An Explicitly Parallel Architecture for Packet Parsing in Software Defined Networks

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

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

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

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

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

I. INTRODUCTION

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

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

II. OUR TASK FORMULATION

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

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

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

III. Architectural Details

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

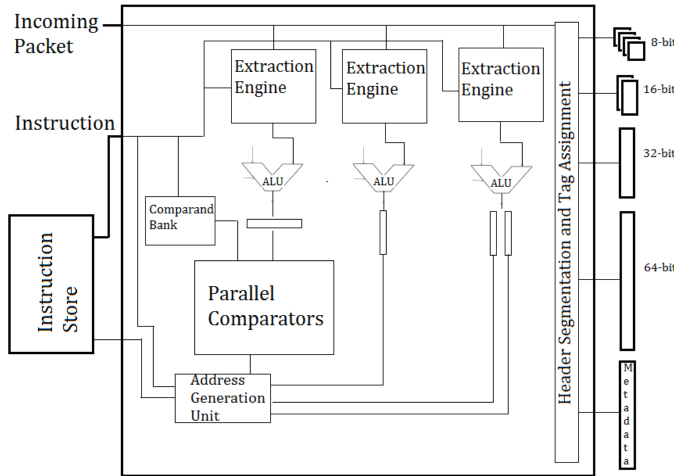

Fig. 1. Overall architecture

The main functional units are as follows:

A. Three Extraction Engines

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

B. Parallel Comparators

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

C. Header Segmentation

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

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

D. Address Generation Unit

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

E. Arithmetic and Logic Units

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

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

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

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

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

IV. EXPERIMENTAL RESULTS

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

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

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

TABLE I. AREA RESULTS FOR A SINGLE PARSER INSTANCE

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

TABLE II. POWER RESULTS FOR A SINGLE PARSER INSTANCE

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

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

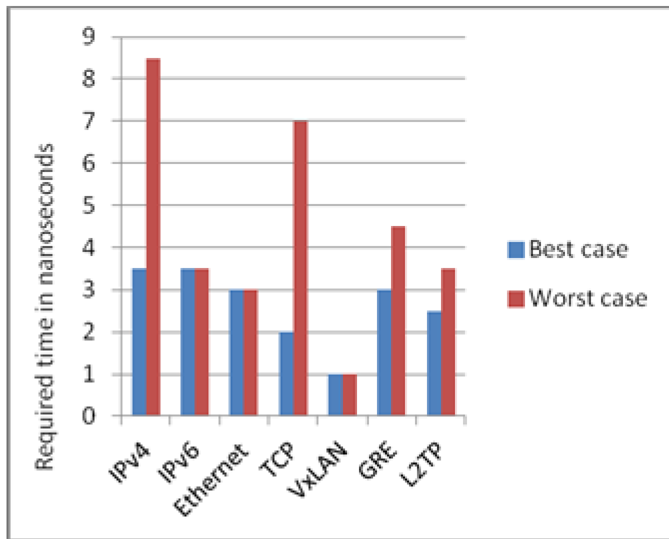

Fig. 2. Time required for parsing various headers

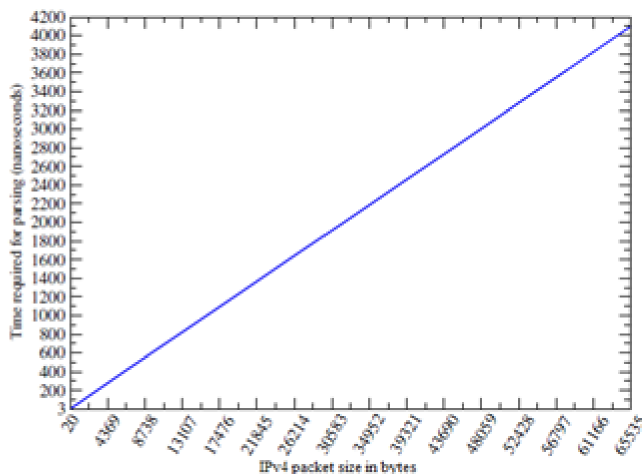

Fig. 3. Time required for parsing IPv4 packets of varying sizes

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

V. Conclusion and Future Work

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

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

References

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

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

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

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

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

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

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

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

[9] Jeferson Santiago da Silva, François-Raymond Boyer, and J.M. Pierre Langlois, "P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs," in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, 2018, pp. 147-152.

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

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

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

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