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

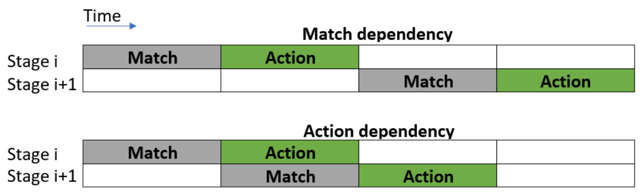


FIGURE 1. **Delaying Match and Action as a result of dependencies.**

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

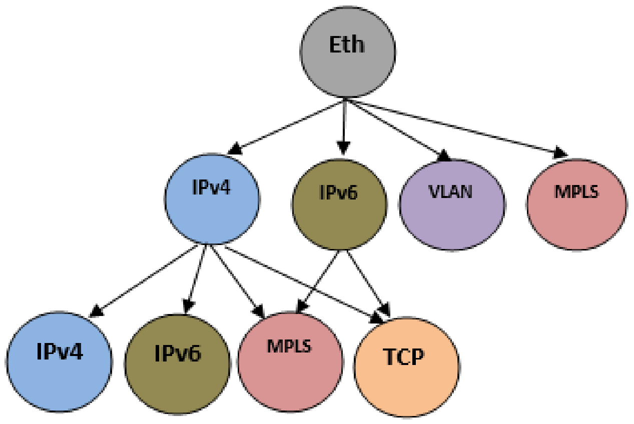


FIGURE 3. **Parse graph with three levels.**

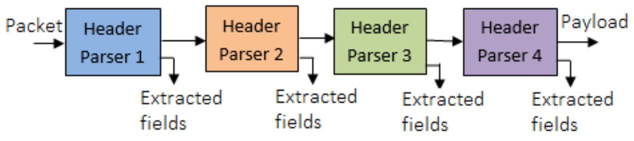


FIGURE 4. **A packet parser with four header parsers.**

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

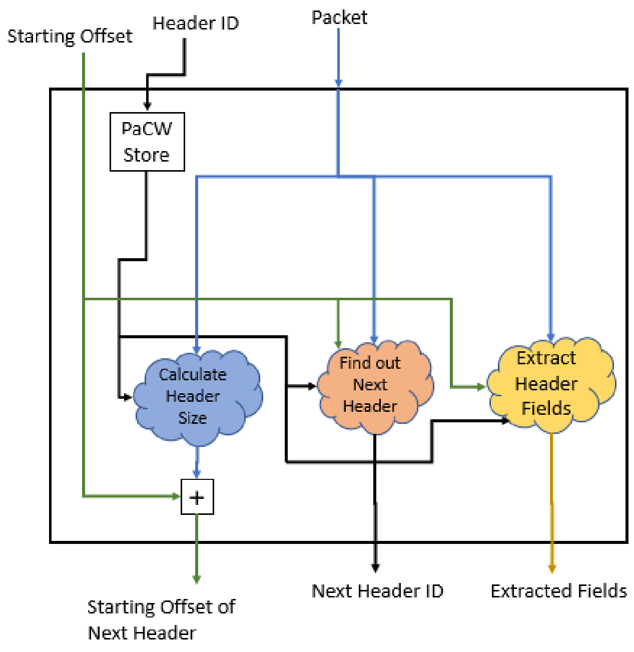


FIGURE 5. High-level view of the header parser.

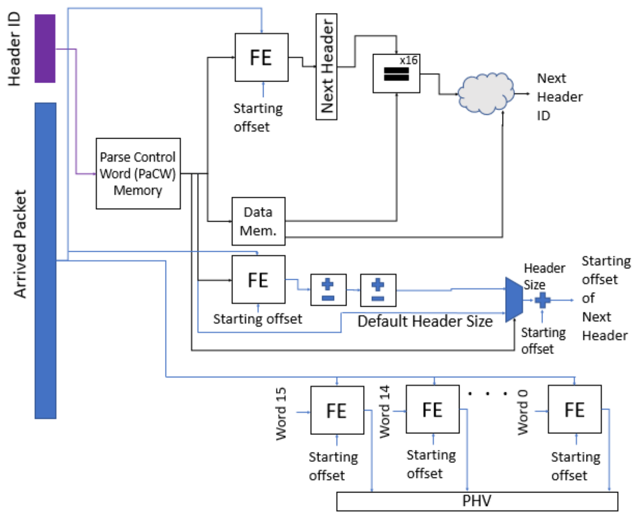


FIGURE 6. Internals of the header parser.

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

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

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.

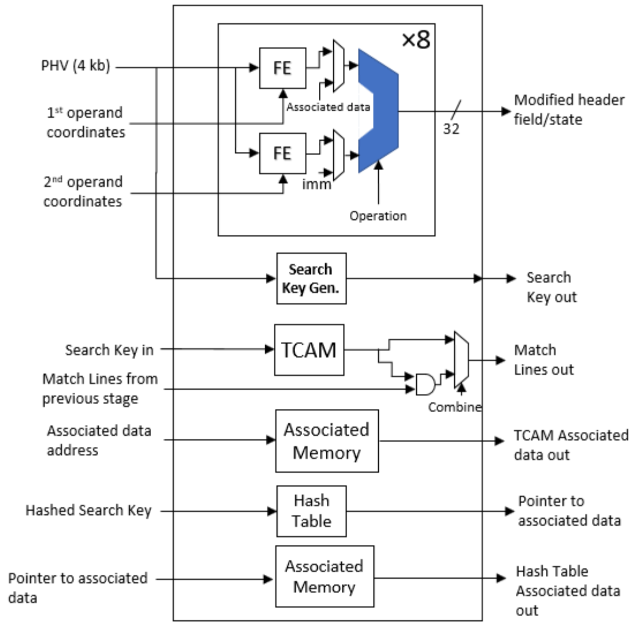


FIGURE 7. Internals of a packet processing stage.

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,

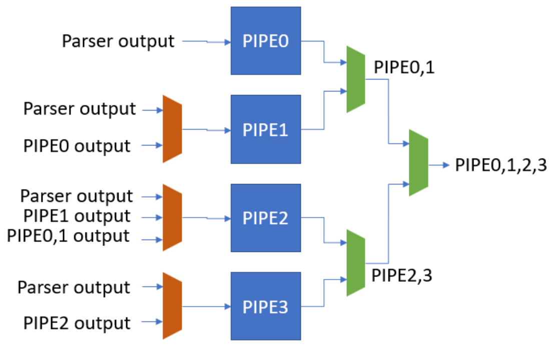


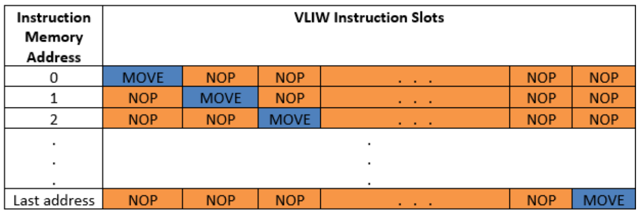
FIGURE 8. Pipeline configuration components.

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.

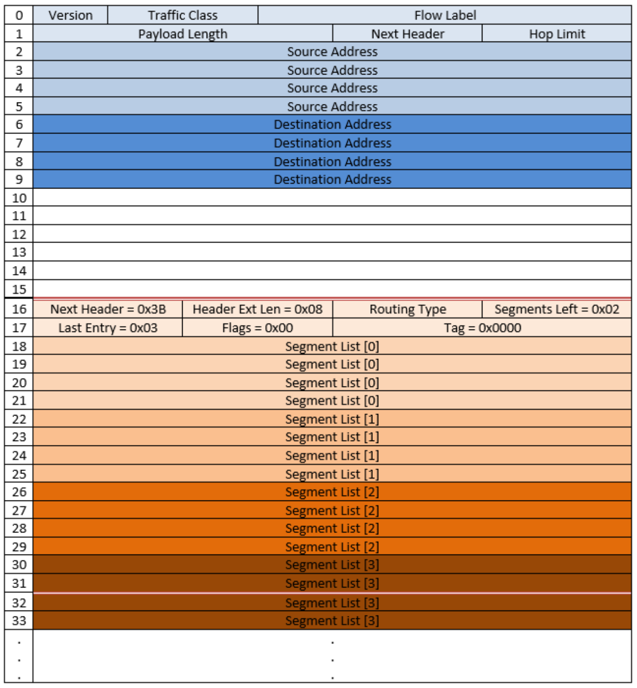


FIGURE 11. Outline of PHV after the parsing is complete.

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.

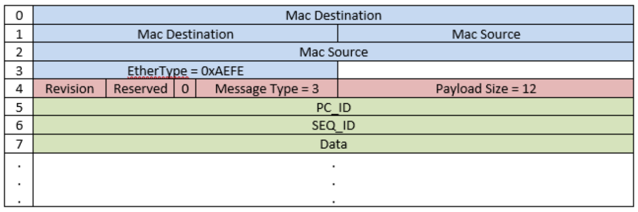


FIGURE 12. Outline of PHV after the parsing is complete.

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

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

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.

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

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

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.

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.

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.

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

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.