ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

A custom processor for protocol-independent packet parsing

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Zolfaghari H., Rossi D., Nurmi J. (2020). A custom processor for protocol-independent packet parsing. MICROPROCESSORS AND MICROSYSTEMS, 72, 0-11 [10.1016/j.micpro.2019.102910].

Availability:
This version is available at: https://hdl.handle.net/11585/739313 since: 2020-02-29

Published:
DOI: http://doi.org/10.1016/j.micpro.2019.102910

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

This is the final peer-reviewed accepted manuscript of:

Hesam Zolfaghari, Davide Rossi, Jari Nurmi, A custom processor for protocol-independent packet parsing, Microprocessors and Microsystems, Volume 72, 2020, 102910, ISSN 0141-9331.

The final published version is available online at:

https://doi.org/10.1016/j.micpro.2019.102910

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

A Custom Processor for Protocol-Independent Packet Parsing

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*- 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.**

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 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], [6] and [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], [10], [11], [12] and [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 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], [11], [12] and [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.

II. 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.

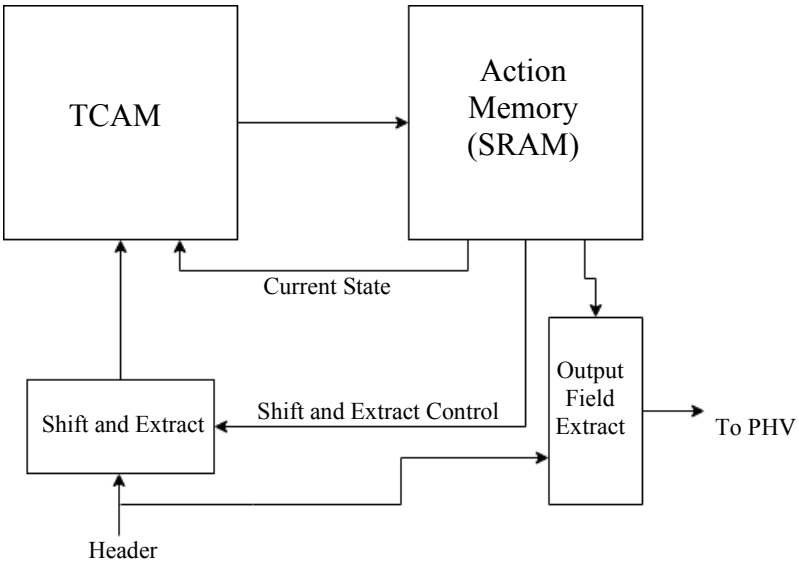

Fig. 1. Programmable parser in [6]

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

III. 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.

IV. 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.

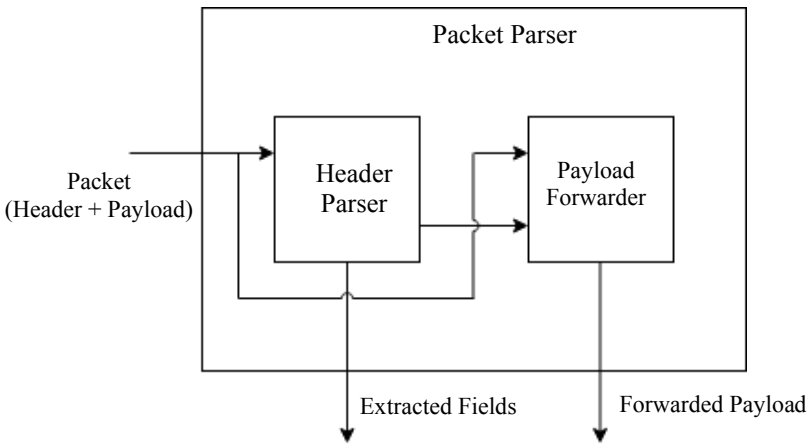

Fig. 2. The new packet parser

A. 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] and [17].

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

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.

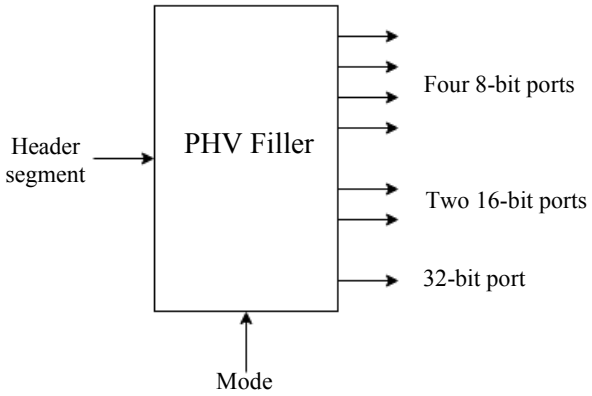

Fig. 3. PHV Filler

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

a) 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.

b) 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.

c) 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.

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

d) 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 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, 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.

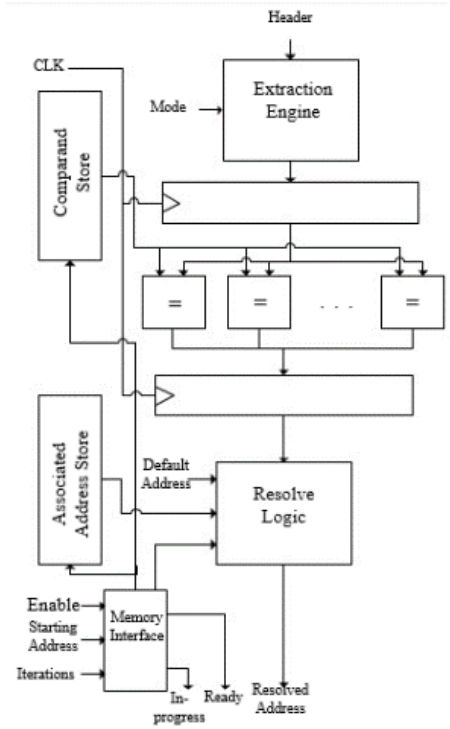

Fig. 5. The internal components of the NHRU

e) 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.

f) 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 I.

TABLE I. 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

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

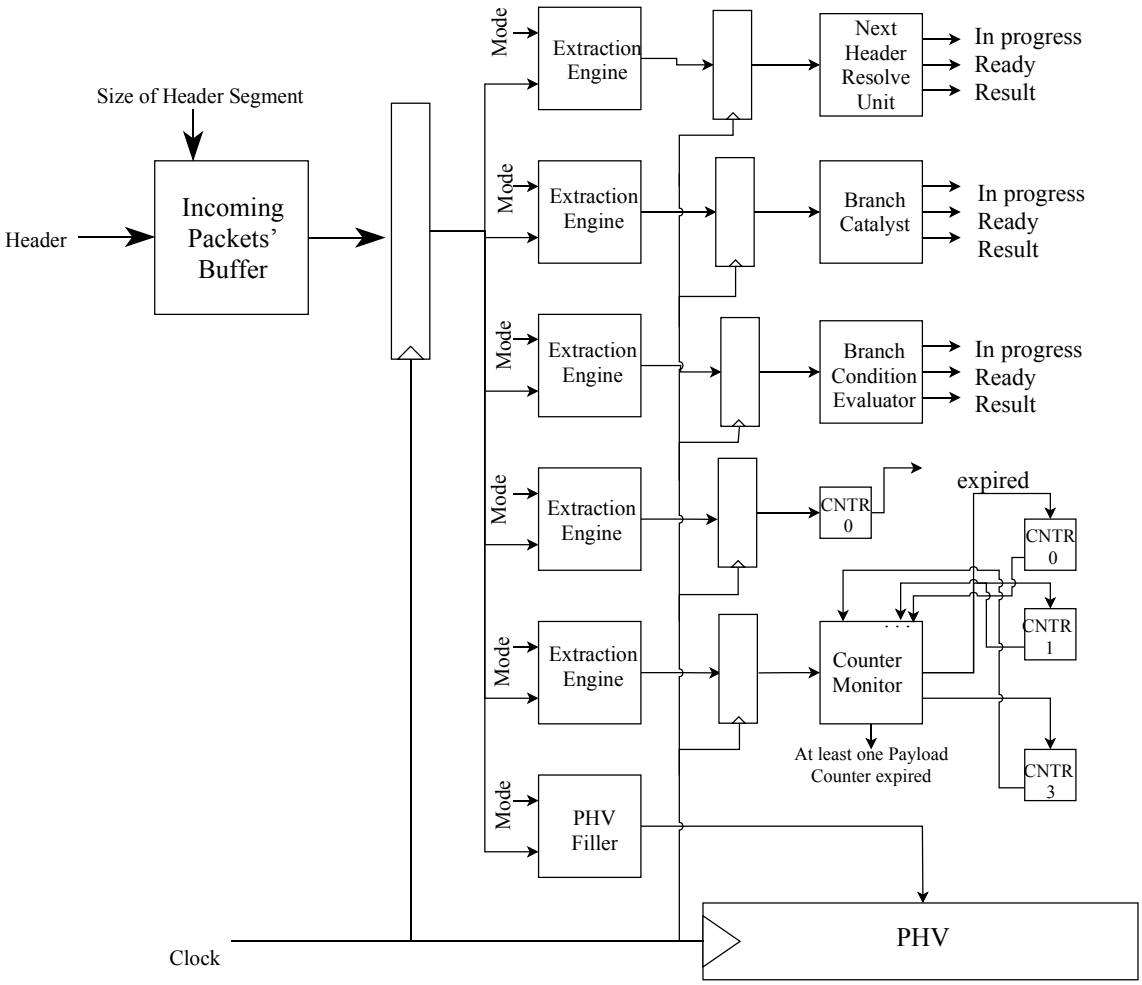

Fig. 6. Internals of the explicitly parallel header parser

B. *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.

V. 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:

A. *Fetch Header (FH)*

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

B. *Extraction (EX)*

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

C. *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.

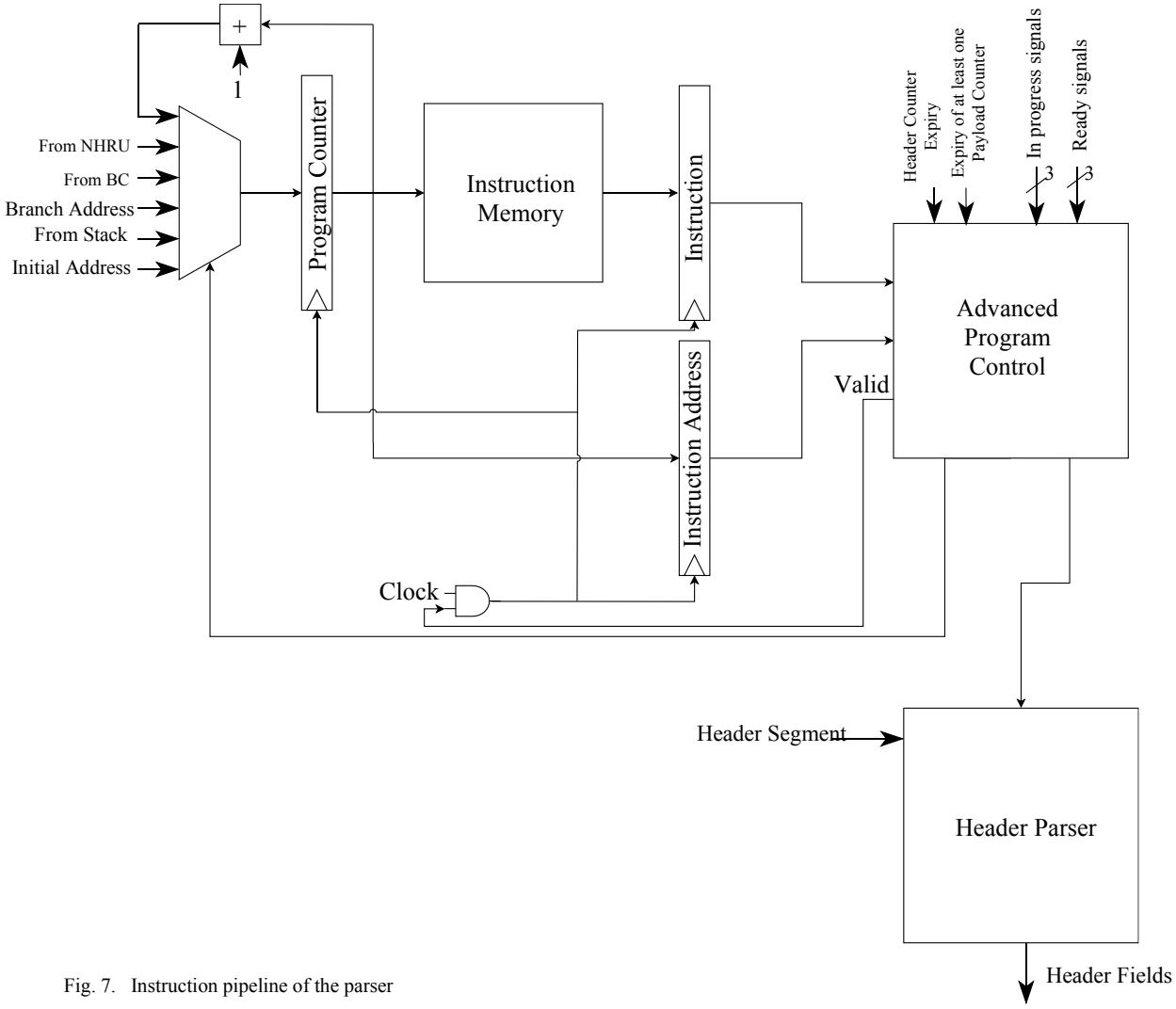

Fig. 7. Instruction pipeline of the parser

VI. Instruction Format

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

TABLE II. 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

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.

VII. Parsing Example

A. *Parsing Ethernet*

In this section we illustrate how a parsing subroutine written in P4 could be mapped to and executed on our parser. Fig. 8 and Fig. 9 illustrate the Ethernet header and parser definition in P4 respectively.

```
header Ethernet_h
{
   bit<48>  dstAddr;
   bit<48>  srcAddr;
   bit<16>  etherType;
}
```

Fig. 8. Header defintion for Ethernet

```
parser parse_ethernet
{
   extract(ethernet);
   return select(latest.etherType)
   {
      0x8100 : parse_vlan;
      0x8847 : parse_mpls;
      0x0800 : parse_ipv4;
      0x86dd : parse_ipv6;
      default: ingress;
   }
}
```

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

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.

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

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.

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

```
header IPv4_h
{
  bit<4>        version;
  bit<4>        ihl;
  bit<8>        diffserv;
  bit<16>       totalLen;
  bit<16>       identification;
  bit<3>        flags;
  bit<13>       fragOffset;
  bit<8>        ttl;
  bit<8>        protocol;
  bit<16>       hdrChecksum;
  bit<32>       srcAddr;
  bit<32>       dstAddr;
  varbit<320>   options;
}
```

Fig. 11. Definition of IPv4 header in P4

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.

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

VIII. 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.

A. *Parsing individual headers*

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

TABLE III. 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

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

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

B. *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 IV. Workload number 4 results in the smallest throughput value which is slightly over 27 Gbps.

TABLE IV. 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

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.

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						
$I_{Next\ Header}$										FI	FH	EX	WB

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

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		
$I_{Next\ Header}$												FI	FH	EX	WB

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

C. Implementation details

The architecture is implemented in VHDL. We have synthesized it on 28 nm UTBB FD-SOI technology in worst-case operating conditions (1.0V, 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.

Tables V and VI 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 IV in which parsing of the headers in the last 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.

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

TABLE V. 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	30864	63002
Packet Header Vector	15976	32631
Instruction Memory	93052	190057
Total Area	145427	358838

TABLE VI. 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

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 163408 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.

IX. 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

[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.

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.

ACKNOWLEDGEMENT

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

[2] Shahbaz, M., & Feamster, N. (2015, June). The case for an intermediate representation for programmable data planes. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (p. 3). ACM.

[3] Song, H. (2013, August). 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 (pp. 127-132). ACM.

[4] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S. T., Agrawal, A., ... & McKeown, N. (2016, August). Programmable packet scheduling at line rate. In Proceedings of the 2016 ACM SIGCOMM Conference (pp. 44-57). ACM.

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

[6] Bosshart, P., Gibb, G., Kim, H. S., Varghese, G., McKeown, N., Izzard, M., ... & Horowitz, M. (2013). Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. ACM SIGCOMM Computer Communication Review, 43(4), 99-110.

[7] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., ... & Licking, S. (2016, August). Packet transactions: High-level programming for line-rate switches. In Proceedings of the 2016 ACM SIGCOMM Conference (pp. 15-28). ACM.

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

[9] Attig, M., & Brebner, G. (2011, October). 400 Gb/s programmable packet parsing on a single FPGA. In 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems (pp. 12-23). IEEE.

[10] Santiago da Silva, J., Boyer, F. R., & Langlois, J. M. (2018, February). 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 (pp. 147-152). ACM.

[11] Benáček, P., Puš, V., Kubátová, H., & Čejka, T. (2018). P4-To-VHDL: Automatic generation of high-speed input and output network blocks. Microprocessors and Microsystems, 56, 22-33.

[12] Benácek, P., Puš, V., & Kubátová, H. (2017). Automatic Generation of 100 Gbps Packet Parsers from P4.

[13] Cabal, J., Benáček, P., Kekely, L., Kekely, M., Puš, V., & Kořenek, J. (2018, February). 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*(pp. 249-258). ACM.

[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] Fisher, J. A., Faraboschi, P., & Young, C. (2009). VLIW processors: Once blue sky, now commonplace. IEEE Solid-State Circuits Magazine, 1(2).

[16] 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.

[17] 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.