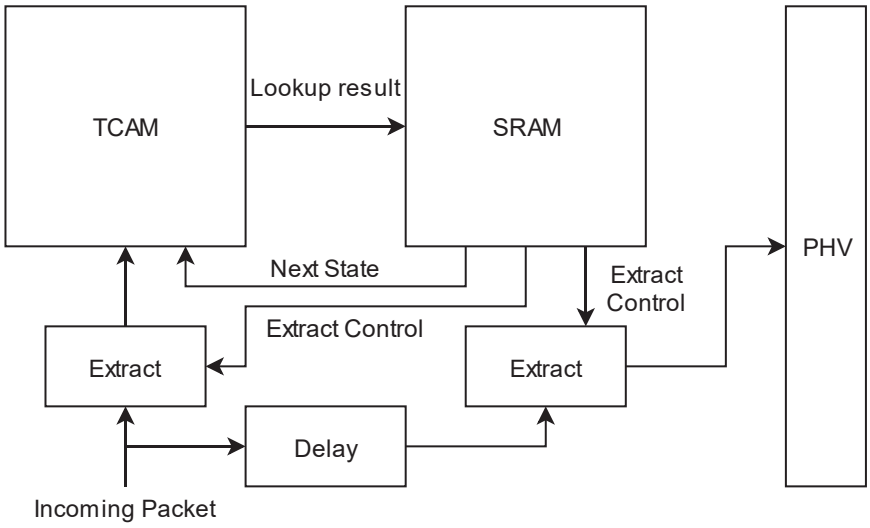

Figure 4. Parser used in RMT architecture (adapted from [26])

Let's assume that the datapath width of this parser can accommodate the minimum-sized IPv4 header. When IPv4 header arrives, the value of Protocol field is extracted for determining the next header, but the size of the header must be determined as well. Because at this point it is not yet known whether what follows the minimum-sized IPv4 is IPv4 optional fields or the next header. Therefore, the value of Internet Header Length (IHL), which contains the size of the header in terms of 32-bit words is also extracted and appended to the search key. Since the search key is now comprised of 2 header fields, the number of actual search keys is the result of multiplication of the number of possible values of the 2 fields. There are 11 valid values for IHL (0x5-0xF). Assuming that the parser is programmed to parse 8 different headers following the IPv4 header, there are 88 entries within the TCAM for the next state transition. Therefore, 34 percent of the TCAM entries are used for IPv4 alone. The reason for this is that all actions are based on the match result and there is one lookup table for all matches, whether they are for determining the next header or the size of the header. Use of narrower datapath width is also inefficient because for multiword headers such as IPv4 and IPv6, more states will be required.

The parser in RMT is not the only parser that employs TCAMs. The parser in [72] uses multiple instances of TCAM. The datapath width of this parser is 32 bits. The incoming data is written to an 88-byte bus. The parser is organized in a pipeline of *slices*. Each slice receives the 32-bit state output of its preceding slice and 32 bits of frame data. Together, they are used as a 64-bit search key for looking up into the TCAM located in the slice. The matching entry determines the action, which updates the state and defines how frame data should be written to the 88-byte bus. This parser is similar to RMT parser, except that there is a pipeline of TCAM-SRAM pairs instead of just one pair.

3.3 An Alternative to TCAM-based State Machine

TCAMs are very powerful devices for searching. Their ability to store don't care values makes them a high-performance solution for LPM matching. However, prefix matching is not dealt with in packet parsing. The most area- and power-efficient means for programmable packet parsing is using program control instead of strict Match-Action using TCAM. Program control logic is far simpler than the logic of TCAM. With program control, the parser turns into a custom processor. Program control is also a form of state maintenance and transition because the value of

Program Counter (PC) can be considered state and the instruction it points to represents action.

One of the important design choices is the processing width. It is important to note that wide processing width does not mean throughput higher than in architectures with narrower processing width. The reason for this is that when looking up the next header field, it takes a number of cycles until the result is available. During these cycles the parser must wait. The cycles are required for field extraction, lookup, and resolving. The chosen processing width is 32 bits. It is also possible for this parser to read 8 or 16 bits from the buffer of incoming packets, but the processing width is 32 bits. So, 8- and 16-bit reads from the buffer are retrieved in 32-bit zero-extended format. An instruction is associated with each processing unit. The instruction specifies what fields must be written to the PHV. In addition, they specify what fields must be extracted for determining the next header and the size of the header. The next header is determined by comparing the field containing indicator of next header with the values associated with the header under parsing. These values are stored in a parameter memory that can provide them in parallel. The comparisons are also done in parallel to speed up the process. Figure 5 illustrates the new packet parser that employs program control for state maintenance.

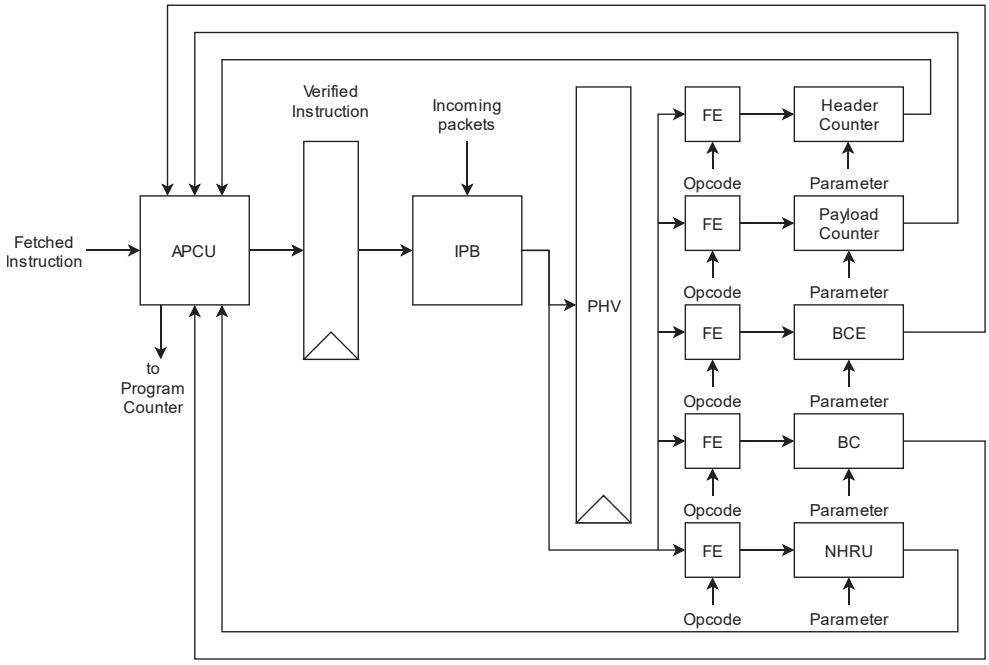

Figure 5. The proposed packet parsing processor

The difference compared to the TCAM-based approach is that in this architecture there is a small number of comparison functional units for parsing different headers. They are loaded with the extracted value from the header and the parameter values. The extracted value of next header field is compared only with values associated with the header being parsed, whereas in TCAM-based approach this value is compared with all TCAM entries. Furthermore, there is no need to append a state-value to this field for comparison. The parse program for parsing IPv4 header has 15 instructions because IPv4 has 15 32-bit words. If minimum-sized IPv4 header is encountered, the instructions after the 5th instruction will be skipped.

3.3.1 Functional Units

The main functional units of this packet parser are listed in Table 3.

Table 3. Functional units of the new packet parser

Functional Unit	Purpose
Incoming Packets' Buffer (IPB)	Storing the incoming packets and providing them in 8-, 16-, and 32-bit units upon request.
Advanced Program Control Unit (APCU)	Providing the correct instruction address to the PC. In addition, providing no-operation (NOP) instruction until the correct instruction reaches the functional units.
Field Extractors (FE)	Extracting the fields containing header size, payload size and next header
Next Header Resolve Unit (NHRU)	Compares the next header identifier with the values associated with header under parsing
Branch Catalyst (BC)	Speeds up multiway branching by comparing all ways in parallel. Useful for evaluating the value of multibit flag fields
Branch Condition Evaluator (BCE)	Evaluates the condition of the branch. If the branch condition is satisfied, program flow changes
Header Size Counter	Initialized to the size of current header. When reaches zero, causes a branch to the first instruction for parsing next header
Payload Size Counter	Initialized to the size of current header. When reaches zero, causes the APCU to branch to the beginning

3.3.2 Instruction Format

The instruction fields are elaborated in Table 4. There is an instruction field for each of the functional units present in this architecture. Therefore, the proposed architecture is a Very Long Instruction Word (VLIW) architecture. VLIW

architectures are discussed in detail in [90]. These architectures are also referred to as Explicitly Parallel Instruction Computing (EPIC) [91].

Table 4. Instruction fields

Instruction field	Width (bits)	Purpose
Branch condition	4	Specifies the condition for branch
Branch type	3	Specifies the type of branch
Branch offset	7	Offset for branching
Field extraction opcode 0	5	The field to extract for branch catalyst
Field extraction opcode 1	5	The field to extract for NHRU
Field extraction opcode 2	6	The field to extract for BCE
Field extraction opcode 3	5	The field to extract for header counter
Field extraction opcode 4	5	The field to extract for payload counter
PHV filler opcode	8	Specifies how the arrived header segment must be written to PHV
Size of header segment	2	Requests a 1-, 2-, or 4-byte unit from the IPB
Parameters Memory Address	5	Address of the entry containing parameters for parsing the header
Stack in select	1	Whether the address of current or next instruction should be pushed to the stack
Stack push	1	Causes the selected value to be pushed to the stack

3.3.3 Instruction Pipeline

The instruction pipeline stages are elaborated in Table 5.

Table 5. Instruction pipeline stages

Stage	Outcome
Instruction Fetch (FI)	Instruction fetched from the instruction memory is written to instruction register
Decode (D)	The fetched instruction is validated and if necessary nullified
Fetch Header (FH)	An 8-, 16- or 32-bit unit of the arrived packet is written to the PHV
Extract (X1)	The programmer-specified portion of the latest arrived header segment is extracted
Compare (X2)	The extracted segment is subject to comparison or condition evaluation
Resolve (X3)	The outcome of comparison or condition evaluation is determined

3.4 Throughput Evaluation

In this section the throughput of the designed packet parser is evaluated. The width of the datapath is 32 bits and the operating frequency is 1.19 GHz. This means that the maximum possible throughput will be 38.08 Gbps. The outcome of experiments reveals if this ideal throughput value will be achieved when different workloads are run on this architecture. The intention is to determine how many packet parser instances are required for sustaining aggregate throughput of 640 Gbps.

3.4.1 Parsing Individual Headers

The first step in the evaluation of the designed packet parser's performance is measuring the achieved throughput when individual headers are parsed. A number of headers have been chosen for this purpose as shown in Figure 6. For headers with variable length, the parsing time and achieved throughput depend on the size of header under parsing. Fixed size headers result in better throughput because no evaluation of fields indicating existence of optional fields is required.

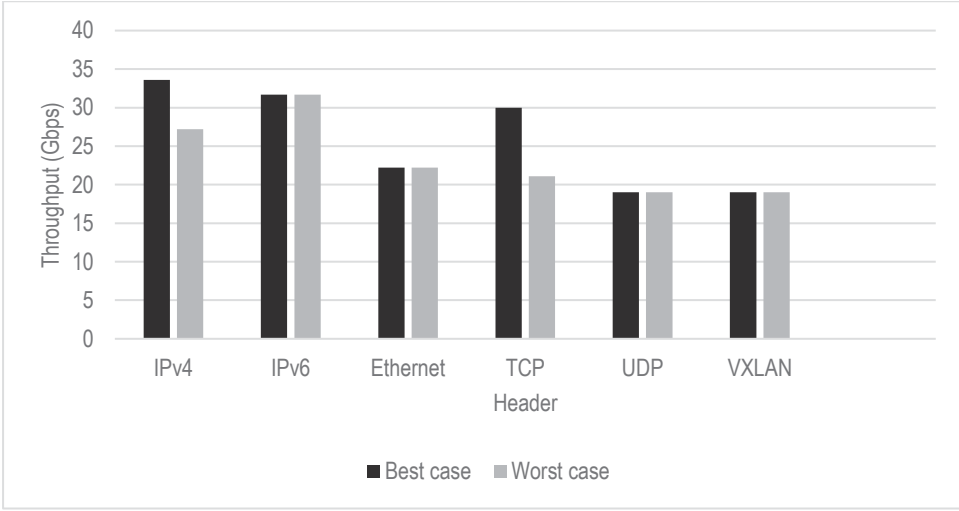

Figure 6. Throughput when parsing individual headers

Throughput depends also on the size of the payload. Large packets result in higher throughput because the content of the payload does not need evaluation. The payload is simply forwarded. Figure 7 illustrates the variation in throughput with

Ethernet, IPv4 and IPv6 packets with payload size of 46, 128, 512, and 1024 bytes. The IPv4 packet under study has the minimum-size header. They have differing starting values because different headers result in different throughput values. As the payload size increases, the achieved throughput approaches the ideal throughput. As can be seen, larger packets result in better throughput. In different deployments, packets of different sizes will be encountered. In order to provide guaranteed performance, the lowest achieved throughput must be used as the basis.

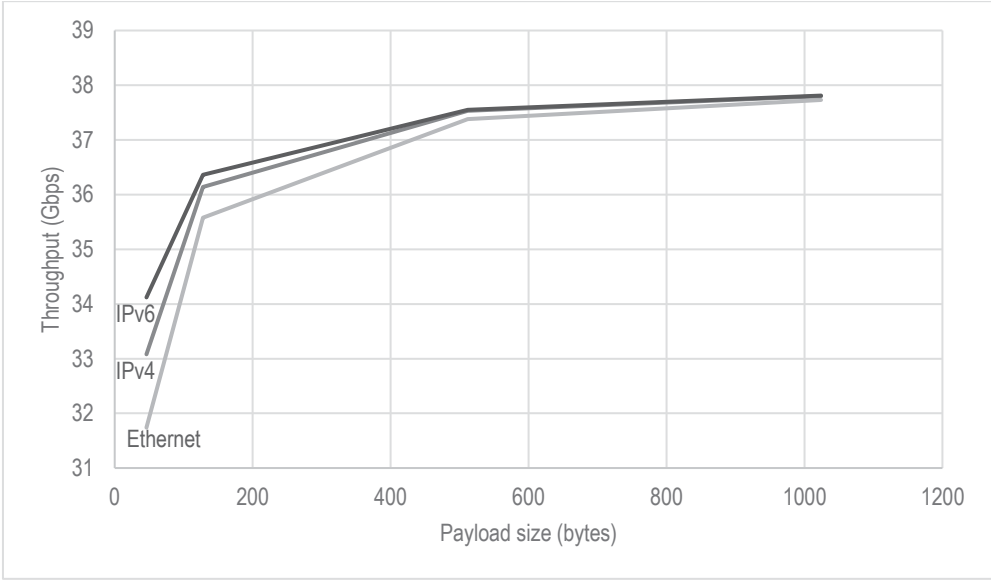

Figure 7. Resulting throughput when parsing Ethernet, IPv4, and IPv6 packets with 46-, 128-, 512-, and 1024-byte payload

3.4.2 Parsing Header Stacks

In order to evaluate the throughput of the designed parser under more demanding workloads, the throughput when parsing packets with the following two stacks of headers has been tested:

-Basic: Ethernet - IPv4 - TCP

-Full: Ethernet - 2×VLAN - 2×MPLS - IPv4 - TCP

IPv4 and TCP are both minimum sized in both header stacks mentioned above in order to limit the achievable throughput. The resulting throughputs are reported in Table 6.

Table 6. Achieved throughput when parsing basic and full header stacks

Header stack	Total size of headers (bits)	Parsing time (cycles)	Parsing throughput (Gbps)
Basic	432	28	18.36
Full	560	42	15.87

It should be noted that the two stacks based on which the throughput values in Table 6 have been achieved are not fixed stacks. In other words, each packet that has been parsed by this parser in this experiment could have had a different sequence of headers. The values in Table 6 correspond to those packets whose header sequence is according to the two header stacks mentioned above. If fixed headers were used, the resulting throughput would be higher. Both basic and full header stacks have Ethernet, IPv4 and TCP within them. However, the achieved throughput is lower than the resulting throughput when each of these headers were individually parsed. The reason for this is that when a branch must be made to the first instruction in charge of parsing the next header, a number of dead cycles are encountered during which parsing cannot occur. The number of these cycles increases if the latest header segment read from the buffer of incoming packets contains indication of next header. This is the case in Ethernet because the next header begins right after EtherType field. A similar condition occurs in MPLS as well. Although MPLS does not have a next header field, it does have an S bit that must be evaluated to determine if another MPLS label follows the current label.

3.4.3 Enhancements for achieving higher throughputs

In the performance evaluations so far, each instruction issues a read request to the buffer of incoming packets in order to retrieve a 32-bit header segment. For an ingress parser that parses the packets as they arrive, the number of cycles required for serving this request depends on port speed. With 10 Gbps ports, as in the case of ports in [26], it takes 3.2 ns until 32 bits arrive. This is equivalent to 4 cycles in a 1.19 GHz system. During the cycles that the requested data is not available, no parsing can take place because the requested data is not available yet.

Since the parser issues a read request in each instruction and each read request for retrieving 32 bits results in idle cycles, higher throughput and better utilization

of functional units is achieved if the parser switches to parsing a packet arriving on another port. If a parser is assigned to parse packets coming through four 10 Gbps ports and if it switches to parsing of the next port each cycle, by the time it reaches serving a given port again, 4 cycles have elapsed, and the data is ready. This means turning the packet parsing processor into a multithreaded processor with support for 4 threads. Each thread serves its corresponding port. For supporting multithreading, all components that maintain state must have an independent instance for their own thread. Stateful components include the program counter, APCU, PHV, header and payload counters. Field extractors, NHRU, BC, and BCE do not maintain state, and hence can be shared by different threads.

Another technique that can be used for boosting throughput is reading header segments from the buffer in large bursts. Doing so requires some modification to the components used. For instance, the field extractors will require larger multiplexers. However, requests for large bursts can be issued when the payload of the packet is being forwarded. Reading in large bursts is only beneficial if during parsing of headers the parser has not kept pace with the rate of data arrival. Otherwise it causes stalls because it cannot increase the speed of the port.

With the performance-enhancing techniques mentioned here, the question is, what is the right number of ports to assign to a packet parser instance. This is equivalent to the question of what is the ideal number of threads to support in each packet parsing processor? The next question to solve is what is the optimum size of header segment that must be requested in each instruction. As seen in the previous section, the greatest performance loss is encountered when program flow changes to parsing of next header. In order to better understand this, consider parsing of two consecutive headers H_i and H_{i+1} shown in Figure 8. It is assumed that both headers are 32 bits wide. Therefore, one instruction per header is sufficient for parsing. Figure 9 illustrates timing diagram of the instruction pipeline when parsing the two headers. The next header indicator within H_i is extracted at t_{n+3}. At t_{n+4}, its value is compared with the expected next header indicator values. Based on the outcome of the comparison, the PC is loaded with the address of the instruction for parsing $header_{i+1}$ at t_{n+6}. During the cycles t_{n+3} to t_{n+8} no header fields are written to the PHV. The goal is to choose the right number of ports to utilize the parser efficiently and avoid idle cycles.

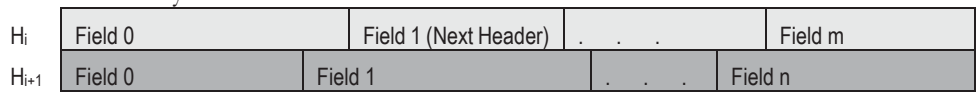

Figure 8. Two consecutive 32-bit headers

Instruction	Time												
	t_n	t_{n+1}	t_{n+2}	t_{n+3}	t_{n+4}	t_{n+5}	t_{n+6}	t_{n+7}	t_{n+8}	t_{n+9}	t_{n+10}	t_{n+11}	t_{n+12}
Instruction for H_i	FI	D	FH	X1	X2	X3							
Instruction for H_{i+1}								FI	D	FH	X1	X2	X3

Figure 9. Timing diagram for instruction pipeline when parsing two consecutive headers

Figure 10 illustrates the timing diagram for the instruction pipeline of an 8-threaded packet parser that has been assigned parsing of packets arriving through 8 ports. Associated with each of the ports is a thread. At each cycle, the parser switches to the next thread. It is assumed that the same consecutive headers H_i and H_{i+1} arrive through the ports. By the time the parser switches back to a given thread, the field extraction and loading of PC have been done, so there is no need for stalling. Those cycles that would have been wasted by stalling the pipeline are now used for parsing headers of packets coming through the other ports. This can be seen in the last row of Figure 10 where the instruction from thread corresponding to port P_j is fetched at t_{n+8}. This instruction is used for parsing H_{i+1}. Now, the only question that must be solved is the size of header segment to read from the buffer in each instruction. It takes 8 cycles until the parser switches back to a given port. During these cycles, 8 bytes have arrived. Therefore, each instruction reads 8 bytes. For headers whose size is not an integer multiple of 8 bytes, the corresponding instructions request reads of smaller units.

Port	Header	t_n	t_{n+1}	t_{n+2}	t_{n+3}	t_{n+4}	t_{n+5}	t_{n+6}	t_{n+7}	t_{n+8}	t_{n+9}	t_{n+10}	t_{n+11}	t_{n+12}	t_{n+13}
P_j	H_i	FI	D	FH	X1	X2	X3								
P_{j+1}	H_i		FI	D	FH	X1	X2	X3							
P_{j+2}	H_i			FI	D	FH	X1	X2	X3						
P_{j+3}	H_i				FI	D	FH	X1	X2	X3					
P_{j+4}	H_i					FI	D	FH	X1	X2	X3				
P_{j+5}	H_i						FI	D	FH	X1	X2	X3			
P_{j+6}	H_i							FI	D	FH	X1	X2	X3		
P_{j+7}	H_i								FI	D	FH	X1	X2	X3	
P_j	H_{i+1}									FI	D	FH	X1	X2	X3

Figure 10. Timing diagram for instruction pipeline of the 8-threaded packet parsing processor

3.5 Implementation Results

With the modifications discussed above, a single parser can serve eight 10 Gbps ports. Therefore, the throughput of one packet parser instance is 80 Gbps. For support of 640 Gbps aggregate throughput, eight packet parser instances are required. Table 7 outlines the number of components required in each instance and the total area and power dissipation. The ASIC technology used is 28nm Fully Depleted Silicon on Insulator (FD-SOI). The operating conditions are (SS, 0.9V, 125°C). The synthesis tool under use is Synopsys Design Compiler J-2014.09-SP4.

Table 7. Area and power dissipation values for components of an 80 Gbps packet parser

Component	Number of instances	Total area (μm^2)	Total power (mW)
PHV	8	149181.28	80.24
APCU	8	3587.76	47.20
Header counter	8	2766.08	11.36
Payload counter	8	2766.08	11.36
Instruction Memory (256×72)	1	44564.47	9.88
Read port for Instruction Memory	1	10450.56	14.83
Parameter Memory (32×448)	1	34426	3.46
Read port for Parameter Memory	1	8684	42.74
NHRU	1	920.33	0.73
BC	1	452.49	0.48
BCE	1	293.65	0.42
Total	-	258092.7	222.7

The total area of an 8-threaded packet parser that sustains 80 Gbps throughput is 258092.7 μm^2. For sustaining aggregate throughput of 640 Gbps, 8 processor instances are required. Since the instruction memory and parameter memory are relatively small, they are hosted on memories made of registers. Therefore, independent read ports can be easily added. In other words, there is one instruction memory and one parameter memory for all 8 processor instances, each of which runs 8 independent threads. Another benefit of sharing the instruction and parameter memories is that initialization process takes less time. The area of this parser must be compared with RMT's parser. In [26] parser components have been categorized into 4 classes. Table 8 outlines these classes and their corresponding components in the proposed architecture.

Table 8. Correspondence of packet parser components in RMT and the proposed architecture		
Component class	Components in RMT parser	Equivalent components in this architecture
1	TCAM	APCU, Parameter memory with 8 read ports
2	SRAM	Instruction memory with 8 read ports
3	Header identification and field extraction	NHRU, header counter, payload counter, BC, BCE
4	PHV	PHV

The total area of components equivalent to the TCAM-SRAM pair is 0.79 M gates in this architecture. This is in contrast to the 1.6 M gate figure of RMT parsers. For class 3 component, the area in RMT and this architecture are 0.35 M and 0.17 M gates respectively. The total area for all PHV instances in this architecture is 3.65 M gates and matches the value provided in [26]. Total gate count is 4.6 M and 5.6 M gates in this architecture and RMT parser instances respectively. As will be discussed in the section to follow, the area difference increases as more parser instances are instantiated for supporting higher aggregate throughput values.

3.5.1 Discussion of results

A 50% saving in area has been achieved for Match-Action memories of the parser by implementing an alternative mechanism for protocol-independent packet parsing. In TCAM-based approach, the search key is compared with all entries of the TCAM whereas in this architecture the search key is compared with the relevant values only. In addition, since a non-lookup mechanism has been used for maintaining the boundary between headers, the number of next header entries does not need to be as many as the TCAM entries. Hence, these values are hosted on memories made of registers. For such memories it is easy to add an independent port. This is not possible with TCAMs. In TCAM-based solutions, as more parser instances are added, each instance must have its own TCAM instance.

The area difference becomes more noticeable as the number of parser instances is increased for sustaining higher throughputs. In RMT parser, the TCAM-SRAM pair must be replicated for each parser instance. This approach is not scalable for instantiating tens of parser instances for achieving high throughputs. In this architecture, on the other hand, the memories are shared simply by adding extra read

ports. The elimination of TCAM is beneficial not only from chip area perspective, but from power dissipation point of view as well. According to [20], the power requirement for an 80 Gbps non-programmable packet parser that does not contain TCAM is around 400 mW. Comparing this figure with an instance of the designed 80 Gbps packet parsers, programmability is achieved at roughly 50% of that power requirement. No power dissipation has been provided in [20] for TCAM-based programmable parser. For a programmable parser that employs TCAM the power dissipation is far above this figure. Information regarding TCAMs are scarce as they do not come with standard cell libraries by default. As a result, it is not possible to compare the power dissipation of the proposed solution with a TCAM-based parser. However, it can be confidently said that this architecture is far more power efficient.

The achieved throughput can be enhanced by increasing the operating frequency. The fact that register-based memories have been used makes increasing the frequency a lot easier because registers are not the limiting factor in frequency scaling. The limiting factors are memories and the critical path of combinatorial components such as field extractors. By internally pipelining the field extractors, the potential timing constraint violations can be eliminated. Actual SRAMs and TCAMs, on the other hand, cannot be clocked beyond a certain point. The synthesis experimentations revealed that even at 2.0 GHz the timing constraints are still met. All the results in this chapter, however, correspond to operating frequency of 1.19 GHz.

4 AN ON-THE-FLY PACKET PRE-PROCESSOR

The implementation results of the programmable packet parser in the chapter 3 are promising. Due to the small area footprint of this parser, there is a lot of silicon real estate that can be utilized for enhancing functionality and throughput. In this chapter, packet processing capabilities will be provided for the packet parser of chapter 3. The content of this chapter is based on P_{IV}.

4.1 Use Cases for Processing Packets on the Fly

Processing the packets as they arrive can enhance the throughput. The entity that is involved with the packets as they arrive is the packet parser. Therefore, on-the-fly packet processing is performed inside or in close proximity to the packet parser. Integrity checking operations such as checksum validation can be easily performed by the parser by adding a few functional units such as ALUs. Assuming that there is a parameter problem within the arrived packet, the following benefits are achieved:

-If the packet must be discarded, it will not enter the pipeline. At high line rates, the packets compete for entering the pipeline. Less competition means less waiting time in the buffers and higher throughput for the rest of the packets. In addition, each packet consumes the computational and lookup resources of the system. By making the drop decision already at the parser, wasting units of the packet processing subsystem is avoided.

-If a packet destined to the sender must be generated, the MAUs are visited with the correct lookup address. Otherwise, by the time parameter problem in the packet is detected, the table containing the address of the packet's original source may have been passed. In this case, the packet must be recirculated which reduces throughput and increases latency.

-Certain tasks such as packet fragmentation and reassembly are best handled by a processor rather than a pipeline. For this reason, flexible ASICs such as Cisco UADP contain micro-engines to handle such operations.

Based on these reasons and considering the fact that the packet parser presented in chapter 3 is very lightweight, it is enhanced with packet processing functionality.

Since real packet processing involves looking up some form of destination address, the kind of packet processing dealt with in this chapter is packet pre-processing. However, it is also possible to include a tiny lookup table for highly recurring flows or those having high QoS requirements. Furthermore, the absence of lookup tables does not make this architecture less of a packet processor. An interface to lookup table(s) is sufficient for it to be considered as a full packet processor.

4.2 Architecture

Compared to packet parsing, packet processing is more complicated. Therefore, the fine-grain multithreaded processing model of the parser in chapter 3 must be turned into a simultaneously multithreading (SMT) in order to maintain line rate. It is still assumed that there are 64 × 10 Gbps ports in the system. At operating frequency of 1.19 GHz, it takes 8 cycles for a 64-bit segment to arrive. Therefore, on average, 8 single-cycle operations can be performed for each arriving segment. The PHV is written to by both the parser and the packet preprocessor containing a 32-bit ALU. 1024 bits of the PHV are reserved for the packet preprocessor to write. The packet preprocessor can read from the entire PHV. When the PHV traverses through the pipeline, the values calculated by the packet preprocessor can be used in addition to the values written by the parser. For instance, the pipeline can be programmed to use the IPv4 address provided by the packet preprocessor as the search key. During parsing, the packet preprocessor checks the integrity of the IPv4 header. If all the header fields have correct value, the packet preprocessor copies the destination address to a known position within the 32 entries reserved for the packet preprocessor. On the other hand, if there is a parameter problem within the header, the packet preprocessor copies the source address to the designated entry so that an ICMP message is sent to the source of the packet.

The preprocessor uses both branches and conditional execution for program control. Conditional execution is an efficient program control mechanism that avoids flushing of the pipeline. It is useful when a PHV entry must be modified only if a certain condition is fulfilled. However, in more complex programs, there is a need for branches. For instance, a header may contain different Type-Length-Value (TLV) messages, each of which requires different processing. In this scenario, conditional execution cannot be used as a replacement for branches because processing of each TLV requires a different subprogram.

4.3 Packet Preprocessor in Action

In this section it will be seen how the packet preprocessing functional unit operates in parallel to the packet parsing system. Two use cases have been provided. Preprocessing of IPv4 packets and packet fragmentation. The operations for these two use cases are executed in ingress and egress respectively. Table 9 elaborates the register index range, width, and the component writing to each range of PHV entries.

Table 9. Register index of PHV entries

Index range	Width (bits)	Written to by
R0-R31, R96-R127	8	Parser
R32-R79, R128-R175	16	Parser
R80-R95, R176-R191	32	Parser
R192-R223	32	Preprocessor

4.3.1 Preprocessing of IPv4 Header

Processing of IPv4 packets requires a fair amount of integrity checking prior to address lookup. If these integrity checking operations reveal a parameter problem, the packet is discarded and a new IPv4 packet is generated destined to the source of the original packet. The new IPv4 packet contains an ICMPv4 Parameter Problem Message. Table 10 outlines the IPv4 header fields for which integrity checking must be performed.

Table 10. Integrity checking operations on IPv4 header fields

Header field	Required checking
Ver	Must be equal to 4
IHL	Must be greater than or equal to 5
Total Length	Must be greater than or equal to 20
TTL	Must be greater than 0
Checksum	Must match the checksum calculated after packet arrival

In RMT architecture, tables are divided into the stages. Operations such as checksum verification require a number of cycles. This requires use of multiple MAUs. During the cycles that checksum is being calculated, it is possible to look up either the source

IP address or the destination IP address, but not both. If a problem is found with the header, there is the possibility that the packet has gone past the table containing the matching entry for the source IP address. In this case, the packet must start from the beginning of the pipeline to look up the source address because the ICMP message must be sent to the packet's source. By having the packet preprocessor take care of these operations, this kind of packet recirculation is avoided. Table 11 contains instructions executed on the packet preprocessor for parsing and preprocessing IPv4 header. Registers in this table follow the numbering defined in Table 9.

Table 11. Instructions executed on the packet preprocessor during arrival of IPv4 header

Time	Packet Parser	Packet Preprocessor	Comments
t_0	-		
t_1	-		
t_2	-		
t_3	R0 <- (Ver, IHL) R32 <- (Ver, IHL, DSCP, ECN) R64 <- Total Length R128 <- Identification R160 <- (Flags, Fragment Offset)	R192 <- R32 + R64	1st and 2nd words of IPv4 header written to PHV
t_4	-	R192 <- R192 + R128	
t_5	-	R192 <- R192 + R160	
t_6	-	R194 <- R0 AND 0x0000000F	Retrieving IHL
t_7	-	R195 <- SHR4(R0)	Obtaining the value of Ver
t_8	-	r0 <- R194 >= 0x5	
t_9	-	r0 <- R195 == 4	
t_{10}	-		
t_{11}	R1 <- TTL R8 <- Protocol R33 <- (TTL, Protocol) R65 <- Header Checksum R129 <- upper_halfword(Source IP)	R192 <- R192 + R33	3rd and 4th words of IPv4 header written to PHV

	R161 <- lower_halfword(Source IP) R176 <- Source IP		
t_{12}	-	R192 <- R192 + R65	
t_{13}	-	R192 <- R192 + R129	
t_{14}	-	R192 <- R192 + R161	
t_{15}	-	r0 <- R1 < 1	Checking TTL
t_{16}	-		
t_{17}	-		
t_{18}	-		
t_{19}	R34 <- upper_halfword(Destination IP) R66 <- lower_halfword(Destination IP) R80 <- Destination IP	R192 <- R192 + R34	5th word of IPv4 header written to PHV
t_{20}	-	R192 <- R192 + R36	
t_{21}	-	R193 <- SHR16 R192	
t_{22}	-	R192 <- R192 + R193	
t_{23}	-	R192 <- NOT (R192)	
t_{24}	-	r0 <- R192 == R65	Comparing the calculated checksum with the value of Checksum field

4.3.2 Fragmentation of IPv4 Packets

Packet fragmentation is required when the size of a packet is greater than the MTU of the path to which it must be forwarded. In this case, the packet must be fragmented into multiple packets such that the size of each fragment is no more than the value of MTU. Figure 11 contains the algorithm for fragmenting IPv4 packets.

```
if(Total_Length > MTU)
{
        if(DF == true)
        {
                send_icmp_destination_unreachable(); //Type = 3, Code = 4
        }
        else
        {
                payload_size = Total_Length - (IHL*4);
                fragment_header = original_header;
                NFB = (MTU - (IHL * 4))/8;
                Append(NFB*8);
                Fragment_offset = 0;
                MF = true;
                Total_Length = (IHL*4) + (NFB*8);
                send();
                remaining_payload = payload_size - (NFB*8);
                //And now for producing the other fragments
                selectively_choose_options(byte_size_of_selected_options);
                do
                {
                        fragment_header_byte_size = byte_size_of_selected_options + 20;
                        Fragment_offset = Fragment_offset + NFB;
                        if(fragment_header_byte_size + remaining_payload > MTU)
                        {
                                MF = True;
                                NFB = (MTU - fragment_header_byte_size)/8;
                                append(NFB*8);
                                fragment_payload_size = NFB*8;
                                Total_Length = fragment_header_byte_size + (NFB*8);
                        }
                        else
                        {
                                MF = false;
                                fragment_payload_size = remaining_payload;
                                append(remaining_payload);
                                Total_Length = fragment_header_byte_size + remaining_payload;
                        }
                        send();
                        remaining_payload = remaining_payload - fragment_payload_size;
                }while(remaining_payload > 0);
        }
}
```

Figure 11. Procedure for fragmenting IPv4 packets

The procedure for fragmenting IPv4 packets is straightforward. However, the presence of IPv4 header options can make it a bit complicated because some options must be included in all fragments, some must be included only in the first fragment and some may be subject to removal in case of fragmentation. When fragmenting an IPv4 packet that contains options, for each option, it must be decided if the option

should be included in each of the fragments or not. Furthermore, the size of each of the options has an impact on the amount of payload data that can be included in the fragment. In the ingress pipeline, when a matching entry is found for the destination address of an IPv4 packet, the MTU of the corresponding path is also retrieved. Then, contents of the PHV are written to the buffer. Once the packet is scheduled for transmission, the egress parser retrieves the packet from the buffer and starts parsing. The most efficient way to handle fragmentation of IPv4 packets whose header contains options is to make the decision for each option during egress parsing. Furthermore, the packet preprocessor is the best computational component for maintaining a loop in which the required number of fragments are created. Assuming that the MTU is 576 bytes, consider the IPv4 datagram in Figure 12. This 600-byte datagram contains Loose Source and Record Route. The MTU is 576 bytes, so the packet must be fragmented. The accompanying option must be included in all fragments. This packet will be fragmented into two packets. Table 12 contains instructions that are executed on the packet preprocessor for fragmenting the IPv4 packet in Figure 12. Registers in this table follow the numbering defined in Table 9.

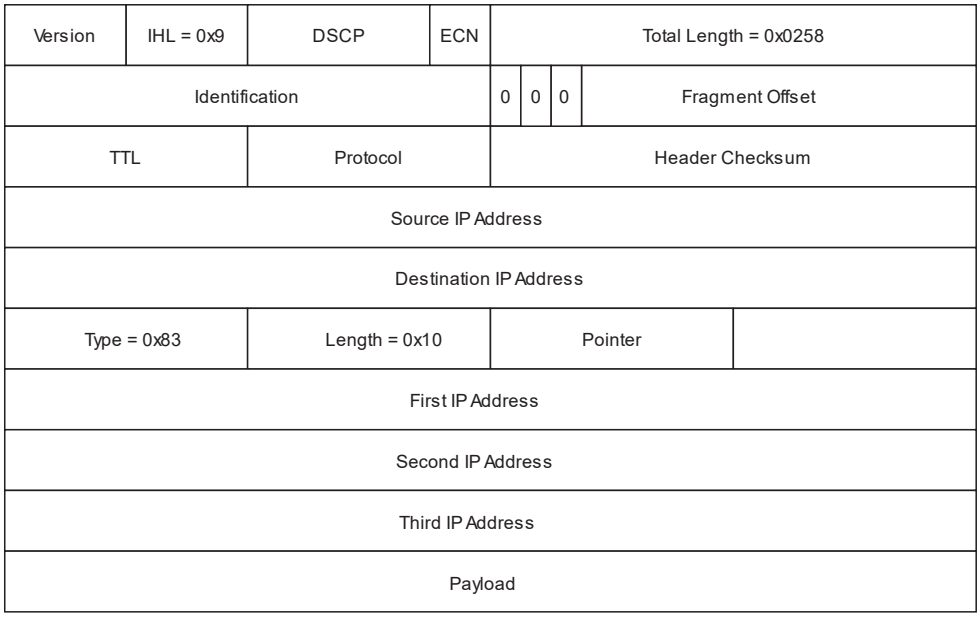

Figure 12. IPv4 header containing option

Table 12. Instructions executed by the egress parser

Time	Packet Parser	Packet Preprocessor	Comments
t_0	-		
t_1	-		
t_2	-		
t_3	R0 <- (Ver, IHL) R32 <- (Ver, IHL, DSCP, ECN) R64 <- Total Length R128 <- Identification R160 <- (Flags, Fragment Offset)	R192 <- R0 AND 0x0000000F	1st and 2nd words of IPv4 header written to PHV. The value of IHL is obtained.
t_4	R1 <- TTL R8 <- Protocol R33 <- (TTL, Protocol) R65 <- Header Checksum R129 <- upper_halfword(Source IP) R161 <- lower_halfword(Source IP) R176 <- Source IP	R192 <- SHL2(R192)	3rd and 4th words of IPv4 header written to PHV. The value of IHL is multiplied by 4.
t_5	R34 <- upper_halfword(Destination IP) R66 <- lower_halfword(Destination IP) R80 <- Destination IP	R203 <- 0	5th word of IPv4 header written to PHV. R203 which is designated to contain byte size of all must-copy options is initialized
t_6	R2 <- Type R9 <- Length R16 <- Pointer R35 <- upper_halfword(option word) R67 <- lower_halfword(option word) R81 <- option word	r0 <- R2(7)	1st option word written to PHV. Checking the highest bit of Type field.
t_7	R82 <- option word	(r0) R203 <- R203 + R9	2nd option word written to PHV Conditionally adding the size of current option to size of must-copy options.
t_8	R83 <- option word	(r0) R193+0 <- R81+0	3rd option word written to PHV Conditionally copying current option to the space reserved for must-copy options.
t_9	R84 <- option word	(r0) R193+1 <- R82+1	4th option word written to PHV Conditionally copying current option to the space reserved for must-copy options.
t_{10}		(r0) R193+2 <- R83+2	Conditionally copying current option to the space reserved for must-copy options.
t_{11}		(r0) R193+3 <- R84+3	Conditionally copying current option to the space reserved for must-copy options.
t_{12}			

t_{13}	R223 <- MTU	R64 > R223	Total Length > MTU
t_{14}		-	
t_{15}		-	
t_{16}		R160(14)	Evaluating DF
t_{17}		-	
t_{18}		-	
t_{19}		R204 <- R64 – R192	Calculating payload size
t_{20}		R205 <- R223 – R192	
t_{21}		R205 <- SHR3(R205)	Number of fragment blocks
t_{22}		R206 <- 0xABCDEFAB	Code representing first fragment
t_{23}		R207 <- SHL3(R205)	Number of payload bytes to be included in the fragment
t_{24}		Submit to egress	Submit to the egress pipeline
t_{25}		R208 <- R204 – R207	Calculating the size of remaining payload
t_{26}		R209 <- R203 + 0x00000014	Calculating the size of fragment header
t_{27}		R160 <- R160 + R205	Updating fragment offset
t_{28}		R210 <- R208 + R209	Adding byte size of remaining payload and fragment's header size
t_{29}		R210 > R223	Checking if the sum of size of header and remaining payload exceeds MTU
t_{30}		R206 <- 0xABCDEFAA	Code representing last fragment
t_{31}		R211 <- R208	Fragment's payload size
t_{32}		Submit to egress	Submit to the egress pipeline
t_{33}		R208 <- R208 – R211	Updating remaining payload size
t_{34}		R208 > 0	Checking if there is payload remaining

Those options that must be included on all fragments are written to registers R193 to R202 of the register space reserved for the packet preprocessor. In addition, the total size of these options is also stored so that the amount of payload to be appended to each fragment can be determined. As each fragment is sent, a code is written to a designated PHV entry. There are distinct codes for a non-fragment packet, first fragment, fragments after the first fragment and before the last one and the last fragment. The code is looked up in the egress pipeline and the instructions corresponding to each one of them is executed, as each of them requires different processing. For instance, for a packet that does not require fragmentation, the options written to registers R193-R202 are ignored. This is also true for the first fragment, as its header is exactly the same as that of the original packet.

4.4 Implementation Results

Table 13 outlines the total area and power for the components required in each packet preprocessor instance. The ASIC technology used is 28nm FD-SOI. The operating conditions are (SS, 0.9V, 125˚C). The synthesis tool under use is Synopsys Design Compiler J-2014.09-SP4. Timing constraints have been verified for operating frequency of 1.19 GHz.

Table 13. Area and power dissipation of the components of a single packet preprocessor

Component	Area (μm^2)	Power dissipation (mW)
Instruction decode, operand retrieval, and operand forwarding	23161	33.5
ALU	1044	3.5
Program Control	448	5.9
Instruction Memory (1K × 32b)	15717.60	3.69

4.4.1 Discussion of results

This architecture enables processing of packets as they arrive. Based on the reasoning in chapter 3, 8 bytes are read from IPB every 8 cycles. Instead of header segments sitting idle in the PHV until the rest of the header fields are written, processing starts already at this point. Although RMT architecture contains 7168 ALUs, some actions such as checksum calculation must be mapped to ALUs across different MAUs. For

such actions, presence of 224 ALUs in a single MAU is of little benefit. The proposed architecture reduces the chance of need for recirculation if the chain of MAUs is not sufficient for a given action. Another issue that this architecture solves is that RMT has match resources coupled with action resources. Each MAU contains 32K ternary entries and 106K exact match entries. It is possible to look up speculatively if the outcome of the action determines whether to match or not. However, if both outcomes of the action each require matching on the same tables but using different keys, speculation is not beneficial. This is problematic for use cases in which match resources from the whole chip must be combined. The main purpose of the architecture proposed in this chapter is to perform the required preprocessing so that the issues discussed here do not hinder throughput.

In addition to the preprocessing IPv4 packets, a similar role can be taken for IPv6 packets. For instance, it can check the value of *Hop Limit* field and if necessary, discard the packet or generate a message to the original sender of the packet. The actual list of use cases is limited only by the number and complexity of available and upcoming network protocols. The programmable nature of this architecture does not tie it to any specific set of protocols.

Given that a single packet preprocessor sustains 10 Gbps throughput, the total area and power dissipation for 640 Gbps packet preprocessing is 2.58 mm^2 and 2.98 W respectively. These values are pessimistic because the instruction memory has been replicated per packet preprocessor instance due to unavailability of multiported SRAMs. For instance, in the presence of two-ported SRAM, the memory cells will be shared by two packet preprocessors. Hence the resulting area will be less than the value provided here. In order to interpret the area and power values properly, it should be considered that commercial switch chips are somewhere between 300 to 700 mm^2 in area and 150 to 350 W in power [69]. Therefore, the total area and power of the extra logic required for 640 Gbps packet preprocessing is negligible. Focusing on IPv4 traffic, the exact gain in throughput depends on the percentage of packets having a parameter problem or requiring fragmentation. For the latter, the proposed architecture acts as an enabler because in the absence of a processor-based component for calculating the fragmentation-specific parameters and sending the required number of fragments to the egress pipeline, fragmentation is not possible and the packet processing architecture simply has to drop the packets requiring fragmentation.

5 EXPLORING CROSSBAR ALTERNATIVES

Crossbars are used extensively in programmable packet processing hardware for providing flexibility. Two primary use cases for crossbars are selecting the header fields for forming the search key and for selecting the input to ALUs. Since a programmable data plane allows any field to be used as the basis for forming the search key or for being the input to a given ALU, crossbars are one of the enabling components and main contributors to the area. In this chapter, further details are provided on the crossbars in RMT and the alternatives are explored for better area efficiency. The content of this chapter is based on P_V.

5.1 Crossbars in RMT

In each of the 32 MAUs within RMT, two 640-bit search keys are generated from the 4096-bit PHV. One of the search keys is meant for TCAMs and the other for SRAM-based exact match tables. In [26], it is mentioned that each bit of the search key is driven by a 224-to-1 multiplexer, which is made of a binary tree of and-or-invert (AOI22) gates. The provided area for this gate is 0.65 μm^2. The total area for match key crossbars across the whole chip is 6 mm^2.

The fact that a 224-to-1 multiplexer has been used for every bit of the search key implies that there are certain constraints for selecting the input for match tables. For instance, the smallest unit for selection from the PHV is an entry within the PHV. Furthermore, there are constraints for placing fields of different width within the search key. The starting position for bytes, halfwords, and words in the resulting search key has a bit index, which is an integer multiple of 8, 16, and 32 respectively. Not all of the multiplexers require distinct select inputs. Some of the multiplexers can share the select lines. The total number of distinct select lines within a MAU is 1280 bits.

Action crossbars provide the operand inputs for each ALU in a given MAU. In RMT architecture, there are 8-bit, 16-bit and 32-bit ALUs. The action engines take input from the PHV and action memory. The first input to a given action engine is from the PHV. The second input is from the PHV or action memory. Smaller units

can be combined so that processing can be performed on a wider unit. For instance, two 8-bit units can be combined into a 16-bit operand. It is unclear whether this combination is performed by the action crossbars or by the action engines. In either case, it is obvious that 16-bit and 32-bit ALUs receive smaller units as input as well as PHV entries of identical width. In [26], it is mentioned that the area calculation for the action crossbar is similar to that of match crossbars. Since there is an ALU per PHV entry and that each ALU requires two operand inputs, two 4096-bit units must be generated out of PHV. This results in the total area of action crossbars to be over 38 mm^2 which is equivalent to the area of 220 TCAM blocks.

5.2 Crossbar alternatives

It is important to explore crossbar alternatives because crossbars occupy a noticeable area and contain considerable amount of wires. In addition, the value for select lines must be stored in wide registers and/or memory blocks. This in turn increases area. This is specifically noticeable in an architecture such as RMT in which wide search keys are generated and there are numerous action engines requiring input.

5.2.1 Alternative Match Crossbar

In this scheme, the entries of each of the two 640-bit search keys are determined on a 32-bit basis. This can be referred to as word-level selection of match keys. For every 32-bit unit of the search key, it is possible to select the following combinations from the PHV:

- Four 8-bit units
- Two 8-bit units and a 16-bit unit
- Two 16-bit units
- One 32-bit unit

For the combination in which there are two 8-bit units and a 16-bit unit, two valid arrangements are possible depending on whether the 16-bit unit comes before or after the two bytes. In order to select match key fields on a 32-bit basis, the multiplexers must be organized in two levels as illustrated in Figure 13. The first level contains multiplexers for selecting fields and the second level has one multiplexer for organizing the selected fields according to the combinations above. In this

alternative, the constraints mentioned in section 5.1 are satisfied. This scheme requires 1080 bits of select lines in each stage.

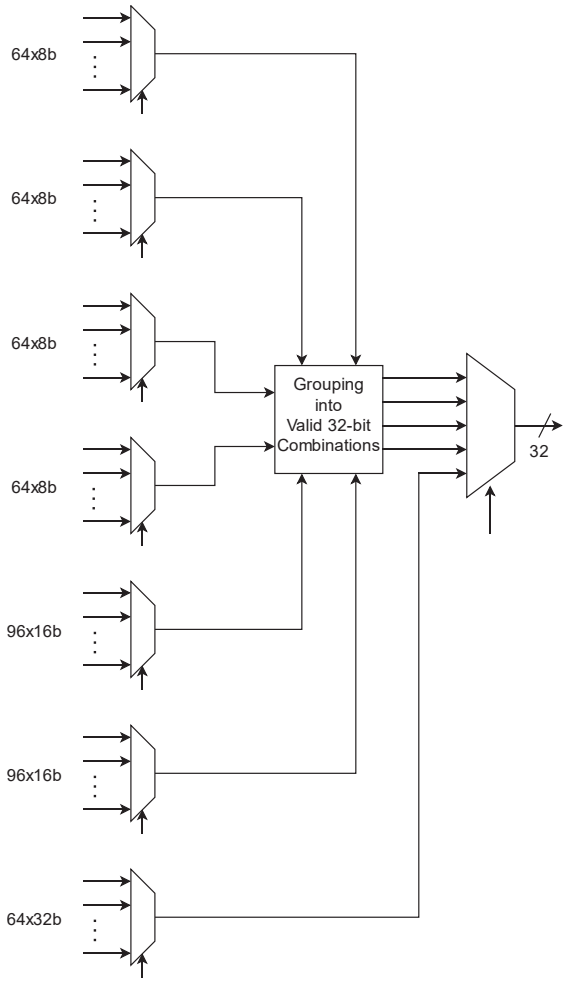

Figure 13. Alternative match crossbar

5.2.2 Alternative Action Crossbars

Action crossbars select inputs to ALUs. For action crossbars the issue of the number of select lines is of more importance because they affect the width of instructions. In RMT, the instruction memory has 32 entries. Furthermore, there are 224 ALUs per stage, each requiring 2 action input selectors.

5.2.2.1 Zero-extending Smaller Units

In this system of multiplexers, as shown in Figure 14, the input to 8-bit ALUs is any of the 8-bit entries of the PHV. For 16-bit ALUs, each input can be any of the 16-bit entries or any of the zero extended 8-bit entries. In a similar manner, for 32-bit ALUs, each input can be any of the 32-bit entries or any zero-extended 16-bit entry. This scheme requires 3328 select lines per stage. Under this scheme, the actual merging takes place by the ALUs. In addition, the parser presented in chapter 3 can be programmed to write an arrived 32-bit word as four 8-bit units, two 16-bit units and a 32-bit unit at the same time in order to tailor the content of headers to the way required by corresponding packet processing functions. As a result, the packet parser can also perform combination of smaller fields that happen to be adjacent in the header. This is illustrated in Figure 15.

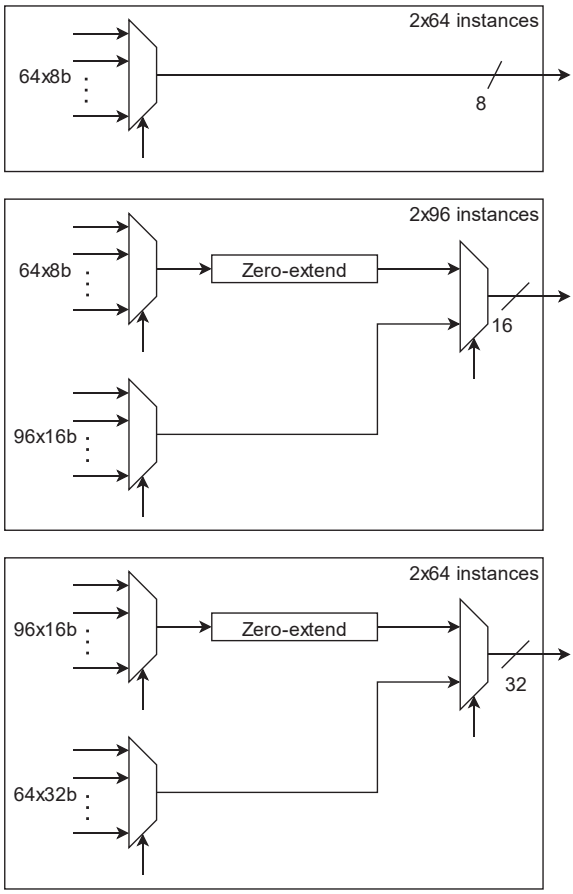

Figure 14. Action crossbar with zero-extension of smaller units

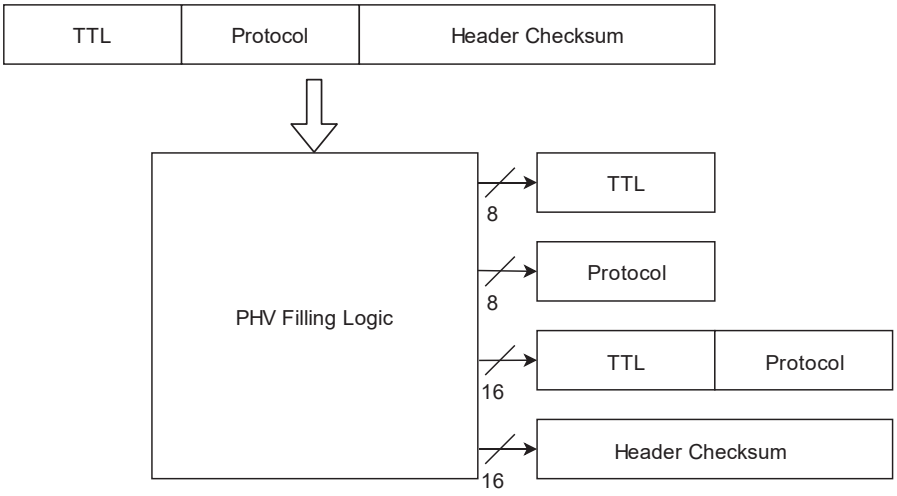

Figure 15. Operation of PHV filling logic when writing the third word of IPv4 header to PHV

5.2.2.2 Combining Smaller Units

Using this crossbar architecture, the input to 8-bit ALUs is any of the 8-bit entries of the PHV. For 16-bit ALUs, each input can be any of the 16-bit entries or any two 8-bit entries combined together to form a 16-bit unit. In a similar manner, for 32-bit ALUs, each input can be any of the 32-bit entries or any two 16-bit entries combined together into a 32-bit unit. This scheme requires 5184 select lines per stage. This action input selection scheme is illustrated in Figure 16.

5.3 Reducing Action Crossbars' Area

The area of crossbars can be reduced significantly by using smaller crossbar variants. In other words, by limiting the number of inputs to the multiplexers, considerable savings can be made. An ALU performing some header field or state modification does not require to access the entire PHV. Some packet processing functions such as IPv4 checksum calculation, the entire header needs to be accessed. The largest possible IPv4 header contains 15 32-bit words, which is a tiny fraction of the entire PHV. The fact that each of the 224 ALUs can read from the entire PHV for each of its inputs is overkill. It is sufficient that each action crossbar reads from a smaller number of PHV entries. The number and location of fields that can be accessed is a design choice.

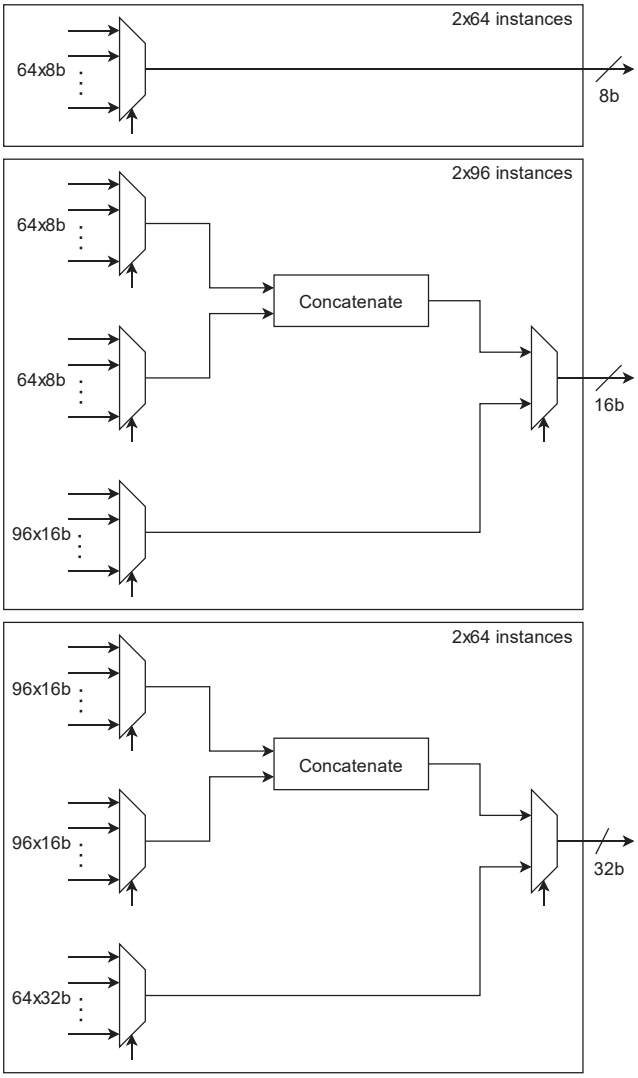

Figure 16. Action crossbar combining smaller units

In order to limit the number of inputs to the action crossbars, the PHV is logically divided into four equal segments. This means that the number of 8-, 16-, and 32-bit entries within each logical segment is 16, 24, and 16, respectively. The action engines within a segment can read from all the PHV entries pertaining to the segment in question. On the other hand, reading from fields located in other segments is subject to limits. Let's consider the action crossbars that zero-extend smaller processing units. In each segment, the input to 8-bit ALUs is any of the 8-bit entries within the segment. For each of the two inputs of 16-, and 32-bit ALUs each, there are 40 input

options but since for 40 options, 6 bits of select input are required, the unused inputs to multiplexers are used to accommodate entries within other segments. Figure 17 illustrates the structure required per segment. For the 16-bit multiplexers marked 32×16b, there are 24 inputs coming from 16-bit PHV entries within the segment. The other 8 inputs come from other segments. With this structure, up to 160 fields from other segments can be read.

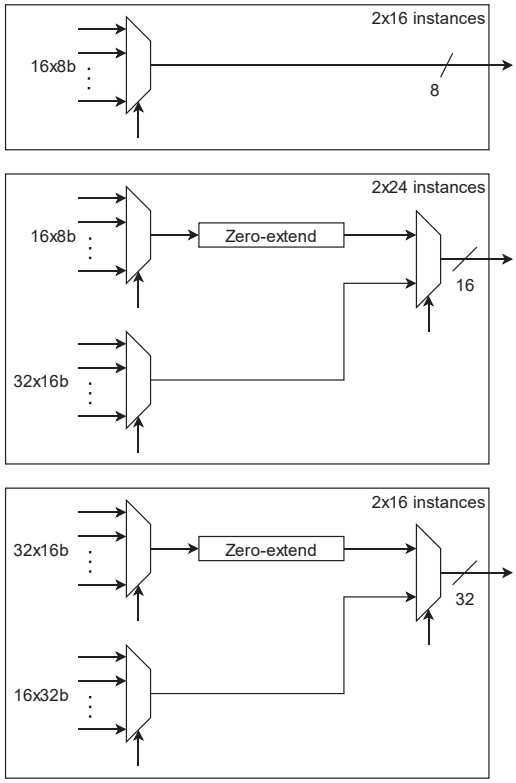

Figure 17. Lightweight action crossbar with zero-extension of smaller units

5.4 Implementation results

In this section, the implementation results are provided in terms of gate area of crossbars and the area of memory cells required for storing the value of select bits. These values are provided in Tables 14 and 15 for match and action crossbar variants respectively. The ASIC technology, backend tool, operating frequency and operating conditions are the same as in previous chapters. For action crossbars, the area of

storage required for select lines is considerably more because action crossbars have more select lines and the instruction memory has 32 entries.

Table 14. Per stage area requirement of match crossbar variants

Crossbar variant	Crossbar area (mm^2)	Memory area (mm^2)
Original RMT	0.187	0.007
Word-level selection	0.174	0.006

Table 15. Per stage area requirement of action crossbar variants

Crossbar variant	Original size		Lightweight crossbars	
	Crossbar area (mm^2)	Memory area (mm^2)	Crossbar area (mm^2)	Memory area (mm^2)
Combination of smaller processing units	0.730	0.967	0.203	0.685
Zero-extension of smaller processing units	0.553	0.625	0.148	0.457

5.4.1 Discussion of results

When using lightweight crossbars, the most important question is whether the use of smaller multiplexers can limit the programmability and performance of the architecture. Let's consider the lightweight variant of action crossbars that combine smaller processing units. Inside each logical segment, there are 112 16-bit multiplexers, each having 8 unused inputs that can be used for reading entries residing in other logical segments. This means that it is possible to read 896 16-bit units that are not resident in a given segment. The total number of 16-bit entries across the whole PHV is 96. Therefore, even in case of segmentation of PHV and limited access to cross-segment fields, the functionality can be maintained by efficient use of resources. The only limitation is that when a field in another segment is required as input, the ALU that has access must be used for the required operation.

As seen from the Tables 14 and 15, for match crossbars, word-level selection of match fields is slightly more area efficient. For action crossbars, zero-extending crossbars occupy smaller area. The lightweight variant brings further savings in area. In chapter 7, a packet processing pipeline whose all ALUs are 32 bits in width will

be presented. For this kind of ALU, the total number of input possibilities is more than each of the ALUs in RMT. Based on the insight gained in this chapter, the required optimizations will be applied to reduce the area.

6 TOWARDS TERABIT-LEVEL PACKET PARSING

As line rates increase, the time window during which a new packet arrives shrinks. For instance, in 400 Gigabit Ethernet (GbE), minimum-sized packets arrive every 1.67 ns. In a digital system operating at frequency of 1.0 GHz, this is hardly equivalent to 2 cycles. Even with minimum-sized Ethernet frames, it is possible for the frame to contain multiple headers. Parsing of multiple headers in a 2-cycle time window is not trivial unless the sequence of headers is known in advance. In this chapter, the details of an architecture for Terabit-level packet parsing will be presented. The content of this chapter is based on P_{VI}.

6.1 The Building Block for Terabit-level Packet Parsing

The packet parser presented in chapter 3 uses a single packet parsing entity to parse all the headers within a packet. It uses program control to use the same functional units for parsing of all headers within a packet. Line rate is sustained because the packets arrive through 10 Gbps ports. The packet parser in this chapter, however, uses multiple parsing entities, each for one of the headers in the packet. The parsing entity is called header parser and it is the building block for Terabit-level packet parsing. These building blocks are put in series to form a pipelined packet parser. The first header parser parses the first header in the incoming packet. The second header parser parses the second header and so on. This is the requirement for Terabit-level switches in which each port operates at hundreds of Gbps.

The header parsers together form a pipeline. In addition, each header parser is internally pipelined. The header data read from the buffer of incoming packets traverses the pipeline of header parsers. Figure 18 illustrates the internals of a header parser. The inputs to each header parser are the Header ID, header data and starting offset. The Header ID is a 4-bit identifier of the header that must be parsed. This ID is only of significance inside the architecture. It is used to retrieve the control signals for the functional units within the header parser. The control signals for parsing a given header are collectively referred to as Parse Control Word (PaCW). Each header parser determines the next header, calculates the size of current header

in order to provide the starting offset of next header and extracts fields of current header into the PHV. Output of each header parser is the input to the next header parser. For the first header, the header ID and starting offset are already set. Each header is parsed over 5 cycles. These cycles are outlined in the table 16.

Table 16. Header parsing stages

Cycle	Operations
1	PaCW is retrieved using the ID of the incoming packet
2	Header fields containing indication of next header and header size are extracted.
3	Extracted next header value is compared in parallel with 16 next header values corresponding to current header. In parallel, the extracted header size field goes through modification.
4	The matching entry for next header is selected. The modified header size value goes through another round of modification.
5	The size of header is now available. Up to 16 words within the header can be extracted in parallel for writing to PHV.

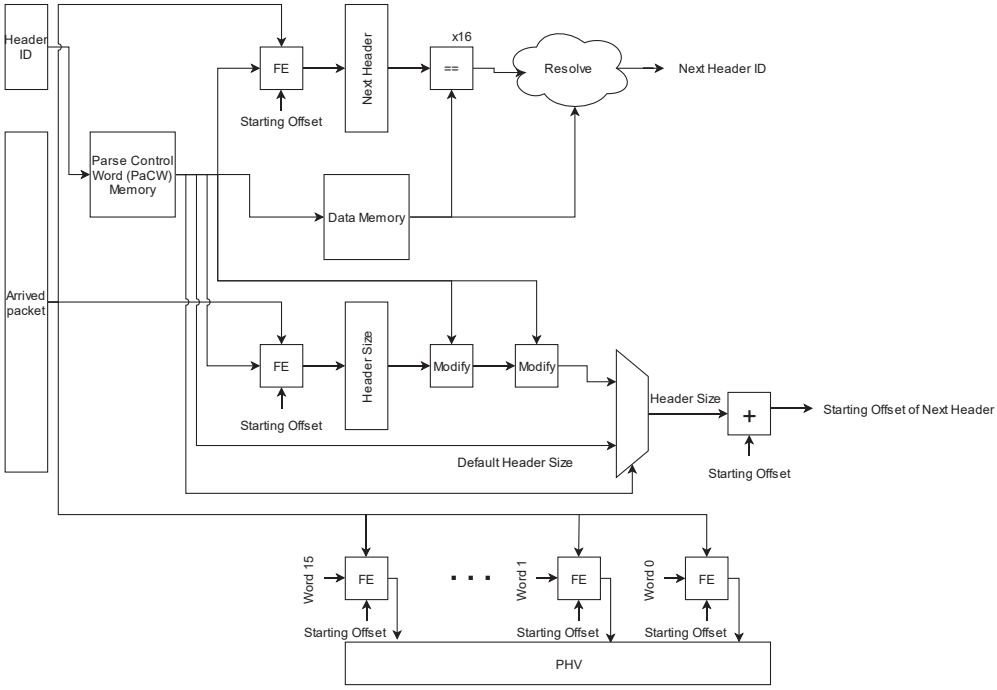

Figure 18. Internals of Header Parser [PVI]

The main difference with the parser in chapter 3 is that the header parser is a pipelined architecture rather than a processor. This should be distinguished from a processor with an instruction pipeline. In addition to this, the header parser's input

is considerably wider. It requests 64 bytes from the buffer of incoming packets. This means that an entire minimum-sized packet can be retrieved at once. Each header parser receives the starting offset from its previous header parser. It calculates the size of the header being parsed and adds it to the starting offset it received to provide the next header parser with the starting offset of the header that must be parsed.

The fact that each header parser parses only one of the headers in the packet does not mean that the header parser has the capability to parse only one header. Rather, it can be programmed to parse 16 distinct headers. Consider the parse graph illustrated in Figure 19. If a packet parser made of header parsers is deployed in an environment in which header sequences within the packets are based on the parse graph in Figure 19, the first header parser parses Ethernet header. The second header parser must be programmed to parse IPv4, IPv6, VLAN, or MPLS. Every header parser finds out what the next header is and signals it to the next header parser. The third header parser must parse IPv4, IPv6, MPLS, or TCP. Interestingly, the second header parser also could parse IPv4, IPv6 and MPLS but for the third header only the third parser is used.

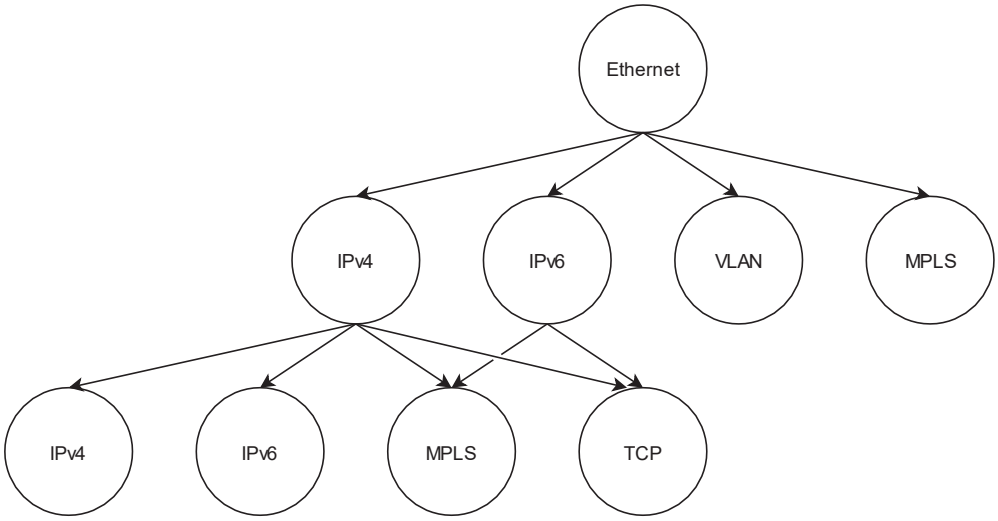

Figure 19. Parse graph with three levels [PVI]

6.2 Using the Header Parsers to Build a Packet Parser

By placing 8 header parser instances in a pipelined organization, a packet parser that can parse packets having up to 8 headers is constructed. For complex headers, a

header parser may not be able to completely parse the header. In such a case, one header parser partially parses the header and the next header parser performs the rest of the parsing of the header in question. The operating frequency is 1.19 GHz. This means that a minimum-sized Ethernet frame is fetched every 0.84 nanoseconds. Therefore, one single packet parser sustains 800 Gbps throughput. For achieving higher throughputs, either the operating frequency must be increased, or the packet parser instances must be replicated. The latter is chosen because the packet parsers are very efficient in area. For a 6.4 Tbps switch, 8 packet parsers are required.

6.3 Implementation Results

Table 17 contains the area and power breakdown of the constituent components of a single header parser. The ASIC technology, backend tool, operating frequency and operating conditions are the same as in previous chapters. Based on the values of Table 17, Table 18 contains the total area and power requirements for 6.4 Tbps packet parsing. These parser instances have total area of 3.61 mm^2 and total power dissipation value of 9.5 W.

Table 17. Area and power dissipation of the components of a header parser (adjusted from P_{VI})

Component	Total area (μm^2)	Total power (mW)
PaCW and Parameter Memories	30 K	52
Field extractors and manipulators	15.9 K	21.64
Comparators and resolving logic	1.1 K	0.96
Total	47.0 K	74.6

Table 18. Area and power dissipation of components required for 6.4 Tbps packet parsing (adjusted from P_{VI})

Component	Number of instances	Total area (mm^2)	Total power (W)
Header Parsers	64	3	4.7
PHV	8	0.61	4.8
Total	-	3.61	9.5

6.3.1 Discussion of implementation results

In order to sustain 6.4 Tbps throughput, 8 pipelined packet parser instances are needed. For this target throughput value, if the 80 Gbps parser presented in chapter 3 is to be used, 80 instances will be needed. If the 40 Gbps parsers used in RMT architecture are to be used, 160 parsers would be needed. There is, however, a fundamental difference between the parser presented in this chapter when compared with the 80 Gbps parser and the 40 Gbps parser. The difference is that the parser in this chapter can parse packets coming from 800 Gbps ports. The parser in chapter 3 and RMT are designed for 10 Gbps ports. 20 instances of RMT parser can sustain aggregate throughput from 80 ports of 10 Gbps which in total is 800 Gbps but they cannot parse packets arriving through a single 800 Gbps port. Similarly, the 80 Gbps parser cannot parse packets coming from an 800 Gbps port, no matter how many parsers have been instantiated. They can however, sustain aggregate 800 Gbps throughput. Despite this, let's compare the area of a 6.4 Tbps packet parser presented in this chapter with the area of 80 instances of 80 Gbps packet parser and 160 instances of RMT parser. The corresponding gate counts are 11 M, 46 M and 56 M respectively. Therefore, the designed parser has 76% and 80% less area compared to parsers in chapter 3 and the RMT parsers instantiated for 6.4 Tbps. It is worth mentioning that each header parser has its own memory instances. No sharing of memory has been used.

At port speeds above 100 Gbps, it is challenging for processor-based solutions to keep pace with the rate of packet arrival. Each 800 Gbps packet parser pipeline of this chapter can sustain an unlimited stream of minimum-sized packets. If a processor-based architecture were to be used for an 800 Gbps port through which an unlimited stream of minimum-sized packets arrives, the requirements would be hardware support for an unlimited number of threads as well as speculative execution engines for each thread. The reason why speculative execution is required is that an entire packet arrives every clock cycle, i.e. every 0.84 ns, while the operations required for determining the next header require multiple cycles. Support for infinite number of threads means infinite amount of hardware, which is not feasible. The pipelined architecture with 8 stages sustains 800 Gbps throughput with far less complexity. Therefore, the fundamental research question of whether to use processor or pipeline has been answered.

7 A FLEXIBLE PACKET PROCESSING PIPELINE

In this chapter the motivation and architectural details will be provided for a low-area packet processing pipeline with enhanced level of flexibility and functionality. The content of this chapter is based on P_{VI}.

7.1 Motivation

Since programmable data plane is a new research area, there have not been many hardware architectures fulfilling the criteria of programmability and performance. The dominant architecture is RMT. It sustains 640 Gbps throughput. A few years after RMT, a new architecture with modifications to RMT appeared. The new architecture was called disaggregated RMT (dRMT). Both architectures were discussed in chapter 2. dRMT is a run-to-completion architecture while RMT is a pipeline. For dRMT, the achievable throughput using all processor instances is at most 1 packet per cycle. Considering its 1.19 GHz operating frequency and 64-byte packets separated by 20-byte unit inserted by the physical layer, the maximum sustained throughput by dRMT is 800 Gbps. As discussed in chapter 6, a pipelined architecture is more suited to the nature of high-throughput packet parsing. The same applies to packet processing as well. Deep pipelines can accommodate extra processing required for some packets whereas in processor-based systems, the issue of extra processing can lengthen the interval at which a new packet must be accepted. The solution to this is simultaneous multithreading but as discussed in chapter 6, a pipelined architecture has far less complexity. Another benefit of pipelined architectures is that the instruction memory is divided into pipeline stages. In other words, each pipeline stage has a portion of the packet processing program. For run-to-completion architectures, each processor instance must have the entire program. As a result, the number of processor instances and/or functional units within them must be subject to limits. Otherwise, the area increase can violate the constraints.

The motivation behind this work is achieving an area-efficient architecture in order to minimize the overall area of multiple pipeline instances required for terabit-level packet processing. In addition, the intention is to overcome RMT's

shortcomings that unnecessarily increase the area, limit the supported workload and use memory resources inefficiently. These shortcomings are listed below:

- Lack of action depth
- Extensive use of matching for program control
- Limited field referencing
- High cost of table combination

7.2 A New Architecture

The underlying component of the new packet processing pipeline is a packet processing stage. Table 19 outlines the functional units available within each stage. Figure 20 illustrates the components within each stage. By having only one TCAM and one exact match table in each stage, the design is greatly simplified. The size of each table and their total number of instances are exactly the same as that of RMT architecture. This means that this architecture is a 512-stage pipeline.

Table 19. Components in each stage of the proposed packet processing pipeline

Functional unit	Number of instances	Purpose
Field Extractors (FE)	18	Extract fields for modification of fields and state
Field- and state-modifier (ALU)	8	ALUs for modifying content of header fields and state
Search key generator (Match crossbar)	2	Crossbars for selecting search key components for exact and ternary matching
TCAM (2K×40)	1	Performing ternary matching
Hash table (1K×64)	4	4-way hash table for exact matching
Associated memories (1K×32)	12	Data associated with match entries in exact and ternary match tables

7.2.1 Program Control

A 10-bit tag is assigned to the packet by the packet parser based on the sequence of headers present in the packet. This tag precisely describes the processing that a packet requires. For instance, it can refer to an IPv4 packet that has TTL value of zero. As the PHV traverses the pipeline, this tag is used to retrieve the instructions for modifying header fields and state. However, this tag is not directly used because its direct use requires an instruction memory with 1K entries at each stage.

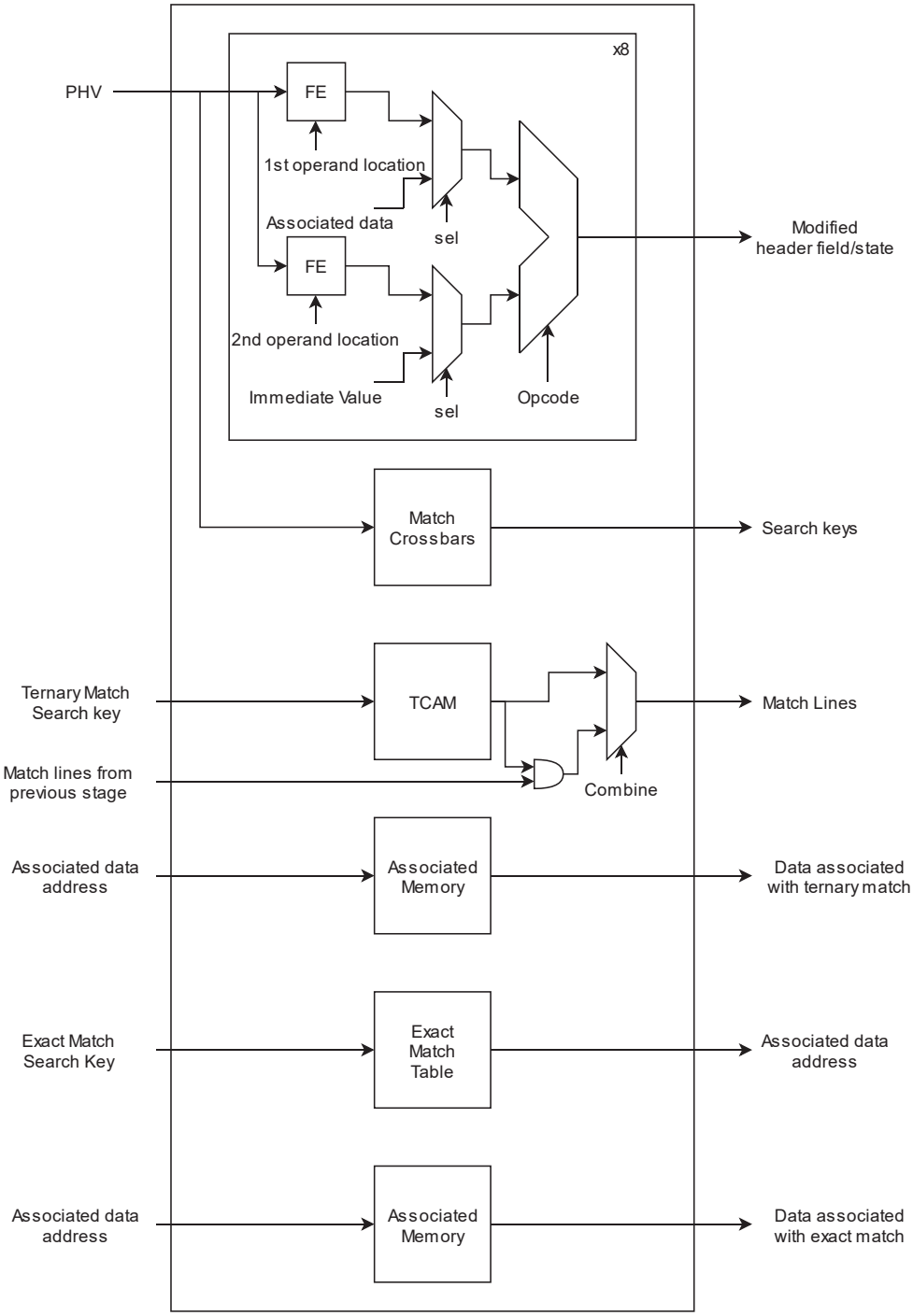

Figure 20. Packet processing stage [P_{VI}]

In addition to large area, if a VLIW instruction differs only in the opcode of one of the functional units, a new instruction memory entry is required, which is inefficient in use of memory. Instead, at each stage the 10-bit tag is used as an index into a 1K × 64-bit memory. Each entry of this memory contains instruction pointers for each of the functional units present in a stage. Once the instruction pointer for each functional unit is available, they are used to retrieve the instruction from a 32-entry memory. Therefore, at each stage, 32^8 combinations are possible while in the naïve approach 1K entries are not sufficient for accommodating 32^8 combinations.

The fact that the tag is used for retrieving the instructions at each stage, allows custom action depth. At each stage, the instructions related to the stage in question are executed. A sequence of instructions is divided into a number of stages. The tag can be changed throughout the pipeline. In addition to the 8 ALUs, there is also a condition evaluator that is present in each stage. It can compare the value of header fields or extract bits. When checking the value of TTL, this unit is used instead of using TTL as a search key for matching.

7.2.2 Combining Tables

Each of the match tables in the architecture is referred to as a physical table. On the other hand, a table required for a specific purpose is referred to as a logical table. For instance, it is possible to combine all ternary match tables for storing over 1 million IPv4 prefixes. In this setting, the logical table with 1048576 entries is constructed using 512 physical tables each having 2K entries. If the required logical table is wider than a single physical table, each of the visited tables receives part of the search key that is designated for it. If the required logical table is deeper than a single physical table but has the same width, each of the visited tables receives the same search key. The first table that returns a match terminates the search. If the required logical table is both wider and deeper than a single physical table, both scenarios above are combined.

When a large number of tables need to be combined, this way of combining tables results in high latency. For instance, in the case of logical table combining all TCAMs, there is a 512-cycle latency for visiting all tables. In order to reduce the latency in such scenarios, the designed architecture has an area-efficient mechanism for combining the tables. In order to elaborate this architecture, some definitions must be made. Pipeline stages are numbered 0 to 511. This 512-stage pipeline is made of 32 smaller pipelines with 16 stages, each of which is called a PIPE16. The starting

index of each PIPE16 is an integer multiple of 16. The PIPE16 instances can be combined in a binary tree manner as can be seen in Figure 21. Note that the complete binary tree has 5 levels. Only a fraction of the pipeline configuration architecture is illustrated for clarity.

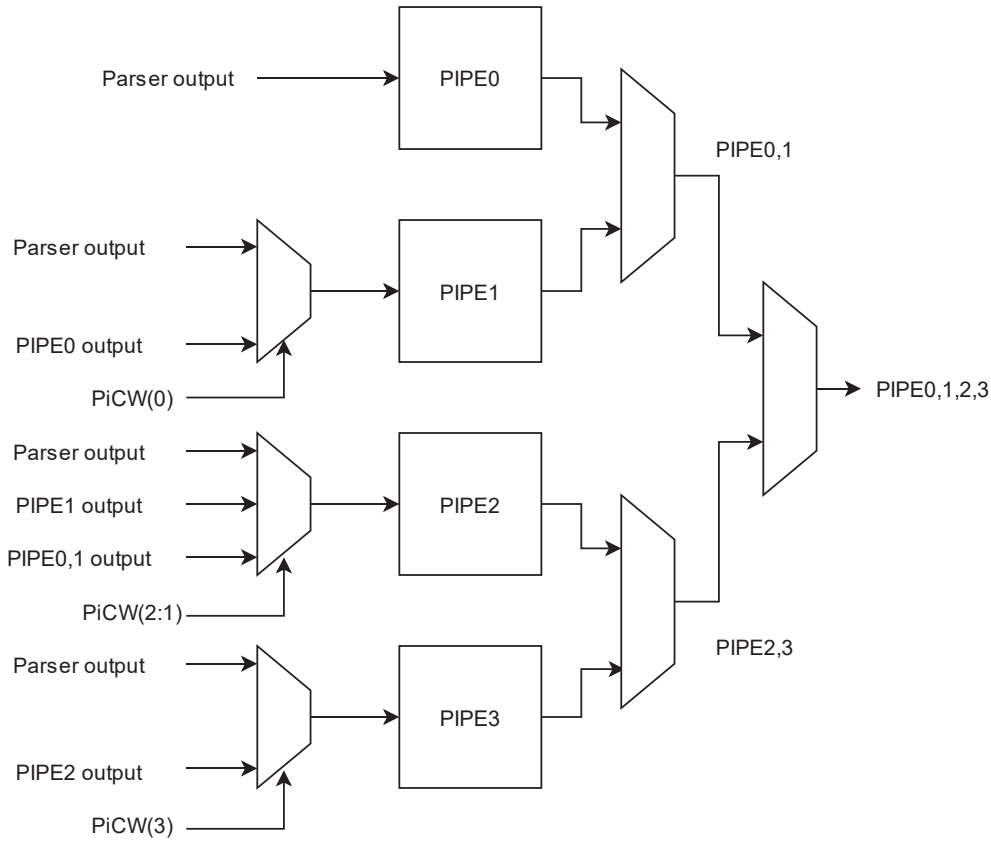

Figure 21. A fraction of the pipeline reconfiguration architecture (adjusted from P_{VI})

The required components are the multiplexers for selecting the input to the PIPE16 instances and priority-based 2-to-1 multiplexers that receive input from the final stage of the PIPE16 instances. For multiplexers providing input to PIPE16 instances, the select lines are set by Pipeline Configuration Word (PiCW). For priority-based 2-to-1 multiplexers, the select lines are set dynamically during the operation of the pipeline by *match found* flags. In addition to the output of the PIPE16 instance that returned a match, the match found flag is also multiplexed so that next-level priority-based multiplexer can function properly. Consider combination of 128 physical tables for making a logical table that is 128 times deeper than a single

physical table. It requires tables from eight PIPE16 instances. By providing the correct select values for the input multiplexers, eight PIPE16 instances can receive the same input and operate in parallel. By doing so, instead of 128 cycles, 16 cycles are required for visiting the tables.

7.2.3 Action Input Selectors

The PHV contains 128 32-bit words. It is also possible to extract four 8-bit units and three 16-bit units from each 32-bit word. Therefore, the action input selectors are 1024-to-1 multiplexers whose width is 32 bits. The area of such a multiplexer is 8100 μm^2 in 28 nm technology. At each stage of the pipeline, there are eight ALUs, each requiring two field extractors. In addition, the condition evaluator also requires a pair of field extractors. As a result, the area of field extractors within one stage of the pipeline is 145800 μm^2 and the total area in the whole pipeline is 74.64 mm^2, which is equivalent to the area of 429 instances of 2K × 32-bit TCAM.

Due to the large area and based on the insight provided in chapter 5, it is wise to consider using smaller action input selectors. The optimization strategies discussed in chapter 5 are based on the principle of dividing the PHV into logical segments and limiting access to PHV entries belonging to other segments. The optimized field selection scheme of use divides the PHV into 8 logical segments, each of which contains one ALU instance. Inside each segment, there are 16 32-bit words. Considering the 8-bit and 16-bit units within the 16 32-bit words, the total number of options for field selection is 128. For PHV entries in other segments, it is only possible to access them in 32-bit units. If an 8-bit field from another segment is required, the whole 32-bit unit in which it is located is read. Therefore, the total number of options for each optimized action input selector is 240. The area of the optimized field selector is 36% of the area of the full field extractors. The first ALU and the condition evaluator still use the large field selectors. The other seven ALUs use the reduced field extractors. The area of field selectors per stage and across all stages is 79000 μm^2 and 40 mm^2, respectively. This is a 46% area reduction with respect to using full field extractors for all functional units.

7.2.4 Pointer-based Header Field Referencing

Sometimes the location of a field that needs to be retrieved as an operand or selected as the destination is not fixed within the header. In these cases, it cannot be accessed

by an absolute index. In Segment Routing (SR), the use of pointer-based field referencing is common. IPv6 Segment Routing Header (SRH) [92], contains a list of IPv6 addresses. The index of the IPv6 address that must be used as the destination address is pointed to by the *Segments Left* field. In the absence of more advanced field-referencing mechanisms, the value of the pointer must be used as a search key. The outcome of the search is an instruction pointer that points to the correct instruction. However, the instructions that are stored for different values of the pointer all have the same opcode. The only difference is in the absolute value of source or destination indexes. This results in inefficient use of lookup and instruction memories.

In this architecture, reading from or writing to a pointer-specified header field can be done without any lookup and using only one instruction. For using an operand that is referred to by a pointer, the pointer field is selected directly as the operand. The ALUs designed have opcode for pointer dereferencing. Once this operation has been executed, the value of the field that has been pointed to by the pointer is available at the specified location. For writing to a location pointed to by a pointer, the first source operand is the value to be written and the second operand is the pointer, both of which can be accessed using absolute address. The operation write to pointer-based location is executed. In this operation, the location specified by the pointer is accessed for writing. When this operation is executed, the writing ALU can override other ALUs because the write location may be located beyond the locations that the ALU in question writes to.

7.3 Implementation results

In this section, the implementation result of the proposed architecture are provided. Table 20 provides an area breakdown of the constituting components of a single packet processing stage. For those components of which multiple instances are present, the total area has been provided. As can be seen, the major contributors to the area are ternary and exact match tables followed by action input selectors. The ASIC technology, backend tool, operating frequency and operating conditions are same as in previous chapters.

Table 20. Area of the constituent components of a packet processing stage (adjusted from P_{VI})

Component	Total area (mm^2)
TCAM	0.180
4-way Exact Match tables	0.125
Field selectors	0.079
Instruction Memory	0.032
PID Map Table	0.031
PHV	0.018
Field- and State-Modifiers	0.012
Search Key Selectors	0.009
Total	0.486

7.3.1 Comparison with other Match-Action Architectures

In this section, the area of the proposed architecture is compared with that of RMT and dRMT. From the perspective of sustained throughput, these architectures are on par with each other. The proposed pipeline sustains 800 Gbps throughput. RMT can also sustain 800 Gbps if its operating frequency is scaled up to 1.25 GHz which is only marginally higher than 1.19 GHz of this architecture. For dRMT, 2 variants have been considered, each with a different value of Inter-Packet Concurrency (IPC). The values in Table 21 correspond to one MAU, one dRMT processor, and one packet processing stage in this architecture. Based on the values of Table 21, Table 22 contains the total area in the three architectures. What is meant by the crossbar area is the area of components required for combining tables. In this architecture, the components required for configuration of pipeline fit into this category. dRMT has more crossbar area because on top of table combination crossbars, there are crossbars for assigning table clusters to dRMT processors. All the three architectures have equal amount of memory for ternary and exact matching.

Table 21. Comparison of the area (mm^2) of RMT, dRMT and the proposed architecture [P_{VI}]

Component	RMT	dRMT (IPC = 1)	dRMT (IPC = 2)	This architecture
Match key config. Reg.	0.021	0.012	0.015	0.000
Match key crossbar	0.187	0.150	0.217	0.009
PHV	0.336	0.998	1.439	0.018
Scratchpad	N/A	0.156	0.156	N/A
Action input selector	1.448	0.523	0.964	0.079
ALUs	0.200	0.050	0.050	0.012
Action output selector	N/A	0.147	0.147	0.000
VLIW instruction table	1.139	1.029	1.029	0.032
Total	3.331	3.065	4.017	0.150

Table 22. Total area for the three architectures under comparison [P_{VI}]

Architecture	Non-crossbar area (mm^2)	Crossbar area (mm^2)	Total area (mm^2)
RMT	106.592	6	112.592
dRMT (IPC = 1)	98.080	11.328	110.128
dRMT (IPC = 2)	128.544	11.328	139.872
This architecture	76.800	1	77.800

7.3.2 Discussion of results

The proposed architecture applies simplifications to RMT and also makes some enhancements. One important thing to note is that the presence of 32 MAUs in RMT architecture does not mean that it is a pipeline of 32 stages. Each MAU is internally pipelined because the operations that must be performed inside MAU require a number of cycles. These operations include search key generation, lookup, action memory access, instruction memory access, and action execution. Considering the number of latency cycles in match and action dependencies, the total number of physical stages in RMT and the proposed architecture are on par with each other.

The main motivation behind a long pipeline in which each stage contains only one ternary match table and one exact match table is eliminating the complex logic for combination of tables and allowing any number of tables to be combined for forming wider and/or deeper tables. Furthermore, some actions must be mapped to a chain of ALUs across multiple stages. One action stage after a match is not sufficient for complex actions. The operation executed in the ALUs in each MAU is dependent on the Match in the same MAU. A strong argument in favor of deep pipelines is that it is better to forward packets to a deep pipeline in which some operations, including both match and action, are executed speculatively, rather than a short pipeline in which packets are likely to be recirculated. Although a deep pipeline has high latency as its main characteristic, it provides guaranteed performance. Once the match result is ready, the outcome of unnecessary speculatively executed operations is discarded.

One of the main design goals of this architecture was efficient use of memory resources. The RMT architecture uses match tables extensively for program control. This means that checking the value of TTL field also requires matching. This architecture, on the other hand, uses combinatorial logic for this purpose and reserves lookup tables for address lookup purposes. Another architectural enhancement introduced in this architecture was support of pointer-based addressing mode. The absence of this kind of addressing mode in RMT causes the instruction memory entries to be exponentially filled in an inefficient manner.

Despite the enhancements, the proposed packet processing pipeline has 31% less area compared to the RMT architecture. This area saving is about 35 mm^2. This is equivalent to 200 TCAM blocks of 2K×40b or more than 2200 instances of 1K×32b SRAMs. Integrating more memories increases the match and action capacity of the architecture, which is one of the performance metrics. Another benefit of an architecture with noticeable area savings is that it allows more instances being integrated into the chip while adhering to the overall area constraint.

8 CONCLUSION

SDN has the concept of software controlling the network at its heart. Despite this, it has provided new opportunities for custom packet processing hardware. There has been a shift from network processors to custom programmable switch ASICs. NFV, as the other major development in computer networks, is about softwarization of network functions and decoupling them from middleboxes. However, it has also provided new opportunities for packet processing hardware by causing a shift from middleboxes to smartNICs. As a result, custom packet processing hardware has not been eliminated. There is a lot of room for innovation. Innovation is needed mostly in hardware architecture, not in ISA. Although the main focus of this thesis was packet processing architecture for switching and routing, the contributions made are also of benefit for architecting smartNICs as well.

8.1 Research Findings

This thesis answered many questions specific to the architectural choices. VLIW is a suitable parallel processing scheme for protocol-independent packet processing. In VLIW, parallelism is dictated explicitly by software. This is in line with the concept of software being in charge. Another benefit of VLIW is that the time required for implementation and verification is far less than that of processors with run-time scheduling hardware. Although neither RMT nor the pipelined architecture in chapter 7 are processors, the fact that they contain multiple ALUs per stage makes them fit the VLIW category.

Deep pipelines and SMT are the dominant architectures for high-performance packet processing. In principle, every computation can be performed by a processor. SMT is the solution for sustaining throughput of packets arriving through ports whose speed is 100 Gbps and above. The major issue with SMT is that implementation complexity can get out of hand. When a large number of simultaneous threads must be supported, it is not possible to add an independent read port to the memory hosting instructions. A pipelined architecture can offer the same performance level using simpler hardware. However, there are certain tasks

that the pipelined architecture is not capable of. The conclusion is that both pipelined and processor-based architectures are required for high-performance protocol-independent packet processing. The ideal combination is a processor-based packet preprocessor, or a preprocessor coupled with a pipelined parser as the parsing and preprocessing subsystem and a pipeline for the packet processing subsystem.

With respect to the question of achieving programmability in packet parsing without using TCAMs, it was seen that using simple binary matching of next header indicator against expected values results in far simpler and more power-efficient hardware. Determining the header size can be done by manipulating header size indicator or if necessary binary matching. As for enhancing the performance of packet parsers, using multithreading in which each thread is in charge of parsing packets arriving through the port to which the thread corresponds is effective. For further performance enhancement, the pipelined parser in which at each clock cycle an entire minimum-size frame is accepted is a promising solution.

There are numerous cases in which the parser can assist packet processing. At a minimum, integrity checking operations can be easily performed by the parser. In addition, the process of fragmentation of variable-sized packets into fixed-size cells can also be handled by the parser. Since match field crossbars are one of the main contributors to area, the packet preprocessor can write the fields used for matching in a designated location of PHV. Doing so helps eliminate match crossbars or use far smaller ones.

Deep programmable pipelines are a good match for flexible layer-2 and layer-3 packet processing. However, some considerations must be taken into account to avoid excessive area and power dissipation. One such consideration concerns instruction memories. Since there is a vast number of memories for storing the instructions, it is necessary to keep the memories small. The way this was achieved in the pipeline of Chapter 7 was use of small instruction memories whose entries can be reused in an efficient manner. The next consideration deals with combination of lookup tables for making tables of custom size. The pipeline of Chapter 7 allows combination of an arbitrary number of tables with least possible amount of hardware. Finally, since each stage has multiple ALUs each of which requiring 2 action input selectors that select from a large number of header fields, it is imperative to subject the number of inputs to some limit. For the designed pipeline, this was done by segmenting the PHV into smaller units. All the header fields in a given logical segment can be read in 3 different sizes while header fields in other segments are read only in 32-bit size, thereby reducing the number of inputs to the multiplexers. Regarding the action input selectors, it is important to note that direct

addressing of header fields is not sufficient. There must be support for pointer-based reads. The same applies for selecting action output destination.

Performance comes not only from hardware, but from software as well. One of the performance-enhancing software techniques used in the designed pipeline was speculatively looking up tables and executing actions. This speculation is instructed by software. The width of PHV allows storing speculative results. From the perspective of throughput and latency, it is far better to speculatively perform matching and action execution rather than recirculating a packet.

The most important finding of this dissertation is that achieving high performance and enhanced functionality does not necessarily come at the cost of increased complexity and power dissipation. The key to achieving high performance without increase in chip area and/or power requirements is making the right architectural choices.

The findings of this dissertation can also be applied to FPGAs as well. Use of low-area hardware architectures is of significance in FPGAs especially because entry-level FPGAs that are more accessible to the research community are more limited in available resources. Both packet parser variants can be implemented on FPGAs as well as they do not require TCAMs. However, some architectural modifications are required to compensate for the lower performance resulting from lower operating frequency of FPGAs.

This dissertation provided insight into packet processing hardware. The insight is not only of significance to packet processing system architects, but to network protocol designers as well. Network protocols are designed mainly with the functional requirements in mind. After reading this dissertation, the protocol designer is familiar with the strengths and weaknesses of packet processing hardware. For instance, in protocols whose header is comprised of multiple words, it is more efficient to place the next header indicator as far as possible from the ending boundary of the header. Furthermore, variable-length headers should be avoided unless absolutely necessary as they put additional performance requirement on packet parsers. IPv6 is an example of a well-designed network protocol. Its enhancements compared to IPv4 are not limited to a much larger address space. The number of integrity checking operations required for IPv6 is considerably smaller than that of IPv4. In addition, the size of IPv6 header is fixed which eases the task of parser. Finally, the position of next header indicator is chosen such that there will be no need for idling the pipeline until the address of instruction for parsing next header becomes available.

8.2 Open Problems and Future Directions

One of the open issues is the degree to which network operators are willing to support flexibility. For instance, one of the questions is whether the programmable data plane must be an architecture that supports flexibility on top of Ethernet or is it desirable to also support any protocol as the first protocol in the header stack of packets. This is a fundamental question as it affects the datapath width and the frequency of packet arrival.

The next major open issue is programming of packet processing hardware. Network administrators prefer to use a high-level language for specifying network policies. There must be tools that translate the policies and provide the operation and configuration codes for the hardware. P4 is one such language. However, its program control mechanism is limited to Match and Action. The RMT architecture is in fact a compiler target for P4. An ideal solution is a language with high-level properties of P4 and yet with the ability to be mapped to various packet processing hardware architectures.

As for future work, the author's plan is to extend this research both in the hardware and software domains. In the hardware domain, the intention is to use both ASICs and FPGAs as the target platform in order to cover the following aspects:

-Programmability of buffer management

-Memory-efficient hash collision resolution

-Enhancing the flexibility and throughput of the designed pipeline

On the software side, the aim is to develop a tool that accepts packet processing requirements at a higher layer of abstraction and generates the configuration and operation codes for the pipeline.

REFERENCES

[1] M. Casado, N. McKeown, and S. Shenker, "From Ethane to SDN and Beyond," *Comput. Commun. Rev.*, vol. 49, no. 5, pp. 92–95, 2019, doi: 10.1145/3371934.3371963.

[2] 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)*, 2015, pp. 103–115.

[3] "Requirements for Separation of IP Control and Forwarding," 2003. https://tools.ietf.org/html/rfc3654 (accessed Nov. 06, 2020).

[4] "Forwarding and Control Element Separation (ForCES) Framework," 2004. https://tools.ietf.org/html/rfc3746 (accessed Nov. 06, 2020).

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Mckeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, 2007, doi: 10.1145/1282380.1282382.

[6] N. McKeown *et al.*, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008, doi: 10.1145/1355734.1355746.

[7] P. Bosshart *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014, doi: 10.1145/2656877.2656890.

[8] 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*, 2013, pp. 127–132, doi: 10.1145/2491185.2491190.

[9] J. Yu, X. Wang, J. Song, Y. Zheng, and H. Song, "Forwarding Programming in Protocol-Oblivious Instruction Set," in *2014 IEEE 22nd International Conference on Network Protocols*, 2014, pp. 577–582, doi: 10.1109/ICNP.2014.92.

[10] S. Li *et al.*, "Protocol Oblivious Forwarding (POF): Software-Defined Networking with Enhanced Programmability," *IEEE Netw.*, vol. 31, no. 2, pp. 58–66, 2017, doi: 10.1109/MNET.2017.1600030NM.

[11] M. Shahbaz and N. Feamster, "The Case for an Intermediate Representation for Programmable Data Planes," 2015, doi: 10.1145/2774993.2775000.

[12] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network

processing as a cloud service," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, 2012, doi: 10.1145/2377677.2377680.

[13] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Commun. Surv. Tutorials*, vol. 18, no. 1, pp. 236–262, 2016, doi: 10.1109/COMST.2015.2477041.

[14] Z. Ni, G. Liu, D. Afanasev, T. Wood, and J. Hwang, "Advancing network function virtualization platforms with programmable NICs," *2019 IEEE Int. Symp. Local Metrop. Area Networks*, vol. 2019-July, pp. 1–6, 2019, doi: 10.1109/LANMAN.2019.8847032.

[15] S. Miano, R. Doriguzzi-Corin, F. Risso, D. Siracusa, and R. Sommese, "Introducing smartnics in server-based data plane processing: The DDoS mitigation use case," *IEEE Access*, vol. 7, pp. 107161–107170, 2019, doi: 10.1109/ACCESS.2019.2933491.

[16] D. Firestone *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 51–66, [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[17] "OpenFlow Switch Specification (Version 1.5.1)." Open Networking Foundation, 2015, [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[18] "Open Systems Intrconnection Basic Reference Model: The Basic Model." International Organization for Standardization, 1994, [Online]. Available: https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip.

[19] "1G/2.5G Ethernet PCS/PMA or SGMII v16.2." XILINX, 2020, [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/gig_ethernet_pcs_pma/v16_2/pg047-gig-eth-pcs-pma.pdf.

[20] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design Principles for Packet Parsers," in *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2013, pp. 13–24, doi: 10.1109/ANCS.2013.6665172.

[21] R. Braden, D. Borman, and C. Partridge, "Computing the Internet Checksum," 1988. https://tools.ietf.org/html/rfc1071 (accessed Nov. 05, 2020).

[22] E. Nordmark and R. Gilligan, "Basic Transition Mechanisms for IPv6 Hosts and Routers," 2005. https://tools.ietf.org/html/rfc4213 (accessed Nov. 05, 2020).

[23] A. Conta, S. Deering, and M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," 2006.

https://tools.ietf.org/html/rfc4443 (accessed Nov. 03, 2020).
[24] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004, doi: 10.1016/j.jalgor.2003.12.002.
[25] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006, doi: 10.1109/JSSC.2005.864128.
[26] P. Bosshart *et al.*, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, 2013, doi: 10.1145/2534169.2486011.
[27] V. C. Ravikumar and R. N. Mahapatra, "TCAM architecture for IP lookup using prefix properties," *IEEE Micro*, vol. 24, no. 2, pp. 60–69, 2004, doi: 10.1109/MM.2004.1289292.
[28] "Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers." https://tools.ietf.org/html/rfc6146 (accessed Nov. 05, 2020).
[29] "Differentiated Services (Diffserv) and Real-Time Communication." https://tools.ietf.org/html/rfc7657 (accessed Nov. 06, 2020).
[30] "Internet Protocol, Version 6 (IPv6) Specification," 2017. https://tools.ietf.org/html/rfc8200 (accessed Nov. 06, 2020).
[31] "New Terminology and Clarifications for Diffserv." https://tools.ietf.org/html/rfc3260 (accessed Nov. 05, 2020).
[32] 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, 2000, doi: 10.1145/354871.354874.
[33] M. Dobrescu *et al.*, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 15–28, doi: 10.1145/1629575.1629578.
[34] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with CuckooSwitch," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, 2013, pp. 97–108, doi: 10.1145/2535372.2535379.
[35] B. Pfaff *et al.*, "The Design and Implementation of Open vSwitch," *Proc. 12th USENIX Symp. Networked Syst. Des. Implementation, NSDI 2015*, pp. 117–130, 2015.
[36] M. Shahbaz *et al.*, "PISCES: A Programmable, Protocol-Independent Software Switch," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 525–538, doi: 10.1145/2934872.2934886.
[37] P. J. McCann and S. Chandra, "Packet Types: Abstract Specification of Network Protocol Messages," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 321–333, 2000, doi: 10.1145/347057.347563.
[38] "The P4 Language Specification." The P4 Language Consortium, 2018,

[Online]. Available: https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.
[39] "P416 Language Specification." The P4 Language Consortium, 2019, [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.0.pdf.
[40] A. Sivaraman *et al.*, "Packet Transactions: High-Level Programming for Line-Rate Switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 15–28, doi: 10.1145/2934872.2934900.
[41] V. Moreno, J. Ramos, P. M. Santiago Del Rio, J. L. Garcia-Dorado, F. J. Gomez-Arribas, and J. Aracil, "Commodity Packet Capture Engines: Tutorial, Cookbook and Applicability," *IEEE Commun. Surv. Tutorials*, vol. 17, no. 3, pp. 1364–1390, 2015, doi: 10.1109/COMST.2015.2424887.
[42] G. Liao, X. Znu, and L. Bnuyan, "A new server I/O architecture for high speed networks," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 255–265, doi: 10.1109/HPCA.2011.5749734.
[43] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015, pp. 5–16, doi: 10.1109/ANCS.2015.7110116.
[44] "DPDK Intel NIC Performance Report Release 20.02," 2020. [Online]. Available: http://fast.dpdk.org/doc/perf/DPDK_20_02_Intel_NIC_performance_report.pdf.
[45] "Data Plane Development Kit." https://www.dpdk.org/ (accessed Jun. 08, 2020).
[46] L. Rizzo, "NetMap: A novel framework for fast packet I/O," in *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012, pp. 101–112.
[47] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi–gigabit packet capturing with multi–core commodity hardware," in *International Conference on Passive and Active Network Measurement*, 2012, pp. 64–73, doi: https://doi.org/10.1007/978-3-642-28537-0_7.
[48] "NetFPGA." https://netfpga.org/ (accessed Jun. 15, 2020).
[49] 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, doi: 10.1109/MM.2014.61.
[50] M. B. Anwer, M. Motiwala, M. Bin Tariq, and N. Feamster, "SwitchBlade: A platform for rapid deployment of network protocols on programmable hardware," in *SIGCOMM'10 - Proceedings of the SIGCOMM 2010 Conference*, 2010, pp. 183–194, doi: 10.1145/1851182.1851206.
[51] G. Brebner and W. Jiang, "High-Speed Packet Processing using Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 8–18, 2014, doi: 10.1109/MM.2014.19.
[52] B. Li *et al.*, "ClickNP: Highly Flexible and High Performance Network

Processing with Reconfigurable Hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 1–14, doi: 10.1145/2934872.2934897.

[53] H. Wang *et al.*, "P4FPGA: A Rapid Prototyping Framework for P4," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 122–135, doi: 10.1145/3050220.3050234.

[54] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," in *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, 2011, pp. 12–23, doi: 10.1109/ANCS.2011.12.

[55] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, "P4-To-VHDL: Automatic generation of high-speed input and output network blocks," *Microprocess. Microsyst.*, vol. 56, pp. 22–33, 2018, doi: https://doi.org/10.1016/j.micpro.2017.10.012.

[56] 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*, 2018, pp. 147–152, doi: 10.1145/3174243.3174270.

[57] J. Cabal, M. Kekely, P. Benáček, V. Puš, L. Kekely, 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*, 2018, pp. 249–258, doi: 10.1145/3174243.3174250.

[58] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-Accelerated Software Router," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 195–206, 2010, doi: 10.1145/1851275.1851207.

[59] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the Bar for Using GPUs in Software Packet Processing," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 409–423.

[60] Y. Go, M. Jamshed, Y. G. Moon, C. Hwang, and K. S. Park, "APUNet: Revitalizing GPU as Packet Processing Accelerator," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 83–96.

[61] "Arista 7060X Series." https://www.arista.com/en/products/7060x-series (accessed Jun. 08, 2020).

[62] J. S. Turner *et al.*, "Supercharging planetlab: A high performance, multi-application, overlay network platform," *ACM SIGCOMM 2007 Conf. Comput. Commun.*, vol. 37, no. 4, pp. 85–96, 2007, doi: 10.1145/1282380.1282391.

[63] "Intel IXP2800 and IXP2850 Network Processors." Intel, 2005.

[64] J. Markevitch and S. Malladi, "A 400Gbps Multi-Core Network

Processor," 2017, [Online]. Available: https://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.22-Tuesday-Pub/HC29.22.80-Architectdure-Pub/HC29,22,810-400gbs-NPU-Markevitch-Cisco.pdf.

[65] "FP4: Delivering performance and capability without compromise." https://www.nokia.com/networks/technologies/fp4/ (accessed Nov. 01, 2020).

[66] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam, "PLUG: Flexible lookup modules for rapid deployment of new protocols in high-speed routers," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, 2009, pp. 207–218, doi: 10.1145/1592568.1592593.

[67] G. Gibb, "Reconfigurable Hardware for Sonftware-Defined Networks," Stanford University, 2013.

[68] "The World's Fastest & Most Programmable Networks." https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/ (accessed Jun. 08, 2020).

[69] S. Chole *et al.*, "dRMT: Disaggregated Programmable Switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 1–14, doi: 10.1145/3098822.3098823.

[70] A. Sivaraman *et al.*, "Programmable Packet Scheduling at Line Rate," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 44–57, doi: 10.1145/2934872.2934899.

[71] V. Shrivastav, "Fast, Scalable, and Programmable Packet Scheduler in Hardware," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 367–379, doi: 10.1145/3341302.3342090.

[72] "Intel Ethernet Switch FM5000/FM6000 1 Gb/2.5 Gb/10 Gb/40 Gb Ethernet (GbE) L2/L3/L4 Chip Datasheet." Intel, 2017.

[73] "NFP-6000 Intelligent Ethernet Controller Family." https://www.netronome.com/static/app/img/products/silicon-solutions/PB_NFP6000.pdf (accessed Nov. 06, 2020).

[74] P. Bosshart, "Programmable Forwarding Planes at Terabit/s Speeds," 2018, [Online]. Available: https://www.hotchips.org/hc30/2conf/2.02_Barefoot_Barefoot_Talk_at_HotChips_2018.pdf.

[75] G. Antichi, T. Benson, N. Foster, F. Ramos, and J. Sherry, Eds., "Programmable Network Data Planes," *Dagstuhl Reports*, vol. 9, no. 3, pp. 178–201, 2019.

[76] B. Sayle *et al.*, *Cisco Catalyst 9000 A new era of intent-based networking*, Second edi. CISCO, 2019.

[77] "25.6 Tb/s StrataXGS® Tomahawk® 4 Ethernet Switch Series." https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series (accessed Nov. 06,

2020).
[78] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 3–14, 2014, doi: 10.1145/2740070.2626292.
[79] "Advanced Network Telemetry." https://www.barefootnetworks.com/use-cases/ad-telemetry/ (accessed Jun. 08, 2020).
[80] 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, pp. 15–28, doi: 10.1145/3098822.3098824.
[81] "Layer 4 Load Balancer." https://www.barefootnetworks.com/use-cases/loadbalancing/ (accessed Jun. 08, 2020).
[82] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the Network be the AI Accelerator?," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 20–25, doi: 10.1145/3229591.3229594.
[83] R. Glebke, J. Krude, I. Kunze, J. Rüth, F. Senger, and K. Wehrle, "Towards Executing Computer Vision Functionality on Programmable Network Devices," in *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, 2019, pp. 15–20, doi: 10.1145/3359993.3366646.
[84] D. R. K. Ports and J. Nelson, "When Should the Network Be the Computer?," in *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019*, 2019, pp. 209–215, doi: 10.1145/3317550.3321439.
[85] A. Checko *et al.*, "Cloud RAN for Mobile Networks—A Technology Overview," *IEEE Commun. Surv. Tutorials*, vol. 17, no. 1, pp. 405–426, 2015, doi: 10.1109/COMST.2014.2355255.
[86] L. M. P. Larsen, A. Checko, and H. L. Christiansen, "A Survey of the Functional Splits Proposed for 5G Mobile Crosshaul Networks," *IEEE Commun. Surv. Tutorials*, vol. 21, no. 1, pp. 146–172, 2019, doi: 10.1109/COMST.2018.2868805.
[87] "Common Public Radio Interface: eCPRI Interface Specification." Ericsson AB, Huawei Technologies Co. Ltd, NEC Corporationand Nokia, [Online]. Available: http://www.cpri.info/downloads/eCPRI_v_2.0_2019_05_10c.pdf.
[88] "1914.1-2019 - IEEE Standard for Packet-based Fronthaul Transport Networks," 2020.
[89] "1914.3-2018 - IEEE Standard for Radio over Ethernet Encapsulations and Mappings," 2018.
[90] J. A. Fisher, P. Faraboschi, and C. Young, "VLIW processors: once blue sky, now commonplace," *IEEE Solid-State Circuits Mag.*, vol. 1, no. 2, pp.

10–17, 2009, doi: 10.1109/MSSC.2009.932433.
[91] M. S. Schlansker and B. R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *Computer (Long. Beach. Calif).*, vol. 33, no. 2, pp. 37–45, 2000, doi: 10.1109/2.820037.
[92] "Segment Routing Architecture," 2018. https://tools.ietf.org/html/rfc8402 (accessed Nov. 03, 2020).

PUBLICATIONS

PUBLICATION
I

An Explicitly Parallel Architecture for Packet Parsing in Software Defined Networks

H. Zolfaghari, D. Rossi and J. Nurmi

2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 1-4

doi: 10.1109/ASAP.2018.8445123

An Explicitly Parallel Architecture for Packet Parsing in Software Defined Networks

Hesam Zolfaghari
Laboratory of Electronics and Communications Engineering
Tampere University of Technology
Tampere, Finland
hesam.zolfaghari@tut.fi

Davide Rossi
Department of Electrical, Electronic and Information Engineering
University of Bologna
Bologna, Italy
davide.rossi@unibo.it

Jari Nurmi
Laboratory of Electronics and Communications Engineering
Tampere University of Technology
Tampere, Finland
jari.nurmi@tut.fi

***Abstract*—Packet parsing is the first step in processing of packets in devices such as network switches and routers. The process of packet parsing has become more challenging due to the increase in line rates and emergence of Software Defined Networking which leads to new protocols being adopted. In this paper, we present a novel architecture for parsing of packets. The architecture is fully programmable and is not tied to any specific protocol. It can be programmed to parse any protocol making it suitable for Software Defined Networks. Compared with the parser used in the Reconfigurable Match Tables, our parser improves supported throughput by a factor of 3.2. Moreover, to achieve the target throughput of 640 Gbps, our parser needs only 2 percent of the number of gates used in the parsers of Reconfigurable Match Tables.**

Keywords— Packet Parsing, Software Defined Networking, Explicit Parallelism, Very Long Instruction Word, Packet Processing Pipeline

I. Introduction

Software Defined Networking (SDN) [1] is the solution to the emergence of a plethora of communication protocols. In recent years, numerous protocols have been proposed. For instance, protocols such as GENEVE [2], NVGRE [3] and VxLAN [4] are only a few of the protocols proposed for network virtualization. With traditional networking approach, the time between proposal of a new protocol and market availability of switches and routers supporting the proposed protocol is counter-productive. This elongated time is due to the complexity in designing, implementing and verifying the functionality of packet processing devices as they need to support a large subset of all network protocols. As a result of these pressures, vendors have shifted towards SDN for simpler hardware and shorter time to market.

Reconfigurable Match Tables (RMT) is an architecture proposed in [5]. The architecture is these days called Protocol Independent Switch Architecture (PISA) and has found its way into industry. Barefoot Network's Tofino is a programmable switch based on the PISA architecture [6]. One of the key components of PISA and any other packet processing system is the packet parser. The architecture of the packet parser used in PISA is based on the parser proposed in [7]. Packet Parsing is the process by which fields within the current header are recognized and extracted to be processed by the Packet Processing Pipeline. In a system designed to be employed in SDN environments, the packet parser must be protocol-independent and programmable. As a result, fixed-function Application Specific Integrated Circuits (ASICs) are not an option anymore. Instead, packet processing architectures must be designed with both goals of programmability and performance. In recent years, a number of architectures have been proposed such as [8], [9] and [10]. However, they use Field Programmable Gate Arrays (FPGAs) as the target device. FPGAs operate at considerably lower frequencies compared to ASICs. Such architectures have to operate on very wide input to achieve decent throughput. For instance, in [9] the datapath width is 320 bits. In [8] the width of the datapath is as wide as 2048 bits. FPGA-based packet parsers do not score well on the latency side. In many real-time packet processing environments, there is a strict ingress to egress latency constraint which should not be violated. Our programmable solution can be implemented on an ASIC and can replace the packet parser used in PISA as it has the same output format. With only 2 percent of the total gates used in the parser in [5], it sustains the same throughput and parses complex variable-length headers in less than 10 nanoseconds.

II. Our Task Formulation

In [5] and [7], the process of Packet Parsing is illustrated by a parse graph in which each state represents an entire header such as Ethernet or IPv4 header. These graphs are at a high level of abstraction. Instead, we focus on a graph in which each state represents parsing of at most four header fields within a given protocol. We present a programmable architecture with datapath width of 64 bits. Each state of the new parsing state machine is represented by one and only one instruction. The motivation for this choice is minimizing parsing latency. We use Very Long Instruction Word (VLIW) kind of instructions because they can do quite a lot of work per cycle [11]. With multiple fields being present in the arrived packet data, each sub-instruction within the VLIW instruction operates on one of the fields. States in the parse graph are analogous to VLIW instructions and we shall refer to states and instructions interchangeably. In a similar manner, state transitions are analogous to branches within the parse program. The architecture also encompasses program control logic to avoid expensive lookups into associative memories at each clock cycle. To further reduce the cost of lookups, there are a finite number of comparators operating in parallel. The parallel comparators compare the header field of choice against members of a comparand set, thereby limiting comparisons only to relevant comparands.

Our parser operates in the streaming mode, meaning that there is no need to buffer the incoming packets prior to parsing. Instead, packets are parsed as they arrive. Streaming parsers are superior in terms of performance and exhibit lower packet processing latency. This parser provides the input to the Match and Action Packet Processing Pipeline such as the one presented in [5]. This means that the parser extracts fields and attaches parsing metadata to them. Based on the accompanying metadata, the required action takes place upon the extracted fields. Metadata includes information such as the port on which the packet arrived and the identification number of the packet.

III. Architectural Details

Being an explicitly parallel architecture, there are a number of functional units operating in parallel [12]. Fig. 1 is a high-level illustration of the internals of our parser which is part of the Match and Action Packet Processing Pipeline. Only the main functional units and connections are presented in this figure for the sake of clarity. Each functional unit has its own field within the VLIW instruction. The VLIW instructions are 128 bits wide. The units perform the requested operation in one clock cycle with the operating frequency being 2 GHz.

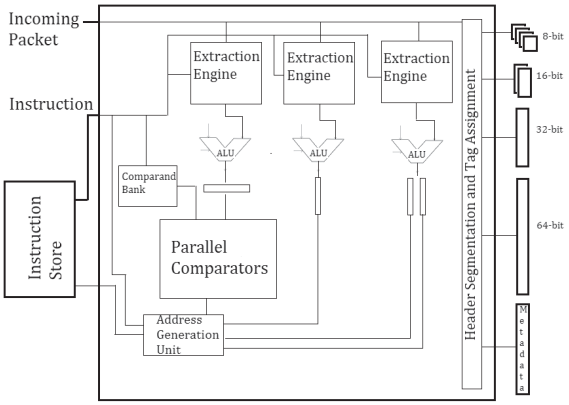

Fig. 1. Overall architecture

The main functional units are as follows:

A. Three Extraction Engines

Within the course of parsing headers, there are fields whose values are of significance for correct parsing. Fields containing the size of header, payload or entire packet are examples of these fields. Moreover, fields signaling the protocol used in the payload portion of the packet fit this category. In order to extract the values of such fields, three parallel Extraction Engines are required. Based on the extracted value the correct state transition takes place in the upcoming cycles. With these information, the parser is able to keep track of intra- and inter-packet boundaries. Moreover, it will perform the right state transitions in order to correctly parse the incoming packet data.

B. Parallel Comparators

Parallel comparators compare a portion of a packet extracted by one of the Extraction Engines against a set of comparands in parallel. This kind of functionality is required in variable-length headers in which the presence of some fields is dependent on the values of some flags. Parallel comparisons help perform the right state transition in a real-time manner without falling behind the rate of packet arrival.

C. Header Segmentation

Similar to the Extraction Engines, this unit is also in charge of extracting fields. Contrary to the fields extracted by the Extraction Engines, fields extracted by this unit will be the output of the parser and input to the Packet Processing Pipeline. According to [13], the parser used in Tofino has a container-based output format, meaning that extracted header fields are placed in containers of 8, 16 and 32 bits of width. These containers form a vector of fields which can be processed in parallel. Our parser has similar output format. We have added a 64-bit container as well which suits large fields.

The parse program specifies how the arrived header should be segmented and extracted. Attached to each set of extracted fields is the parsing metadata which contains information such as the port on which the corresponding packet arrived, the offset from the beginning of the packet and a user-specified tag.

D. Address Generation Unit

This unit provides address of the next instruction which is the address of the next state in the parse state machine. It does so based on the branch condition specified by the current instruction and, if required, the result of parallel comparisons. Branches are based on values of header fields extracted by the Extraction Engines and the internal state of the parser. For instance, when parsing IPv4 packets, if the value of Internet Header Length (IHL) is 5 and the parser has parsed five 32-bit words since the time IHL was received, the parser will branch to an instruction which starts parsing the next header.

E. Arithmetic and Logic Units

The values of fields extracted by the extraction engines may need to undergo some modifications by an Arithmetic and Logic Unit (ALU). There are numerous cases in which this kind of functionality may be desirable. For instance, in IPv4 header, the size of header is encoded in terms of number of 32-bit words while total length of the packet is encoded in terms of number of bytes. Such values must be normalized to a universal encoding so that the state of the parser is updated automatically as contents of the packet arrive without requiring the programmer to update the state manually by means of software. As another example, there are branch conditions that make use of ALUs to resolve the branch result. For instance, in an Ethernet frame, if the value of EtherType is greater than 1500, the field signals the next protocol. Otherwise it indicates the size of payload in bytes.

There are architectural features that are unique to this parser. The first one is that branches have no penalty. This means that even if there are frequent jumps in the program flow, the execution time is the same as for the case in which there are no branches. This is partly due to the fact that instructions require very little decoding. Moreover, there is no program counter register in this architecture. Instead, each instruction carries its own address, thereby, playing the role of a virtual real-time program counter. The architecture uses the so-called bundle instructions which are if-elsif-else instructions, making use of the parallel comparators. In these instructions, all conditions are evaluated in parallel and only the one evaluating to true determines the program flow. These instructions implicitly contain 64 bits of comparands and 32 bits of addresses. Yet these instructions carry only 5 extra bits compared to ordinary instructions. Therefore, the overhead is negligible.

Most parsers rely on Content Addressable Memories (CAM) for matching. Our parser does not employ any form of CAM and yet it does not suffer from any performance penalty. For instance, the parser used in [5] and [6] uses a Ternary CAM (TCAM) whose search key is comprised of an 8-bit value denoting state and 32 bits of header data. We do not need a state index because when being in the set of

states/instructions pertaining to a specific protocol, the state index is implicit. Moreover, in most cases, only few of the TCAM entries need to be searched. For instance, when parsing Ethernet header, in order to determine whether the incoming frame contains Virtual LAN (VLAN) tags, the first 16 bits after the source MAC address must be compared with hexadecimal values of 88A8 and 8100. In our architecture, these two values are referred to as a comparand set. A dedicated memory unit referred to as the comparand bank holds the comparand sets. A comparand set can have an arbitrary number of elements. When a comparand set's index is presented to the comparand bank, the corresponding comparands are loaded into the comparators in parallel. In TCAM-based approach all entries will be searched for matching value while in our solution, which is a lot simpler, only relevant entries are searched which is a lot more efficient. We only use a handful of comparators operating in parallel. Therefore, the resulting area is negligible compared to the 256 × 40 bit TCAM used in [5].

When parsing application-layer headers, the payload section of the packet is not subject to parsing and should be directed to a so-called common data buffer. In our programmable architecture, parse programs for application-layer headers are independent of the size of the packet. The payload section is forwarded to the common data buffer using only one instruction regardless of the size of the payload. The instruction loops back to itself until payload is fully forwarded. Meanwhile, all corresponding counters and states are updated automatically.

IV. Experimental Results

We have implemented the architecture in VHDL and synthesized it on 28 nm UTBB FD-SOI technology in worst-case operating conditions (1.0V, ss, 125°C) using Synopsys Design Compiler J-2014.09-SP4, while power analysis was performed in typical operating conditions at the supply voltage of 1.1V (tt, 25°C). Architectures operating at higher frequencies are more challenging to design due to the timing constraints imposed by higher frequencies. We have verified that the parser can operate at 2 GHz. Table I outlines the synthesis results. In [5], 16 instances of 40 Gbps parsers are used in parallel to achieve aggregate throughput of 640 Gbps. These parsers, which are also synthesized using 28 nm process, have total gate count of 5.6 million. A single instance of our parser supports throughput of 128 Gbps. For achieving 640 Gbps aggregate throughput, we need only 5 instances of our parser. This translates to 114K gates which is only 2 percent of the number of gates required for the parsers used in [5] without causing any performance degradation or limit in programmability. This substantial reduction is to a great degree owing to the elimination of TCAM. According to [5], the TCAM alone requires over 10^6 logic gates. Moreover, we are not employing any form of speculation or prediction of next header. If the next header arrives at the same time as its indicator, it cannot be parsed until the address of the subroutine in charge of parsing it has been resolved. However, with optimized scheduling of instructions, even in this extreme case, the number of dead cycles will be limited to two which equals one nanosecond. Furthermore, very little state is maintained in this architecture. Everything is instructed by software. The parse program instructions which arrive in synchrony with the header fields control the functionality of functional units within the parser. Therefore, the logic is as simple as executing simple instructions such as extraction, basic arithmetic, comparison and condition checking in parallel. Table II outlines the power consumption of a single instance of our parser.

In [7], 64 instances of non-programmable 10 Gbps parsers consume around 450 mW of power in total. As mentioned earlier, for that throughput, we need only 5 instances of our parser. This results in power consumption of 221 mW, not to mention the fact that our parser is programmable, as a result of which it consumes more power than its non-programmable counterpart. Moreover, it operates at 2 GHz frequency while the parsers used in [5] and [7] operate at 1 GHz. Therefore, we have a reduction factor of more than 50 percent compared to [7].

We have programmed the parser to parse a number of headers. Parse programs for most headers have very few instructions. For instance, parse program for IPv6 header requires 8 instructions. Fig. 2 shows time required for parsing a number of headers. The best case denotes the case in which optional fields are not present while the worst case indicates the presence of all optional fields. For fixed length headers, best case and worst case are equal. In calculating parsing time, we have also considered the execution time of the instruction which passes program control to the subroutine in charge of parsing the next header. Therefore, the parsing times are realistic. As we can see, parsing latencies are orders of magnitudes shorter than figures in FPGA-based solutions. In [8], the average parsing latency per header is between 58 and 108 nanoseconds while in our solution headers are parsed in less than 10 nanoseconds. This reflects that the ultra-wide datapath of FPGA-based solutions does not help reach low latencies. Fig. 3 illustrates parsing time for IPv4 packets of different sizes. For minimum-sized IPv4 packet which comprises the header only and no header

TABLE I. Area results for a single parser instance

Number of ports	591
Number of nets	1304
Number of cells	437
Number of combinational cells	364
Number of sequential cells	54
Number of buffers/inverters	91
Number of references	69
Combinational area	4577.596841 µm2
Buf/Inv area	909.840006 µm2
Noncombinational area	6585.664149 µm2
Total cell area	11163.260990 µm2
Total gate count	22800

TABLE II. Power results for a single parser instance

Power group	Internal power	Switching power	Leakage power	Total power
Clock network	0.5928	0.8941	0.0021	1.4891 (3.37%)
Register	17.2796	0.1307	1.2178	18.6281 (42.20%)
Combinational	2.9715	19.9672	1.0866	24.0244 (54.43%)
Total	20.8439 mW	20.9920 mW	2.3066 mW	44.1417 mW

Options nor payload, parsing takes 3 nanoseconds. This equals to a throughput of 53.33 Gbps. For IPv4 packets of maximum size, i.e 65535 bytes, parsing takes 4.1 milliseconds. This translates to throughput of almost 128 Gbps. As we can see, larger packets score better in terms of throughput. This is because the 64-bit container can be utilized. For minimum-sized IPv4 packet with the header only, the 64-bit container remains empty. Although it is possible to pack multiple header fields into a 64-bit container, it is not recommended as it hurts parallelism in the packet processing pipeline. Headers such as IPv6 which contain large fields such as 128-bit source and destination addresses can make better use of the 64-bit container, thereby boosting throughput. For instance, as we can see in Fig. 2, parsing 40-byte IPv6 fixed header takes equal time as parsing 20-byte IPv4 header. On the whole, smaller packets result in greater number of packets being parsed per second while larger packets result in better throughput.

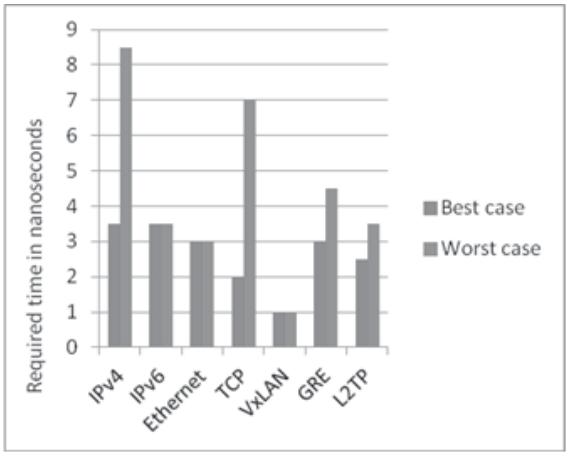

Fig. 2. Time required for parsing various headers

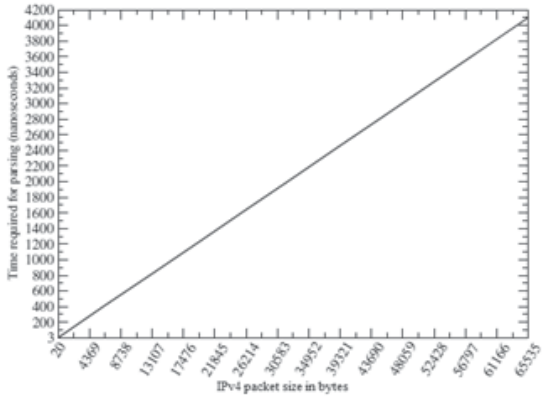

Fig. 3. Time required for parsing IPv4 packets of varying sizes

For links with multiple channels of incoming packets, multiple instances of the parser can be placed per channel to support a higher aggregate throughput.

V. Conclusion and Future Work

In this paper we presented the architecture of a fully-programmable protocol-independent packet parser for Software Defined Networks. As we have seen, the architecture is a lot simpler and yet superior in throughput compared to the parser with similar output format. This proves that SDN, while requiring programmable packet processing, does not require complex hardware. We have also seen that an explicitly parallel architecture suits packet parsing applications very well.

We would like to enhance the architecture of this parser so that it supports even higher throughputs. Moreover, we would like to fully automate the process of packet parsing so that the required instructions are generated after the parsing requirements are described in a high level of abstraction.

References

[1] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," Whitepaper 2012.

[2] Jesse Gross, Ilango Ganga, and T. Sridhar. (2018, March) Geneve: Generic Network Virtualization Encapsulation. [Online]. https://www.ietf.org/id/draft-ietf-nvo3-geneve-06.txt

[3] Pankaj Garg and Yu-Shun Wang. (2015, September) NVGRE: Network Virtualization Using Generic Routing Encapsulation. [Online]. https://tools.ietf.org/html/rfc7637

[4] Mallik Mahalingam et al. (2014, August) Virtual eXtensible Local Area Network (VXLAN): A Framework. [Online]. https://tools.ietf.org/html/rfc7348

[5] Pat Bosshart et al., "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," in ACM SIGCOMM, Hong Kong, 2013, pp. 99-110.

[6] Barefoot Networks, "The world's fastest and most programmable networks," Whitepaper. [Online]. https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/

[7] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown, "Design principles for packet parsers," in ACM/IEEE symposium on Architectures for networking and communications systems, San Jose, 2013, pp. 13-24.

[8] Michael Attig and Gordon Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, 2011, pp. 12-23.

[9] 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," in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, 2018, pp. 147-152.

[10] Pavel Benáček, Viktor Puš, Hana Kubátová, and Tomáš Čejka, "P4-To-VHDL: Automatic generation of high-speed input and output network blocks," Microprocessors and Microsystems, vol. Volume 56, pp. 22-33, February 2018.

[11] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young, "VLIW processors: once blue sky, now commonplace," IEEE Solid-State Circuits Magazine, vol. 1, no. 2, June 2009.

[12] Mark Smotherman, "Understanding EPIC Architectures and Implementations," in 40th Annual Southeast ACM Conference, 2002.

[13] Vladimir Gurevich. (2017, May) Programmable Data Plane at Terabit Speeds. [Online]. https://p4.org/assets/p4_d2_2017_programmable_data_plane_at_terabit_speeds.pdf

PUBLICATION
II

Low-latency Packet Parsing in Software Defined Networks

H. Zolfaghari, D. Rossi and J. Nurmi

2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), 1-6

doi: 10.1109/NORCHIP.2018.8573461

Low-latency Packet Parsing in Software Defined Networks

Hesam Zolfaghari
Laboratory of Electronics and Communications Engineering
Tampere University of Technology
Tampere, Finland
hesam.zolfaghari@tut.fi

Davide Rossi
department of Electrical, Electronic and Information Engineering
University of Bologna
Bologna, Italy
davide.rossi@unibo.it

Jari Nurmi
Laboratory of Electronics and Communications Engineering
Tampere University of Technology
Tampere, Finland
jari.nurmi@tut.fi

***Abstract*— Packet parsing is the first step in processing of packets in devices such as switches and routers. In this paper, we present a totally new program control unit as well as further enhancements for a recently designed packet parser architecture which can parse headers of most commonly used protocols such as Ethernet, IPv4, IPv6 and TCP in a time window shorter than 10 nanoseconds. However, when it comes to parsing variable-length headers and multiple stacked headers, it deviates from its maximum throughput due to inefficiencies in its program control logic. We have designed and employed a more advanced program control logic that improves parsing time of variable length headers such as IPv4 header by up to 48 percent and parsing time of typical header stacks used on the Internet by 16 to 21.42 percent. Our solution can sustain aggregate throughput of 640 Gbps while requiring only 30 percent of the number of gates used in the parser used in the Reconfigurable Match Tables architecture.**

Keywords—Software Defined Networking, Programmable data plane, Packet Parsing, Advanced Program Control

I. Introduction

Software Defined Networking (SDN) is the key to deployment and management of complex networks. New network protocols are being proposed and standardized by both the industry and academia. The internals of the packet processing devices such as switches and routers can no longer accommodate the logic for the aggregate of network protocols proposed and standardized so far. Instead, the data plane of the packet processing systems must be protocol-independent and programmable, so that they can provide the functionality required for network protocols of present and future. This requires thorough analysis of the operations incurred in processing of packets. A common concern for programmable and protocol-independent data plane is that of performance. However, as we will see, such systems can be on par with the conventional systems due to simpler architecture which allows for further optimizations.

Recently, there have been attempts to design programmable data planes. The most notable of these efforts are [1], [2] and [3]. [2] has also been commercialized and its architecture is now the basis of Barefoot Tofino [4]. In this paper we are interested in the problem of packet parsing. There are countless papers in which FPGA-based parsers are proposed. [5], [6] and [7] are just a few examples of such research efforts which achieve throughput on the scale of hundreds of Gigabits per second. However, it should be noted that these architectures achieve this throughput by means of operating on ultra-wide input due to their low frequencies. For instance, in [5], the input width is 2048 bits. Obviously, no transmission medium can transfer this amount of data at once. As a result, the actual throughput is far below the claimed figure. We are interested in parsing solutions that can sustain the line rate and that can parse the packets on the fly without having to buffer them. Moreover, we would like such a solution to be programmable and not tied to any specific set of protocols in order to be in line with the concept of SDN.

II. An Explicitly Parallel Architecture

In [8], we presented a novel programmable packet parser which was an explicitly parallel architecture. Fig. 1 illustrates a high-level view of the architecture. As we saw, explicitly parallel architectures suit packet parsing very well because with each segment of the packet header, there are tasks that can be performed in parallel.

The arrived header segment, which could be 16, 32 or 64 bits in width, is received by three programmable extraction engines. Each one of them can be instructed independently to extract the programmer-specified portion of the header segment. The extraction engines are meant to extract header fields containing size of payload, size of header and next header indicator. The extraction results are registered. One of the registers is the input to two counters, the second register provides input to one counter and the third register provides input to four comparators operating in parallel. The purpose of the counters is to maintain boundaries between headers and packets. They do so by counting down after being assigned value. When they reach zero, they signal it to the address generation unit (AGU) which provides the address of next instruction at each clock cycle. It should be noted that when a counter reaches zero, it does not automatically trigger an action, instead it should be checked manually by software. The parallel comparators compare the value of extracted field with four comparands. The comparands are stored in a memory unit and the programmer loads comparands of their choice whenever required. Therefore, the first input to all comparators is the extracted field but the second input varies from one comparator to the other. Associated with each comparand is a branch address. The outcome of the comparison is also registered and then provided to the AGU. The parallel comparators provide functionality similar to a Ternary Content Addressable Memory (TCAM). The output of the parser is a vector of header fields. The programmer specifies how the arrived header segment should fill the entries of the vector. For instance, with a 16-bit header segment, it is possible to fill two 8-bit entries or one 16-bit entry. The programmer makes the decision based on the structure of the header.

As this parser was the first attempt to provide an architecture which does not require look-up into TCAM at

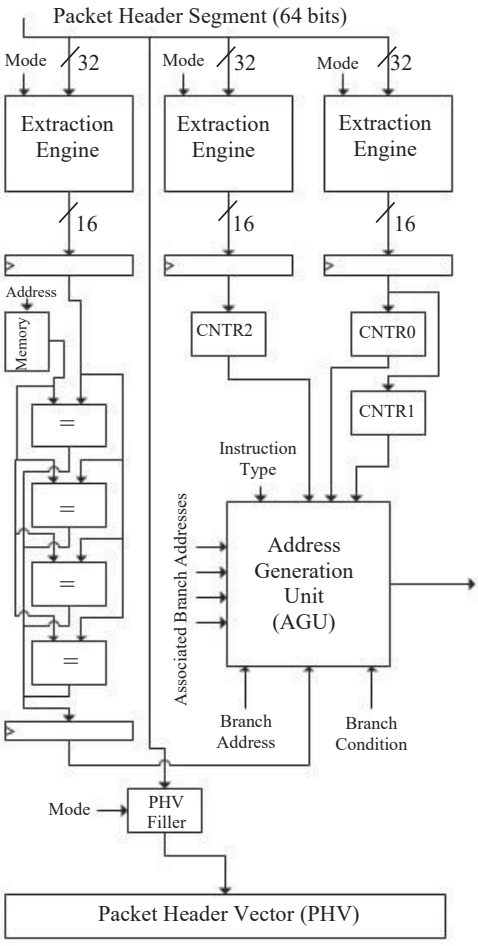

Fig. 1. High-level view of the Packet Parser in [8]

every clock cycle, the address generation unit was not designed for handling deep stacks of headers. Its small area footprint makes it ideal for switches and routers which need to parse a small number of headers stacked on each other. Therefore, when it comes to parsing packets with large number of stacked headers, the actual throughput deviates from the maximum achievable figure due to dead cycles which are caused by instructions which do nothing but check the intra- and inter- packet boundaries. It is essential to check these boundaries for correct operation of the parser. In the new architecture, we have shifted this boundary checking to hardware to relieve the programmer/compiler of this check and to eliminate the dead cycles.

In order to fully understand the issue of dead cycles, consider parsing of IPv4 header. This header contains a field called Internet Header Length (IHL) which specifies the size of the header in terms of number of 32-bit words. Once the word containing the Destination Address has arrived, the parser places it in a 32-bit container. At this point, the parser must check whether it should branch to the subroutine in charge of parsing the next header or it should continue with the IPv4 header and parse the available header options. This decision is made by means of a conditional branch instruction which checks whether the counter that was assigned the value of IHL has reached zero. The result is that on the subsequent clock cycle, the parser has to execute a no-operation (NOP) instruction. This results in a dead cycle which causes deviation from maximum throughput. Moreover, after each header option, this check needs to be performed again, which further decreases throughput. Once the presence of header options is resolved, the header must check whether the payload size is non-zero before proceeding to parsing of next header or forwarding of payload. This is because IPv4 packets without payload are also valid. Therefore, in the best case, parsing of IPv4 header contains two dead cycles. In the worse cases, there will be one dead cycle per option in addition to the two dead cycles just mentioned. The same problem exists with other variable-length headers such as TCP as well. We solve this issue by assigning the task of boundary checking to hardware and leaving the programmer or compiler with only the task of providing valid parse programs regardless of the size of header (in the case of variable-length headers) and size of the payload. The parse programs are written in such a way that they contain all the instructions required for parsing of optional header fields as well. Moreover, for protocols containing trailers, the instruction(s) in charge of parsing the trailer immediately follows the instructions which parse header.

Moreover, we have modified the functional units of the parser in a way that it can operate in true 64-bit mode. In the initial design, extraction engines operated on the lower 32-bit portion of the arrived header. The primary motivation for this design choice was that of hardware simplicity. But as synthesis results revealed, we have a large silicon real-estate that can be utilized in a more efficient manner for better performance while still being minimal compared to the most prominent parsers for SDN such as [1] and [2].

The initial design required precise programming. For instance, if a counter that has just been assigned a value needs to be checked, NOP instructions have to be placed by the programmer or compiler because the counter has not yet received the value due to the pipelined nature of the functional units. Moreover, movement of data between the pipeline stages is also instructed by software. In the new architecture, however, the hardware automatically stalls program flow when necessary. Moreover, all dataflow is organized and handled by hardware. This eases the task of the compiler.

III. Program Flow in Packet Parsing

In order to be able to design an efficient program sequencing logic, we must analyze the nature of program flow in packet parsing programs. A parse program is comprised of instructions, each of which is associated with one of the segments of packet header. An instruction specifies how its associated header segment must be placed into the containers within the PHV. Moreover, if the header segment in question contains fields indicating size of header, size of payload or next header, it instructs the extraction engines to extract these fields in order to update the internal state machine of the parser. Among the instructions comprising a parse program, there are different kinds of branches. One of the most common branches are the ones that jump to the code segment in charge of parsing the next

header. This branch is unconditional for fixed-size headers and conditional for variable-sized headers. Branches are sometimes required within a subroutine or code segment that parses a given header. For instance, presence of some header fields are signaled by flag fields which need to be evaluated for correct branches. If optional fields are not present, the instructions in charge of parsing them must be skipped. Branches are sometimes required based on the value of non-flag header fields. For instance, in Ethernet frames, if the value of EtherType field is 1500 or below, it should be interpreted as the size of the payload in bytes. Otherwise, the value should be used as the basis for determining payload type. Some protocols have a trailer after the payload section. In these cases, after parsing the header and handling the payload, the program flow should return to the trailer parsing code segment. Based on these observations, we classify branches in packet parsing in two broad categories of hardware and software branches. Hardware branches take place automatically in a high-priority manner while software branches occur using branch instructions. Branches required for maintaining intra- and inter-packet boundaries are taken care of by hardware. Therefore, they do not consume any execution time. As such, dead cycles will be eliminated. Other branches are explicitly specified by the programmer.

IV. A New Programmable Packet Parser

In this section, we present the new architecture. Fig. 2 illustrates a high-level view of the architecture. The main functional units are explained below:

A. Packet Header Vector Filler

Similar to the packet parser in [2], our programmer packet parser extracts header fields of the arrived header segment and places them in the Packet Header Vector (PHV) which contains 8-bit, 16-bit and 32-bit containers. We have also four 64-bit containers.

The PHV is 4 kilobits wide and is in fact the output of the parser and the input to the packet processing subsystem in which header fields undergo modification. The unit that places header fields in the PHV is called PHV Filler. It is programmable and has various modes of operation. For instance, given a 64-bit header segment, it can place eight 8-bit fields into the PHV at one clock cycle. Other combinations are possible as well.

B. Resource Monitors

The packet parser in [8] requires the compiler to precisely manage the counters within the parser. The compiler must keep track of internal counters that have been assigned a value and the ones that are idle. In this architecture, the programmer needs to just specify that a counter is needed for initialization. Resource Monitors, marked as RM in Fig. 2, keep track of all counters and automatically pick an idle counter for initialization.

C. Counters

Counters are the means by which inter-header and inter-packet boundaries are detected. There is one counter reserved for headers, marked as CNTR0 in Fig. 2, because at any instance in time, one header is being parsed but there are multiple payload counters, marked as CNTR1-CNTR8, because each next header is the payload of its preceding header. After reset, expiry of counters has highest priority in the program control logic. When a counter is expired, the branch type and branch condition code are ignored. Moreover, upon expiry of a counter, its expiry signal remains asserted for one clock cycle only, after which the counter will become available for initialization.

D. Extraction Engines

The extraction engines used in this architecture are similar to the ones used in [8]. However, they operate on 64-bit data rather than 32-bit data. Moreover, there are 5 extraction engines rather than 3. All extraction engines are identical but each one of them is dedicated for a specific purpose and each one of them can be independently programmed. The information extracted by these extraction engines are necessary to parse each header correctly. The extraction engines are integrated into the unit that uses their output. These units are the Next Header Resolve Unit, The Branch Catalyst, The Branch Condition Evaluator, Counter C0 and the Resource Monitor.

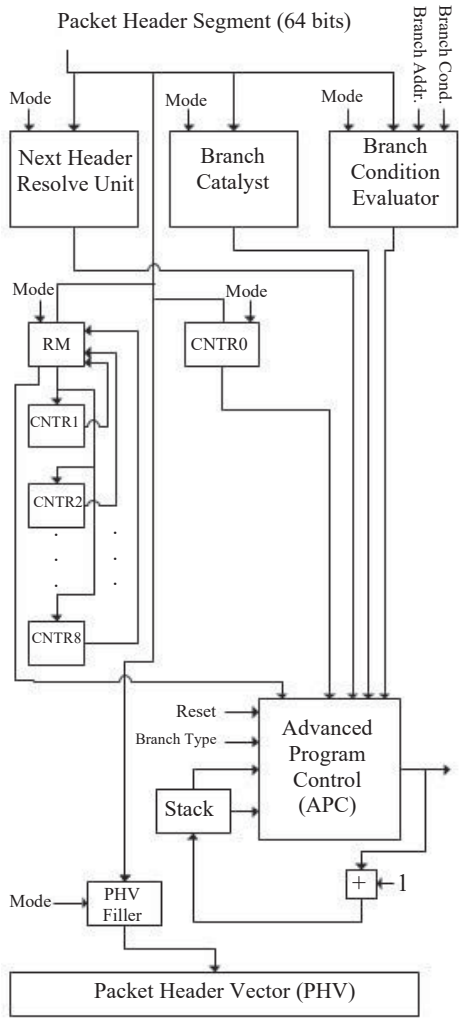

Fig. 2. High-level view of the new packet parser

E. Next Header Resolve Unit

The parser needs to know the next header and the address of the subroutine in charge of parsing the next header. For instance, in IPv4 the Protocol field indicates the next header. This unit determines the next header and provides the starting address of the subroutine in charge of parsing the next header. Fig. 3 illustrates a high-level view of this unit. As we can see, it has a built-in extraction engine that extracts the field containing the ID of the next header. After extraction, it will be compared against a set of expected values in parallel to resolve the next header. There are eight comparators operating in parallel. Associated with each comparand is its corresponding subroutine address. Comparands and associated memories are hosted on two distinct memory units. Each memory access provides eight comparands and their associated subroutine addresses. The number of comparands required for determining the next header may be larger than a memory word can accommodate. For this reason, the memory interface submodule is initialized with the number of times it is allowed to access the two memory units. To avoid wasted cycles, the entries should be filled in decreasing order of prevalence. There is also a default address that is provided to Next Header Resolve Unit in case none of the comparands results in a match. The Next Header Resolve Unit has status signals in-progress and ready to guide the Advanced Program Control in determining the address of the next instruction.

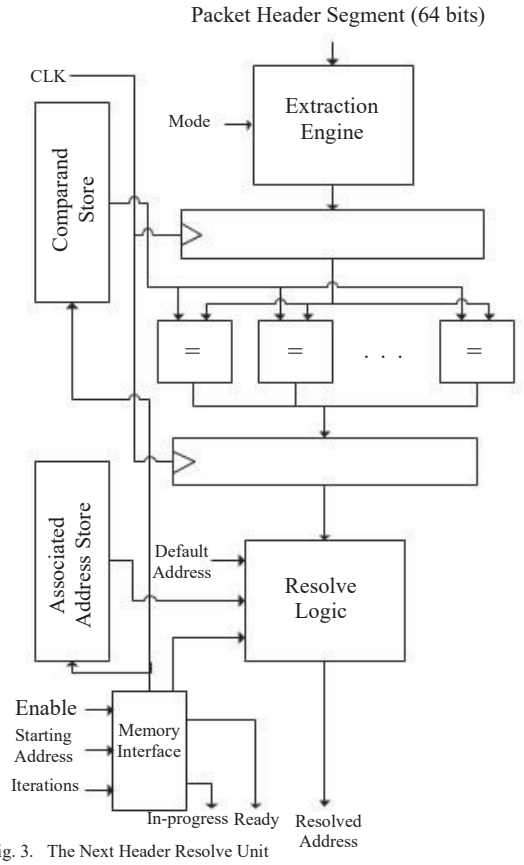

Fig. 3. The Next Header Resolve Unit

F. Branch Catalyst

Some headers have optional fields whose presence is indicated by flag bits. A very good example of such a header is that of Generic Routing Encapsulation (GRE). This header has three flag bits, each signaling the presence of its corresponding field. Therefore, there are 8 possibilities that need to be evaluated without hurting throughput. The purpose of the Branch Catalyst is to speed up branching by extracting the flag bits using a built-in extraction engine and comparing the extracted flag(s) against all valid values at once to resolve the branch in a real-time manner.

G. Branch Condition Evaluator

This unit extracts the programmer-specified segment of header using its built-in extraction engine and checks whether it evaluates to true according to the programmer-specified condition and reference value. The evaluation result is provided to the advanced program control unit to provide the address of next instruction.

V. The Advanced Program Control Unit

At each clock cycle, the advanced program control (APC) unit provides the address of the instruction to be executed on the upcoming cycle. It does so according to the control signals that it constantly monitors as well as branch type specified in the current instruction. Among the control signals, reset has the highest priority and it causes the APC to jump to the initial subroutine. After that, expiry of counters that have been assigned the value of payload or total packet size have the highest priority. They cause the APC to jump to the subroutine which parses the trailer or the initial subroutine if no trailer is present. Next high-priority signal is expiry of the counter holding header size. It causes branch to the next header. If no next header is present, the payload should be forwarded to a buffer for recombination with header fields that will undergo processing. If there is no payload, presence of trailers is checked. If none of the aforementioned signals are active, the branch type is considered. We support the following branch types:

A. Branch Catalyst

This branch type indicates that the address of the next instruction must be provided by the Branch Catalyst. This branch type is typically used when the header contains optional fields whose presence is signaled by flags.

B. Next Header

This branch type signals that at the upcoming cycle, the first instruction in the subroutine in charge of parsing the next header should be executed.

C. Next Header Function Call

This branch type is similar to the previous one except that address of the next instruction which starts parsing the trailer will be saved in a stack so that a return can be made later on. It suits protocols which contain a trailer.

D. Payload Forwarding

This branch type signals that there are no headers anymore and the payload of the packet must be forwarded to the common data buffer for recombination with headers that will undergo processing. Payload is not subject to parsing but

its size should be known to the parser to maintain the boundaries between packets. From the perspective of a parser, payload is anything that is not subject to parsing. For instance, in an Ethernet switch, the layer-3 header is already considered payload unless the switch is capable of performing layer-3 functionality.

E. End of Trailer

This branch type signals that parsing of current trailer is over. At this point the APC must check whether there are more trailers waiting for parsing.

F. Conditional Branch

This branch type specifies conditional branch based on the value of a programmer-specified field within packet header. The condition is specified by a three-bit field within the instruction.

VI. Evaluation

The architecture is implemented in VHDL. We have synthesized it on 28 nm UTBB FD-SOI technology in worst-case operating conditions (1.0V, ss, 125°C) using Synopsys Design Compiler J-2014.09-SP4. Power analysis was also performed in worst-case operating conditions at the supply voltage of 1.1V (tt, 125°C). We have verified that all timing constraints are met for operation at the frequency of 1.0 GHz. In order to make the comparison between this architecture and its predecessor more accurate, we have slightly modified the architecture in [8]. For instance, the original architecture used in [8] only had 2 counters to be assigned the payload or packet size. We have increased this number to 8. Moreover, we have increased the number of parallel comparators to 8. These modifications are mandatory for running the workloads that we will specify here.

Table I compares the two architectures from the perspective of power dissipation. As we can see, the enhancements come at the cost of a 36 percent increase in total power consumption. Table II contains synthesis results of this architecture and modified [8]. As we can see, the extra cost is a 42 percent increase in area. It should be noted that the increase in area and power dissipation is not only due to the APC but as a result of enhancements that ease the task of compiler as well.

Fig. 4 compares the total gate count required for sustaining aggregate throughput of 640 Gbps using current parser, the modified version of [8] and the parser in [2]. All of these parsers have been synthesized on 28 nm technology. We have derived the equivalent gate count by dividing the total area by the area of the smallest NAND2 gate which is 0.3264 μm^2 in the adopted technology. As we can see, despite the extra hardware, we still manage to provide substantial savings in area. While the parser in [2] requires 5.6 million gates, we can achieve the same throughput using only 1.6 million gates. This translates to a 71 percent reduction in area. Although modified version of [8] has the least number of gates, it should be noted that its extraction engines operate on the lower 32-bits of the incoming header. Therefore, its throughput is not always 64 Gbps. As a result, more instances of it are required for supporting aggregate throughput of 640 Gbps. More instances result in more gates.

As we have made architectural modifications for reducing latency when parsing variable-length headers, we would like to consider the case of parsing IPv4 header. Fig. 5 compares time required for parsing of IPv4 headers of different sizes using the two parsers. As we can see, with larger IPv4 headers, the original architecture lags behind considerably. In the case of largest possible IPv4 header which is 60 Bytes long, the speedup is 48 percent. Interestingly, in the new architecture, parsing IPv4 headers with sizes of 20, 24 and 28 bytes takes equal amount of time. The reason for this is the pipelined architecture of the Next Header Resolve Unit. It takes a number of cycles until the result is ready. The parser must stall while it could keep parsing the header if more fields were present.

In order to see the effect of architectural enhancements for better throughput when parsing header stacks, we consider the following workloads for evaluating the performance of our parser and comparing it with its predecessor:

TABLE I. Power Dissipation Comparison of Architecture Variants

	Modified [8]	Current Architecture
Internal Power	17.4 mW	23.3 mW
Switching Power	9.6 mW	14.2 mW
Leakage Power	8.2 mW	10.5 mW
Total Power	35.2 mW	48.0 mW

TABLE II. Area Comparison of Architecture Variants

	Modified [8]	Current Architecture
Number of ports	4491	4500
Number of nets	14423	7860
Number of references	878	529
Combinational area (μm^2)	17230	24089
Buf/Inv area (μm^2)	10757	12136
Noncombinational area (μm^2)	21114	30601
Total cell area (μm^2)	38344	54690

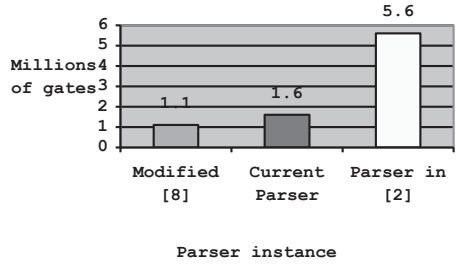

Fig. 4. Total Gate count Required for aggregate throughput of 640 Gbps

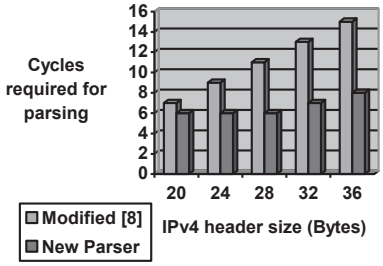

Fig. 5. Comparsion of cycle counts required for parsing IPv4 headers of different sizes

- 1-Ethernet-IPv4-TCP
- 2-Ethernet-IPv4(with two extensions)-TCP
- 3-Ethernet-MPLS-IPv6(with two extensions)-TCP
- 4-Ethernet-2xVLAN-2xMPLS-IPv6 (with two extensions)-TCP

We have written the parse programs for each of the headers present in the stacks above. Table III outlines number of clock cycles required for parsing of the workloads in both [8] and current architecture. We do not consider the payload in the evaluation. Therefore, cycle times reflect only the time required for parsing of headers. As we will see, with increase in the number of stacked headers, the number of dead cycles in [8] will be increased. Most savings come from workloads which contain variable-length headers. This is because the instruction can not place the arrived header into containers of suitable sizes until it makes sure that header or payload counter still has non-zero value. In parsing Ethernet there is one dead cycle, in IPv4 there are two dead cycles. In addition, each IPv4 option also adds one dead cycle because at the end of each extension, the check for header size needs to be done again. Parsing of IPv6 is more straightforward. There will be one dead cycle. However, parsing of IPv6 extension headers incur two dead cycles. Parsing of TCP also incurs two dead cycles.

TABLE III. COMPARISON OF TIME REQUIRED FOR PARSING THE WORKLOADS

Workload	Cycles required in modified [8]	Cycles required in current architecture	Improvement
1	25	21	16 %
2	28	22	21.42 %
3	44	35	20.45 %
4	50	40	20 %

VII. CONCLUSION

In this paper we presented a totally new program control logic for a recently-designed packet parser. We thoroughly studied the nature of packet parsing. We saw the required functional units and program control logic required for efficient parsing of variable-length headers and deep stacks of headers. We have also seen the cost of such enhancements. Once again, we have proved that the use of TCAM is not necessary for parsing and when TCAM-like functionality is desired, similar functionality can be provided by a small number of parallel comparators.

As for future work, we would like to work on some of the shortcomings we came across the new architecture. For instance, the pattern of many headers is such that within a 64-bit segment of the header, there are two 8-bit fields, one 16-bit field and one 32-bit field. The PHV Filler currently cannot fill the containers in one clock cycle if such a pattern is encountered. Providing it with such functionality helps in ultralow-latency environments.

REFERENCES

[1] G. Gibb, G. Varghese, M. Horowitz and N. McKeown, "Design principles for packet parsers," in *ACM/IEEE symposium on Architectures for networking and communications systems*, San Jose, 2013.

[2] 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," in *ACM SIGCOMM*, Hong Kong, 2013.

[3] 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*, Florianopolis, Brazil, 2016.

[4] Barefoot Networks, "The world's fastest and most programmable networks," [Online]. Available: https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[5] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, 2011.

[6] J. S. d. Silva, F.-R. Boyer and J. P. Langlois, "P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, 2018.

[7] P. Benáček, V. Puš, H. Kubátová and T. Čejka, "P4-To-VHDL: Automatic generation of high-speed input and output network blocks," *Microprocessors and Microsystems,* vol. Volume 56, pp. 22-33, February 2018.

[8] 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)*, Milan, 2018.

PUBLICATION
III

A custom processor for protocol-independent packet parsing

H. Zolfaghari, D. Rossi and J. Nurmi

Microprocessors and Microsystems, vol.72 (2019), 1-11

doi: 10.1016/j.micpro.2019.102910

Microprocessors and Microsystems 72 (2020) 102910

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A custom processor for protocol-independent packet parsing

Hesam Zolfaghari [a,*], Davide Rossi [b], Jari Nurmi [a]

[a] Electrical Engineering Unit, Tampere University, Tampere, Finland
[b] Department of Electrical, Electronic and Information Engineering, University of Bologna, Bologna, Italy

ARTICLE INFO

Article history:
Received 18 February 2019
Revised 12 August 2019
Accepted 12 October 2019
Available online 14 October 2019

Keywords:
Software defined networking
Programmable data plane
Packet parsing
Advanced program control

ABSTRACT

Networking devices such as switches and routers have traditionally had fixed functionality. They have the logic for the union of network protocols matching the application and market segment for which they have been designed. Possibility of adding new functionality is limited. One of the aims of Software Defined Networking is to make packet processing devices programmable. This provides for innovation and rapid deployment of novel networking protocols. The first step in processing of packets is packet parsing. In this paper, we present a custom processor for packet parsing. The parser is protocol-independent and can be programmed to parse any sequence of headers. It does so without the use of a Ternary Content Addressable Memory. As a result, the area and power consumption are noticeably smaller than in the state of the art. Moreover, its output is the same as that of the parser used in the Reconfigurable Match Tables (RMT). With an area no more than that of parsers in the RMT architecture, it sustains aggregate throughput of 3.4 Tbps in the worst case which is an improvement by a factor of 5.

1. Introduction

Software Defined Networking (SDN) is the key to deployment and management of complex networks. New network protocols are being proposed and standardized by both the industry and academia. The internals of the packet processing devices such as switches and routers can no longer accommodate the logic for the union of network protocols proposed and standardized so far. Instead, the data plane of the packet processing systems must be protocol-independent and programmable, so that they can provide the functionality required for network protocols of present and future. This requires thorough analysis of the operations incurred in processing of packets. A common concern for programmable and protocol-independent data plane is that of performance. However, as we will see, such systems can be on par with the conventional systems due to simpler architecture which allows for further optimizations.

Recently, there have been research efforts for realizing programmable data plane. These efforts span over the software and hardware abstraction layers. The P4 language, first introduced in [1], is a language for describing the forwarding behavior in packet processing systems. It describes packet processing in the form of match and action. In [2], an intermediate representation for programmable data plane is provided. It is a target-independent instruction set. It bridges the gap between high-level languages and hardware. A similar contribution is made in [3]. In [4], a solution for providing programmable packet scheduling in switches is provided. The most notable of research efforts for the hardware architecture of the programmable data plane are [5–7]. The architecture proposed in [6] is called Reconfigurable Match Tables (RMT). It contains the packet parser proposed in [5] and 32 stages of match and action. Any number of fields could be used to form a match key. The result of the match determines the processing that must be performed on header fields. The architecture in [6] has been commercialized and it is now the basis of the Protocol Independent Switch Architecture (PISA) used in Barefoot Tofino [8].

In this paper we are interested in the problem of packet parsing. There are countless papers in which FPGA-based parsers are proposed. [9–13] are just a few examples of such research efforts which achieve throughput on the scale of hundreds of Gigabits per second. However, it should be noted that these architectures achieve this throughput by means of operating on ultra-wide input due to their low frequencies. For instance, in [9], the input width is 2048 bits. Obviously, no transmission medium can transfer this amount of data at once. In addition, due to the sequential dependency of headers, each header must be parsed in turn in order to extract its fields and determine the following header. Therefore, there will be stalls and the flow of wide data could not be processed at every clock cycle unless the sequence of headers is known and remains unchanged. As a result, the actual throughput is far below the claimed figure. In addition, checksum verification which is needed in many packets must be calculated over

* Corresponding author.
E-mail addresses: hesam.zolfaghari@tuni.fi (H. Zolfaghari), davide.rossi@unibo.it (D. Rossi), jari.nurmi@tuni.fi (J. Nurmi).

https://doi.org/10.1016/j.micpro.2019.102910
0141-9331/

a number of cycles because the addition result must be accumulated and the clock cycle does not allow adding more than two operands. Therefore, it is best to read header data in smaller units and maintain a steady flow. In [10–13] packet parsers are synthesized from P4 definition of headers and parsers. This means that the synthesized architecture is protocol-specific. Having a fixed hardware which provides parsing capability for different headers by means of software is a far more robust solution and more compatible with the goals of SDN. This is the approach taken by the industry and hardly any high-end commercial packet processing system is based on FPGAs. Packet parsers in high-end commercial devices parse packets without buffering them in advance. We take the same approach and present a programmable packet parser for Terabit-scale packet parsing. It could be used for parsing any L2-L4 header. It could also parse application-layer headers unless the header requires deep packet inspection capabilities, in which case the throughput deteriorates.

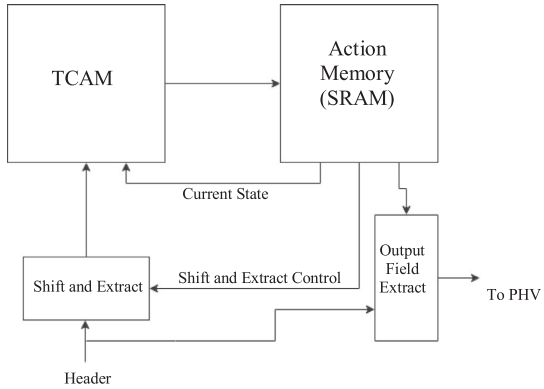

Fig. 1. Programmable parser in [6].

2. State of the art in packet parsing

In this section we investigate the architecture of two packet parsers in commercial use. The first one is the parser used in Intel FM5000/FM6000 Ethernet switches. The architecture of this switch is presented in detail in [14]. This switch series supports 640 Gbps aggregate throughput. The output of this parser contains a 40-bit vector of flags, checksum and an 88-byte bus containing header fields. This parser could be thought of as a state machine whose each iteration consumes successive 4-byte segments of the frame. The operation of this parser is specified by microcode. The parser is comprised of 28 slices each of which represents one of the transitions of the parser's state machine. Each slice receives as input 4 bytes of the frame and the status of the preceding slice. These two inputs are concatenated to form a 64-bit search key which is provided to a Ternary Content Addressable Memory (TCAM). The result of the match determines the action. As a result of the action, the state of the slice as well as the status flags are updated and frame data is placed on the 88-byte bus. We do not investigate this architecture any further for two reasons. It requires microcode for programming and it is not protocol independent. It is an Ethernet switch and the flags are specific to protocols such as Ethernet, IPv4 and IPv6.

We limit our focus to the parser used in [6]. It shares similarities such as use of TCAM and Action SRAM with the parser used in [14]. However, it is programmed using P4 and could be used to parse a wider range of headers. Internally, it is a state machine. As with any state machine, each state is associated with a number of actions. A schematic of this parser is illustrated in Fig. 1. As we can see, the incoming header data is subject to being shifted and extracted in order to form a search key. The amount of shift and the field to be extracted are determined by the current state of the parser. The search key is comprised of the extracted header field as well as the present state of the parser. Together, they form a 40-bit key which is presented to a TCAM. The outcome of the match determines the next state. When the next header arrives, the state determined in the previous cycle is the present state. The present state also specifies how the arrived header must be extracted and written to the Packet Header Vector (PHV) which is a 4096-bit vector comprised of 8-bit, 16-bit and 32-bit entries. Since the TCAM can lookup 40 bits at each clock cycle and the parser operates at clock frequency of 1.0 GHz, it provides 40 Gbps throughput. Programmability is achieved by filling in two separate memory units. The first one is a TCAM-based match table. The second table is the SRAM associated with the TCAM. When a search key is presented to the TCAM, the matching entry will point to a memory location in the action table so that the associated action is executed.

One of the shortcomings of the parser presented in [6] is that its Match-Action nature may result in the TCAM entries being filled in an inefficient manner. For instance, consider parsing of IPv4 headers. At a minimum, the IHL and the Protocol field must be combined to form a search key. IHL has 11 valid values (0x5-0xF). Assuming that the parser is programmed to recognize 8 different next headers, there will be 88 search keys in the TCAM for only 8 different next headers. A more efficient architecture could solve this problem.

TCAMs are powerful devices for searching. When a search key is presented to the device, all entries are searched in parallel. Therefore, the outcome of the search is ready in one clock cycle. TCAMs allow storing *don't care* bits. As a result, they are widely used for Longest Prefix Matching (LPM). Due to their robust search capabilities, they have large area footprint on the chip and the power dissipation figures are relatively high. With this in mind and considering the fact that no address lookup is required in packet parsing, we are motivated to consider alternative ways for determining the next state of the parser while still maintaining the programmability and protocol-independence.

It turns out that the most energy- and area-efficient way to accomplish this is to convert the state machine in question to a processor in which the TCAM and its functionality is replaced by a program control unit. Such a unit, regardless of its degree of complexity, will be far simpler and more energy-efficient than a TCAM. The role of such a unit is to ensure that the right instruction is executed when each segment of the header arrives. We should design parsing-specific branch types for the processor under development. Therefore, in this paper we will go through the process of converting a state machine for packet parsing to a processor. The main changes required are as follows:

- Converting state-specific actions to instructions, or more specifically, to operation codes for functional units
- Converting next state determination to next instruction's address resolution

3. Program flow in packet parsing

The aim of packet parsing is to extract the incoming header so that the packet processing system could perform the required processing on the extracted header fields. Therefore, the general rule is that the parser is not concerned with the content of the header. For instance, header fields such as destination address do not affect parsing. However, there are fields whose value have impact on parsing. For instance, in IPv4, the value of Internet Header Length

(IHL) specifies the length of the header. The parser must examine the value of such fields for correct operation.

In order to be able to design an efficient program control logic, we must analyze the nature of program flow in packet parsing programs. A parse program for a given header is comprised of a number of instructions, each of which is associated with one of the segments of the packet header. A header segment could be of 8-, 16- or 32-bit size. Each of the instructions specifies how its associated header segment must be placed into the containers within the PHV. Moreover, if the header segment in question contains fields indicating size of header, size of payload or next header, it instructs the extraction and use of these fields in order to perform branches within the parse program or branches to parse program for another header. One of the most common branches are the ones that jump to the code segment in charge of parsing the next header. This kind of branch occurs once the current header has been fully parsed. Branches are sometimes required within a subroutine or code segment that parses a given header. For instance, presence of some header fields are signaled by flag fields which need to be evaluated for correct branches. If optional fields are not present, the instructions in charge of parsing them must be skipped. Branches are sometimes required based on the value of non-flag header fields. For instance, in Ethernet frames, if the value of EtherType field is 1500 or below, it should be interpreted as the size of the payload in bytes. Otherwise, the value should be used as the basis for determining payload type.

4. A new programmable packet parser

In this section we present the architectural details of our novel programmable packet parser. Incoming packets go through a buffer called Incoming Packets' Buffer before being read by the parser. However, the packets need not be buffered in their entirety before the parsing can start. The parser is indeed a streaming parser. The aim of the buffer is to control the size of header data that each instruction operates upon. Moreover, different headers have different sizes. For instance, minimum-sized IPv4 header is comprised of 20 bytes while the Ethernet frame is made up of 14 bytes. IPv4 header could be read and operated upon in 4-byte units while for the Ethernet header it could be read in a sequence of two 4-byte and one 2-byte units. In the absence of such a buffer, header data must be read in the smallest unit common among different headers which is inefficient and throughput-degrading. The parser has two sets of output ports. The first set of ports are the ones through which the extracted header fields will be output to be written into the PHV. The second set of ports is used to forward the payload of the packet which is not subject to parsing to a buffer. As we could see in Fig. 2, the new packet parser is comprised of Header Parser and Payload Forwarder.

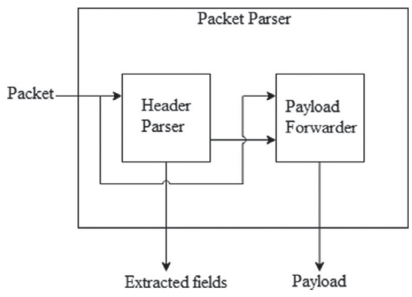

Fig. 2. The new packet parser.

4.1. Header parser

Header Parser is the entity in charge of parsing headers. It reads the header in 4-byte units from the Incoming Packets' Buffer. Due to the presence of multiple fields in each header segment, it is beneficial performance-wise to employ some form of parallelism. We have chosen explicit parallelism as the parallelism model. It is a software-defined form of parallelism and suits the Software Defined Networking paradigm very well. Protocol-independent networking hardware is unaware of protocols and cannot dynamically schedule the instructions at run-time. Instead, all instruction scheduling tasks must be handled by software. Explicit parallelism achieves this by explicitly specifying the required parallelism. Another benefit of such architectures is their simplicity and shorter design and verification time. Such architectures have wide instructions. Basically, there is an instruction field for each of the programmable functional units. The generic name for this class of processors is Very Long Instruction Word (VLIW). VLIW processors are discussed in detail in [15]. Our packet parser is based on the packet parsers proposed in [16-17].

The main components of the Header Parser are PHV Filler and Advanced Program Control Unit.

4.1.1. PHV filler

This unit places the arrived header segment into PHV entries. It has 16 modes of operation. Fig. 3 shows the input and output ports of the PHV Filler. It extracts the incoming header segment into any combination of 8-, 16- and 32-bit units in a way that the sum of the size of the units equals the size of the input header segment. The PHV is organized in 7 separate banks each connected to an output port of the PHV Filler. This separation allows writing to different locations in the PHV simultaneously. These banks together form the entire PHV. At any given instance in time, a maximum of 4 PHV banks receive data to be written.

The PHV Filler has no knowledge of protocols and header structures. It must be programmed for correct functionality. The Advanced Program Control Unit is in charge of ensuring that this unit receives the correct operation code.

4.1.2. Advanced Program Control unit (APC)

Advanced Program Control Unit is the brain of the system. It provides the right instruction for a given header word. It does so according to the control signals that it constantly monitors as well as the branch type specified in the current instruction. Among the control signals, reset has the highest priority and it causes the APC

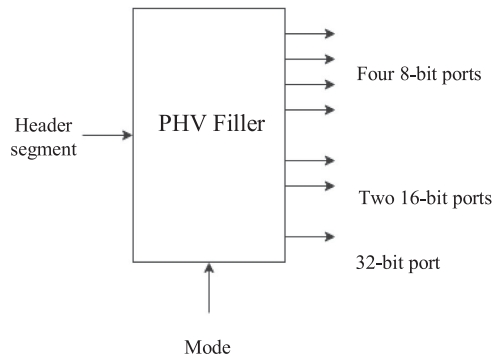

Fig. 3. PHV filler.

Type	Length	Data (Variable Length)
	. . .	
Type	Length	Data (Variable Length)
	. . .	
Type	Length	Data (Variable Length)
	. . .	

Fig. 4. A series of type-length-value sub-headers.

to jump to the initial subroutine. Internally, the APC consists of the following units.

4.1.2.1. Header counter. The Header Counter is a counter that is initialized with the size of the header. After initialization, it will count downwards with arrival of each header segment. Upon expiry, it signals an interrupt to the APC at which point parsing of next header or forwarding of the payload must begin.

4.1.2.2. Payload counters. There are four Payload Counters in the APC. They are used to hold two distinct values:

- Size of sub-headers
- Size of packet payload or the entire packet

What is meant by a sub-header is a part of a header which has a specified size and associated data. They do not have next header indicator because they are part of a main header which may or may not have next header indicator. For instance, IPv4 options are optional extensions to the IPv4 header. IPv4 options except the basic ones have a Length field specifying the size of the option. When a Payload Counter is used to hold the size of sub-headers, the stack is initialized with a return address so that once the option has been parsed, a return could be made to the return address at the top of stack. Alternatively, a Payload Counter is initialized with the size of the payload or entire packet if any of the headers has a field containing these values. The counter is initialized by the Header Parser but it is used by the Payload Forwarder.

4.1.2.3. Stack. The APC contains a stack to which the address of the current or following instruction can be pushed. If any of the Payload Counters expires and the Stack is non-empty, the address at the top of the Stack is popped and loaded into the Program Counter. As mentioned above, the stack is used in conjunction with the Payload Counters. Assume that a header contains a number of sub-headers each of which has a Type identifier, a Length indicator and the associated data. This is illustrated in Fig. 4. In the parse program, one of the instructions must be designated for extracting the Type and Length for branching to the right set of instructions and initializing a Payload Counter. The address of this instruction is pushed to the stack. Each time a sub-header is parsed, the Payload Counter expires and the address at the top of the stack is loaded into the program counter so that the next sub-header could be evaluated and parsed. This process continues until the main header is over.

4.1.2.4. Next Header Resolve Unit (NHRU). The parser needs to know the next header and the address of the subroutine in charge of parsing the next header. For instance, in IPv4 the Protocol field indicates the next header. This unit determines the next header and provides the starting address of the subroutine in charge of parsing the next header. Fig. 5 illustrates the internals of this unit. As

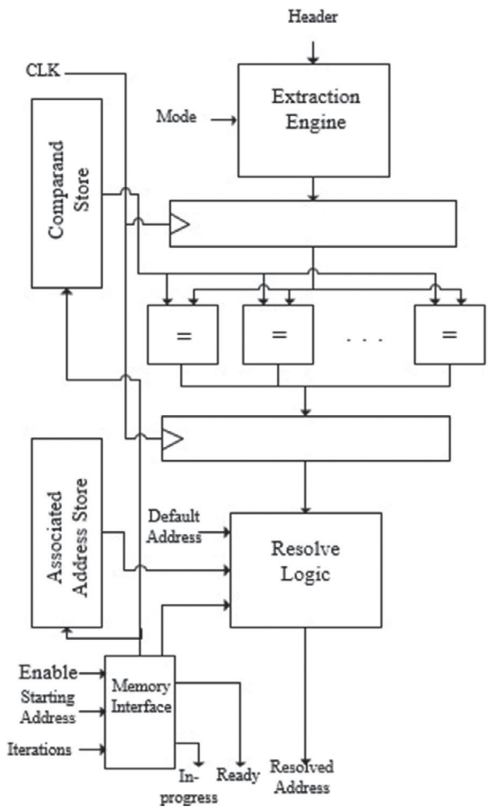

Fig. 5. The internal components of the NHRU.

we can see, there is a dedicated extraction engine for this unit. It extracts the field containing the identifier of the next header. The value of this field will be compared against a set of expected values in parallel to resolve the next header. We call this set of expected values a comparand set. In our architecture, each entry within the comparand set is 16 bits wide and the memory storing them can provide 8 entries in parallel. There are 8 comparators operating in parallel. Associated with each comparand is its corresponding subroutine address. Comparands and associated memories are hosted on two distinct memory units. The memory hosting associated addresses also provides eight entries in parallel. The number of comparands required for determining the next header may be larger than a memory word can accommodate at each address. In such a case, more than one memory address holds comparands. Similarly,

Table 1
Control signals monitored by the APC.

Control signal	Corresponding action
Reset	Jump to the first instruction
Expiry of header counter	Jump to subroutine in charge of parsing the next header or start payload forwarding
Expiry of any of the payload counters	Jump to the address at the top of stack if stack is non-empty else start payload forwarding
Branch type in the fetched instruction	Depending on the branch type, load the program counter with the value provided by the NHRU, BC or BCE

the associated addresses will occupy more than one memory entry. For this reason, the memory interface submodule is initialized with the number of times to access the two memory units until a match is found. To avoid wasted cycles, the entries should be filled in decreasing order of prevalence. In other words, the most expected values should be placed in the first comparand memory location that is accessed. For instance, a comparand set for resolving the next header of IPv4 is {0x0001, 0x0002, 0x0006, 0x0009, 0x0011, 0x0029, 0x0033, 0x0073}. They are all standardized values. The corresponding entry in the memory hosting associated addresses will have starting address for parsing of ICMP, IGMP, TCP, IGP, UDP, IPv6, AH and L2TP headers respectively. There is also a default address that is provided to Next Header Resolve Unit in case none of the comparands results in a match. The Next Header Resolve Unit has status signals *in-progress* and *ready* to guide the APC in determining the address of the next instruction.

4.1.2.5. Branch Catalyst (BC). Some headers have optional fields whose presence is indicated by flag bits. A very good example of such a header is that of Generic Routing Encapsulation (GRE). This header has three flag bits, each signaling the presence of its corresponding field. Therefore, there are 8 possibilities that need to be evaluated without degrading throughput. The purpose of the Branch Catalyst is to speed up branching by extracting the flag bits using a dedicated extraction engine and comparing the extracted flag(s) against all valid values at once to resolve the branch in a real-time manner. Architecturally, it is similar to the Next Header Resolve Unit, except that only one access is made to the memory units hosting comparands and associated memory addresses.

4.1.2.6. Branch Condition Evaluator (BCE). This unit extracts the programmer-specified segment of header using its built-in extraction engine and checks whether it evaluates to true according to the programmer-specified condition and reference value. The evaluation result is provided to the APC to resolve the branches.

The control signals based on which the APC operates are outlined in decreasing order of priority in Table 1.

Fig. 6 illustrates a high-level view of the internals of the Header Parser.

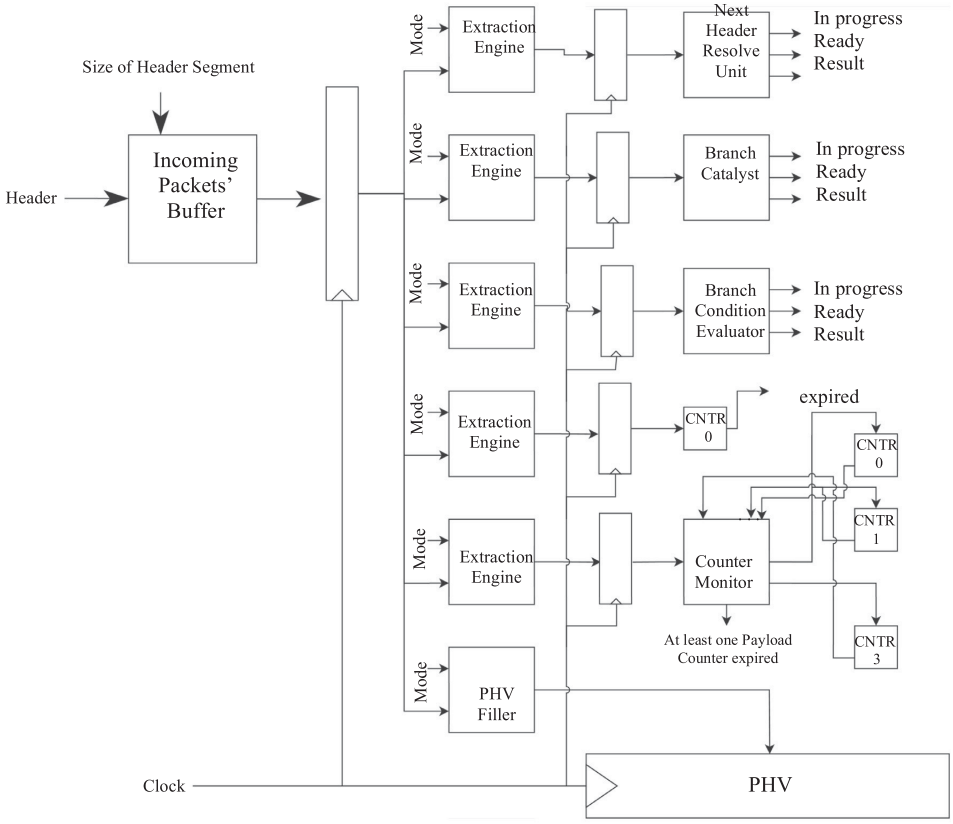

Fig. 6. Internals of the explicitly parallel header parser.

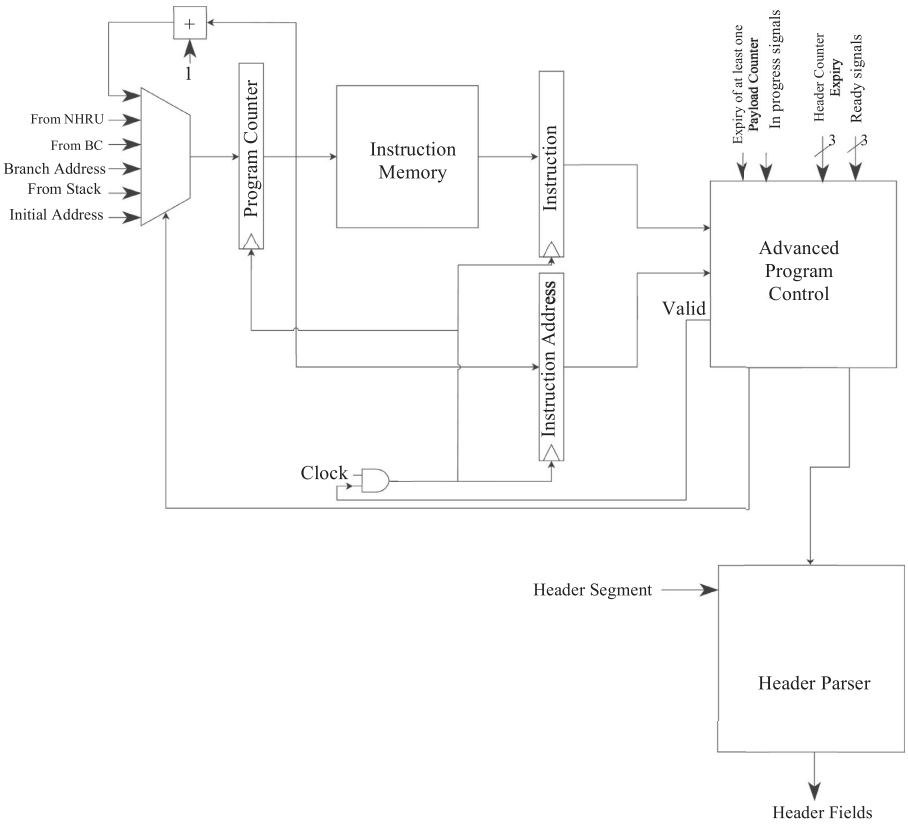

Fig. 7. Instruction pipeline of the parser.

4.2. Payload forwarder

The Payload Forwarder is the unit in charge of forwarding the payload of the packet into a buffer to which modified header fields will be written once the processing of the packet is done. The Payload Forwarder can read the Incoming Packets' Buffer in 32-Byte units which is 8 times wider than the widest unit the Header Parser could read. Payload Forwarding is more straightforward than header parsing. It uses the value of the Payload Counter which contains the remaining size of the payload to determine the size of data it requests from the buffer until the Payload Counter expires.

5. Pipelined organization

For operation at 2.0 GHz frequency, fetching and execution of instructions occur separately and in a pipelined manner. As each functional unit has its own field within the instruction, there is little need for instruction decoding. The functional units within the parser perform the execution stage. As we saw in Fig. 6, the internals of the packet parser are also pipelined. Therefore, the execution stage is made up of the following single-cycle stages.

5.1. Fetch header (FH)

At this stage, as much of the header as specified by the instruction is retrieved for operations at the upcoming stages.

5.2. Extraction (EX)

At this stage, the retrieved header segment is subject to extraction by extraction engines and PHV Filler.

5.3. Writeback (WB)

At this stage the extracted fields are written to the PHV.

Resolving the branches occurs at the beginning of the execution stage, i.e., at the FH stage. Branches have a penalty of one cycle.

Fig. 7 illustrates the instruction pipeline.

6. Instruction format

The instructions are 96 bits wide and comprised of 21 fields. Table 2 specifies the instruction fields, their width and use.

The instructions do not need decoding and can be fed to the packet parser once they have been fetched. A No Operation (NOP) instruction has value of zero for all extraction mode fields and the size of next header segment field.

7. Parsing example

7.1. Parsing Ethernet

In this section we illustrate how a parsing subroutine written in P4 could be mapped to and executed on our parser. Figs. 8 and

Table 2
Instruction fields.

Instruction field	Width (bits)	Use
Branch type	2	Specifies the type of branch
Branch condition	3	Specifies the branch condition for conditional branch instructions
Extraction mode_0	5	Specifies the extraction mode for the extraction engine dedicated to Next Header Resolve Unit
Extraction mode_1	5	Specifies the extraction mode for the extraction engine reserved for Branch Catalyst Unit
Extraction mode_2	5	Specifies the extraction mode for the extraction engine dedicated to Branch Condition Evaluator
Extraction mode_3	5	Specifies the extraction mode for the extraction engine reserved for Header Counter
Extraction mode_4	5	Specifies the extraction mode for the extraction engine designated for Payload Counters
Address_0	6	Next Header Comparands' Starting Address
Next header resolve iterations	7	Specifies the number of consecutive memory locations the Next Header Resolve unit may access starting from the initial address until a match is found
Address_1	6	Branch Catalyst Comparands' Address
Address_2	6	Header Counter Target Value Address
Address_3	6	Payload Counters' Target Value Address
Header segment size	2	Specifies the size of header segment to operate on. Valid sizes are 0 byte, one byte, two bytes and 4bytes
PHV filler operation mode	4	Determines how the incoming header should be broken down into fields.
PHV_address_0	4	The location of the extracted field in the first bank containing 8-bit entries
PHV_address_1	4	The location of the extracted field in the second bank containing 8-bit entries
PHV_address_2	6	The location of the extracted field in the third bank containing 8-bit entries, in the first bank containing 16-bit entries as well as in the bank containing 32-bit entries
PHV_address_3	6	The location of the extracted field in the fourth bank containing 8-bit entries as well as in the second bank containing 16-bit entries
Stack data in select	1	Selects whether the value to be pushed into the stack is the address of the current instruction or the following instruction
Stack push	1	Instructs a push operation to the stack
Unused	7	Currently unused

```
1   header Ethernet_h
2   {
3       bit<48>  dstAddr;
4       bit<48>  srcAddr;
5       bit<16>  etherType;
6   }
```

Fig. 8. Header defintion for Ethernet.

```
1   parser parse_ethernet
2   {
3       extract(ethernet);
4       return select(latest.etherType)
5       {
6           0x8100 : parse_vlan;
7           0x8847 : parse_mpls;
8           0x0800 : parse_ipv4;
9           0x86dd : parse_ipv6;
10          default: ingress;
11      }
12  }
```

Fig. 9. Source code for parsing Ethernet header in P4 language.

Fig. 9 illustrate the Ethernet header and parser definition in P4 respectively.

As we can see, the subroutine for parsing Ethernet, has the statement extract, which indicates that fields of this header must be extracted. On our parser, parsing of Ethernet is done using 4 instructions as shown in Fig. 10. The P4 source code also specifies selecting the parsing function for the next header based on the value of Ethertype field.

I_0 reads 4 bytes from the buffer at t_1 and writes it to a 4-byte container within the PHV at t_3. These 4 bytes are part of the 6-byte Destination MAC address. The next instruction, I_1 also reads 4bytes but writes them to two distinct 2-byte containers because the first 2 bytes belong to the Destination MAC address while the second 2 bytes belong to the Source MAC Address. The third instruction, I_2, reads the lower 4 bytes of the Source MAC Address and writes it to a 4-byte container. By now, contents of Destination and Source Address fields are in the PHV. Instruction I_3 whose branch type indicates a jump to the address provided by the NHRU, reads the 2-byte Ethertype field at t_4, extracts it at t_5 for writing to the PHV. At the same time, the field extractor in the NHRU extracts it for using it to find out the next header. At the same time, the memory address containing the comparands for Ethernet's next header is provided to the memory hosting the values. At t_6, Ethertype is written to a 2-byte container within the PHV. In parallel, Ethertype value is compared with the values at the memory address provided in the previous clock cycle. These values are 0x8100, 0x8847, 0x0800 and 0x86DD. They have been loaded into the memory in advance. At t_7, the comparison result is evaluated and the instruction address associated with the matching entry is selected. For instance, if the Ethertype had value of 0x86DD, the address of the subroutine containing the instructions for parsing IPv6 header is selected. At t_8, the address selected in the previous clock cycle is loaded into the program counter.

7.2. Parsing IPv4 header

Fig. 11 illustrates the definition of IPv4 header in P4 language. Parsing of IPv4 header is more complex than parsing Ethernet header because its length is variable. The fixed part contains five

Executed Instructions	Time							
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
I_0	FI	FH	EX	WB				
I_1		FI	FH	EX	WB			
I_2			FI	FH	EX	WB		
I_3				FI	FH	EX	WB	

Fig. 10. Instructions for parsing the Ethernet header.

```
1   header IPv4_h
2   {
3     bit<4>      version;
4     bit<4>      ihl;
5     bit<8>      diffserv;
6     bit<16>     totalLen;
7     bit<16>     identification;
8     bit<3>      flags;
9     bit<13>     fragOffset;
10    bit<8>      ttl;
11    bit<8>      protocol;
12    bit<16>     hdrChecksum;
13    bit<32>     srcAddr;
14    bit<32>     dstAddr;
15    varbit<320> options;
16  }
```

Fig. 11. Definition of IPv4 header in P4.

32-bit words. Up to ten 32-bit words may exist after the fixed words.

Fig. 12 illustrates the pipeline stages of the executed instructions for parsing minimum-sized IPv4 header. At t_2, the first instruction is in the Extract stage of the instruction pipeline. Parallel to the extraction performed by the PHV Filler, the extraction engine in the Header Counter extracts the IHL field and the extraction engine in the Payload Counter extracts Total Length. At t_3, the extraction performed by the PHV Filler is written to the PHV. Furthermore, both Header Counter and Payload Counter are initialized. Therefore, starting from t_4, their value will be decremented based on the size of the header segment read from the buffer at each clock cycle. Again, at t_4, header fields in the second word of the header are written to the PHV. At the same time, the value of the Protocol field is extracted by the NHRU in order to start resolving the next header. The third, fourth and fifth header words are written at t_5, t_6 and t_7 respectively. Execution of I_4 causes expiry of the Header Counter. As a result, at t_8, the first instruction from the subroutine for parsing the next header must be fetched.

Table 3
Time required for parsing of different headers.

Header	Shortest parsing time (cycles)	Longest parsing time (cycles)
IPv4	8	18
IPv6	13	13
MPLS	4	4
Ethernet	7	7
TCP	8	18
VxLAN	5	5
GRE	4	12
L2TP	10	13

8. Experimental results and discussion

In this section, we evaluate the performance of the parser in terms of how well it could parse individual headers as well as stacks of headers when operating at a clock frequency of 2.0 GHz. After this evaluation, we present the implementation details of the parser.

8.1. Parsing individual headers

We have chosen a number of commonly used protocols for this purpose. Table 3 contains the time taken to parse the chosen headers.

The difference in parsing time for some headers is due to variable length of some headers such as GRE. For fixed headers such as IPv6, parsing time is constant. There are interesting observations to make from Table 3. For instance, maximum-sized L2TP header contains 128 bits and it takes 13 cycles to fully parse this header. This is the same duration required for parsing of IPv6 header that consists of 320 bits. The reason for this is the fixed nature of IPv6 header. L2TP header is a variable-sized header in which existence of some fields are indicated by flags located in the first 16 bits of the header. The parser must extract these flags and use them to make the right branch in the program. Fig. 13 illustrates the instruction pipeline diagram for parsing the minimum-sized L2TP header. At time instance t_3 the Branch Catalyst unit starts comparing the flags with programmer-specified values. Comparisons are

Executed Instructions	Time							
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
I_0	FI	FH	EX	WB				
I_1		FI	FH	EX	WB			
I_2			FI	FH	EX	WB		
I_3				FI	FH	EX	WB	
I_4					FI	FH	EX	WB

Fig. 12. Instruction pipeline diagram for parsing IPv4 header.

Executed Instructions	Time									
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
I_0	FI	FH	EX	WB						
I_1							FI	FH	EX	WB

Fig. 13. Instruction pipeline diagram for parsing L2TP header.

Executed Instructions	Time												
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
I_0	FI	FH	EX	WB									
I_1		FI	FH	EX	WB								
I_2			FI	FH	EX	WB							
I_3				FI	FH	EX	WB						
$I_{\text{Next Header}}$										FI	FH	EX	WB

Fig. 14. Instruction pipeline diagram for parsing Ethernet and branching to its next header.

performed in parallel in order to minimize wasted cycles. At t_6 the correct instruction is fetched.

8.2. Parsing header stacks

Next, we consider a number of header stacks. We have chosen the following header stacks:

1-Ethernet-IPv4-TCP
2-Ethernet-IPv6-TCP
3-Ethernet-IPv6-ICMPv6 (Destination unreachable)
4-Ethernet-MPLS (three stacks)-IPv6-UDP

The time required for parsing these stacks is presented in Table 4. Workload number 4 results in the smallest throughput value which is slightly over 27 Gbps.

Similar to the case of parsing individual headers, we can observe variations in throughput. Fig. 14 illustrates the instruction pipeline diagram for parsing the Ethernet header and a potential next header.

The next header indicator is located in the last 16 bits of Ethernet header. At t_4 the instruction in FH stage contains branch type of next header, therefore the fetched instruction has to be flushed. At t_6 the process of finding the next header begins. At t_9 the instruction in the subroutine in charge of parsing the next header is fetched. Conversely, headers such as that of IPv6 have different characteristics. Fig. 15 illustrates the instruction pipeline diagram for parsing IPv6 and branching to the subroutine in charge of parsing the header following IPv6 header. As can be seen, there is only one wasted cycle. The reason for this is that the Next Header field is located at the second word of the IPv6 header and by the time the header is entirely parsed, the address of next header subroutine has been resolved.

Table 4
Time required for parsing of four different header stacks.

Protocol stack	Total size of headers (bits)	Parsing time (cycles)
1	432	25
2	592	28
3	656	35
4	592	43

Table 5
Area results for different components of the parser.

Component	Area (μm^2)	Area (Gate count)
Advanced program control	342	698
Header parser	3800	7761
Payload forwarder	1393	2845
Parameter memories	30,864	63,002
Packet Header Vector	15,976	32,631
Instruction memory	93,052	190,057
Total area	145,427	358,838

Table 6
Power dissipation of different components of the parser.

Component	Power dissipation (mW)
Advanced program control	1
Header parser	9
Payload forwarder	4
Parameter memories	90
Packet Header Vector	54
Instruction memory	291
Total power dissipation	449

8.3. Implementation details

The architecture is implemented in VHDL. We have synthesized it on 28 nm UTBB FD-SOI technology in worst-case operating conditions (1.0 V, ss, 125 °C) using Synopsys Design Compiler J-2014.09-SP4. Power analysis was also performed in worst-case operating conditions at the supply voltage of 1.0 V (ss, 125 °C). We have verified that all timing constraints are met for operation at the frequency of 2.0 GHz.

Tables 5 and 6 present area and power dissipation results for the components comprising one parser instance. The total area consumed by 16 parser instances in the RMT architecture is 1.7 mm^2 in 28 nm ASIC technology.[1] Total gate count is 5.6 M of which over 1 M is contributed by the TCAM [6]. 16 parser instances sustain aggregate throughput of 640 Gbps. We must now determine how many instances of the new programmable parser are required for sustaining the aggregate throughput of 640 Gbps. A distinctive feature of this parser is that it provides variable latency when parsing different headers. Let's take the most demanding workload from Table 4 in which parsing of the headers in the last

[1] Source: Private correspondence with designers of RMT parser.

Executed Instructions	Time														
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
I_0	FI	FH	EX	WB											
I_1		FI	FH	EX	WB										
I_2			FI	FH	EX	WB									
I_3				FI	FH	EX	WB								
I_4					FI	FH	EX	WB							
I_5						FI	FH	EX	WB						
I_6							FI	FH	EX	WB					
I_7								FI	FH	EX	WB				
I_8									FI	FH	EX	WB			
I_9										FI	FH	EX	WB		
$I_{\text{Next Header}}$												FI	FH	EX	WB

Fig. 15. Instruction pipeline diagram for parsing IPv6 and branching to its next header.

workload takes 43 cycles which is roughly equal to 22 nanoseconds. This translates to a throughput of about 27 Gbps which is the least achievable throughput value compared to the other workloads. This throughput figure is more than enough for the aggregate traffic from two 10 Gbps ports. Therefore, in a switch with 64 10 Gbps ports, 32 parser instances are enough. This analysis is based on extreme conditions but in order to provide a guaranteed lower bound on the throughput, we do not consider more optimistic workloads.

When calculating the total area of multiple instances of our parser, we must bear in mind that not all the components need to be replicated. The parameter memories and instruction memory will be shared by all the parser instances. Each parser instance will have independent access. In our architecture, the TCAM is replaced by the APC. Since it uses some of the functional units required for parsing, we take the sum of the area of both in order to compare the resulting value with that of TCAMs. In our architecture, the total area of units in charge of determining the next state is 163,408 μm^2 which translates to 334 K gates.[2] This is 66% reduction in area of next state resolving logic. Total area of parsers in this organization is 0.8 mm^2 or 1.6 M gates. Compared to 1.7 mm^2, this is a 53% reduction in area. If we use the area required by the parser instances in RMT, we could fit 128 parser instances. Together, they support aggregate throughput of 3.4 Terabit per second.

Since there is no mention of RMT parser's power dissipation figure, it is not possible to perform a precise comparison for power dissipation. However, due to large difference in area and elimination of TCAM, the power savings must also be noticeable.

9. Conclusion and future work

In this paper we presented a novel programmable packet parser that does not rely on a TCAM to provide the required functionality. We designed all the functional units required for protocol-independent packet parsing. Our design of a packet parsing-oriented program control unit resulted in 53% saving in area compared to the parser used in the RMT architecture.

We saw that different headers exhibit different behaviors and affect the throughput of the parser differently. For some headers, a protocol-independent parser cannot provide the same throughput as a dedicated parser and that is the cost of programmability. However, the benefits of programmability and protocol independence outweigh the occasional wasted cycles. Moreover, considering the fact that packet processing resources such as lookup tables are shared among packets arriving from different ports, and that packets have to wait for their turn to use shared resources, there is no point in maintaining maximum possible throughput in packet parsing as there will be cycles in which packets have to wait during packet processing.

As for future work, we would like to investigate the throughput gain achievable by not binding the parser instances to ports and instead assigning the arrived packet to a free packet parser instance. In addition, the decoupling of header parsing and payload forwarding logic allows overlapping of payload forwarding with parsing of a new packet's header. This results in more efficient use of system resources and improvement in throughput. The exact amount of improvement is dependent on traffic patterns and must be investigated. Another area which could further be explored is enhancing the throughput of a single parser instance. This is possible by running the parser at higher frequencies. In order to scale the frequency noticeably further, we must optimize the code and increase the depth of the instruction pipeline as well as the registers in the functional units. Another way to increase the throughput is to read header words in units larger than 32 bits.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

We hereby express our gratitude to professor Nick McKeown from Stanford University for providing us with the details on the area of the parser used in [6]. We would also like to thank Mr. Glen Gibb for providing us with invaluable comments and insight. We acknowledge the Finnish DELTA network and The Pekka Ahonen Fund for providing partial funding for this project.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, D. Walker, P4: programming protocol-independent packet processors, ACM SIGCOMM Comp. Commun. Rev. 44 (3) (2014) 87–95.

[2] The gate count is obtained by dividing the area by the area of the smallest NAND2 gate in the deployed 28 nm ASIC library.

[2] M. Shahbaz, N. Feamster, The case for an intermediate representation for programmable data planes, in: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, ACM, 2015, June, p. 3.
[3] 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, ACM, 2013, August, pp. 127–132.
[4] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.T. Chuang, A. Agrawal, N. McKeown, Programmable packet scheduling at line rate, in: Proceedings of the 2016 ACM SIGCOMM Conference, ACM, 2016, August, pp. 44–57.
[5] G. Gibb, G. Varghese, M. Horowitz, N. McKeown, Design principles for packet parsers, in: Architectures for Networking and Communications Systems, IEEE, 2013, October, pp. 13–24.
[6] P. Bosshart, G. Gibb, H.S. Kim, G. Varghese, N. McKeown, M. Izzard, M. Horowitz, Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN, ACM SIGCOMM Comput. Commun. Rev. 43 (4) (2013) 99–110.
[7] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, S. Licking, Packet transactions: high-level programming for line-rate switches, in: Proceedings of the 2016 ACM SIGCOMM Conference, ACM, 2016, August, pp. 15–28.
[8] Barefoot Networks, "The World's Fastest and Most Programmable Networks," [Online]. Available: https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.
[9] M. Attig, G. Brebner, 400 Gb/s programmable packet parsing on a single FPGA, in: 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, IEEE, 2011, October, pp. 12–23.
[10] J. Santiago da Silva, F.R. Boyer, J.M. 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, ACM, 2018, February, pp. 147–152.
[11] P. Benáček, V. Puš, H. Kubátová, T. Čejka, P4-To-VHDL: automatic generation of high-speed input and output network blocks, Microprocess. Microsyst. 56 (2018) 22–33.
[12] Benácek, P., Puš, V., & Kubátová, H. (2017). Automatic generation of 100 Gbps packet parsers from P4.
[13] J. Cabal, P. Benáček, L. Kekely, M. Kekely, V. Puš, 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, ACM, 2018, February, pp. 249–258.
[14] Intel® Ethernet Switch FM6000 Series Product Brief https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.
[15] J.A. Fisher, P. Faraboschi, C. Young, VLIW processors: once blue sky, now commonplace, IEEE Solid-State Circuits Mag. 1 (2) (2009).
[16] H. Zolfaghari, D. Rossi, 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), IEEE, 2018, July, pp. 1–4.
[17] H. Zolfaghari, D. Rossi, J. Nurmi, 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), IEEE, 2018, October, pp. 1–6.

Hesam Zolfaghari is a doctoral student at Tampere University. His-research interest is design of programmable and protocol-independent packet processors for Software Defined Networking with special focuses on low on-chip area, low power dissipation and minimized packet processing latency. This includes design of abstraction layers starting from the instruction set all the way down to the microarchitecture of both packet parsing and packet processing subsystems within high-performance switches and routers.

Davide Rossi, received the PhD from the University of Bologna, Italy, in 2012. He has been a post doc researcher in the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi" at the University of Bologna since 2015, where he currently holds an assistant professor position. His research interests focus on energy efficient digital architectures in the domain of heterogeneous and reconfigurable multi and many-core systems on a chip. In these fields he has published more than 80 paper in international peer-reviewed conferences and journals.

D. Sc. (Tech) **Jari Nurmi** is Professor at Tampere University (formerly Tampere University of Technology), Finland since 1999. He works on embedded computing, wireless localization, and software-defined radio/networks. He held various positions at TUT 1987–1994 and was the Vice President of SME VLSI Solution Oy 1995–1998. Since 2013 he is also a partner at research spin-offs. He has supervised 25 PhD and 138 MSc theses, and been opponent/reviewer of 40 PhD theses worldwide. He is senior member of IEEE, and in steering committees of three international conferences (chairman in two). He has edited five Springer books, and published over 350 international publications.

PUBLICATION
IV

An Explicitly Parallel Architecture for Packet Processing in Software Defined Networks

H. Zolfaghari, D. Rossi and J. Nurmi

2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), 1-7

doi: 10.1109/NORCHIP.2019.8906959

An Explicitly Parallel Architecture for Packet Processing in Software Defined Networks

Hesam Zolfaghari
Electrical Engineering Unit
Tampere University
Tampere, Finland
hesam.zolfaghari@tuni.fi

Davide Rossi
Department of Electrical, Electronic, and Information Engineering
University of Bologna
Bologna, Italy
davide.rossi@unibo.it

Jari Nurmi
Electrical Engineering Unit
Tampere University
Tampere, Finland
jari.nurmi@tuni.fi

***Abstract*— Programmable data plane is a key enabler of Software Defined Networking. By making networking devices programmable, novel networking services and functions could be realized by means of software running on these devices. In this paper, we present a lightweight packet processor that could process the packets on the fly as they arrive. As we will see, the area of this packet processor is smaller than a packet parser employing Ternary Content Addressable Memory. As an added benefit, the designed packet processor could also reduce the traffic to the lookup tables on the chip. Moreover, its use is not limited to switches and routers. It could also be used in the Network Interface Cards and offload packet processing tasks. Despite its packet processing capabilities, packet processor instances required for sustaining aggregate throughput of 640 Gbps have area equivalent to the packet parser instances in the Reconfigurable Match Tables Architecture.**

Keywords*— *On-the-fly packet processing, programmable data plane, Explicit Parallelism, Packet Processor

I. Introduction

Networking gear has traditionally been fixed-function and tied to a specific set of protocols. Recently, the benefits of programmable data plane have forced the industry and academia to design programmable networking devices. Commercially available programmable packet processing systems are Barefoot Tofino [1], Intel FlexPipe [2] and Netronome Agilio [3]. Benefits of programmability in the data plane are as follows:

- Support for newer networking protocols
- Simpler networking devices
- Telemetry and network troubleshooting
- Offloading computation to the network

With programmable data plane, support of different network protocols is made possible by means of software updates. The architectures presented in [4], [5] and [6] are all programmable. All it takes for a device to provide new functionality is to run the required software. Although it sounds counter-intuitive, programmable networking devices are architecturally simpler considering the range of functionality that they could provide. If this range of functionality were to be provided using conventional design principles, a large amount of protocol-specific state must be permanently stored.

One of the strongest driving forces of deploying programmable data plane is the rich set of telemetry and network troubleshooting facilities. In [7], [8] and [9] network state is embedded into the packets and later on used for diagnostics and telemetry. With the data plane being programmable, any internal state of the device which is of significance to the network could be written to the packet and later on used for analysis. In fact a network administrator-defined header could be inserted into the packet to carry such state. Finally, the latest trend in programmable data plane is in-network computing. In this paradigm, computational tasks are offloaded by networking devices. For instance, in [10], the programmable network device is used as a neural network accelerator.

The packet parsing and packet processing subsystems are the main components of networking devices such as switches and routers. Earlier, we designed a programmable packet parser in [11] and [12]. In this programmable packet parser, the output format is identical to that of the packet parser in [1], [4] and [5]. The internal architecture of our parser, however, is fundamentally different. Rather than a state machine that relies on a Ternary Content Addressable Memory (TCAM) to derive the next state, our parser uses a program control unit. In this paper, we have augmented the parser with functional units for packet processing. As a result, it is a Packet Processor and we refer to it as such in this paper. As we will see, for an aggregate throughput of 640 Gbps, the total area of our Packet Processor instances equals that of the packet parsers in Reconfigurable Match Tables (RMT) architecture while providing wide range of packet processing capabilities in addition to packet parsing.

This paper is organized as follows. In section II the motivation for this architecture is elaborated. In section III the architecture is presented followed by a packet processing example in section IV. Finally, the evaluation results will be presented in section V.

II. Motivation

The set of packet processing operations that must be performed on a packet can be categorized under two classes:

- Standardized operations
- Deployment-specific operations

Standardized operations are the ones that must be performed on a packet because the protocol standard instructs to do so. For instance, in IPv6, a packet's Hop Limit must be examined and if its value is zero, the packet must be discarded. The packet parser could do this as the packet is arriving. However, the treatment for a packet that has a given Destination Address is deployment specific. For instance, it could lead to the packet being discarded or the packet being forwarded to all egress ports. The packet parser in its conventional form could not be involved with this level of packet processing unless it has a lookup table at its use, in which case it could do almost any deployment-specific packet processing.

Our packet parser reads the header of the incoming packet in units of maximum 4 bytes wide. Throughout the time of a packet's arrival at which point the packet parser is involved, many packet processing tasks could be performed. For

instance, validity checking of a packet and field modifications that do not depend on the outcome of table lookups could be already done. By doing so, by the time the packet has fully arrived, packet processing has been partially done. This greatly reduces the time required for processing of packets, as the packets spend less time in the packet processing device. Processing of header fields during the course of parsing comes at a negligible cost.

In addition, the parsers in the RMT architecture have their output multiplexed into a serial pipeline. This means that if new packets are constantly arriving through the ingress ports, the output of the parser has to wait until its turn for entering the pipeline comes. With 16 parser instances and one packet processing pipeline, this waiting could last as long as 16 cycles during which a lot could be done. By starting packet processing already at the time the packet starts arriving, this waiting time is exploited for packet processing.

In order to enhance the chance of being able to perform all the required processing for a packet, the concept of packet flows is fully exploited. The parser is equipped with tiny binary lookup tables that contain the lookup result for the most recently arrived packets. The lookup result is retrieved from the main processing pipeline which contains lookup tables. If a match is found in the tiny lookup tables, the corresponding action that could not have otherwise been possible could be performed in the Packet Processor. In this case the packet could bypass the main packet processing pipeline. Even if a match is not found, the packet enters the main pipeline in partially processed form which means that it requires less processing time.

III. Architecture

The packet processing system comprises a programmable packet parser and a programmable packet processor. The two entities work in harmony for on-the-fly packet processing. The architecture of the programmable packet parser is discussed in detail in [12]. It extracts the header fields of the incoming packets and writes them to a storage location called Original Header Word (OHW) entries. These entries are marked R16-R31. The programmable Packet Processor does not wait for the packet parser to fully parse a packet before it starts processing of the packet. It starts as soon as an entry is written to the OHW. Fig. 1 illustrates a high-level view of the Packet Parser and the Packet Processor.

The packet processor is comprised of 4 Groups of Functional Units (GFU). Each GFU has 8 functional units which are sufficient for processing a number of header fields. Together, these GFUs perform all the processing required for the headers of an arriving packet. Fig. 2 illustrates a GFU. As we can see, each GFU contains two ALUs, two shifters, a Merge Unit, a Load and Store unit, a Lookup Table and a Parameters Unit. The Load and Store Unit, Lookup Table and Parameter Unit are stateful units. For instance, it is possible to store the number of packets associated with a specific search key in the memory located in the Load and Store Unit. All the other units are stateless units.

The complete instruction set contains 64 instructions. Depending on their semantics and the functional unit which executes them, there are 8 classes of instructions. Instruction classes have been outlined in TABLE I.

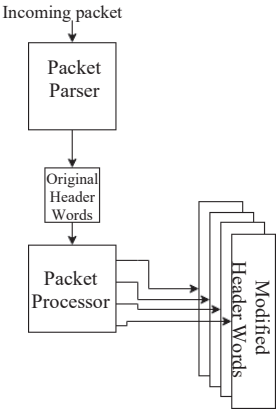

Fig. 1. High-level view of the Packet Parser and the Packet Processor

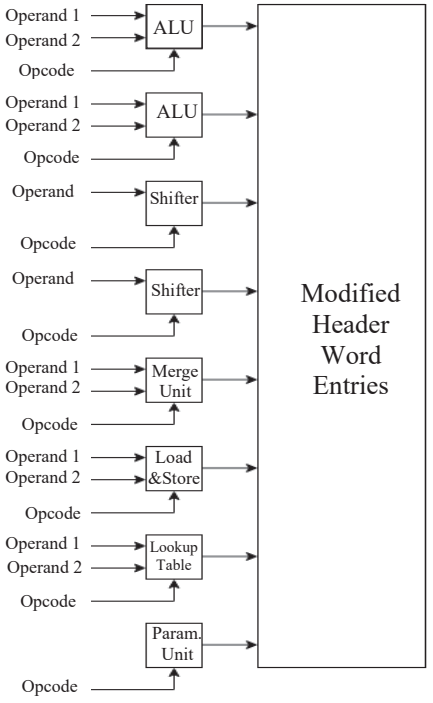

Fig. 2. A Group of Functional Units (GFU)

Merge Word instruction is used to merge the corresponding bytes of a 32-bit word into a target word in the specified manner. For each byte in the target word, there is a choice of two from the corresponding byte position of the two source words. It is used for forming search keys and finalizing the words of a header. Save Search Key in conjunction with Load and Store instructions allow to save a key generated during processing of a packet and associate data with it so that they could be later on retrieved if the same search key is encountered. Bit Extract instruction is useful for reading flag bits such as Explicit Congestion Notification (ECN) bits. Load Parameter is used for inserting four different types of values:

TABLE I. INSTRUCTION SET

Instruction Class	Operations	Mnemonic
Condition Evaluation	Check for equality, greater than or less than conditions	CEQ, CG, CL
Arithmetic and Logic	Arithmetic and Logic operations	ADD, SUB, AND, OR, NOT, MOVE, XOR, NAND
Shift	Logical left and right in 1-, 4-, 8- and 16-bit units	SHL1, SHL4, SHL8, SHL16, SHR1, SHR4, SHR8, SHR16
Merge	Merge the corresponding bytes of two words	MERGE0000, MERGE0001, …, MERGE1111
Lookup	Search key lookup and lookup table maintenance	Save Search Key, Lookup
Bit Extraction	Extract the specified bit within a byte	BE0, BE1, … , BE7
Load and Store	Load and Store operations	Load, Store
Load Parameter	Insert header template, processing operand, status or field value	Load Param(i)

- Header templates such as header words for Internet Control Message Protocol (ICMP)
- Unique values for fields intended to contain unique values such as a connection identifier
- System status such as status of queues
- Operands for functional units

The functional units receive up to two operands from four different sources:

- Modified Header Word (MHW) entries in the GFU (R0-R15)
- Original Header Word (OHW) entries (R16-R31)
- Certain entries of Modified Header Word in other GFUs (R32-R63)
- An immediate value

There is one OHW space which is shared by all GFUs. All operands from the OHW entries are subject to extraction as a result of which an 8- or 16-bit field within the 32-bit word is extracted and zero-extended to 32 bits prior to execution. Alternatively, the selected 32-bit OHW entry could be operated upon without any field extraction.

In addition to the operands, the functional units receive a one-bit predicate input from the 16-bit condition status (r0-r15) to which the result of condition evaluation instructions and lookup result is written. Each GFU has an independent condition status word. The predicate input is used for conditional execution. Fig. 3 illustrates the inputs to the ALU.

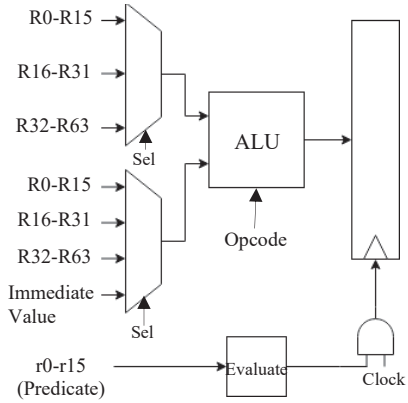

Fig. 3. Inputs to the ALU

There is an MHW space per GFU. MHW storage is a multiport register file in which each functional unit has its own writing space. This means that no other functional unit could write to the range of locations reserved for the functional unit in question. Two entries are reserved per functional unit within the GFU. In contrast, any functional unit could read from the writing space of both itself and all other functional units. If the nature of a given header allows, each functional unit produces a result that is read on the next cycle by another functional unit and this process goes on until the header is fully processed. This is an ideal scenario and also requires that the software written for the program also makes use of the functional units properly. The datapath of each GFU contains logic for operand forwarding. Thus, the functional units need not wait for the input to be written to the MHW if another functional unit has produced the value in the preceding clock cycle. Independent GFUs could also communicate, but it requires a number of cycles from the time a functional unit in a GFU has produced the result until another functional unit in another GFU could read that value because there is no cross-GFU forwarding. OHW entries contain arriving header words written by the parser.

A. Data Types

All data types are 32 bits wide. The parsing subsystem could read from a buffer of incoming packets in 8-, 16- and 32-bit units but writes them to 32-bit entries within the OHW storage. Therefore, everything that the packet processing subsystem reads from this space is 32 bits wide.

B. Memories

There are four different memory blocks in each GFU:

- Memory for holding the search key entries in the binary lookup table
- Memory for holding one-time-usable values
- Memory for holding parameter values
- Memory for load and store operations

C. Program Control

There are three key mechanisms for ensuring correct program flow.

- Branches based on header field values
- Branches based on results produced by the Packet Processor
- Conditional execution performed by the functional units of the Packet Processor

The functionality for both branches mentioned above is provided by the program control unit of the packet parser. The purpose of branches based on results produced by the functional units of the Packet Processor is to avoid a long stream of instructions whose execution result is not written to the MHW entries. Such streams of instructions waste cycles and prevent functional units from being used for other instructions.

D. Packet Processing Metadata

The metadata is a data structure containing information regarding the packet under processing. The rest of the system uses the metadata of a packet for further decisions regarding a packet. Metadata fields are outlined in Table II.

TABLE II. METADATA FIELDS

Metadata Field	Width (bits)	Purpose
Incoming Port	6	Indicates the port through which the packet has arrived.
Processing Status	2	Indicates whether processing is done and also whether the packet requires further processing in the main pipeline.
Discard	1	Specifies if the packet should be discarded.
Destination	2	Specifies if the packet should be sent to: -Main processing pipeline -Outgoing port -Control Plane -Internal Buffer
Destination Port Bitmap	64	Specifies the outgoing port(s) through which the packet must be transmitted.

IV. ILLUSTRATED EXAMPLE

In this section, we illustrate how this architecture processes IPv4 packets. We have specifically chosen IPv4 because it constitutes a great portion of Internet traffic. In addition, it requires quite a lot of processing even in the absence of header options. We will demonstrate how this programmable architecture can handle this workload without any protocol-specific hardware.

At a minimum, once an IPv4 packet arrives at a router, the checksum must be verified to detect possible errors during transmission. The Time to Live (TTL) field must be evaluated to see if the packet can continue its path or if it must be discarded. If it can continue, the value of TTL must be decremented and as a result of this, the checksum must be recalculated. The Destination Address field of the header must be used as a key to lookup into the forwarding table to determine the outgoing port(s) for the packet. In this illustrated example, we assume that the packets carry no header options. But to increase the workload to some extent, we perform some additional integrity checking on the header. Fig. 4 contains the packet processing functions to execute on this Packet Processor.

```
1  void process_ipv4_packet(ipv4_packet *p)
2  {
3          verify_ipv4_packet(p);
4          check_TTL(p -> TTL);
5          update_checksum(p);
6          lookup_destination_address(p ->
   Destination_Address);
7          check_DF(p -> flags);
8  }
9  void verify_ipv4_packet(ipv4_packet *p)
10 {
11         verify_version(p);
12         verify_IHL(p);
13         verify_Total_Length(p);
14         verify_checksum(p);
15 }
```

Fig. 4. Source code for processing IPv4 packets

As we can see, it contains functions for verifying the contents of Version, Internet Header Length (IHL) and Total Length fields. In order to clarify the contents of these functions, Fig. 5 illustrates the contents of the function that checks the value of TTL.

```
1  void check_TTL(ipv4_packet *p)
2  {
3          if(p -> TTL == 0)
4          {
5              insert_ICMPv4_header(BAD_HEADER);
6              drop(p);
7          }
   }
```

Fig. 5. Source code for checking the TTL field

The compiler or the programmer must decide how to assign these functions to the processing resources of the system. The processing resources of this architecture are:

- GFUs
- Functional units within each GFU
- Registers

We assign all integrity checking functions other than checksum checking to GFU 0. Checksum calculation and update could share the intermediate results for more efficient execution, therefore we assign both functions to GFU 1. We assign checking of Don't Fragment (DF) flag to GFU 2 and finally GFU 3 performs address lookup. Tables III, IV, V and VI contain the instructions that will be executed at each clock cycle in GFUs 0, 1, 2 and 3 respectively. t_0 is the time at which the first header word of the incoming IPv4 header is written to the OHW storage. It should be noted that the tables contain the instructions that are executed in each of the GFUs and not the

complete source code written for processing of IPv4 packets. Furthermore, any instruction that uses a given OHW entry as operand must be scheduled for execution at least 2 cycles after the packet parser has written it to the corresponding OHW entry. This is to ensure that the operand fetch logic has retrieved the correct value.

TABLE III. INSTRUCTIONS EXECUTED IN GFU 0

Time	Operation
t_0	-
t_1	-
t_2	R4 <- SHR4 R16(3)
t_3	r0 <- R4 CEQ 0x04 R3 <- R16(3) AND 0x0F
t_4	(r0) R14 <- Param(15); r4 <- R3 CG 0x04; R4 <- SHL4 R3
t_5	(r4) R14 <- Param(15); r0 <- R16(4) CEQ R4; r4 <- R16(4) CG R4
t_6	r15 <- r0 OR r4
t_7	r15 <- NOT r15
t_8	(r15) R14 <- Param(15)

In GFU 0, the value of Version field is obtained to be compared with the immediate value of 4. Similarly, the value of IHL is obtained to ensure that it is greater than or equal to 5. IHL is then shifted left four bits to derive the header size in bytes. Then it is compared with the value of Total Length to ensure that the entire size of the packet is greater than or equal to the header size in bytes. If any of the above-mentioned conditions is not fulfilled, ICMP bad header is inserted.

TABLE IV. INSTRUCTIONS EXECUTED IN GFU 1

Time	Operation
t_0	-
t_1	-
t_2	R0 <- R16(4) + R16(6)
t_3	R0 <- R0 + R17(4)
t_4	R0 <- R0 + R17(6)
t_5	R0 <- R0 + R18(4); R3 <- MOVE R0
t_6	R0 <- R0 + R18(6); R3 <- R3 + R19(4)
t_7	R0 <- R0 + R19(4); R3 <- R3 + R19(6)
t_8	R0 <- R0 + R19(6); R3 <- R3 + R20(4)
t_9	R0 <- R0 + R20(4); R3 <- R3 + R20(6)
t_{10}	R0 <- R0 + R20(6); R2 <- R18(3) - 0x01
t_{11}	R4 <- SHR16 R0; R6 <- SHL16 R2
t_{12}	R0 <- R0 + R4; R6 <- SHL8 R6
t_{13}	R1 <- NOT R0; R8 <- R18(7) MERGE R6
t_{14}	r0 <- R1 CEQ 0x00; R7 <- SHR16 R8
t_{15}	(NOT r0) R14 <- Param(15); R3 <- R3 + R7
t_{16}	R6 <- SHR16 R3
t_{17}	R3 <- R3 + R6
t_{18}	R9 <- R8 MERGE R3

In GFU 1, all 16-bit words of the header are added together for verifying the checksum. These additions take 10 cycles because the addition result must be accumulated and there are at least 10 items to be added together. R0 holds the summation result. At t_5 addition result calculated up to that point is copied into R3 so that the calculation of the new checksum could begin in parallel. As we can see, during t_5-t_{11} at each cycle there are two ALU or two shift operations occurring without conflict.

At t_{10}, TTL is decremented and stored in R2. The decremented TTL value which is one byte wide is then shifted all the way to take the most significant byte position. This takes two clock cycles. Then it is merged with the third word of the header so that it contains the updated TTL. Then a copy of this updated word is shifted right 16 bits for updating the checksum. Once the updated checksum is calculated, it is merged with the updated TTL and the unchanged Protocol field at t18.

TABLE V. INSTRUCTIONS EXECUTED IN GFU 2

Time	Operation
t_0	-
t_1	R14 <- Param(15)
t_2	r4 <- R16(4) CG R14
t_3	r14 <- BE5 R17(1)
t_4	r15 <- r4 AND r14
t_5	(r15) R15 <- Param(14)

GFU 2 executes instructions for checking the DF flag. At t1, R14 is loaded with the value of maximum allowed packet size. The router has previously calculated this using Path MTU Discovery. At the following clock cycle, the value of Total Length field which contains the size of the entire packet in bytes is compared with the value of R14. At the next clock cycle, the DF flag is extracted for evaluation. At t4, the two conditions of packet size being larger than maximum allowed packet size and DF flag being set are subject to an AND operation. At t5, ICMP Destination Unreachable is inserted if the compound condition calculated in the preceding cycle turns out to be true.

TABLE VI. INSTRUCTIONS EXECUTED IN GFU 3

Time	Operation
t_0	-
t_1	-
t_2	-
t_3	-
t_4	-
t_5	-
t_6	R12 <- Lookup R20(7)
t_7	-
t_8	(r6) R10 <- Load R12

GFU 3 performs lookup operation. At t6, Destination Address field of the arrived packet is submitted to the lookup table in the GFU for lookup. It takes one cycle until a matching entry is found and it takes another cycle to derive the address of the entry which contains the associated data which in this case is the set of ports to which the packet must be sent. The associated entry is loaded at t8 if a match has been found.

V. EVALUATION

In this section we present the implementation results. The Packet Processor has been implemented in VHDL. We have synthesized it on 28 nm UTBB FD-SOI technology in worst-case operating conditions (1.0V, ss, 125°C) using Synopsys

Design Compiler J-2014.09-SP4. Power analysis was also performed in worst-case operating conditions at the supply voltage of 1.0V (ss, 125°C). We have verified that all timing constraints are met for operation at the frequency of 2.0 GHz.

Table VII contains the area of the packet processing units called atoms in [6] and the equivalent functional units of our Packet Processor. Since the atoms perform only field modification and not lookup, we have also excluded the effect of the lookup tables on the values of table VII. It should be noted that we have not implemented the atoms in [6]. Instead we have provided the area results for the aggregate of functional units which provide functionality equivalent to the atoms in [6]. The atoms in [6] were synthesized to a 32-nm standard cell library. Using (1), we convert their area to equivalent 28 nm values so that we could compare them with the results of our own implementation.

$$\text{Area in 28 nm} = \text{Area in 32 nm} \times (28/32)^2 \qquad (1)$$

TABLE VII. AREA COMPARISON OF ATOMS IN [6] AND EQUIVALENT FUNCTIONAL UNITS IN THIS WORK

Atom	Area in [6] (μm^2)	Area in this work (μm^2)	Saving (%)
Stateless	1059	700	34
ReadWrite	191	90	53
ReadAddWrite	330	245	26
Predicated ReadAddWrite	605	418	31
IfElse ReadAddWrite	753	630	16
Subtract	1164	834	28

The Stateless atom could only perform arithmetic, logic, relational and conditional operations on operands. The Read/Write atom has the smallest area among the stateful atoms. The distinctive feature of atoms with higher area is the addition of logic levels for predication and logic for performing predicate-specific actions. From the perspective of functionality, the functional units in this work are far superior to the atoms because as we can see from Table 4 and Table 5 of [6], the functional unit in the stateful atoms performs either addition or addition and subtraction. The rest of the logic is for evaluating the predicate. In our architecture, with two ALUs in each GFU, action associated with each outcome for the predicate could be done in parallel using different functional units. It should also be noted that our architecture runs at 2.0 GHz while the architecture in [6] runs at 1.0 GHz.

Table VIII contains the area results of different components of a single instance of our Packet Processor. The table presents area also in terms of number of gates so that the results could be more easily compared with the results in [4] and [5]. For each component, the equivalent gate count is obtained by dividing the total cell area by the area of the smallest cell used in the technology. It should be noted that memories have an independent read port for each Packet Processor instance and hence they could be shared without contention. In other words, they need not be replicated per Packet Processor instance. In order to be able to make a precise comparison between the area of this Packet Processor and the Packet Parser in RMT, we must answer the critical question of how many Packet Processor instances are required for sustaining the target aggregate throughput of 640 Gbps. Assuming that the packets at each port arrive on a 10 Gbps link and that the packets are all minimum-sized, a new packet will arrive every 67 nanoseconds. Equivalently, there are around 15 M packets per port per second.

TABLE VIII. AREA FIGURES FOR A SINGLE PACKET PROCESSOR INSTANCE SYNTHESIZED TO 28 NM UTBB-FDSOI

Component	Area (μm^2)	Area (gate count × 1000)
Packet parsing logic	5193	11
Packet parsing parameter memories	96593	197
Program Control Logic	342	0.7
Packet processing functional units	56612	116
Operand storage, fetch and forwarding	75896	155
Instruction memory	344650	704

We also assume that all Layer-2 functionality is performed by an interface between the link and the Packet Processor which performs IPv4 routing. As we saw in section IV, the processing takes 19 cycles which in a 2.0 GHz system is equivalent to 9.5 nanoseconds. Therefore, one Packet Processor instance per 6 ports is sufficient for sustaining the line rate. However, we assign a Packet Processor per 4 ports in case some of the packets require further processing. Assuming that these Packet Processors will be in the same port configuration as in [5], 16 Packet Processor instances are sufficient to sustain the 640 Gbps throughput. The sum of the area of 16 instances is roughly 5.4 M gates which is slightly smaller than the area of the packet parser instances in [5].

VI. CONCLUSION AND FUTURE WORK

In this paper we presented a fully programmable architecture for protocol-independent packet processing. We have seen that by enhancing the previously designed packet parser with packet processing functionality which comes at a marginal cost, a great deal of packet processing could be done on the packets even if a match is not found in the tiny lookup tables. As for future work, we would like to enhance the throughput of the system and fine-tune it for high-end workloads and applications. In addition, requirement for use of more advanced functional units with the aim of supporting as wide range of protocols while maintaining protocol-independence must be investigated.

ACKNOWLEDGMENT

We hereby express our gratitude to Mr. Glen Gibb from Barefoot Networks for the invaluable comments regarding our idea. This work has been partially supported by the Finnish DELTA doctoral training network.

REFERENCES

[1] The World's Fastest & Most Programmable Networks https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/

[2] Intel Ethernet Switch Fm6000 Series 10/40 GbE Low Latency Switching Silicon https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf

[3] Agilio FX SmartNIC https://www.netronome.com/products/agilio-fx/

[4] Gibb, Glen, George Varghese, Mark Horowitz, and Nick McKeown. "Design principles for packet parsers." In Architectures for Networking and Communications Systems, pp. 13-24. IEEE, 2013.

[5] Bosshart, Pat, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN." ACM SIGCOMM Computer Communication Review 43, no. 4 (2013): 99-110.

[6] Sivaraman, Anirudh, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. "Packet transactions: High-level programming for line-rate switches." In Proceedings of the 2016 ACM SIGCOMM Conference, pp. 15-28. ACM, 2016.

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

[8] Jeyakumar, Vimalkumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. "Millions of little minions: Using packets for low latency network programming and visibility." In ACM SIGCOMM Computer Communication Review, vol. 44, no. 4, pp. 3-14. ACM, 2014.

[9] Handigol, Nikhil, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. "I know what your packet did last hop: Using packet histories to troubleshoot networks." In 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pp. 71-85. 2014.

[10] Sanvito, Davide, Giuseppe Siracusano, and Roberto Bifulco. "Can the network be the AI accelerator?." In Proceedings of the 2018 Morning Workshop on In-Network Computing, pp. 20-25. ACM, 2018.

[11] Zolfaghari, Hesam, Davide Rossi, and Jari 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), pp. 1-4. IEEE, 2018.

[12] Zolfaghari, Hesam, Davide Rossi, and Jari Nurmi. "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), pp. 1-6. IEEE, 2018.

PUBLICATION
V

Reducing Crossbar Costs in the Match-Action Pipeline

H. Zolfaghari, D. Rossi and J. Nurmi

2019 IEEE 20th International Conference on High Performance Switching and Routing (HPSR), 1-6

doi: 10.1109/HPSR.2019.8808105

Reducing Crossbar Costs in the Match-Action Pipeline

Hesam Zolfaghari
Department of Electrical Engineering
Tampere University
Tampere, Finland
hesam.zolfaghari@tuni.fi

Davide Rossi
Department of Electrical, Electronic and Information Engineering
University of Bologna
Bologna, Italy
davide.rossi@unibo.it

Jari Nurmi
Department of Electrical Engineering
Tampere University
Tampere, Finland
jari.nurmi@tuni.fi

***Abstract*— Software Defined Networking (SDN) is a new networking paradigm in which the control plane and data plane are decoupled. Throughout the recent years, a number of architectures have emerged for protocol-independent packet processing. One such architecture is the Protocol Independent Switch Architecture (PISA). It is a programmable and protocol-independent architecture composed of a number of Match and Action stages. Inside each of these stages is a crossbar to generate the search key and another crossbar to provide the input to the Action Units. In this paper, we design and explore alternative interconnection schemes with the aim of finding the most area- and power-efficient interconnection structure. Moreover, we propose further modifications to the interconnection structure, as a result of which the on-chip area of both match and action crossbars will be reduced by more than 70 % and power dissipation will be reduced by 25.8 % and 23.1 % for match and action crossbars respectively.**

Keywords—Software Defined Networking, Protocol Independent Switch Architecture, Crossbar

I. Introduction

Packet processing is a domain in which performing matching and applying the actions associated with the match result are the key operations. The match operation could be used to determine the outgoing port for a packet or determining the kind of processing that a group of fields requires. Programmable packet processing systems must have the means to process different protocols each of which requires corresponding match and action operations. Therefore, a programmable packet processing system should be able to generate a search key using any subset of the header fields. Moreover, the functional units performing tasks other than lookup must be able to operate on any of the header fields. This is achieved by the use of flexible crossbars.

There are a number of programmable networking devices already available on the market. Intel FlexPipe [1], Cavium XPliant [2] and Barefoot Tofino [3] are the most notable of such devices. Due to the abundance of flexible tables and processing units, we limit our focus to the Barefoot Tofino. It can be programmed using the P4 language [4] for providing a wide range of packet processing functionalities. The internals of Tofino are based on the Protocol Independent Switch Architecture (PISA) which is a switch architecture first proposed in [5]. PISA is comprised of a programmable packet parser based on the architecture proposed in [6]. The programmable parser extracts header fields and places them in placeholders of three different sizes, 8-bit, 16-bit and 32-bit entries. These entries together form a 4096-bit vector of header fields. This vector is called Packet Header Vector (PHV) and it traverses a pipeline of 32 stages. Each stage contains a match part and an action part. Using a large crossbar, two 640-bit search keys are generated for table lookup in the match stage. The result of the lookup determines the required actions to be performed in the subsequent action stage. The action stage contains an action unit for each one of the entries. Therefore, there are action units of the different bit widths referred to earlier. For each action unit, there is a multiplexer which selects the inputs. The first input to the action units is from the entries of the PHV and the second input is either from the PHV or action memories containing packet processing parameters. Being a 32-stage pipeline, the crossbars make a noticeable contribution to the overall area of the chip. For instance, the area of match crossbars is 6 mm^2 in total [5].

There are Match and Action crossbars in each pipeline stage of the PISA. According to [5], each of the 1280 bits of the search key are driven by a 224-to-1 multiplexer. The multiplexers are constructed using a binary tree of AOI22 gates. The choice of 224-to-1 multiplexers indicates that there is an alignment constraint for the placement of header fields into the search key. This means that bytes can be placed into any location within the search key, 16-bit fields must be placed at even locations and 32-bit fields must be placed at locations whose index is a multiple of four. Such constraints help limiting the complexity of crossbars.

The Action crossbar in PISA provides input to the Action Engines. The first input to the Action engines is selected from the PHV while the second input is selected from either the PHV or the Action memories. The Action crossbar allows for combining smaller header fields. For instance, two 16-bit fields could be combined to form a 32-bit input to a 32-bit Action Engine.

The crossbars used in the original paper are not the only crossbars that can fulfill the interconnection requirements. In this paper we devise alternatives for providing the required interconnection, and compare area and power dissipation of each alternative. Furthermore, we justify the use of more lightweight crossbars and observe the results. The rest of the paper is organized as follows: In section II, different alternatives for implementing match and action crossbars will be evaluated. In section III, actual requirements for match and action crossbars will be analyzed. In sections IV and V, we propose smaller crossbars and present the required architectural modifications. Section VI presents the experimental results followed by the conclusion.

II. Alternative Crossbar Architectures for PISA

There are numerous strategies for forming the match key as well as selecting the inputs to the Action Engines. In this section we explore some of the alternatives with the aim of finding the most efficient solution.

The match crossbar in PISA operates at bit level, meaning that while adhering to the alignment constraint mentioned earlier, each bit of the search key is selected independently. Consequently, there are 10240 bits of select inputs required in total for the multiplexers.

A. Alternative Match Crossbars

1) Byte-level match field selection

Alternatively, we could select input to the search key on byte-level basis while still maintaining the alignment constraint. The PHV could be thought of as being comprised of 512 bytes and the search key to be generated as 160 bytes. Therefore, we could fill the search key using 160 512-to-1 8-bit multiplexers. In this organization, 1440 bits of select are required.

2) Word-level match field selection

Another solution is to operate at 32-bit basis using two levels of multiplexers. In the first level, there are four 64-to-1 8-bit multiplexers, two 96-to-1 16-bit multiplexers and one 64-to-1 32-bit multiplexer. Using these multiplexers, different combinations of fields could be combined to form 32 bits of the search key. For instance, two 8-bit fields and one 16-bit field could be combined together. The multiplexer at the second stage selects the combination of first-level multiplexers' outputs. Fig. 1 depicts the internals of this solution for filling in 32 bits of the 1280-bit search key. For other 32 bits of the search key, the same structure must be replicated. Some of the multiplexers could share the select lines. For generating a 1280-bit search key using this structure, 1160 bits of select are required.

We have implemented the alternatives in VHDL and synthesized them on 28 nm UTBB FD-SOI technology in worst-case operating conditions (1.0V, ss, 125°C) using Synopsys Design Compiler J-2014.09-SP4. Power analysis was performed in typical operating conditions at the supply voltage of 1.1V (tt, 25°C). The same methodology has been used throughout the whole paper. Timing analysis confirms that all crossbars mentioned in this paper can run in a 1.0 GHz system.

Table I compares the above-mentioned alternatives for the Match crossbar in terms of area and power. The given values are for one stage of the Match-Action pipeline. Moreover, their output generates a 640-bit match key. As we can see, the byte-level crossbar has the largest area and power dissipation values. In addition, they have more levels of logic compared to other crossbars. As a result, scaling the frequency beyond a point requires using registers. This increase latency and area. Using the word-level interconnection scheme results in 6 % reduction in area and 13 % reduction in power dissipation compared to the original scheme used in [5].

B. Alternative Action Crossbars

The use of interconnection schemes in which the width of select values is extremely wide should be avoided because that will increase the required width for instruction memory.

1) Bit-level selection

For the Action crossbar, if the same approach as the Match Action is adopted, 32768 select bits are required for the multiplexers.

2) Byte-level selection

If, we choose to select the input on byte-level, 512 multiplexers are required, each being a 512-to-1 8-bit multiplexer. Under this scheme, the number of select bits will be reduced to 4608 bits.

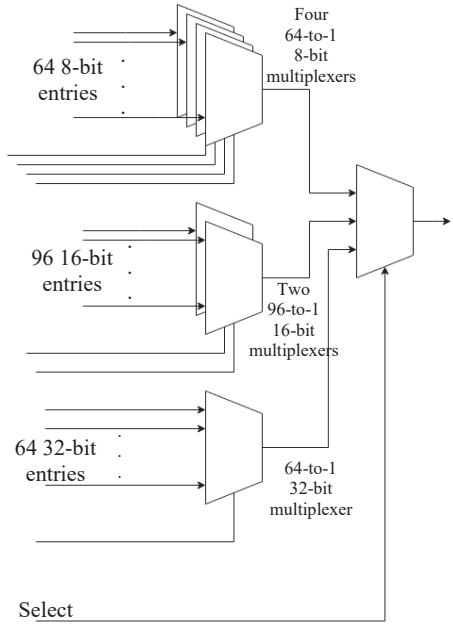

Fig. 1. Generating 32 bits of the search key using two levels of multiplexers

TABLE I. COMPARISON OF MATCH CROSSBAR ALTERNATIVES

Interconnection Strategy	Area (μm^2)	Total Power Dissipation (mW)
Bit-level	125650	137.5
Byte-level	256772	156
Word-level	118724	119.4

3) Combination of smaller processing units

In this implementation, the input to the 8-bit Action Engines comes from any of the 8-bit entries of the PHV, while the input to the 16-bit Action Engines comes either from combination of any two 8-bit entries or any of the 16-bit entries. Similarly, the input to 32-bit Action Engines is selected from any two 16-bit entries or any of the 32-bit entries of the PHV. Therefore, the input to 16-bit and 32-bit action engines comes from two levels of multiplexers. For this scheme, 3648 bits of select are required. Fig. 2 illustrates multiplexers required for an action unit of each size under this scheme.

4) Zero-extension of smaller processing units

In the final Action crossbar that we consider, 8-bit Action Engines take any of the 8-bit entries as input while 16-bit action engines receive either a 16-bit entry or a zero-extended 8-bit entry whose width is 16 bits after zero-extension. Similarly, 32-bit action engines take any 32-bit entry or any 16-bit entry which has been zero-extended to 32 bits. This scheme requires 1664 bits of select which is the smallest number among the solutions discussed so far. However, the actual merging of smaller entries requires two steps: Selecting two zero-extended values as input to a given Action Engine

and using the Deposit Byte operation code for the Action Engine in question to merge the inputs together. Therefore, the Action crossbar by itself cannot perform the actual merging and requires the assistance of the action engine. However, the simplicity of this scheme makes it an interesting choice.

Table II presents area and power values for different Action crossbars. The given values are for one stage of the Match-Action pipeline. Similar to the match crossbars, byte-level crossbars are the least efficient alternative. The most important observation is that the action crossbar with zero-extension is the most efficient crossbar. Compared to the bit-level interconnection scheme, use of the two-level crossbar with zero-extension capability results in 52.7 % and 46.7 % reduction in area and power dissipation respectively.

TABLE II. COMPARISON OF ACTION CROSSBAR ALTERNATIVES

Interconnection Strategy	Area (μm^2)	Total Power Dissipation (mW)
Bit-level	804160	880
Byte-level	1643343	992.4
Combination of smaller processing units	503423	656.2
Zero-extension of smaller processing units	379992	468.7

III. REQUIRED CROSSBAR COMPLEXITY

In this section, we analyze the actual required complexity for both Match and Action crossbars.

A. Required Match Crossbar Complexity

Typically, a search key is comprised of header fields whose semantics allow for the concept of matching. For instance, in Internet Protocol version 4 (IPv4), the destination address field is used to lookup into the forwarding table to determine the outgoing port of the packet. As another example, the Next Header field in Internet Protocol version 6 (IPv6) is the match key for determining the kind of processing that the header following IPv6 header requires. In many processing scenarios, the search key is made up of multiple header fields placed next to each other. For instance, Protocol, source address and destination address from IPv4 header as well as source port and destination port from Transmission Control Protocol (TCP) header can be used together to form a search key for retrieving the flow-specific parameters.

As we can see, not all the header fields have the semantics to be used as a search key. Therefore, use of crossbars that receive input from the whole entries of the PHV is unnecessary. Smaller crossbars could be used without degradation in functionality.

B. Required Action Crossbar Complexity

Determining the right degree of complexity for Action crossbars is more complicated than that of Match crossbars. In order to do so, various packet processing actions must be taken into consideration. We have considered commonly used protocols because protocol-independent packet processors should be capable of processing them for the transition to these devices to be feasible. We are primarily interested in the number of inputs to each action because the number of inputs determines the complexity of the required crossbar. The inputs are mainly header fields. However, metadata could also be the input.

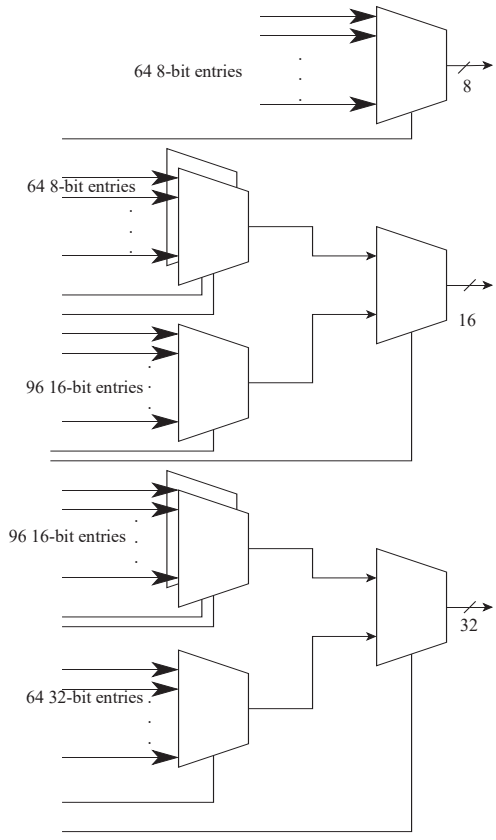

Fig. 2. Part of the Action crossbar in which smaller processing units can be combined

At one extreme, we encountered actions which require only one or two inputs. A prime example of this group of actions is decrementing the Time to Live (TTL) field in Internet Protocol version 4 (IPv4) header. Fig. 3 illustrates the corresponding subroutine.

```
1  void IPv4_TTL_Decrement(ipv4_packet* p)
2  {
3          //TTL is located at byte offset 8
4          if(*(p + 8) == 0)
5          {
6                  drop(p);
7          }
8          else
9          {
10                 *(p + 8) -= 1;
11         }
12 }
```

Fig. 3. Source code for decrementing Time to Live in IPv4

At the other extreme, there are actions which require access to the entire header. In other words, they operate on all

the fields existing in the header. An example of such actions is the Internet checksum [7] used in IPv4 and Internet Control Message Protocol version 4 and 6. Fig. 4 illustrates the subroutine for calculating the checksum in IPv4.

There are important observations to make from the Internet checksum calculation. It takes 16-bit words as input. Therefore, the programmable parser must place incoming header into the 16-bit entries of the PHV. Operations such as TTL decrement which operate on smaller input must copy the 16-bit word containing the TTL into smaller entries of the PHV to perform the desired operation. Alternatively, if the programmable parser places the header fields into entries that best match the fields' actual size, they need to be combined into 16-bit words for Internet checksum calculation. Being one's complement sum, the additions must be performed by 32-bit Action Units. Therefore, all the 16-bit words must be copied to 32-bit entries. Once all the required additions are performed, the lower 16-bit portion of the result will be copied to a 16-bit entry.

```
1  void calculate_IPv4_checksum(ipv4_packet* p)
2  {
3          unsigned int sum = 0;
4          unsigned char IHL= *(p + 0) & 15;
5          unsigned int temp;
6          for(int i = 0; i <= IHL * 4; i = i + 2)
7          {
8                  //Adding 16-bit words
9                  temp = (unsigned int)(*(p + i) * 256)
   + (*(p + i + 1));
10                 sum += temp;
11         }
12
13         //as long as there is overflow
14          while(sum >= 65536)
15         {
16                 sum = (sum & 65535) + (sum>>16);
17         }
18         sum = ~sum;
19         //disposing the value of the upper 16 bits
20         sum = sum & 65535;
21 }
```

Fig. 4. Source code for calculating IPv4 checksum

Between the two extremes mentioned above, there are actions which require a handful of header fields. For instance, in order to determine which queue a packet must be sent to in an IPv6 router with Quality of Service (QoS) support, Traffic Class, Flow Label and outgoing port are required.

Based on this discussion, we can come up with the following taxonomy of header fields:

- Fields as they appear in the technical specification of a networking protocol: For instance, the Version field is a 4 bit field in the header of Internet Protocol version 6.
- Fields in the form required for packet processing actions: As an example, Source and Destination Addresses are 32-bit fields in Internet Protocol version 4, but in order to calculate checksum, they must be broken into 16-bit fields.
- Fields as placed in the entries of the PHV by the programmable parser and crossbar: The PHV in the PISA architecture contains entries of three sizes. Header fields whose size does not exactly match these entries have to share placeholder with other fields or span over more than one placeholder. Examples of such header fields are the 4-bit Internet Header Length field in Internet Protocol version 4 and 20-bit label field in Multiprotocol Label Switching.

IV. Reducing Crossbar Costs

As we saw in the previous section, only a fraction of fields existing in a header are used for forming match keys. In addition, not all packet processing actions require all the header fields. Based on this observation, the packet parser and crossbars in the Match-Action pipeline can be modified with the following goals:

- Tailoring the header fields to the packet processing actions: Part of the time required for processing a packet is contributed to by transferring them to a suitable PHV entry. If we could limit this part of the processing, the overall packet processing time will be reduced. This specifically concerns the action crossbar in which smaller processing units are zero-extended. Using these crossbars, the actual merging of smaller processing units takes another cycle.
- Reducing the complexity of crossbars by limiting the number of inputs: The crossbars in PISA allow for full interconnection, meaning that all entries are the input to each of the multiplexers. Except for tasks such as checksum calculation, this is rarely encountered.

Based on the arguments above, we would like to limit the number of PHV entries that can be selected as the input to its corresponding Action unit. In other words, we can think of the PHV as being divided into independent segments. Inside each segment, the input to each entry's Action unit can be any other entry within the segment. However, only few of the entries of other segments can be accessed. The programmable parser extracts incoming header fields and places them into entries of the PHV segments. In order to limit the communication of independent segments, it can place a given header field into more than one segment, meaning that duplicates can exist among the segments. Moreover, it can extract an incoming 32-bit header such that it fills two 8-bit and one 16-bit entries in one PHV segment and at the same time it fills two 16-bit entries of another PHV segment. By doing so, the fields will be tailored to packet processing tasks. It should be pointed out that the programmable parser does this according to the software that is written for it. As such, it does not have any built-in knowledge of protocols.

We have chosen the value of 4 as the number of independent segments. Alternatively, we could think that the number of inputs to the multiplexers is divided by four. This means that inside each segment, there will be 16 8-bit entries, 24 16-bit entries and 16 32-bit entries. In addition, the match crossbar receives as input only a quarter of the entries present in the whole PHV. This is equivalent to the number of entries in one segment. However, the match crossbar receives an even number of inputs from each of the four segments.

V. Architectural Requirements

The first item that must be modified is the packet parser. For each independent PHV segment, it must have a separate output port. We use the packet parser proposed in [8] and [9]. It has a program control unit as opposed to a Ternary Content Addressable Memory (TCAM). As a result, its area is considerably lower than that of the parser used in [3], [5] and [6]. Reduced area leaves room for further modifications. A high-level view of this parser is illustrated in Fig. 5. As we can see, there are four 8-bit, two 16-bit and one 32-bit output port. At any given instance in time, only a fraction of these ports has valid data. Data at these ports will be written to the PHV. The PHV is organized as a number of banks. Therefore, each of the aforementioned ports can write to its corresponding bank once it has valid data. The only required modification is replication of the PHV Filler component inside the parser so that incoming header fields can be extracted independently. This means increase in the width of the instruction. The instruction field associated with the PHV Filler is 4 bits wide. In the new architecture, 16 bits will be required for this purpose. In addition, the location of extracted fields within the PHV is specified by software. 20 bits are required to specify the location of extracted fields. In the modified parser, 48 bits are required in total for specifying location of extracted fields in the PHV.

Another required modification is the resizing and arrangement of banks that together comprise the PHV. The width of the PHV is still 4 Kilobits. Only the banks comprising the PHV will be resized.

The most performance-critical aspect is interconnection of independent segments. We must ensure that limiting the number of inputs to crossbars has as little effect as possible on the packet processing capabilities and throughput of the architecture. In order to maximize the savings in area and power dissipation, we consider the action interconnection scheme in which zero-extended values of smaller processing units will be the input to the next larger multiplexer. Under this organization, in each segment, there will be 16 8-bit 16-to-1, 24 16-bit 64-to-1 and 16 32-bit 64-to-1 multiplexers. Both 16-bit and 32-bit 64-to-1 multiplexers have 24 inputs that could come from the other segments because for both of them the total number of inputs from their own segment is 40. These 24 input lines could be used to pass the value of entries in other segments. With 24 16-bit multiplexers each having 24 inputs that could come from other segments, there is room for 576 16-bit entries which is far more than the total number of 16-bit entries in the entire PHV. Similarly, with 16 32-bit multiplexers each having 24 unused inputs, 384 32-bit inputs can be accommodated which is well above the total number of 32-bit entries of the PHV. Each multiplexer with unused inputs can take an equal number of entries of matching size from the other segments. For instance, each of the 64-to-1 16-bit multiplexers in a given segment could take 8 entries from each of the other segments to fill its 24 unused inputs. Alternatively, one such multiplexer could take all the 16-bit entries of another segment with its adjacent multiplexer taking all the 16-bit entries of another segment as its inputs.

In addition, because the number of action engines matches the number of PHV entries, each entry has an action unit reserved for it and hence, it is not required to take the same entry as the second input to the action unit in question. Consequently, each multiplexer inside a segment can take one entry from the other segments. As each segment contains 56 entries, each segment can in total take 56 entries belonging to other segments. Therefore, independent segments can also communicate reasonably well. The only limitation is that when an entry residing in another segment is required, only certain action engines have access to it. However, once the value of an entry residing in another segment is written to an entry in the current segment, then all action units of the current segment have access to it.

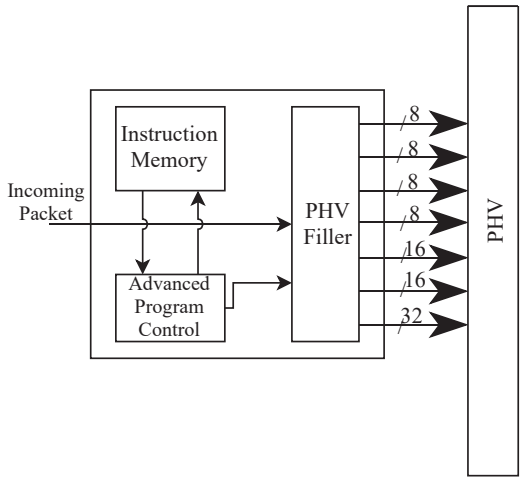

Fig. 5. High-level view of the packet parser

The deparser does not require any modification because the overall size of the PHV remains the same. Moreover, in the original architecture too, there will be PHV entries that contain only intermediate results that must be discarded.

VI. Experimental Results

In this section, we present the costs and savings of modifications. Table III compares area and power of the two parser variants. As we can see, the increase in area is only 3.4%. The increase in switching power is due to presence of more registers in the modified parser. However, as the parser accounts for less than 1 % of the chip area [5], this increase can be neglected. The savings reflected in the upcoming tables outperform this increase. Tables IV and V present area and power results of lightweight match and action crossbars respectively. For each one of the crossbar variations discussed in section II, the figures for the corresponding lightweight crossbar is provided. It should be noted that values in table V reflect area and power results of the crossbars in all four segments within a Match-Action stage, not only one segment. The area and power saving columns specify how much area and power has been saved for each crossbar with respect to its corresponding larger instance. Since for all variants, the number of inputs to the multiplexers in the crossbars is divided by four, the savings in area are all in the same range of 70 %. The saving in power dissipation is at least 25 % for the match crossbars and at least 23 % for the action crossbars. The ratio of crossbar areas is maintained. For instance, the word-level match crossbar is still the most lightweight crossbar. It is also worth mentioning that using lighter crossbars, the frequency could be scaled up more easily because there will be fewer levels of logic.

TABLE III. COMPARISON OF PARSER IMPLEMENTATIONS

	Baseline Parser	Parser with multiple outputs
Total Area (μm^2)	29900	30937
Internal Power (mW)	7.7	8.2
Switching Power (mW)	4.2	11.7
Leakage Power (mW)	4.5	4.8
Total Power (mW)	16.4	24.7

TABLE IV. AREA AND POWER DISSIPATION COMPARISON OF LIGHTWEIGHT MATCH CROSSBARS

Match Crossbar Variation	Area (μm^2)	Saving in area (%)	Total Power Dissipation (mW)	Saving in power dissipation (%)
Bit-level	34885	72.2	93.5	32
Byte-level	67499	73.7	101.8	34.7
Word-level	32813	72.3	88.5	25.8

TABLE V. AREA AND POWER DISSIPATION COMPARISON OF LIGHTWEIGHT ACTION CROSSBARS

Action crossbar variation	Area (μm^2)	Saving in Area (%)	Total Power Dissipation (mW)	Saving in power dissipation (%)
Bit-level	223264	72.2	598.4	32
Byte-level	431996	73.7	651.6	34.3
Combination of smaller processing units	140260	72.1	503.2	23.3
Zero-extension of smaller processing units	107000	71.8	360.3	23.1

VII. CONCLUSION

In this paper, we devised alternative interconnection schemes while maintaining the interconnection requirements. Based on the experimental results, we determined the most area- and power-efficient crossbars for the Match-Action pipeline. In addition, by analyzing headers and packet processing actions, we proposed use of smaller crossbars using which area will be decreased by at least 70 %. The savings in power dissipation are between 23 and 34 %. By careful placement of header fields in PHV entries, any possible performance degradation will be eliminated. Moreover, by making the compiler aware of cross-segment interconnection restrictions within a Match-Action stage, the changes proposed in this paper will be invisible to the programmer.

ACKNOWLEDGMENT

We would like to acknowledge the Finnish DELTA network as well as The Pekka Ahonen Fund for providing the partial funding for this project.

REFERENCES

[1] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[2] Cavium's XPliant™ Ethernet Switch Supports the Emerging Open Ecosystems https://www.cavium.com/Caviums-XPliant-Ethernet-Switch-Supports-The-Emerging-Open-Ecosystems.html

[3] Barefoot Networks, "The world's fastest and most programmable networks," [Online]. Available: https://barefootnetworks.com/resources/worlds-fastest-mostprogrammable- networks

[4] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G. & Walker, D. (2014). P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 44(3), 87-95.

[5] Bosshart, P., Gibb, G., Kim, H. S., Varghese, G., McKeown, N., Izzard, M., Mujica, F. & Horowitz, M. (2013, August). Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In ACM SIGCOMM Computer Communication Review (Vol. 43, No. 4, pp. 99-110). ACM.

[6] Gibb, G., Varghese, G., Horowitz, M., & McKeown, N. (2013, October). Design principles for packet parsers. In Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on (pp. 13-24). IEEE.

[7] Rijsinghani, A. (1994). Computation of the internet checksum via incremental update.

[8] Zolfaghari, H., Rossi, D., & Nurmi, J. (2018, July). 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) (pp. 1-4). IEEE.

[9] Zolfaghari, H., Rossi, D., & Nurmi, J. (2018, October). 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) (pp. 1-6). IEEE.

PUBLICATION
VI

Flexible Software-Defined Packet Processing Using Low-Area Hardware

H. Zolfaghari, D. Rossi, W. Cerroni, H. Okuhara, C. Raffaelli and J. Nurmi

IEEE Access, vol. 8 (2020), 98929-98945

doi: 10.1109/ACCESS.2020.2996660

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received April 26, 2020, accepted May 15, 2020, date of publication May 22, 2020, date of current version June 5, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2996660

Flexible Software-Defined Packet Processing Using Low-Area Hardware

HESAM ZOLFAGHARI[1], (Graduate Student Member, IEEE), DAVIDE ROSSI[2], (Member, IEEE), WALTER CERRONI[2], (Senior Member, IEEE), HAYATE OKUHARA[2], (Member, IEEE), CARLA RAFFAELLI[2], (Senior Member, IEEE), AND JARI NURMI[1], (Senior Member, IEEE)

[1]Electrical Engineering Unit, Tampere University, 33720 Tampere, Finland
[2]Department of Electrical, Electronic, and Information Engineering, University of Bologna, 40126 Bologna, Italy

Corresponding author: Hesam Zolfaghari (hesam.zolfaghari@tuni.fi)

This work was supported in part by the 5G-FORCE Project, Finnish DELTA Doctoral Training Network, the Grant awarded by the Nokia Foundation, and the collaboration grant provided by HiPEAC network.

ABSTRACT Computer networks are in the Software Defined Networking (SDN) and Network Function Virtualization (NFV) era. SDN brings a whole new set of flexibility and possibilities into the network. The data plane of forwarding devices can be programmed to provide functionality for any protocol, and to perform novel network testing, diagnostics, and troubleshooting. One of the most dominant hardware architectures for implementing the programmable data plane is the Reconfigurable Match Tables (RMT) architecture. RMT's innovative programmable architecture enables support of novel networking protocols. However, there are certain shortcomings associated with its architecture that limit its scalability and lead to an unnecessarily complex architecture. In this paper, we present the details of an alternative packet parser and Match-Action pipeline. The parser sustains tenfold throughput at an area increase of only 32 percent. The pipeline supports unlimited combination of tables at minimum possible cost and provides a new level of flexibility to programmable Match-Action packet processing by allowing custom depth for actions. In addition, it has more advanced field-referencing mechanisms. Despite these architectural enhancements, it has 31 percent less area compared to RMT architecture.

INDEX TERMS Software defined networking, programmable packet processing, low-area hardware, programmable data plane.

I. INTRODUCTION

Computer and communication networks have been subjected to a significant paradigm change in the last decade, leading to the emergence and subsequent consolidation of network programmability solutions and technologies, such as Software Defined Networking (SDN) [1] and programmable data plane [2]. In particular, the innovation introduced by SDN is represented by the separation of the *control plane* from the *data plane*, which have been traditionally co-existent and tightly coupled within network forwarding devices, such as switches and routers. Due to the increasing complexity of modern networks and the high level of flexibility required by newly emerging services, this tight coupling caused significant complications in managing network infrastructures, forcing operators and service providers to adopt solutions that were strictly dependent on the features offered by specific equipment vendors [3].

With the separation between control and data planes, SDN-enabled devices can specialize on how packet processing and forwarding operations can be efficiently executed in the data plane, whereas the decision on what kind of processing must be performed and where to forward each packet (or flow of packets) is left to a *logically* centralized component located in the control plane, the so-called *SDN controller*. This approach opens a completely new set of possibilities to make the network truly programmable: once an open and standard interface has been defined between control and data planes, the SDN controller can be used as a means to instruct network devices on how to act on the packets in the data plane, independently of any vendor-specific implementation.

The associate editor coordinating the review of this manuscript and approving it for publication was Yulei Wu.

The most noteworthy and widespread SDN control plane solution is represented by the OpenFlow protocol [4], which allows SDN applications to abstract the network infrastructure and program the behavior of the underlying set of forwarding nodes in terms of *Match-Action* packet processing. A set of matching rules (including wildcards) is applied to layer-2 to layer-4 header fields in order to specify packet flows with arbitrary levels of granularity. Then, each packet of a given flow is treated according to the actions specified in the corresponding matching rule. This approach simplifies internal switch operations and, at the same time, allows unprecedented flexibility in traffic control and steering capability.

However, the programmability features offered by OpenFlow at the control plane are limited by the dependence on a set of pre-defined protocol headers and on a static processing pipeline inside the switches. Therefore, a step forward is represented by the inception of programmable data plane approaches, such as protocol-oblivious forwarding [5] and the P4 language [6]. More specifically, the latter allows to dynamically reconfigure the data plane processing system at deployment time, making it protocol independent as well as target independent, thus giving programmers the possibility to describe the packet-processing pipeline in an abstract way independent of the specific hardware solution adopted.

In this scenario, the SDN concept has become a key enabler also for 5G networks where radio, transport and cloud domains cooperate to offer ubiquitous connectivity services to people and objects [7]. To meet the performance requirements of an unpredictable amount of different applications, flexible and scalable architectures and functionalities are introduced in 5G deployments. In addition, the trend is to consider commercially available packet-based solutions in the transport network, e.g., the Ethernet standard. Recently, the new concept of flexible Radio Access Network (RAN) has been considered that, coupled with Network Function Virtualization (NFV) and SDN control capability, allows to configure the network with different functional splits in transport network nodes [8]. This solution is expected to be dynamic enough to face with virtual resource instantiation needs, the so-called network slices, and can require different packet formats as specified by the relevant standards [9]–[11]. In this context, the possibility to have a programmable packet processing pipeline is crucial to implement high speed flexible forwarding. Reconfigurations may be needed when a different functional split is required to meet changing slice requirements.

As a result of these efforts to make the network truly programmable, both in the control and the data plane, there is a clear need for flexible and protocol-independent hardware-based packet processing systems. One of the reference architectures based on the Match-Action principle is represented by the Reconfigurable Match Tables (RMT) [12], also adopted by commercial switch chips such as Barefoot Tofino [13]. However, as we will see in section 2, there are a number of limitations associated with this architecture. As a result of these limitations, the architecture is unnecessarily complex.

From the perspective of hardware architecture, the programmable data plane is still in its infancy. In this paper, we present a programmable packet parser and a flexible packet processing pipeline. The parser sustains aggregate throughput of 6.4 Tbps which is 10 times that of the parser in RMT architecture, but the area increase is only 32%. The packet processing pipeline allows unlimited combination of lookup table resources with the minimum possible hardware costs. As a result of this support for unlimited table combinations, the resources are more efficiently used. In addition, it allows the action depth to be freely determined by the programmer. We achieve area reduction of up to 44% with respect to the latest Match-Action architectures.

The remainder of the paper is organized as follows. In section 2, we discuss related work and main motivations behind our approach. The main contributions of this work, a new packet parser and a flexible packet processing pipeline, are discussed in sections 3 and 4 respectively. The contributions are evaluated in section 5, followed by a conclusion on this work.

II. RELATED WORK AND MOTIVATIONS

A. RELATED WORK

The first attempt to separate IP control and forwarding functions was made within the Internet Engineering Task Force (IETF) Network Working Group and resulted in the Forwarding and Control Element Separation (ForCES) architecture [14], [15]. These documents define the framework, including the primary functions of a forwarding element and the communication requirements between forwarding and control elements. Then, the Ethane network architecture was introduced, in which the traffic flow management is handled by a centralized controller [16]. An Ethane-capable switch establishes a connection with the controller that contains the overall image of the network. The switches do not need to discover and locally store the network topology, which greatly reduces the state that must be maintained by the switches.

The next major breakthrough toward the SDN approach as we know it today was the introduction of OpenFlow as a standard protocol for communication between the data plane and the control plane [4]. The early motivation of running experimental protocols on real network infrastructures led to the availability of commercial Ethernet switches enabled to OpenFlow and implementing the Match-Action packet processing dictated by that control plane protocol. More specifically, all OpenFlow switch operations are based on a set of tables against which cross-layer packet headers are matched, and each table entry specifies a given action or set of actions to be applied to each matching packet. Typical actions include forwarding the packet to one or multiple output ports, dropping the packet, rewriting some of the header fields, or sending the packet to the OpenFlow controller for further analysis and decision making.

The idea of a logically centralized controller, which is the pivotal concept in SDN, simplifies the internal operations to be performed by network nodes. It also encourages the idea of making them protocol-independent, so that by installing any set of rules in the tables inside the switches, their behavior can be programmed accordingly. The term Protocol-Oblivious Forwarding was coined, and a generic high-level instruction set was presented in [5]. In a similar attempt, but with a lower layer of abstraction, an instruction set was presented in [17] in order to act as an intermediate layer between many packet processing hardware architectures and packet processing software. In other words, it acts as a target-independent machine model.

On the programmable data plane level, the P4 language was introduced in [6]. In P4, the problem of processing packets is formulated in the form of Match-Action processing. However, unlike OpenFlow, P4 abstracts the switch as a programmable parser followed by a protocol-independent Match-Action pipeline. Contrary to the primitive and protocol-specific actions defined in OpenFlow, the actions in P4 are not tied to any specific protocol. P4 also allows definition of compound actions by combining the primitive actions. It should be noted that OpenFlow and P4 are meant for different purposes, namely communicating with the central controller and programming the data plane respectively, but since both define actions and a similar abstraction of the switch, we made a comparison of the two here.

Custom architectures with varying levels of programmability for processing of network packets gained popularity both in research and industry in late 1990s and early 2000s. In those days, these devices were called protocol processors and later on network processors. The major hurdle for the widespread adoption of these devices was the complex procedure for programming of some of these devices as some of them required microcode-style programming. In addition, each vendor had its proprietary means of programming their devices. For this reason, network processors failed to gain widespread popularity.

As a result of research efforts on separation of forwarding and control plane of networking devices that later on led to introduction of Software Defined Networking (SDN), the need for hardware-based packet processing systems re-emerged. However, this time with special focus on protocol-independence and programmability. The new term was programmable data plane. Since the debut of the concept, there have not been many architectures for this purpose.

The most dominant architecture was first introduced in [12] and [18]. It is based on the Match-Action principle, meaning that programmer-specified header fields are used to form a search key which is provided to a match table. The outcome of the match determines the action, which is the required processing on the packet. In [19], high-speed packet processing is addressed in both software and hardware domains. On the software side, it provides guidelines for arranging packet processing programs for high-throughput execution. On the hardware side, it provides alternative architectures for action units of Match-Action switches. The work in [20] decouples the sets of match tables from action stages and replaces the action stages of RMT with packet processors. Due to this disaggregation, the architecture is called Disaggregated RMT (dRMT). Each dRMT processor operates in run-to-completion mode. Once a packet is sent to a dRMT processor, it remains there until the entire program is executed. Therefore, a single dRMT processor is comparable to the entire RMT pipeline in terms of functionality.

Commercial programmable switch chips have replaced fixed-function chips. Examples of these devices include Barefoot Tofino [21] and Tofino 2 [22], Broadcom Trident 3 [23], Tomahawk 3 [24], Tomahawk 4 [25], and Innovium Teralynx [26]. An interesting observation is that most of these architectures are similar in that they contain a programmable packet parser followed by a flexible pipeline with a number of stages and tables. The difference is in the supported throughput, supported workloads, size of tables, programmability, and flexibility.

In the meantime, numerous solutions based on Field Programmable Gate Array (FPGA) have appeared. FPGAs run at considerably lower frequencies compared to ASICs. In order to sustain high throughputs, the FPGA is configured to implement protocol-dependent hardware for the workload that is to be run on the device. This means that the architecture contains protocol-specific state. This is in contrast to architectures such as RMT that contain no protocol-specific state and achieve functionality for different protocols via purely software means. Another issue with using FPGAs for packet processing is that Ternary Content Addressable Memory (TCAM) has to be emulated through the embedded memory blocks. With protocol-specific hardware architecture and ultrawide datapath, FPGAs achieve raw throughput in the range of a few hundred Gbps for packet parsing as in [27]. For packet processing, the achievable throughput is in the range of 100 Gbps [28], [29]. For Terabit-level throughput, ASICs are the only solution. Therefore, FPGA-based solutions are not within the scope of this paper.

1) A CLOSER LOOK AT MATCH-ACTION ARCHITECTURES

The Protocol Independent Switch Architecture (PISA) has its roots in the RMT architecture that first appeared in [12]. It is currently the underlying basis of commercial products such as Barefoot Tofino and Tofino 2. According to [30], Barefoot Tofino contains 4 pipelines, each of which is based on RMT. In this paper we refer to RMT and PISA interchangeably despite potential differences. The two main components of PISA are the parser instances and the pipeline. The parsers extract a part of the arrived header and append a tag to it to form a search key which is presented to a TCAM. The outcome of this matching determines the action to be performed. The main action for the parser is to write the header fields to a 4-Kb register called Packet Header Vector (PHV). The pipeline consists of 32 Match-Action stages through which the PHV traverses. Each stage starts by generating a search

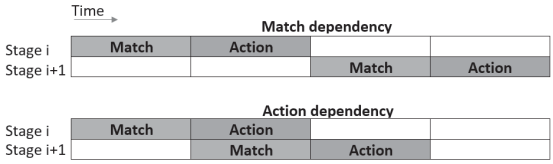

FIGURE 1. Delaying Match and Action as a result of dependencies.

key and providing it to the exact and ternary match tables. The outcome of the match then determines the instructions that must be executed by the action engines.

Depending on the dependency in the packet processing program running on the architecture, it is possible that matching in the next stage begins while action execution in the current stage is still ongoing, or alternatively, the next stage has to wait until the current action execution is entirely over until matching in the next stage begins. Match dependencies occur when a field under modification in a stage must be used for forming the search key in the subsequent stage. Action dependencies occur when field being modified in an action stage needs to be used as input for action in subsequent action stage. Fig. 1 illustrates the timing of Match-Action operations in two consecutive Match-Action stages in case of dependencies. It should be noted that both match and action operations take a number of cycles each.

Each of the Match-Action stages contains 16 TCAM blocks for ternary matching. In addition, there are 106 SRAM blocks that can be configured for exact match, action, and statistics purposes. The dimensions of TCAMs and SRAMs are 2K $\times$ 40 and 1K $\times$ 112 bits respectively. The action subunit contains 224 action engines, one for each PHV entry. Each Match-Action stage is referred to as a physical stage because it directly corresponds to its physical implementation. Sometimes, the match capacity of a physical stage is not sufficient for the required use case. In these cases, the capacity from multiple physical stage can be combined. The combined stages are referred to as a logical stage. For instance, it is possible to combine all 32 Match-Action stages into one logical stage in order to store 1 million IPv4 prefixes in all the TCAMs available on the chip.

dRMT [20] is also a Match-Action architecture but instead of being a pipeline, it is a processor or in other words, a run-to-completion architecture. As a result, each processor must have the entire packet processing program in its instruction memory. The overall dRMT architecture consists of 32 Match-Action processors each of which contains 32 action engines. As opposed to the RMT architecture in which a set of lookup tables are assigned to stages, in dRMT, sets of tables called clusters can be selected to be assigned to a given processor by means of crossbars. As such dRMT has disaggregated the packet processing units and the lookup tables.

One of the major design choices for hardware-based packet processing systems is that of pipeline versus processor. We believe that a pipelined architecture such as that of RMT is more suited to packet processing for a number of reasons:

Packets arrive at high speeds and must each undergo a set of steps. A pipeline achieves this inherently. If a pipeline is deep enough, the extra processing required by a packet can be accommodated without hurting throughput. In a run-to-completion processor, if a packet requires extra processing, the processor cannot accept a new packet at the designated interval unless it supports a large number of independent threads to avoid falling behind. The high-end commercial products we referred to earlier use pipelined architecture.

Second, the RMT architecture already has quite a lot of crossbars. dRMT architecture goes even further by allowing table clusters to be assigned to the processors. Crossbars contribute to the area and power dissipation of the chip.

Last but not least, the run-to-completion nature of dRMT limits the number of action engines and the depth of the instruction memory attached to them. Because the packet remains assigned to a dRMT processor until all required processing is done, the instruction memory in each dRMT processor contains the whole program, while in a pipelined architecture, the program is divided into instruction memory in each stage. In order to increase the supported throughput, multiple dRMT processors are instantiated. The contents of the instruction memory of different processor instances is identical. Therefore, we must limit the number of Arithmetic Logic Unit (ALU) instances to limit the overall memory size across all processor instances.

B. MOTIVATION

The motivation behind this work is overcoming the shortcomings in the PISA architecture. These shortcomings result in a high area overhead and inefficient use of resources such as match tables and instruction memories. We explore these shortcomings in this section.

1) SHORTCOMINGS OF CURRENT MATCH-ACTION ARCHITECTURES

Based on the discussion above, we maintain our main focus on the RMT architecture. These shortcomings are as follows:

Use of TCAMs for packet parsing: TCAMs are powerful devices for matching. They can search all their entries in parallel and provide the matching entries in one clock cycle. The capability to store *don't care* values and the availability of a built-in priority encoder makes them perfect for wildcard and longest prefix matching (LPM). However, wire-speed packet parsing could be performed more area- and power-efficiently without using TCAMs.

Lack of action depth: In the PISA architecture, there is only one stage of action execution after each match stage. Actions such as IPv4 checksum verification and calculation require a number of action stages. In order to fulfill such criteria in the PISA architecture, match tables in the next match stage must be used for the same purpose, which is wasteful. An improved PISA must have configurable action depth. In other words, what is desired is Match $+ \sum$ Action.

Match-based program control: PISA architecture strictly uses matching for program control. For instance, in order to check if the Hop Limit of an IPv6 packet is zero, it matches it against the entries of a table. This strict use of lookup tables for program control wastes match entries. As we will see, there are alternative means for program control whereas for address lookup there is no other alternative other than using TCAM- or SRAM-based tables.

Limited field referencing: PISA architecture allows only directly specified header fields to be used as source operands or the destination. Some protocols require more advanced means of addressing the header fields. For instance, the field for reading or writing could be specified by another header field. Using a header field that acts like a pointer as a search key to obtain an instruction that directly specifies the right field leads to inefficient filling of instruction memory entries.

High cost of table combination: The PISA architecture supports table combination for making wider and/or deeper tables in each match stage. Hardware support for table combination can be very complex. Due to the large number of combinations and complexity of combining states, an area-efficient way to provide hardware support for table combinations is allowing groups of 2^n tables to be combined. In such as system, if, for instance, a given logical table has the width of 120 bits and depth of no more than 6144, the actual table will be 160 bits wide and has depth of 8192. This table is 1.7 times larger than the required table. Providing hardware support for any combination is very expensive due to the number of possible combinations. It is not clear to what degree hardware support for combining tables has been provided in PISA. In case of limited support, tables will be assigned inefficiently, and capacity will be lost. Conversely, if full support is provided, the hardware cost is very high.

In order to increase the utilization of tables, a tag can be appended to the search key so that the table could be reused for as many purposes as there are different combinations of the tag value. If there are not enough tables remaining for the lookup requirements of a packet, the packet must be recirculated to access the tables that it had surpassed in the first round of traversing the pipeline. Recirculation cuts throughput of the pipeline by half. In addition, once a packet is about to be recirculated, it has to compete with other packets that try to enter the pipeline. However, if we could assign no more than the required number of tables for building a logical table, tables would be assigned in a far more efficient manner. In addition, this gives the possibility to provide narrower physical tables. This results in significant savings in area.

2) SIGNIFICANCE OF LOW-AREA MATCH-ACTION PIPELINES

Low-area architectures enable lower fabrication costs and increase production yield. When it comes to packet processing architectures, low area becomes critical because these architectures contain substantial amount of memory for exact and ternary match tables. Savings in area allow integrating more on-chip memories for match tables, thereby increasing the match capacity, which is one of the metrics for evaluating switch chips.

When it comes to Terabit-level packet processing, the issue of low area becomes far more crucial because pipeline instances must be replicated in order to sustain throughput. For instance, Barefoot Tofino contains four independent pipelines [30]. Each packet processing pipeline in a high-end programmable switch contains hundreds of memory blocks. Area optimizations ensure that physical constraints are met and that the pipeline instances can fit into the chip. Therefore, in the architecture presented in this paper, low-area design is a key goal.

III. A NEW PROGRAMMABLE PACKET PARSER

A packet is made up of a number of headers. The parser starts with the first header and finds its way into subsequent headers. How deep the parser digs into the packet depends on the number of headers present in the packet and functionality of the parser. A network switch is concerned with layer-2 headers, whereas a router or layer-3 switch uses the contents of layer-3 headers as well. Therefore, the functionality of the device in which the parser is deployed defines how deep the headers must be parsed. Layer-4 systems such as TCP Offload Engines require the contents of the layer-4 header. The most extreme case of parsing a packet is Deep Packet Inspection (DPI) in which the payload of the packet is examined as well. DPI is more advanced than packet parsing as it has to be aware of the patterns of application data in the subject application. We are not concerned with DPI in this paper.

A packet parser operates in state machine manner for traversing headers. Even the simplest parsers that only parse one header need to maintain states to provide the required functionality when dealing with the header and payload of the packet. For correct operation, the parser requires precise information regarding the following points:

- Current header under parsing
- Progress made so far in parsing the current header
- Next header
- Size of current header
- Whether current header is the last header
- When to switch to parsing the next header

Packet parsing is a straightforward problem. What makes parsing of some headers more complex than that of others is their variable length. With such headers, calculating the size of the header in a real-time manner considering the line rate could become challenging. For instance, in Generic Routing Encapsulation (GRE) header, presence of four of the fields are dependent on the value of three flag bits. The total size of the GRE header varies depending on which flags are set. As another example, in an Ethernet frame, if the value of EtherType field is 0×8100, VLAN tag is present. This adds 4 bytes to the size of the header. Some headers have a field indicating the size of the header. However, such indications use different encodings. For instance, in IPv4 header, the size

Packet\Time	t	t+1	t+2	t+3	t+4	t+5	t+6
P_1	$P.H_1$	$P.H_2$	$P.H_3$	$P.H_4$			
P_2		$P.H_1$	$P.H_2$	$P.H_3$	$P.H_4$		
P_3			$P.H_1$	$P.H_2$	$P.H_3$	$P.H_4$	
P_4				$P.H_1$	$P.H_2$	$P.H_3$	$P.H_4$

FIGURE 2. Parsing of headers in a pipelined manner.

of the header in terms of the number of 32-bit words is indicated by the IHL field. In IPv6 Extension Headers, the size of the extension header in terms of number of bytes minus the first 8 bytes is given. Therefore, the parser must interpret these values correctly for correct operation.

The toughest workload for a packet processing system including the parser is when a minimum-sized packet arrives every clock cycle. This requires the toughest performance guarantees because minimum-sized packets strain the resources of the system. In other words, it is easier to achieve higher throughputs when non-minimum-sized packets arrive because the payload of the packet does not require processing. Therefore, it relaxes the strain on the resources of the system. However, for the throughput figure of a packet processing system to be reliable, minimum-sized packets are the basis for evaluation. In an 800 Gigabit/s link, a new Ethernet frame arrives every 0.84 nanoseconds. This means that a system operating at clock frequency of 1.19 GHz that reads an Ethernet frame every clock cycle can sustain 800 Gbps throughput. If each frame contains multiple headers that must be parsed, they cannot be parsed in one clock cycle and the parser lags behind. The solution is to have the packet go through a number of header parsers, each in charge of parsing one of the headers in the packet. Fig. 2 illustrates the stages that four packets will go through with respect to time. $P.H_n$ refers to parsing of n_{th} header within the packet. In this illustration, it is assumed that each of the four packets has four headers to be parsed and that parsing of each of the headers takes one clock cycle.

These header parsers are equal in the generic parsing functionality. However, each one of them is specialized for parsing the headers of a specific layer. This means that the first header parser is programmed to parse all possible headers that appear first in the packet. The second header parser has the program to parse all the headers that appear as second header in the packet and so on. Fig. 3 is an illustration of a parse graph with three levels.

Parse graph is a tree-like data structure with nodes corresponding to headers. Nodes in level n of the tree represent possible n_{th} header of the packet. For instance, in Fig. 3, the second header of the packet in this setting could be IPv4, IPv6, VLAN, or MPLS. If the header parser discussed so far is to be used for parsing packets based on this parse graph pattern, the first header parser must have the program to parse Ethernet header. The next header parser must have the programs for parsing IPv4, IPv6, VLAN and MPLS. The third header parser must be able to program IPv4, IPv6, MPLS and

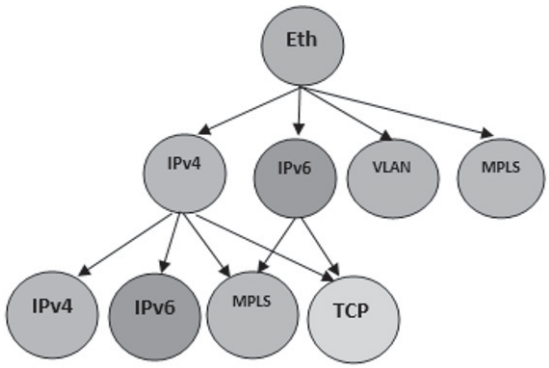

FIGURE 3. Parse graph with three levels.

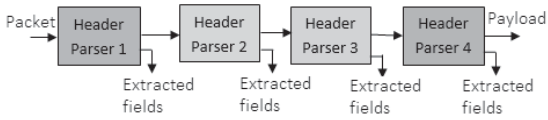

FIGURE 4. A packet parser with four header parsers.

TCP. One important observation is that some headers appear in more than one level. For instance, in the parse graph of Fig. 3, IPv4 and IPv6 headers can appear both as second and third headers. In order to sustain the throughput, second and third header parsers must both have the program to parse these headers. Another interesting observation is that two distinct headers of a given layer can both have the same next header. Referring back to the parse graph in Fig. 3, both IPv4 and IPv6 can have MPLS as the next header. In the implementation, both these cases must be mapped to the same program.

Fig. 4 illustrates the packet parser that Fig. 2 is based on. Each header parser provides the starting offset of the next header to the subsequent header parser. Fig. 5 illustrates a high-level view of the internals of the header parser. The functional units within the header parser are used for finding out the next header, calculating the size of the header, and writing the header fields to PHV entries. These functional units operate in a manner similar to the corresponding functional units in [31].

Internally in our packet parser, each header is represented by a 4-bit Header ID. This representation is only of significance for programming the parser and is independent of encodings used in headers. This value is used to retrieve the Parse Control Word (PaCW) which provides the control signals for the functional units within the header parser.

Information in the PaCW is the minimum information required for correctly parsing a header. The fields within the PaCW and their descriptions are outlined in Table 1. In addition to the PaCW, there are some data associated with each of the headers supported by a header parser. Table 2 outlines

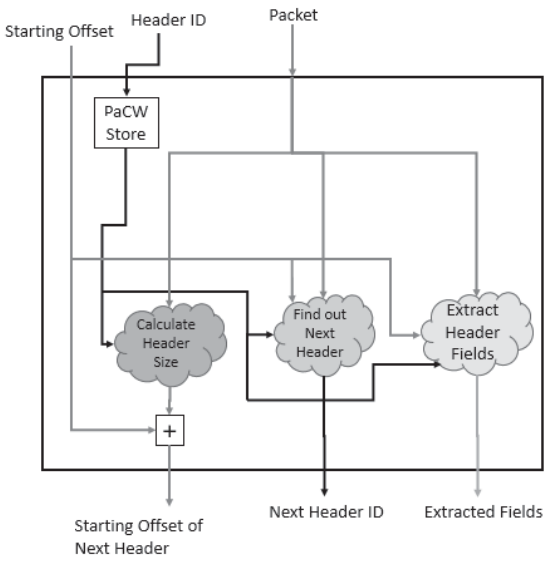

FIGURE 5. High-level view of the header parser.

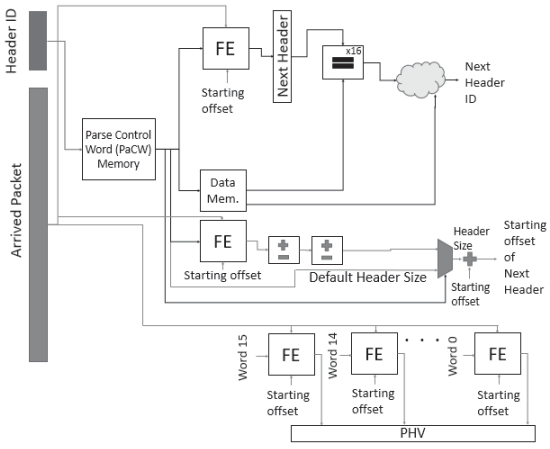

FIGURE 6. Internals of the header parser.

these data. Fig. 6 illustrates the internals of the packet parser in more detail.

The fixed latency for parsing of headers by a header parser is 5 cycles. Since a header parser is internally pipelined, it can be thought of as having five single-cycle stages. Therefore, it could accept a new packet on each clock cycle.

Each header parser can be programmed to parse up to 16 distinct headers. The internal stages of the header parser are as follows:

- Retrieval of PaCW: The PaCW is fetched from the PaCW Store based on the header ID provided by the previous header parser. If this is the first header parser, the correct header ID has already been configured.
- Next Header and Header Size field extraction: In this stage, the fields that contain indication of the next header and header size are extracted using field extractors (FE). If such fields are not present, the PaCW instructs the parser to use other means for calculating the next header and header size.
- Comparison: The value of fields extracted in the previous cycle is compared with the data associated with the header in question. Meanwhile the shifter is shifting the value of the field containing the header size if the PaCW instructs it to do so.
- Resolving: The highest-priority matching entry is used as the basis for determining the next header and current header size. At the same time, an ALU modifies the original or shifted value of the header field containing the header size.
- Header field extraction: In this stage, fields of the header are extracted to be written into the PHV.

As we can see from Fig. 6 and the stages elaborated above, neither finding out the next header nor calculating the header size requires the use of TCAM in our architecture. For finding out the next header, the value of the next header field is extracted and compared in parallel with 16 values associated with the current header. If there is no next header field, default header associated with the current header is selected. For calculating header size, the field containing header size is extracted and passed through a shifter and an ALU. It is also possible to assign the default size of the current header as header size.

As mentioned earlier, the main building block of our packet parser is the header parser. When dealing with use cases and packets that have more than one header for parsing, using more than one header parser inside the packet parser allows the flow of one minimum-sized packet per clock cycle to progress without stall. Header parser n parses the n_{th} header. Otherwise the packet has to be recirculated in which case throughput is degraded. Another benefit of having multiple header parsers inside the packet parser is that if a header is too complex to be parsed using the resources of one header parser, it is parsed by more than one header parser. In this case, each one of the header parsers involved partially parses the header until it is fully parsed.

A. PARSING EXAMPLES

1) PARSING GRE HEADER

The GRE header starts with a nibble containing three flag bits indicating presence of three 32-bit words in the header. In the first parsing stage, the PaCW for parsing GRE header is fetched. In the second stage, the Protocol Type field in the GRE header is extracted using the byte offset information in PaCW. The most efficient way of calculating the header size is by extracting the flag bits and mapping each value to the corresponding header size. Otherwise, the flag bits have to be added one by one and the result must be multiplied by 4 to obtain the header size in bytes. Therefore, in this stage,

TABLE 1. Parse control word (PaCW) entries.

Field	Width (bits)	Purpose
Default header size	6	Default size of the header in bytes.
Offset of next header field	6	Byte offset of the field containing indication of next header
Offset of header size field	6	Byte offset of the field containing header size
Default next header	4	The header that follows current header by default
Opcode for header size manipulation	5	Operation for manipulating the value of the field containing header size
Operand for field manipulation	6	The second operand for manipulating the value of the field containing header size
Header size MUX select	2	Select whether the size of current header should be based on lookup, field manipulation result or the default size associated with this header
Next Header MUX select	1	Select whether the next header should be based on lookup result or the default next header associated with the current header
Last header	1	Whether current header is the last header for parsing
Payload forwarding on no match	1	Whether payload forwarding should be started after current header if no match is found for next header

the flags are also extracted. In the third stage, the value of the Protocol Type field is compared with the comparands. In parallel, the value of flags is also compared with all the possible values. In the fourth stage, the associated data of highest-priority matching entry is selected for next header and header size. In the final stage of parsing, all the header fields present in the header are written to the PHV in parallel.

2) PARSING IPv6

IPv6 header is relatively straightforward to parse. In the first stage of parsing, the PaCW corresponding to IPv6 header is retrieved. In the second stage, based on the information contained in the PaCW and the starting offset provided to the header parser, the Next Header field is extracted. Since the size of IPv6 header is fixed, the PaCW does not contain any information regarding the location of a field specifying the header size. Instead, it contains value of 40 as the default header size. In the third stage of parsing, the value contained in Next Header field is compared in parallel with 16 comparands to find a match. In the fourth stage, the highest-priority matching comparand is used as the basis for determining the next header. In the final stage, the ID of next header is presented to the next header parser and all fields contained in the IPv6 header are written to the PHV in parallel.

TABLE 2. Data associated with each header.

Data	Width (bytes)	Definition
Next Header Comparands	32	16 values to be compared in parallel with the value of the field containing next header indicator in order to find out the header following current header
Next Header Associated IDs	8	Header IDs associated with each of the Next Header Comparands
Header Size Comparands	16	16 values to be compared in parallel with the value of the field indicating the size of the header
Header Size associated values	12	Header size (in bytes) associated with each of the Header Size Comparands
Tag/Packet ID	10	Describes the packet in detail and is used as basis for performing the required packet processing

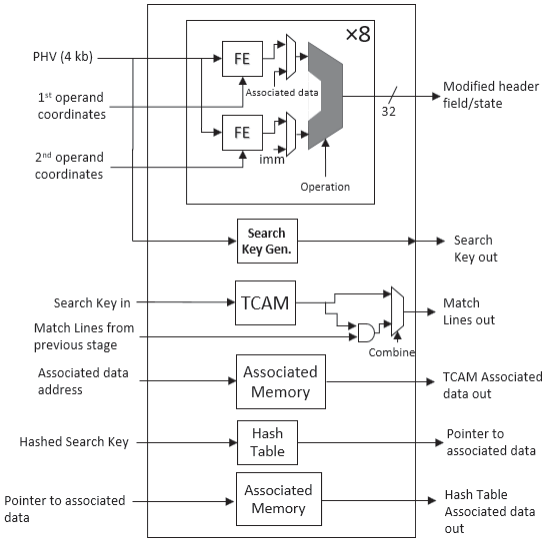

FIGURE 7. Internals of a packet processing stage.

IV. A FLEXIBLE PACKET PROCESSING PIPELINE

The packet processing pipeline is made of packet processing stages each of which performs part of the processing. Fig. 7 is an illustration of a packet processing stage, which is the

fundamental building block of this pipeline. The number of these stages is 512, indexed from 0 to 511. During these stages, action execution as well as matching overlap. The set of packet processing operations within a stage is determined by the packet ID assigned by the parser. The packet ID can be modified in the pipeline as a result of condition evaluation or an earlier match operation.

Besides action execution in each stage, an exact match operation is executed in which the hashed values of an exact match search key is presented to a 4-way hash table to retrieve the data associated with it. A ternary match operation is also executed in which the table hosting the search keys is a ternary table, meaning that it can store don't care values as well.

The main functional units within a packet processing stage are as follows:

- Field extractors (FEs): Extract 8-, 16- and 32-bit fields from the PHV for processing.
- Field- and state-modifiers: There are eight field- and state-modifiers in each stage. They perform logical and arithmetic operations on header fields and state. Field modifiers are 32-bit units that take two inputs. The first input is either a header field or state, and the second input is either a header field or an immediate value. Each field modifier can write to 16 designated locations within the PHV.
- Search key generators: Construct a 40-bit search key by selecting the constituent fields from the PHV.
- TCAM: Each packet processing stage contains one 2048-entry TCAM. It takes a search key as input and provides *match lines* at the upcoming cycle. There are as many match lines as the number of entries within the TCAM. A value of 1 at a given position in the match line indicates that the corresponding entry matched the search key.
- Hash tables: Each stage contains four hash tables for 4-way hashing. Each table is constructed using a 1K $\times$ 64-bit SRAM block. Hash tables contain key-value entries. Key is the search key and the value is a 10-bit tag, also referred to as packet ID (PID).

Once an exact match search key is provided, it is hashed in order to retrieve the position of the search key within the hash table. All ways are accessed in parallel. The value associated with the matching way is selected. The tag becomes the new tag, which is the basis for instruction and data retrieval.

Both ternary and exact match tables have memories associated with them. They contain packet processing parameters such as header templates and header field values or statistical state associated with a search key. The choice of whether to use the TCAM or the hash tables depends on the kind of search required. For instance, for looking up IPv4 addresses, the TCAMs are great because they can perform single-cycle LPM search. If, on the other hand, the Tag for processing an IPv6 Extension Header is to be obtained, the hash tables must be used.

A. PROGRAM CONTROL

The instructions to execute at each stage are determined by the value of a 10-bit tag. This tag is first set by the parser. This tag is used to retrieve the instructions at each stage. It gives detailed information about the packet. For instance, a given value could be used for an IPv6 packet whose Hop Limit is zero. In this case, the instructions for making an ICMPv6 Time Exceeded Message are fetched. When using the same tag in a number of stages, part of the required actions is executed in each of the stages involved and thereby the requirement of custom action depth is achieved. What makes this architecture flexible is that the 10-bit tag could be changed as the packet traverses the pipeline. These features allow implementation of actions that are far more complex than OpenFlow v1.5.1 [32] actions. Each stage has the following functional units for program control:

- PID Map Table: This table maps the 10-bit ID of the incoming packet to a 64-bit value which contains instruction pointers for each of the functional units within the packet processing stage. This means that each functional unit has a separate instruction memory that can be independently addressed. By using this technique, many distinct instruction combinations can be achieved without using a deep instruction memory. The mapping for each PID and each stage is decided by the programmer. The PID map table is allocated from the SRAM blocks available at each stage. Therefore, it does not consume any additional area compared to SRAM blocks in RMT and dRMT.
- Condition evaluator: This unit performs operations such as bit extraction and magnitude comparison. The result of this unit's operation can be used to change the 10-bit tag, which in turn changes the program flow.

In this architecture, there are condition flags to represent the status of the latest lookup in ternary and exact match tables. The evaluation of these flags can also be the basis for program control.

B. COMBINING TABLES

A 512-stage pipeline is a deeply pipelined architecture. The latency is directly associated with the number of stages. Before reaching a verdict on the latency of this pipeline, let's review some of the latency figures of the original PISA architecture when it comes to dependencies. In the original PISA architecture, there is a 12-cycle latency for match dependencies and 3-cycle latency for action dependencies [33]. The reason for this is that if, under dependency conditions, the operation of functional units of different match stages is overlapped, the old header field values will be used for search key generation or action execution. Therefore, delays are configured to ensure that the succeeding match stage will use the updated PHV.

In our architecture, accessing each table takes a cycle. Two cycles after accessing the match table, the outcome of whether a match was found or not is known. In case of

positive match, another two cycles are required to obtain the corresponding value stored in the associated memory. The 4-cycle latency after accessing tables is a fixed value, whether one table has been accessed or multiple tables. The two tables that are visited during the cycles required for retrieving the associated data are simply ignored. No stalling or delay configuration occurs in our architecture. The cost of losing two tables is considerably less than that of losing 16 tables, as is the case in RMT. The two ignored tables could be used for speculative lookup. This way, possible wasting of lookup resources is eliminated.

Any number of tables could be combined for making wider and/or deeper tables. As the packet traverses the pipeline, one table is visited at each stage. If a logical table wider than a physical table is desired, at each stage part of the whole search key is presented to the lookup table within the stage. The resulting match lines are transferred from one stage to the next stage and ANDed together until the whole search key has been looked up. Then the final match line which is the result of AND operation on all of the match lines is used to retrieve the associated state. For making a logical table whose depth is more than a single physical table, the entries of the logical table could be arranged in such a way that physical tables that are visited first have higher priority. The same search key is presented to all the tables involved. Once a match is found, the packet's tag is changed to indicate that the packet no longer requires the same lookup procedure. Making wider and deeper tables is similar and contains both of the procedures mentioned here.

Our flexible pipeline has the means to reduce the latency when a considerable number of physical tables must be combined for accommodating more entries. Each 16-stage unit whose starting index is an integer multiple of 16 is called a PIPE16. Therefore, there are 32 PIPE16 instances in our pipeline, indexed from 0 to 31. The output of a PIPE16 instance is the input to its successor PIPE16. PIPE16 instances can be configured to run in parallel to reduce the latency when 32, 64, 128, 256, or 512 tables are to be combined for making deeper tables. For instance, if the desired depth of a logical table is 64 times that of a single physical table, four PIPE16 instances run in parallel and latency is cut by a factor of four. In this scenario, all the four PIPE16 instances receive the same PHV as input. The pipeline stage that follows these four parallel PIPE16 instances takes the PHV output of the PIPE16 instances that has had the highest priority. The input to the PIPE16 instances can be configured. A 64-bit software-defined Pipeline Configuration Word (PiCW) sets the desired configuration. When running PIPE16 instances in parallel, 100% utilization of the tables involved is achieved if the desired number of physical tables is a power of two. If this is not the case and utilization of tables is the most high-priority criterion, the pipeline can be configured for its conventional configuration, in which each stage receives the output of its immediately preceding stage. Fig. 8 illustrates the pipeline and the components that make the reconfiguration possible. For space-saving reasons,

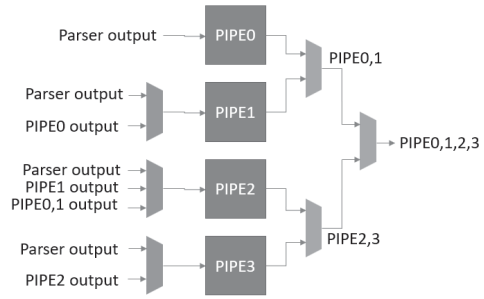

FIGURE 8. Pipeline configuration components.

only the first four PIPE16 instances are shown. The illustrated architecture is repeated for the rest of the PIPE16 instances and the resulting binary tree has three more levels. The key component that picks the higher-priority match outcome is a priority-based 2-to-1 multiplexer. The select line for these priority-based multiplexers are set by the match found flags of the two PIPE16 instances that provide their output to the priority MUX. They also multiplex the value of match found flag so that the next-level multiplexers can function correctly. By having a binary tree of these components, it is possible to run selected PIPE16 instances in parallel. The other component required for the configuration is the set of multiplexers that provide the input to PIPE16 instances. PiCW is the set of values for the select lines of these multiplexers. If the pipeline is configured in its basic form in which the packets have to traverse all the stages, the latency is 430 nanoseconds because the operating frequency is 1.19 GHz. The terabit-level switches of Nexus 9200 family from Cisco have latency figures close to two microseconds [34]. Therefore, even the worst-case latency of our architecture is in reasonable range.

What is meant by input to a PIPE16 instance is the input to the first stage within the PIPE16 in question. For instance, input to PIPE16_{30} means input to stage 480, which is the first stage within PIPE16_{30}. For all stages after the first stage of a PIPE16 instance, the only input is the output of the preceding stage. For instance, for stage 17 which is located in PIPE16_1 but is not its first stage, the only input option the output of stage number 16.

C. INPUTS TO FIELD- AND STATE-MODIFIERS

Field extractors provide the input to the functional units including field- and state-modifiers. The PHV contains 128 32-bit words. This translates to 384 16-bit and 512 8-bit units as well. The reason why there are 384 16-bit units is that for a given PHV word called word_i, $\text{word}_i(31{:}16)$, $\text{word}_i(23{:}8)$, and $\text{word}_i(15{:}0)$ are extracted as 16-bit units. Field extractors are in fact multiplexers with 1024 inputs. Each of the field- and state-modifying instructions have fields for specifying the location of a field within the PHV. When 8- and 16-bit fields are selected, they are zero-extended to 32 bits. Field extractors are one of the major contributors to chip area due to

Instruction Memory Address	VLIW Instruction Slots					
0	MOVE	NOP	NOP	. . .	NOP	NOP
1	NOP	MOVE	NOP	. . .	NOP	NOP
2	NOP	NOP	MOVE	. . .	NOP	NOP
.				.		
.				.		
.				.		
Last address	NOP	NOP	NOP	. . .	NOP	MOVE

FIGURE 9. Instruction memory layout for pointer-based write.

the number of pipeline stages and the fact that each field- and state-modifier requires two field extractors. Therefore, it is desirable to evaluate the possibility of optimizations for saving area. In [35] different alternatives for field extractors are compared. We consider two optimization strategies. In both strategies, it is assumed that the PHV is logically divided into eight equally sized groups.

Based on the observation that it is not necessary for all field extractors to be able to read from the whole PHV, each field- and state-modifier is allowed to access all of the fields within its group but only some of the entries of other groups. In other words, cross-group field retrieval is more limited. In the second optimization strategy, full field extraction capability is available for entries of a group. However, entries of other groups are read only in 32-bit units in order to reduce the number of inputs to the multiplexer and thus have a lighter multiplexer. If an 8- or 16-bit field from the entries pertaining to other groups is required, it must be extracted using the field- and state-modifiers.

Both optimization strategies result in use of multiplexers with 240 inputs as field extractors which occupy 36% of the area of 1024-input multiplexers. The resulting saving is not limited to the crossbars. The number of SRAM blocks required to hold Very Long Instruction Word (VLIW) instruction slots will be reduced too because the instructions will slightly shrink.

D. MORE ADVANCED MEANS OF HEADER FIELD REFERENCING

As mentioned earlier, one of the limitations of the PISA architecture is that its sole means of referencing header fields is directly specifying them in the instruction. If one of the header fields is a pointer specifying the header field for reading or writing, the pointer field has to be used as a search key. The outcome of this match points to the instruction that reads from or writes to the correct field within the PHV. This causes the instruction memory to be inefficiently filled by instructions that are in principle the same. Fig. 9 illustrates the layout of the instruction memory when one of the fields in the header contains the index of the field to which a value must be written. This writing is achieved by using the MOVE instruction. There is an action engine for each PHV entry and each VLIW instruction slot corresponds to an action engine.

As we can see, all these instructions are in principle the same. The only difference is the location of the VLIW instruction slot containing the MOVE instruction. The PHV in RMT architecture contains 224 fields of three different widths. There is an action engine per PHV field. In the worst case, as many as 224 instruction entries will be filled according to the pattern in Fig. 9.

In our architecture, we do not need to use any form of matching in such scenarios. Field modifiers have a specific opcode for reading the content of a header field whose location is specified by a pointer. The location of the pointer within the PHV must be known in advance so that it could be directly referenced. After reading the pointer and executing this opcode, the field referenced by the pointer is provided at the output of the field modifier. In addition to this, there is an opcode for writing to a field specified by a pointer. When this opcode is executed, the location pointed to by the pointer is assigned the intended value even if the destination field is beyond the range of locations to which the writing field modifier can write. For this to be feasible, the writing field modifier overrides all other field modifiers.

E. PACKET PROCESSING EXAMPLES

1) IPv6 SEGMENT ROUTING

Segment Routing (SR) is a type of source routing in which the source determines the nodes that a packet must visit. SR has been discussed in detail in [36]. SR can be implemented using MPLS or IPv6. In the latter case, an IPv6 extension header called Segment Routing Header (SRH) is required. Here we consider SR using IPv6 SRH. In this packet processing walkthrough, we assume that a router based on the architecture proposed in this paper is the endpoint for the arriving IPv6 packet. This means that Destination Address (DA) is the same as the router's address. We also assume that Hop limit is greater than 1 and that SRH immediately follows the fixed IPv6 header. Fig. 10 contains the pseudo-code that must be executed on our architecture.

Since IPv6 extension headers are all independent headers, the SRH has already been parsed by the parser and the corresponding 10-bit tag has been assigned. Each of the header fields referred to in Fig. 10 have a determined place within the PHV.

Fig. 11 illustrates the outline of PHV after parsing is complete. The instructions executed in each stage are outlined in Table 3. It is assumed that R124, R125, R126 and R127 contain the IPv6 address assigned to the device.

Processing in stage 0 begins by comparing DA with the address assigned to the device. After each comparison instruction there is a change label instruction to change the program flow if necessary. Four comparisons are required because IPv6 addresses are 128 bits wide. Selecting the current segment from the list of segments requires pointer-based read. Before pointer-based read can be done, the value of the pointer must be manipulated so that it points to the correct PHV entry.

Due to the width of IPv6 addresses, writing the segment pointed to by the updated value of Segments Left takes four

```
if(Segments_Left == 0)
{
   process_next_header();
}
else
{
   max_last_entry  =  ( Hdr_Ext_Len / 2 ) - 1;
   if((Last_Entry > max_last_entry) or (Segments_Left > (Last_Entry+1))
   {
      Send_ICMP_Parameter_Problem();// Code 0.
   }
   else
   {
      Segments_Left--;
      destination_address = Segment_List[Segments_Left];
      if(Hop_Limit <= 1)
      {
         send_ICMP_Time_Exceeded(); //to the Source Address
      }
      else
      {
         Hop_Limit--;
         Resubmit();
      }
   }
}
```

FIGURE 10. Pseudo-code for IPv6 SRH processing.

0	Version	Traffic Class	Flow Label	
1	Payload Length		Next Header	Hop Limit
2	Source Address			
3	Source Address			
4	Source Address			
5	Source Address			
6	Destination Address			
7	Destination Address			
8	Destination Address			
9	Destination Address			
10				
11				
12				
13				
14				
15				
16	Next Header = 0x3B	Header Ext Len = 0x08	Routing Type	Segments Left = 0x02
17	Last Entry = 0x03	Flags = 0x00	Tag = 0x0000	
18	Segment List [0]			
19	Segment List [0]			
20	Segment List [0]			
21	Segment List [0]			
22	Segment List [1]			
23	Segment List [1]			
24	Segment List [1]			
25	Segment List [1]			
26	Segment List [2]			
27	Segment List [2]			
28	Segment List [2]			
29	Segment List [2]			
30	Segment List [3]			
31	Segment List [3]			
32	Segment List [3]			
33	Segment List [3]			
.	.			
.	.			
.	.			

FIGURE 11. Outline of PHV after the parsing is complete.

TABLE 3. Instructions executed in each stage for IPv6 SRH Processing.

Stage	Instruction	Comments

0	CMPEQ R6, R124
	SUB R17, R16, 1
	MOV R35, R17
		Compare DA with address of this device.
		Decrement Segments Left.
		Move the 2nd word of SR header to R35.

1	CMPEQ R7, R125
	SHL4 R17, R17
	SHR1 R16, R16(Hdr Ext Len)
	MOV R34, R16
	CHNGLBL (comparison flag zero)
		Compare DA with address of this device.
		Multiply the decremented Segments Left by 4.
		Shift Hdr Ext Len one bit to the right (div by 2).
		Move the 1st word of SR header to R34.
		Change Tag if outcome of comparison was negative.

2	CMPEQ R8, R126
	SUB R16, R16, 1
	Add R36, R35, 1
	ADD R17, R17, 18
	CHNGLBL (comparison flag zero)
		Compare DA with address of this device.
		Decrement the shifted value of Hdr Ext Len.
		Increment Last Entry.
		Add 18 to R17 to obtain the word offset of the Segment.
		Change Tag if outcome of comparison was negative.

3	CMPEQ R9, R127 CHNGLBL (comparison flag zero)	Compare DA with address of this device. Change Tag if outcome of comparison was negative.
4	CHNGLBL (comparison flag zero) CMPEQ R34, 0	Change Tag if outcome of comparison was negative. Compare Segments Left with zero.
5	CMPG R34 (Segments Left), R36 CHNGLBL (comparison flag one)	Compare if Segments Left is greater than Last Entry+1. Change Tag if outcome of comparison was positive.
6	CMPG R34 (Segments Left), R16 CHNGLBL (comparison flag one)	Compare Segments Left with Hdr Ext Len. Change Tag if outcome of comparison was positive.
7	CHNGLBL (comparison flag one)	Change Tag if outcome of comparison was positive.
8	MOVINDRCT R6, R17 Add R17, R17, 1	Use R17 as pointer to data and move the contents of pointed position to R6 (IPv6 DA). Increment the pointer.
9	MOVINDRCT R7, R17 Add R17, R17, 1 Ternary Lookup R6	Use R17 as pointer to data and move the contents of pointed position to R7 (IPv6 DA). Increment the pointer. Lookup the new IPv6 DA.
10	MOVINDRCT R8, R17 Add R17, R17, 1 Ternary Lookup R7	Use R17 as pointer to data and move the contents of pointed position to R8 (IPv6 DA). Increment the pointer. Lookup the new IPv6 DA.
11	MOVINDRCT R9, R17 Ternary Lookup R8	Use R17 as pointer to data and move the contents of pointed position to R9 (IPv6 DA). Increment the pointer. Lookup the new IPv6 DA.
12	Ternary Lookup R9	Lookup the new IPv6 DA.

cycles (stages 8 to 11). As soon as the first word of the new IPv6 DA is known, ternary lookup begins (stage 9). An interesting observation is that Segments Left, which acts as a pointer, is already updated in stage 0, so that the process of retrieving the segment to which it points can be started although at this point it is not clear whether it contains a positive value. This kind of execution is speculative. If at

TABLE 4. Tags used in processing of IPv6 SRH.

Label	Meaning
A	An IPv6 packet with SRH has been received.
B	The IPv6 packet with SRH is destined to this device.
C	Segments Left field of SRH is zero.
D	Parameter problem detected in the SRH. ICMP message must be sent.
E	The lifetime of the IPv6 packet is over.

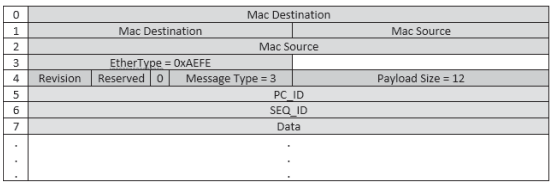

FIGURE 12. Outline of PHV after the parsing is complete.

any point the value of Segments Left turns out to be invalid, the changes can be discarded.

As we can see, the label modification instruction has been extensively used. Table 4 contains the designated labels and their meaning. In this table, labels are referred to with letters because their actual value is implementation-specific and is not of significance in the discussion here. Each of these labels is the basis for retrieving the instructions in each stage. Change of label causes change in program flow.

2) 5G FRONTHAUL TRAFFIC

Common Public Radio Interface (CPRI) is an interface-defining standard for communication between Radio Equipment Control (REC) and Radio Equipment (RE) using the fronthaul transport network. eCPRI is the enhanced CPRI. It connects the eREC and eRE via transport network. eCPRI messages can be encapsulated in Ethernet or IP packets. Here we consider the encapsulation in Ethernet.

Fig. 12 illustrates outline of PHV after parsing is complete. R0-R3 contain Ethernet header, R4 contains eCPRI common header and R5-R7 contain eCPRI Generic Data Transfer message.

The parser has already marked the packet as an eCPRI message. The 1-byte field Message Type from the eCPRI common header is selected as an exact match search key. In this scenario, the value of this field indicates the presence of Generic Data Transfer message after the common header. eCPRI messages have an identifying field called PC_ID at the beginning of the eCPRI message. Depending on the message type, the width of this field is a byte, 2 bytes or 4 bytes. We cannot know the width of this field until the outcome of looking up Message Type is available. To reduce the latency, we generate three exact match search keys, each corresponding to the 3 different sizes of PC_ID field. This way, we don't have to wait until the outcome of matching Message Type is available. It is also beneficial from the perspective of using the tables efficiently because by the time the outcome of matching Message Type is available, two tables are traversed. The outcome of matching PC_ID reveals how the data in the eCPRI message must be handled.

TABLE 5. Area and power of header parser components.

Component	Total area (μm^2)	Total power (mW)
PaCW and Parameter Memories	30 K	52
Field Extractors and manipulators	15.9 K	21.64
Comparators and resolving logic	1.1 K	0.960

TABLE 6. Area and power dissipation of 6.4 Tbps packet parser.

Component	Number of instances	Total area (mm^2)	Total power (W)
Header parsers	64	3	4.7
PHV	8	0.617	4.8

V. EVALUATION AND DISCUSSION

The packet parser and the packet processing pipeline have been implemented using VHDL. The implementation has been synthesized using Synopsys Design Compiler J-2014.09-SP4 on 28 nm FD-SOI technology. The results correspond to supply voltage of 0.9 V and worst-case operating conditions (ss, 125°C). The implementation meets the timing constraints at operating frequency of 1.19 GHz. Post-synthesis simulation has been performed using Mentor Questa.

A. PACKET PARSER RESULTS

Table 5 presents the area and power dissipation of the main constituent components of a single header parser instance. The total area of a header parser instance is 47000 μm^2 and the total power dissipation is 74.6 mW. Table 6 outlines the area and power dissipation of components of a 6.4 Tbps packet parser that can parse packets with depth of eight headers. This packet parser is made of eight pipelines of header parsers. Each such pipeline contains eight header parser instances and can sustain throughput of 800 Gbps. By having eight of these pipelines in parallel, aggregate throughput of 6.4 Tbps can be supported.

The total area of all packet parser instances required for 6.4 Tbps throughput is 3.617 mm^2 or 7.38 M gates. The total area of packet parsers in [12] is 5.6 M gates for 640 Gbps throughput. For reaching 6.4 Tbps throughput, the number of parser instances must be increased by a factor of 10. This causes the resulting total area to be 56 M gates. This means that we have increased the throughput by a factor of 10 whereas the increase in area has been only 32 %. The area difference is equivalent to the area of 137 instances of 2048×32 TCAM blocks.

TABLE 7. Area of the components in a packet processing stage.

Component	Total Area (μm^2)
TCAM	180 K
Field selectors	145.8 K
4-way Exact Match Tables	125.7 K
Instruction Memory	32.7 K
PID Map Table	31.4 K
PHV	18.5 K
Field- and State- Modifiers	12 K
Search Key Selection	9.3 K

B. PACKET PROCESSING PIPELINE RESULTS

Table 7 contains details on the area of the main components of a single packet processing stage. The components in the table are ordered according to their area. For components having multiple instances in each stage, the total area of the instances is given.

As we can see, the major contributor to the area is the TCAM. The next major contributors to the area are field selectors. In section 4 we discussed optimizations for field selectors. The area of the proposed lightweight field selectors is 36% of the original field selectors. In addition, using these crossbars causes the width of field- and state-modifying instructions to shrink. In this optimization, the first field modifier and the condition evaluator still use the large input selectors. The other field modifiers use the light-weight input selectors. By using the lightweight field selectors, an area equivalent to 37 mm^2 can saved. This saving is equivalent to the area of 214 TCAM blocks of 2K $\times$ 32 bits.

All the memories used for storing PIDs, instructions, and search keys are industry-standard dual-ported memories. The control plane can write to these memories while the device is operating. It does so by communicating with the centralized controller using a protocol such as OpenFlow. The area occupied by the memories comes not only from components required for reading, writing, and storing data, but also from built-in test components.

C. COMPARISON WITH OTHER MATCH-ACTION ARCHITECTURES

In this section, we compare the area of our architecture with that of RMT and dRMT. Table 8 compares the area of different components in each stage of the three architectures under comparison. Since dRMT architecture is a processor, the values correspond to one processor instance. For dRMT, we have considered two variants each with a different value for Inter-Packet Concurrency (IPC). It is assumed that all these architectures have equal amount of memory to host both ternary and exact match search keys as well as the data associated with them. The values for RMT and dRMT architectures have been taken from [20] and converted into values that would be obtained after synthesis using 28 nm technology. We have, however, taken the value of match crossbars and ALUs from [12]. According to [12], the total area of match key crossbars in RMT architecture is 6 mm^2, which means that in each stage the area of match crossbars is 0.187 mm^2.

TABLE 8. Area per stage (mm^2).

Component	RMT	dRMT (IPC = 1)	dRMT (IPC = 2)	This architecture
Match key config. Reg.	0.021	0.012	0.015	0.000
Match key crossbar	0.187	0.150	0.217	0.009
PHV	0.336	0.998	1.439	0.018
Scratchpad	N/A	0.156	0.156	N/A
Action input selector	1.448	0.523	0.964	0.079
ALUs	0.200	0.050	0.050	0.012
Action output selector	N/A	0.147	0.147	0.000
VLIW instruction table	1.139	1.029	1.029	0.032
Total	3.331	3.065	4.017	0.150

From the values in the table we can see a noticeable difference in the area of PHV when comparing the area of PHV in our architecture with that of RMT or dRMT architecture variants. The key to understanding this difference is understanding that a stage in RMT architecture is a logical stage. In our architecture, on the other hand, all stages are physical. Each of the Match-Action units in RMT is internally pipelined because there are quite many operations such as search key generation, header field retrieval, match result combination, memory access, etc. taking place in each logical stage and since RMT operates at 1.0 GHz frequency, there is no way that all these operations can take place in one cycle. Therefore, the PHV must be propagated from one physical stage to the next stage. The actual number of physical stages in RMT can be estimated based on the match and action latency values. As a result, the fact that our architecture has 512 stages does not mean that the overall cost of PHV instances in our architecture is more than that of RMT architecture. In fact, the total area of PHV instances in the two architectures are on par with each other.

Table 8 has an entry called Match key configuration register. In our architecture, we have a lookup instruction for ternary matching and another instruction for exact matching.

In the decode stage of both these instructions the components of the search key are selected in the decode stage. Therefore, we do not have any register to hold match key configuration. This indicates that our architecture is more flexible in supporting diverse set of search keys.

One of the issues with the analysis in [20] is the way the area of ALUs has been estimated. From [12], the authors of [20] have used the 7.4% share of contribution of action engines to overall area as the basis for calculating the area

of ALUs. In order to obtain the total area of RMT, they have used the 200 mm^2 value from [18]. This value represents a lower bound on the area of a commercial 640 Gbps switch chip. There is no evidence that this value represents the total area of RMT. Besides that, the process technology associated with this value has not been mentioned in [18]. Another issue with the values calculated by [20] is that the ratio of the area of ALUs in RMT and dRMT is inconsistent with the number and width of ALUs used in the two architectures. According to our experiments, the area of a 16-bit ALU is half the area of the corresponding 32-bit ALU. Similarly, the area of an 8-bit ALU is a quarter of the area of the 32-bit ALU with same functionality. Instead of the estimation in [20], we use the per-bit gate count provided in [12] because it is based on results from implementation. According to [12], each action engine requires less than 100 gates per bit. Based on this assumption, the area of a 32-bit action engine is around 1500 μm^2. We assume that all action engines used in the three architectures being compared are equal in internal architecture.

As for action output selectors, each ALU in our architecture writes to a fixed set of locations within the PHV. The ALUs together cover the whole PHV. The area of action output selectors in our architecture is almost zero in mm^2 scale.

Based on the values of Table 8, Table 9 contains the area for all stages of the two pipelined architectures and in the case of dRMT architecture variants, the area for all processors. Furthermore, the area for table combination logic is provided. In dRMT architecture, there is logic for both table combination within a cluster and assignment of clusters to processors. In our architecture, there is tiny logic for configuring the organization of pipeline. This area corresponds to the multiplexers providing input to the PIPE16 instances and the 2-to-1 priority-based multiplexers receiving the output of certain PIPE16 instances.

According to Table 9, RMT architecture has 44 % more area than our architecture. dRMT variants have 41 % and 79 % more area than our architecture despite lacking the features of the architecture presented in this work. In order to be able to interpret these numbers, we should compare them with the latest area figures for commercial switch ASICs, which are 300-700 mm^2 [20]. All the architectures under comparison are within this range. However, our architecture is notably ahead of others in area-efficiency. The savings in area can be used for integrating more TCAMs and/or exact match tables and thereby increasing the match capacity of the system.

TABLE 9. Area for all processors plus interconnect (mm^2).

Architecture	Non-crossbar area (mm^2)	Crossbar area (mm^2)	Total area (mm^2)
RMT	106.592	6	112.592
dRMT (IPC = 1)	98.080	11.328	110.128
dRMT (IPC = 2)	128.544	11.328	139.872
This architecture	76.800	1	77.800

VI. CONCLUSIONS

In this paper, we presented the architecture of a programmable packet parser and a flexible packet processing pipeline. The parser supports 6.4 Tbps throughput without relying on expensive TCAMs. As a result, its area is very modest for its level of performance. The packet processing pipeline allows fine-grained table assignment and unlimited combination of tables at minimum possible cost. It also provides more advanced features such as custom action depth, alternative program control, and an addressing mode for pointer-based read and write. All of this is achieved while still being considerably more area efficient than the current Match-Action architectures, namely the RMT and dRMT architectures.

Chip area is a measure of complexity of the logic inside a chip. For a given functionality and performance level, a chip with lower area is more desirable. Digital ICs are subject to various constraints. One such constraint is area. The significance of low-area design is that the savings in area could be used for providing more complex logic for enhanced functionality. In packet processing architectures, this saving can be exploited for more functional units. By doing so, the functionality and/or supported throughput of the system will be enhanced.

Performance comes not only from the hardware side, but from the software side as well. One of the techniques used in the packet processing examples presented in this paper was software-based speculative execution. When a match is in progress, the possible actions can be executed speculatively. When the match result is ready, the outcome of the corresponding action is committed, and the other results are discarded. By doing so, the overall latency of match and action is reduced.

As for future work, we intend to work further on the architecture for supporting higher throughputs and providing further flexibility. The idea of breaking the pipeline into PIPE16 instances with the aim of reducing latency when deeper tables are required, can be expanded for having multiple independent pipelines, each of which processes packets with the same packet processing requirements. This enhances packet-level parallelism. Each packet is dispatched to the corresponding pipeline depending on its needs. Different pipelines deal with different packets. The architectural components required are dispatch logic and independent deparser at the end of each independent pipeline. We also plan to develop a P4 compiler for this architecture.

REFERENCES

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[2] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *Proc. IEEE 19th Int. Conf. High Perform. Switching Routing (HPSR)*, Bucharest, Romania, Jun. 2018, pp. 1–7.

[3] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 27–51, 1st Quart., 2015.
[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[5] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 32–127.
[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Chlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.
[7] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger, and L. Dittmann, "Cloud RAN for mobile networks—A technology overview," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 405–426, 1st Quart., 2015.
[8] L. M. P. Larsen, A. Checko, and H. L. Christiansen, "A survey of the functional splits proposed for 5G mobile crosshaul networks," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 146–172, 1st Quart., 2019.
[9] *eCPRI Interface Specification V1.2-Common Public Radio Interface*, Ericsson AB, Huawei Technol., NEC Corp., CPRI, Nokia, Espoo, Finland, Jun. 2018.
[10] *IEEE Approved Draft Standard for Packet-Based Fronthaul Transport Networks*, IEEE Standard P1914.1/D5.3, Sep./Nov. 2019, pp. 1–92.
[11] *IEEE Standard for Radio over Ethernet Encapsulations and Mappings*, IEEE Standard 1914.3-2018, Oct. 2018, pp. 1–77, doi: 10.1109/IEEESTD.2018.8486937.
[12] 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," in *Proc. ACM SIGCOMM Conf.*, Hong Kong, Aug. 2013, pp. 99–110.
[13] Barefoot Networks. *The World's Fastest & Most Programmable Networks*. Accessed: Apr. 25, 2020. [Online]. Available: https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/
[14] H. Khosravi and T. Anderson, *Requirements for Separation of IP Control and Forwarding*, document RFC 3654, IETF, Nov. 2003.
[15] L. Yang, R. Dantu, T. Anderson, and R. Gopal, *Forwarding and Control Element Separation (ForCES) Framework*, document RFC 3746, IETF, Apr. 2004.
[16] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, Kyoto, Japan, Aug. 2007, pp. 1–12.
[17] M. Shahbaz and N. Feamster, "The case for an intermediate representation for programmable data planes," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res. (SOSR)*, Santa Clara, CA, USA, Jun. 2015, pp. 1–6.
[18] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *Proc. Archit. Netw. Commun. Syst.*, San Jose, CA, USA, Oct. 2013, pp. 13–24.
[19] 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 *Proc. Conf. ACM SIGCOMM Conf. (SIGCOMM)*, Florianópolis, Brazil, 2016, pp. 15–28.
[20] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "DRMT: Disaggregated programmable switching," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Los Angeles, CA, USA, Aug. 2017, pp. 1–14.
[21] Barefoot Networks. *World's Fastest P4-Programmable Ethernet Switch ASICs*. Accessed: Apr. 25, 2020. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino/
[22] Barefoot Networks. *Second-Generation of World's Fastest P4-Programmable Ethernet Switch ASICs*. Accessed: Apr. 25, 2020. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino-2/
[23] Broadcom. *High-Capacity StrataXGS Trident 3 Ethernet Switch Series*. Accessed: Apr. 25, 2020. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series
[24] Broadcom. *12.8 Tb/s StrataXGS Tomahawk 3 Ethernet Switch Series*. Accessed: Apr. 25, 2020. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56980-series
[25] Broadcom. *25.6 Tb/s StrataXGS Tomahawk 4 Ethernet Switch Series*. Accessed: Apr. 25, 2020. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series
[26] Innovium. *Teralynx: The World's Most Scalable Switch Family-1.2 Tbps Through 12.8 Tbps With Industry Leading Analytics, Lowest Latency and Programmability*. Accessed: Apr. 25, 2020. [Online]. Available: https://www.innovium.com/teralynx/
[27] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *Proc. ACM/IEEE 7th Symp. Archit. Netw. Commun. Syst.*, Brooklyn, NY, USA, Oct. 2011, pp. 12–23.
[28] 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, Sep. 2014.
[29] G. Brebner and W. Jiang, "High-speed packet processing using reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 8–18, Jan./Feb. 2014.
[30] P. Bosshart, "Programming forwarding planes at terabit/s speeds," presented at the Hot Chips, Symp. High Perform. Chips, 2018. [Online]. Available: https://www.hotchips.org/archives/2010s/hc30/
[31] H. Zolfaghari, D. Rossi, and J. Nurmi, "A custom processor for protocol-independent packet parsing," *Microprocessors Microsyst.*, vol. 72, Feb. 2020, Art. no. 102910.
[32] Open Networking Foundation. (Mar. 26, 2015). *OpenFlow Switch Specification Version 1.5.1*. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf
[33] G. Gibb, "Reconfigurable hardware for software-defned networks," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, USA, 2013. [Online]. Available: https://stacks.stanford.edu/file/druid:ns046rz4288/gibb-thesis-augmented.pdf
[34] CISCO. *Nexus 9200 Compare Models*. Accessed: Apr. 25, 2020. [Online]. Available: https://www.cisco.com/c/en/us/products/switches/nexus-9000-series-switches/nexus-9200-models-comparison.html
[35] H. Zolfaghari, D. Rossi, and J. Nurmi, "Reducing crossbar costs in the match-action pipeline," in *Proc. IEEE 20th Int. Conf. High Perform. Switching Routing (HPSR)*, Xi'an, China, May 2019, pp. 1–6.
[36] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, *Segment Routing Architecture*, document RFC8402, IETF, Jul. 2018.

HESAM ZOLFAGHARI (Graduate Student Member, IEEE) is currently pursuing the Ph.D. degree with Tampere University. His research interests are design of programmable and protocol-independent packet processors for software defined networking with special focuses on low on-chip area, low-power dissipation, and minimized packet processing latency. This includes design of abstraction layers starting from the instruction set all the way down to the microarchitecture of both packet parsing and packet processing subsystems within high-performance switches and routers.

DAVIDE ROSSI (Member, IEEE) received the Ph.D. degree from the University of Bologna, Italy, in 2012. He has been a Postdoctoral Researcher with the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi," University of Bologna, since 2015, where he currently holds an assistant professor position. His research interests focus on energy efficient digital architectures in the domain of heterogeneous and reconfigurable multi and many-core systems on a chip. This includes architectures, design implementation strategies, and runtime support to address performance, energy efficiency, and reliability issues of both high end embedded platforms, and ultra-low-power computing platforms targeting the Internet of Things (IoT) domain. In these fields he has published more than 100 articles in international peer-reviewed conferences and journals. He was a recipient of the Donald O. Pederson Best Paper Award, in 2018.

WALTER CERRONI (Senior Member, IEEE) is currently an Assistant Professor of communication networks with the University of Bologna, Italy. His recent research interests include software-defined networking, network function virtualization, service function chaining in cloud computing platforms, intent-based northbound interfaces for multidomain/multitechnology virtualized infrastructure management, and modeling and design of inter-data and intra-data center networks. He has coauthored more than 120 articles published in well renowned international journals, magazines, and conference proceedings. He serves/served as a Series Editor for *IEEE Communications Magazine*, an Associate Editor for the IEEE COMMUNICATIONS LETTERS, and a Technical Program Co-Chair of the IEEE-sponsored international workshops and conferences.

HAYATE OKUHARA (Member, IEEE) received the Ph.D. degree from Keio University, Kanagawa, Japan, in 2018. He is currently a Postdoctoral Researcher with the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi," University of Bologna, Bologna, Italy. His research interest includes low-power VLSI system design.

CARLA RAFFAELLI (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in electronic and computer engineering from the University of Bologna, Italy, in 1985 and 1990, respectively. She is currently an Associate Professor with the University of Bologna. She is the author or coauthor of more than 150 conference papers and journal articles mainly in the field of optical networking and network performance evaluation. Her research interests include performance analysis of telecommunication networks, switch architectures, optical networks, and 5G networks. She actively participated in many National and International research projects, such as the EU funded ACTS-KEOPS, the IST-DAVID and the e-photon/One and BONE networks of excellence. She has served as a Technical Program Committee Member in several Top International Conferences, such as ICC and ONDM and the Technical Program Committee Co-Chair in ONDM 2011. Since October 2013, she has been a member of the editorial board of the journal *Photonic Network Communications* (Springer). She is the Director of the International Telecommunications Engineering master's degree at the University of Bologna. She regularly acts as a Reviewer of top international conferences and journals.

JARI NURMI (Senior Member, IEEE) has been working as a Professor with the Electrical Engineering Unit, Tampere University, TAU (formerly Tampere University of Technology, TUT), Finland, since 1999. He is currently working on embedded computing systems, system-on-chip, approximate computing, wireless localization, positioning receiver prototyping, and software-defined radio and -networks. He holds various research, education, and management positions at TUT, since 1987. He was the Vice President of the SME VLSI Solution Oy, from 1995 to 1998. He has supervised 25 Ph.D. and over 140 M.Sc. theses. He has edited five Springer books and has published over 350 international conference papers and journal articles and book chapters. He is a member of the Technical Committee on VLSI Systems and Applications at the IEEE CASS. He is also an Associate Editor/Handling Editor of three international journals.