Low-latency Packet Parsing in Software Defined Networks

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

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

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

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

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

I. INTRODUCTION

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

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

II. AN EXPLICITLY PARALLEL ARCHITECTURE

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

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

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

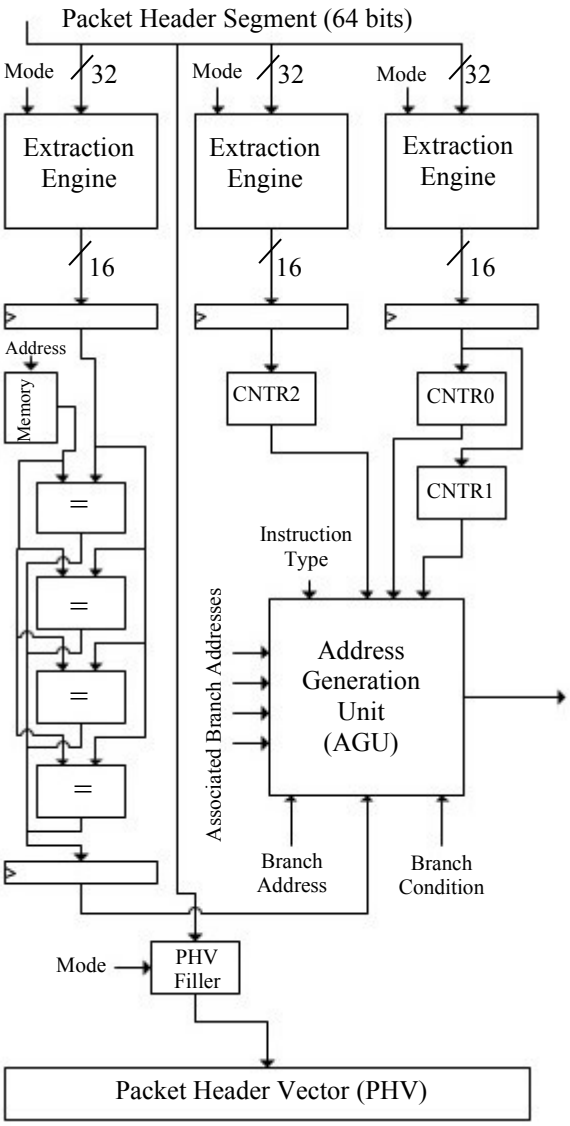

Fig. 1. High-level view of the Packet Parser in [8]

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

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

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

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

III. Program Flow in Packet Parsing

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

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

IV. A New Programmable Packet Parser

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

A. Packet Header Vector Filler

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

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

B. Resource Monitors

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

C. Counters

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

D. Extraction Engines

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

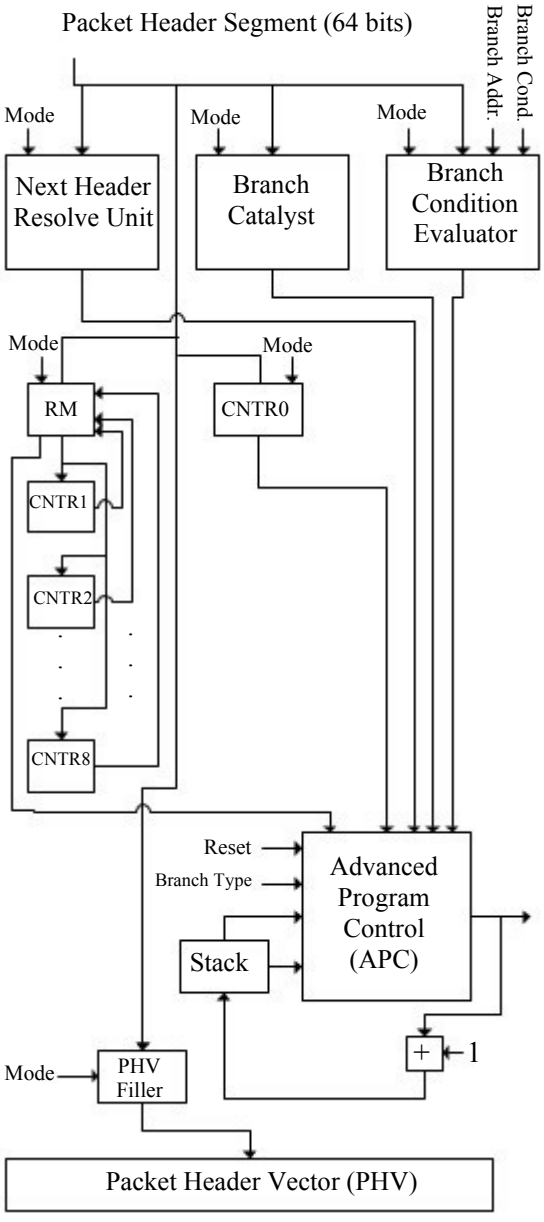

Fig. 2. High-level view of the new packet parser

E. Next Header Resolve Unit

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

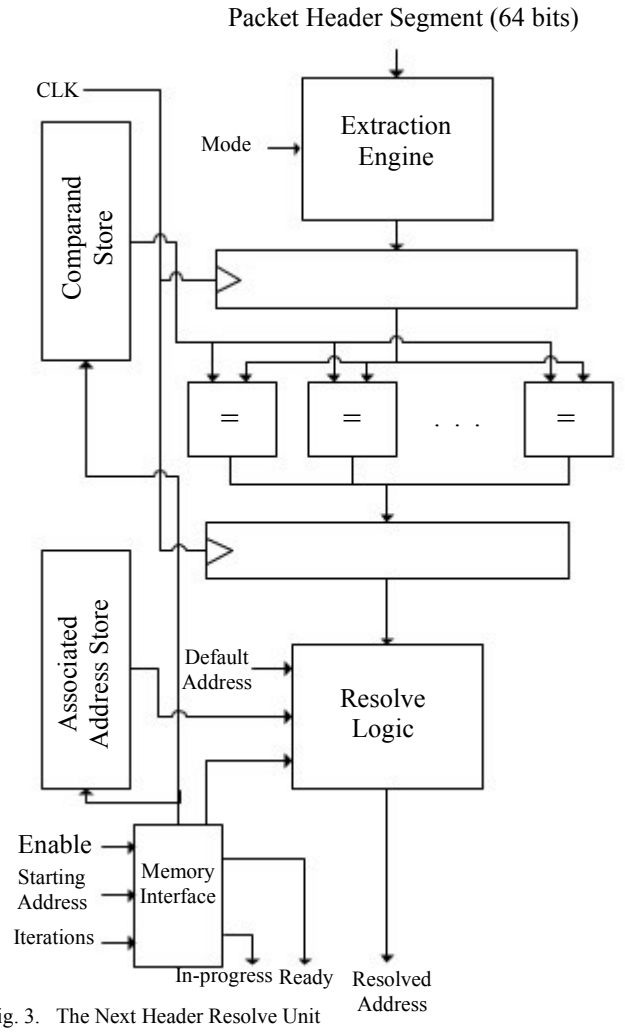

Fig. 3. The Next Header Resolve Unit

F. Branch Catalyst

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

G. Branch Condition Evaluator

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

V. The Advanced Program Control Unit

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

A. Branch Catalyst

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

B. Next Header

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

C. Next Header Function Call

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

D. Payload Forwarding

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

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

E. End of Trailer

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

F. Conditional Branch

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

VI. EVALUATION

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

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

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

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

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

TABLE I. POWER DISSIPATION COMPARISON OF ARCHITECTURE VARIANTS

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

TABLE II. AREA COMPARISON OF ARCHITECTURE VARIANTS

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

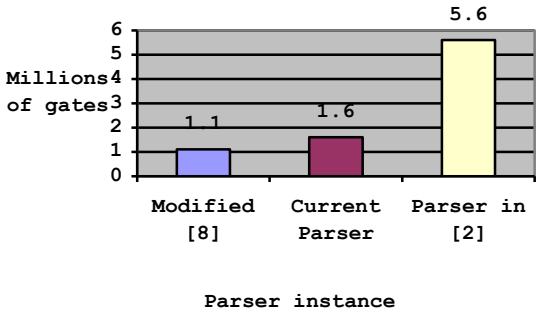

Fig. 4. Total Gate count Required for aggregate throughput of 640 Gbps

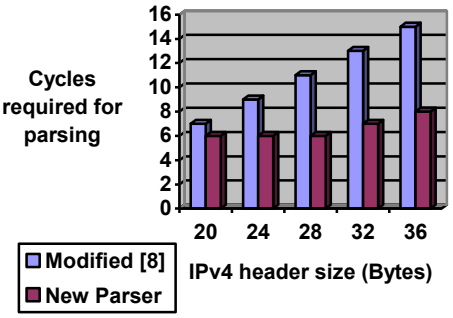

Fig. 5. Comparsion of cycle counts required for parsing IPv4 headers of different sizes

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

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

TABLE III. COMPARISON OF TIME REQUIRED FOR PARSING THE WORKLOADS

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

VII. CONCLUSION

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

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

REFERENCES

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

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM*, Hong Kong, 2013.

[3] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown and S. Licking, "Packet Transactions: High-Level Programming for Line-Rate Switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, Florianopolis, Brazil, 2016.

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

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

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

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

[8] H. Zolfaghari, D. Rossi and J. Nurmi, "An Explicitly Parallel Architecture for Packet Parsing in Software Defined Networks," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Milan, 2018.