Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput

Jakub Cabal
CESNET a.l.e.
Prague, Czech Republic
cabal@cesnet.cz

Pavel Benáček
CESNET a.l.e.
Prague, Czech Republic
benacek@cesnet.cz

Lukáš Kekely
CESNET a.l.e.
Prague, Czech Republic
kekely@cesnet.cz

Michal Kekely
Netcope Technologies
Brno, Czech Republic
kekely@netcope.com

Viktor Puš
Netcope Technologies
Brno, Czech Republic
pus@netcope.com

Jan Kořenek
IT4Innovations Centre of Excellence,
FIT BUT
Brno, Czech Republic
korenek@fit.vutbr.cz

ABSTRACT

As throughput of computer networks is on a constant rise, there is a need for ever-faster packet parsing modules at all points of the networking infrastructure. Parsing is a crucial operation which has an influence on the final throughput of a network device. Moreover, this operation must precede any kind of further traffic processing like filtering/classification, deep packet inspection, and so on.

This paper presents a parser architecture which is capable to currently scale up to a terabit throughput in a single FPGA, while the overall processing speed is sustained even on the shortest frame lengths and for an arbitrary number of supported protocols. The architecture of our parser can be also automatically generated from a high-level description of a protocol stack in the P4 language which makes the rapid deployment of new protocols considerably easier. The results presented in the paper confirm that our automatically generated parsers are capable of reaching an effective throughput of over 1 Tbps (or more than 2 000 Mpps) on the Xilinx UltraScale+ FPGAs and around 800 Gbps (or more than 1 200 Mpps) on their previous generation Virtex-7 FPGAs.

CCS CONCEPTS

• **Hardware** → *Networking hardware*; *Hardware accelerators*; *Hardware description languages and compilation*;

KEYWORDS

packet parser; HLS; P4; Ethernet; high-speed networks; VHDL

ACM Reference Format:
Jakub Cabal, Pavel Benáček, Lukáš Kekely, Michal Kekely, Viktor Puš, and Jan Kořenek. 2018. Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput. In *FPGA '18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174250

FPGA '18, February 25–27, 2018, Monterey, CA, USA

ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174250

1 INTRODUCTION

The speeds of network links are growing very fast. This holds true not only in the core of carrier networks but also in a wide variety of application-specific cases. Parsing is a crucial operation that must precede any kind of further traffic processing like filtering/classification, deep packet inspection, and even basic routing/switching. With network lanes currently operating at hundreds of gigabits per second, and terabit requirements on the horizon, an effective design of packet parser capable of lossless wire-speed processing poses a major challenge.

Apart from high performance, current networks also require parsers to support an extensive set of various protocols. Furthermore, these requirements are changing rapidly as network protocols are constantly evolving. This trend is spearheaded by the success of Software Defined Networking and the end of network "ossification" that comes with it. The changing nature of networking ecosystem thus clearly favors flexible (programmable) technologies like FPGAs over fixed ASIC implementations. On top of that, utilization of some form of High-Level Synthesis in parser description is a must.

Fastest of current FPGA-based packet parsers are able to achieve a *raw (theoretical) throughput* of little over 400 Gbps by utilizing very wide (up to 2 048 b or 256 B) data buses. No approaches capable of 1 Tbps parsing has been presented so far. Nearly all of the existing parsers support some form of a higher-level protocol stack description followed by an automatic or semi-automatic HDL code generation. But the key issue, that still remains largely unaddressed in all of the previous works in this area, is the *effective throughput* achievable by the parsers at different traffic patterns. In the worst case, when bursts of the shortest possible packets are processed, the effective throughput of existing approaches drops to only a small fraction of advertised raw throughput as they can process only a single packet per clock cycle. This degradation of effective throughput is getting more severe as the raw throughput (bus width) increases and becomes unbearably large for wire-speed processing even at 100 Gbps or 400 Gbps.

The shortest allowed length of L2 Ethernet frame is 64 B (512 b). At the same time, a typical implementation of FPGA packet parser

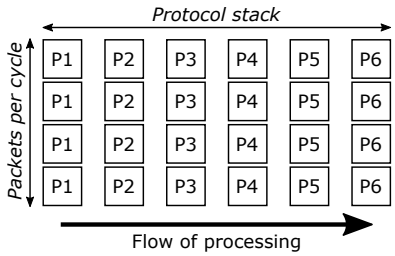

Figure 1: Proposed concept of main parser structure.

at 100 Gbps uses data bus that is 512 b wide and running at a clock frequency of at least 195 MHz. The shortest 64 B Ethernet frame fits nicely into a single data bus word. But what about 65 B, 66 B and similar frame lengths? When a few bytes spill into the second word, the rest of that word remains unused, yielding effective throughput of only around a half of the parser raw capacity. When data bus is wider than the length of the shortest frames, the effective throughput is even further reduced by insufficient packet rate. For example, a 400 Gbps parser operating with 2 048 b wide bus achieve effective throughput of only a fourth of its raw capacity when processing the shortest 64 B frames. Therefore, a processing of multiple packets or their fractions per clock cycle must be possible to achieve higher effective throughput of parsing.

This paper presents a novel packet parser design that not only advances achievable raw throughput of parsing in FPGAs above 1 Tbps mark but more importantly enables to retain sufficiently high effective throughput to guarantee the wire-speed processing of even the shortest packets at these speeds. In order to achieve such high degree of performance, the main feature of our proposed parser is to maximally exploit the means for massive processing parallelism that modern FPGAs offer.

The illustration of our key design concept is shown in Fig. 1. The complexity of fast packet parsing is divided into multiple simple parsers or analyzers (individual squares), where each of them can process at most one packet per clock cycle and parse only one protocol header from a supported protocol stack (P1, P2 . . .). The simple parsers are organized in a matrix-shaped structure that enables utilization of parallelism in two orthogonal dimensions. Firstly, each incoming packet traverses the structure from left to right going through a pipeline of simple parsers of different protocols, thus protocol stack complexity is divided into multiple steps (columns). Secondly, processing of multiple packets can start in every clock cycle as multiple simple parsers are working in parallel at each pipeline stage. In other words, the proposed matrix structure of our parser enables to use FPGA resources for scaling of both protocol stack complexity (width of the matrix) and achieved packet rate (height of the matrix).

The contribution of our work is a novel packet parser design for FPGAs that posses the following 3 key features:

(1) unprecedented raw throughput of over 1 Tbps;
(2) processing of multiple packets per clock cycle that leads to sufficient effective throughput for wire-speed processing;
(3) modular structure supporting automatic generation of HDL implementation from a high-level P4 description.

The rest of this paper is organized in the following manner. In section 2, we introduce several published approaches to parser design and compares them with our work. Section 3 describes our parser design concept and architecture in depth. In section 4, we provide a quick overview of the P4 language and description of the supported automated generation of the proposed parser from a P4 description. Section 5 contains results of achieved effective throughput, chip area and latency of our parser architecture in different configurations. Finally, the last section concludes the paper and discusses the obtained results.

2 RELATED WORK

Many fundamentally different approaches to FPGA packet parser design are present in published works with various benefits and disadvantages. However, many of them fail to scale well in a high-speed deployment that we aim for. Furthermore, none of them provide any robust and practical approach towards retainment of sufficient effective throughput ratio in the worst case.

Kobierský et al. [5] implement packet parsing using finite state machines generated from an XML description of protocol headers. This work shows achievable throughput of up to 20 Gbps. However, effective further scalability of the approach is poor. The size of generated FSMs (number of their states) rises rapidly with protocol stack complexity and data bus width, creating a performance bottleneck. Also, the crossbar used for extraction of fields values do not scale well with rising data bus width.

Unique Kangaroo parsing architecture of Kozanitis et al. [6] stores packets in memory and employes on-chip associative memory to perform a speculative lookahead into stored data. Based on lookahead results, the packet format is sequentially constructed, even moving through several headers in a single step (clock cycle). Authors showed achievable throughput of this architecture to be up to 40 Gbps line rate. However, this approach has the architectural limitation of storing the packets in the memory and fetching their data afterward. When scaling for higher throughput, the memory soon becomes a bottleneck. Also, higher complexity of parsed protocol stack can lead to considerably harder lookahead operation.

Wang et al. [10] introduce a rapid prototyping framework for mapping of a domain-specific HLS source code (P4 language) to the HDL description in BlueSpec language. The paper doesn't provide a detailed description of the generated parser's architecture but it provides results for its latency and throughput. The results show a raw throughput of up to 10 Gbps. But the effective throughput of the generated parser degrades with the growing complexity of supported protocol stack. With our parser design, we aim at throughput independent on the number of supported protocols to achieve wire-speed processing even in complex use cases. Also, the scalability of the parser from [10] on higher throughputs is questionable due to the generation of BlueSpec instead of HDL.

Another example of a domain-specific HLS generation of parsers is provided by Attig and Brebner in [1]. They propose their own Packet Parsing (PP) language to describe the structure of protocol headers and the methods which define parsing rules. From PP source code a pipelined implementation of a parser is generated. Thanks to the extreme pipelining and data bus width, generated parsers are able to achieve raw throughputs of up to 400 Gbps.

However, the results indicate a heavy price for such throughput in terms of the chip area and the latency. Also, authors themselves acknowledge the fact, that the effective throughput of their approach on the shortest frames is up to four times lower than presented raw throughput results (so only up to 100 Gbps). They propose to allocate the whole parser multiple times in parallel to workaround this issue, which would lead to even heavier resource price.

Similar pipeline architecture of parser is also proposed and elaborated by Kekely et al. in [3, 4]. Their approach achieves similar raw throughput of around 400 Gbps, but with considerably lower resource usage and latency compared to [1]. This is achieved thanks to hand-optimized implementation and not that heavy usage of pipelining. Authors also acknowledge the issue of degraded effective throughput of their parser on the shortest frames. They even propose a partial mitigation of the issue by unaligned frame starts and data bus words shared between parts of two consecutive frames. However, their approach is only sufficient for wire-speed processing of up to 100 Gbps (parsers with at most 512 b wide data bus). Further scaling of throughput (bus width) leads to similar throughput degradation as without this approach. Furthermore, in the original papers, the generation of the parser from the high-level description is not present at all. This is only supplemented later by Benáček et al. in [2], where they describe and generate these parsers using the P4 language.

The architecture of parser proposed in this paper is to some extent similar to pipelined approaches presented in the previous two paragraphs as protocol stack processing is divided into multiple parallel header parsers (columns in Fig. 1). However, the major contribution of this paper is the extension of parallelism utilization in the parser design towards another orthogonal dimension (packets per cycle or rows in Fig. 1). This extension in conceptual design is the key contribution that separates our parser from all previous works and enables us to achieve considerably higher effective throughput leading to lossless wire-speed processing.

3 PACKET PARSER DESIGN

The following section provides a bottom-up description of parser's architecture concept. We start with the proposition of our data convey bus protocol, more specifically the architecture of data word that would enable transfer of multiple packets per clock cycle. After that, the description of single protocol parser (analyzer) follows. Finally, the top-level parser architecture following the conceptual design from the Introduction is elaborated.

3.1 Data Convey Interface

The interface is used for transfer of packet's data into and through the whole parser. As already mentioned in the Introduction, it must be possible to transfer multiple packets (or their parts) in one bus word in order to retain the high ratio of effective to raw throughput even at higher speeds (bus widths).

To enable transfer of multiple packets per clock cycle, we define the data bus word structure illustrated in Fig. 2. The figure also shows an example of possible packet placements under the proposed bus structure. One should notice that without the support of multiple packets per clock cycle, each of the depicted data frames should occupy separate word on the bus (5 words would be required), but word sharing enables more dense packing (only 3 words are needed). Now for the proposed structure (bottom of the Fig.), each data word of the bus consists of several *regions*. These restrain the maximal number of packets per cycle as at most one packet can start and one end (can be a different one) in each region. Each region is further separated into *blocks* of data *elements* (items) to constraint possible positioning of packet starts. All packets must start aligned with the start of a block, but can end in any element (e.g., frames A and B both end in the middle of a block).

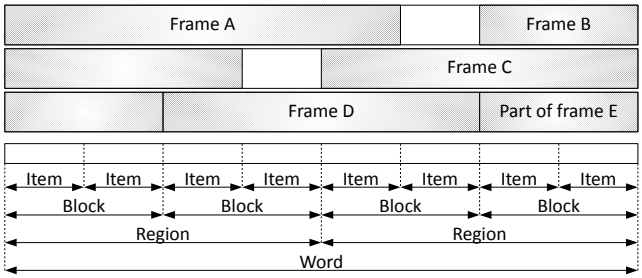

Figure 2: An example of possible packet placements under the proposed data bus structure (for $n = r = b = 2$).

General description of the proposed data word structure enables for definitions of multiple buses with different parameters. We formally describe them by the following four attributes:

- *Number of regions (n)* directly corresponds to the maximal number of packets transferred in each word.
- *Region size (r)* defines the number of blocks in each region, thus affects the size of overhead for very short packets. Usage of values that are powers of 2 is recommended here.
- *Block size (b)* states the number of elements in each start alignment block, thus controls the alignment overhead for frames. To simplify the processing complexity, usage of values that are powers of 2 is recommended.
- *Element width (e)* defines the size of the smallest distinguishable piece of data in bits. In networking, we always work with bytes (octets of bits), but in general, other values can be also utilized.

Using these main attributes, we derive bus word width in bits as: $dw = n \times r \times b \times e$. Now, we can also specify that illustration in Fig. 2 shows a bus with parameters $n = r = b = 2$.

When considering the processing of Ethernet frames, the aforementioned parameters of the bus should be configured to appropriate values. As already mentioned Ethernet operates with bytes (octets) as the smallest data elements – therefore we let $e = 8$. Lower layers of Ethernet (PCS/PMA layers) usually operate with frame starts aligned at 8 B blocks (lanes) – so $b = 8$ is convenient. Size of a region should correspond with the size of the smallest allowed packets (64 B) as smaller regions would needlessly allow transfer of more packets per word that is possible and on the other hand, larger regions would reduce effective bus saturation for the shortest packets – therefore we let $r = 64/b = 8$. Using these attribute values ($r = b = e = 8$) and considering the shortest packets to be 64 B long, the bus structure impose no more than $b - 1 = 7$ bytes of alignment

T [Gbps]	f [MHz]	n [-]	dw [b]
100	200	1	512
200	400	1	512
200	200	2	1024
400	400	2	1024
400	200	4	2048
800	400	4	2048
800	200	8	4096
1600	400	8	4096

Table 1: The throughput of selected combinations of bus attributes and frequency.

overhead per packet. Furthermore, as lower layers of Ethernet operate with larger overhead per packet (20 B of preamble and IFG), our bus enables us to achieve effective throughput sufficient for wire-speed processing of Ethernet packets even in the worst case.

In the previous paragraph, we left the value of attribute n unset, because we want to use it to control the number of transferred packets per clock cycle and also the total width of the bus. Thanks to different values of n and appropriately selected frequency, we can easily scale the supported throughput of the parser. Tab. 1 shows some considered configurations of the bus, where T stands for the achievable throughput, f stands for the FPGA frequency, n is the number of regions and dw is the total width of the data bus.

3.2 Simple Protocol Analyzers

The simple analyzers are the basic building blocks of our parser architecture. Fig. 3 shows how they are internally arranged. Each simple analyzer is able to start a processing of at most one packet per clock cycle and extract data from a single specific protocol header. From these, it computes control information for the next analyzer in the processing pipeline. The input information necessary to parse a single protocol header consists of: (1) packet data from the corresponding and the previous region of the data bus, (2) offset of the current word counted from the last start of packet, (3) type of the expected protocol header and (4) offset of the current header start. One should note that for $n > 2$, every simple analyzer does not have to operate with the full data bus width, only with two of its regions. The output information is similarly simple and includes: (1) type of the next expected protocol, (2) offset of the next header start and (3) extracted data of the current protocol header.

The Common logic block remains the same in all protocol analyzers. It is used to extract correct bytes from a data bus that corresponds to a protocol header based on given offsets. Only Protocol logic is specifically designed for each of the parsed protocols, it computes offset and type of the next header in a protocol-specific manner based on extracted data. As headers are not always aligned to region boundaries, the analyzer has to look into two regions, not just one. The data extraction is started when the end of a protocol header is detected in the current region of the data word. If the current region is occupied by two ends of the same protocol header, the second header is parsed by the protocol analyzer in the next region. This solution allows parsing of each frame on the bus without any stalling of the incoming data stream.

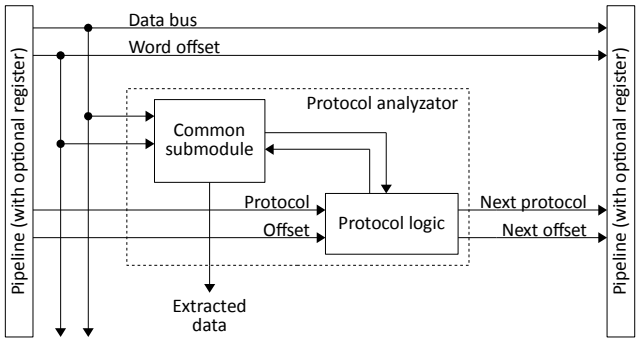

Figure 3: Internal structure of a single protocol analyzer.

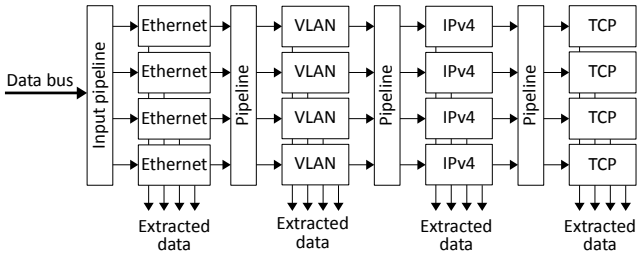

Figure 4: An illustration of complete parser arrangement.

3.3 Parser Top-Level Design

Our parser uses pipelining to achieve high working frequency and thus high throughput. However, pipeline steps are optional and we can control the trade-off between the frequency and used logic. For example, if many pipelines are enabled, the frequency (throughput) rises at the cost of longer latency and increased resources usage. Therefore, by careful selection of pipeline positions in the parser, we can find optimal configurations for any given use case.

Fig. 4 shows the example of parser pipeline with $n = 4$ and support of Ethernet, VLAN, IPv4 and TCP parsing. The shown pipeline arrangement corresponds to the conceptual matrix-like schema from Fig. 1. Each pipeline stage (column) contains one kind of protocol analyzers. The number of analyzers (in a pipeline stage) is equal to the number of regions n. Each protocol analyzer contains an inner bypass to solve the situations when the protocol is not found in processed data (i.e., the protocol analyzer is skipped if the currently processed packet does not contain the protocol which is being analyzed by the protocol analyzer). Thanks to this feature, the protocol analyzers can be arranged in a simple pipeline with a constant latency. This property also makes adding of new protocols into the pipeline very easy and it does not require any changes to the current protocol analyzer's architecture.

The shown arrangement contains two types of pipeline modules - *input pipeline* and *internal pipelines*. The input (first) pipeline stage also generates the initial control data for the first protocol analyzer stage (expecting Ethernet header at offset 0). There is also a word counter inside that counts the number of transferred words from the last packet start (i.e., word offset value). Both pipeline types

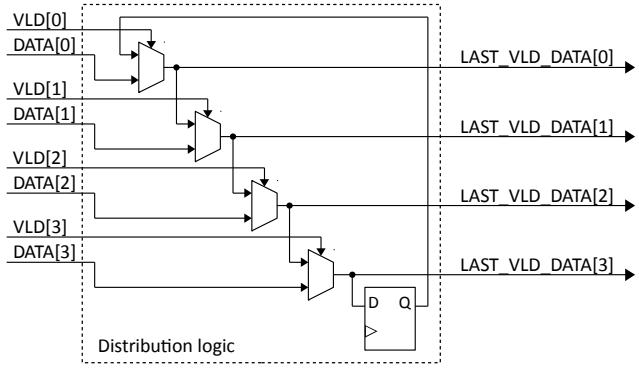

Figure 5: Internal structure of control data distribution login in pipelines.

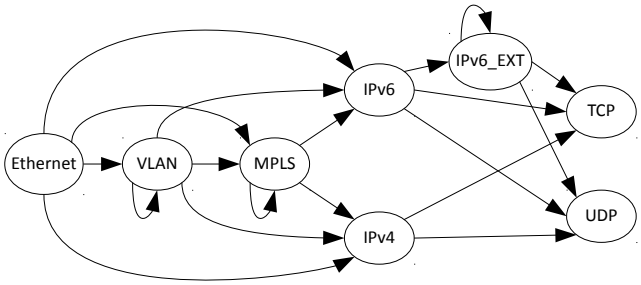

Figure 6: An example of protocol stack graph.

contain optional registers to achieve better timing and a logic for the distribution of control data across corresponding regions. The main task of the distribution block is to deliver control data to the correct protocol analyzer (region) in the next stage.

Fig. 5 shows the distribution logic for a control data. New valid control data are distributed to every following region until the start of a new packet is detected. Notice that each pipeline has registers to store data from the last region in the word because these data can be necessary for parsing in the next clock cycle (i.e., the remainder of header data is present in the following data word).

An example of the possible protocol stack that can be parsed by our parser is shown in Fig. 6. The order of protocol analyzer stages in the created pipeline directly depends on the selected protocol stack. The planning of parser stages order that accommodates any required stack is discussed in the next section of this paper, as we intend to use the $P4_{14}$[8] language for an automatic generation of packet parsers according to our proposed arrangement. This allows us to describe support for different protocol stacks very easily.

4 P4 HIGH-LEVEL SYNTHESIS

P4: Programming Protocol-independent Packet Processors is a high-level, platform-agnostic language. It represents a recent contribution to the broader idea of SDN and the SDN ecosystem. Its main purpose is to provide a way to define packet processing functionality of network devices, paying attention to reconfigurability, protocol independence and target (platform) independence. The idea of programmable data plane in P4 language was introduced in [7]. There are two versions of P4 standard currently available: $P4_{14}$[8] and $P4_{16}$[9]. The $P4_{14}$ is earlier language specification which syntax was firstly introduced in 2014. The $P4_{14}$ standard is based on the abstract forwarding model which is capable to process the P4 program. The main advantage of this approach is easy portability to different platforms because the software tool is responsible for mapping of the P4 program to the target architecture. The disadvantage comes from the property of easy portability. That is, we cannot use the advanced functionality of a target because it is not used in the abstract forwarding model (the pattern matching for example). The $P4_{16}$ specification solves this problem directly in the language, as it allows us to describe the advanced functionality in the form of a library which is distributed with a network device. The library is then used by a user in a $P4_{16}$ program. The following text briefly introduces the $P4_{14}$ specification because we consider it to be more currently known by the P4 community.

Using relatively simple syntax, the $P4_{14}$ allows to define five basic aspects of packet processing:

- **Header Formats** describe recognized protocol headers.
- **Packet Parser** describes the (conceptual) state machine used to traverse packet headers from start to end, extracting field values as it goes.
- **Table Specification** defines how the extracted header fields are matched in possibly multiple lookup tables (e.g., exact match, prefix match, range search, and so on).
- **Action Specification** defines compound actions that may be executed for packets.
- **Control Program** puts all of the above together, defining the control flow mainly among the tables.

For our work, only the first two aspects of $P4_{14}$ are relevant. Header format description may look like this:

```
header_type ethernet {
   fields {
      dst_mac  : 48;
      src_mac  : 48;
      ethertype : 16;
   }
}
```

The description simply lists fields of the packet header and their width in bits. The example above shows the situation for a static header where the header length is the sum of lengths of all fields. This can't be done for protocols with a variable header length. The P4 solves this situation by the header length definition in form of an expression which uses fields from the protocol header declaration to compute the header length. Header format description with variable length may look like this:

```
header_type ipv6_ext {
    fields {
        next_hdr     : 8;
        total_len    : 8;
    }
    length : (total_len + 1) * 8;
    max_length : 1024;
}
```

Packet parser description constructs a parse graph using the header format description, for example:

```
header ethernet eth;
parser ethernet {
   extract(eth);
   switch(eth.ethertype) {
      case 0x8100: vlan;
      case 0x9100: vlan;
      case 0x800: ipv4;
   }
}
```

The example uses `switch` and `extract` statements. The `extract` instructs the parser to examine input packets and look for the data defined in the header. The parsed data is then used in the `switch` statement to determine the next state (protocol) to process.

4.1 Mapping from P4$_{14}$ to Parser Architecture

The transformation process from P4$_{14}$ to VHDL was introduced by Benáček et al. in [2]. The architecture of our parser is designed to be compatible with the parser generator from the mentioned paper: the parser consists of protocol analyzers and pipelines which form the processing chain of given protocol stack. Therefore, we can reuse the already presented algorithm for generation of our parser pipeline. In cooperation with authors of [2], we managed to integrate our parser into their generator, enabling it to create parsers with much higher effective throughput. The following text briefly introduces the transformation from P4$_{14}$ description to VHDL.

The main idea of the transformation is based on the topological ordering of Parse Graph Representation (PGR). The PGR is defined as an acyclic oriented graph (loop edges are allowed) from the P4's Packet Parser description. Each node of that graph represents one protocol header and each edge represents the next parsed protocol. A transition between nodes is based on a condition which is also inferred from the P4's Packet parser description. The PGR also contains loop edges which are used for the representation of more instances of the same protocol (e.g., two or more VLAN headers). Each non-finite node contains the edge to a special *Unknown* state. This state is not directly described in a P4 program but it is implicitly required by the transformation process because it represents the situation when none of the provided conditions match (in the currently processed protocol). The topological ordering of PGR nodes is based on a mark which is identified by the depth-first search (DFS) algorithm. The key for the ordering of nodes is to identify the latest possible usage of the protocol in a PGR. After that, we can connect all modules in any non-decreasing order, where: (1) each node will be translated to VHDL and (2) every two nodes will be divided by a pipeline module. The described planning algorithm fulfills the following requirements:

(1) The planned structure of modules forms a pipeline (i.e., it follows the processing flow of our parser in the protocol stack dimension).
(2) Protocol at the end of the transition has to be processed after the protocol at the beginning of the transition.
(3) Topologically ordered nodes (connected into the pipeline) are able to cover all paths through the PGR and we are able to parse any given combination of protocols because the PGR is acyclic and node skipping is allowed.

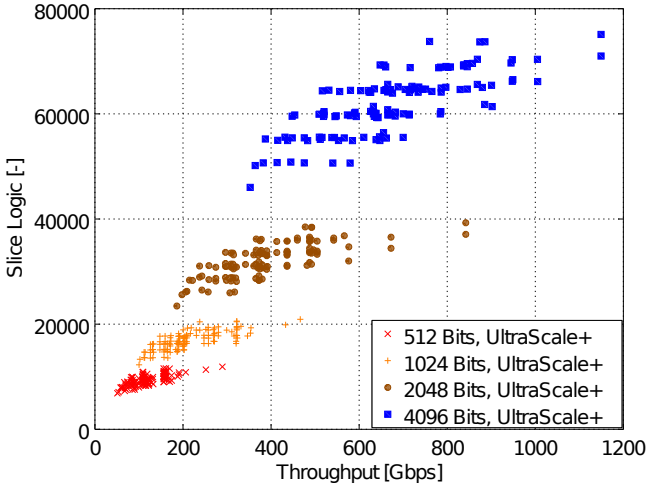

Figure 7: The relation between throughput and used resources of the simple L2 parsers on the UltraScale+ FPGA.

5 RESULTS

In this section, we provide a comparison of our generated parsers for two different protocol stacks:

- *full* – Ethernet, 4×VLAN, 4×MPLS, IPv4 or IPv6 (2×extension headers), TCP or UDP
- *simple L2* – Ethernet, IPv4 or IPv6 (2×extension headers), TCP or UDP

Thanks to the description in P4$_{14}$, we are able to easily generate parsers with the support of any protocols rather quickly. Of course, too big and complicated protocol stacks may require huge amounts of FPGA resources, but we will discuss it later in this section.

We described both mentioned protocol stacks in a P4$_{14}$ language. Then, translated the P4 source and synthesized it with different settings of data bus word width (512, 1024, 2048 and 4096 bits) and configuration of enabled pipeline registers. In all cases, we use data bus parameters that allow sufficient effective throughput for wire-speed processing of even the shortest packets ($r = b = e = 8$ with varying n as stated in subsection 3.1). The synthesis results of all the combinations form the state space of parsers with different throughput, working frequency, latency and resource usage. All values provided in the following text are after the synthesis for the Xilinx Virtex-7 XCVH870T FPGA or the Xilinx UltraScale+ XCVU7P FPGA using the Xilinx Vivado 2017.2 design tool. The achieved results were searched for sets of Pareto optimal parsers. From these, we can pick the best-fitting parser configurations for applications with different requirements. Notice that the FPGA resource usage is expressed as a sum of required LUTs and registers because it allows us to reflect the influence of enabled pipelines. Furthermore, selected Pareto optimal parsers were also integrated into our FPGA firmware and tested under real network conditions.

5.1 Simple L2 Protocol Stack

Figures 7 and Fig. 8 show the resource utilization and achieved throughput of generated simple L2 parsers, the first shows the UltraScale+ results and the second the Virtex7 results. Each point

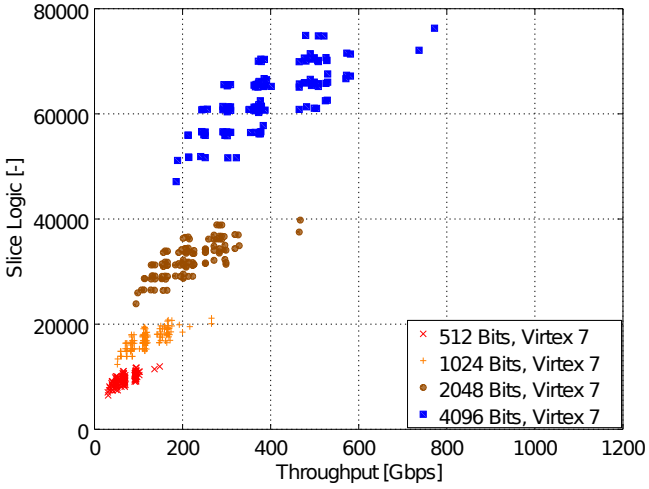

Figure 8: The relation between throughput and used resources of the simple L2 parsers on the Virtex-7 FPGA.

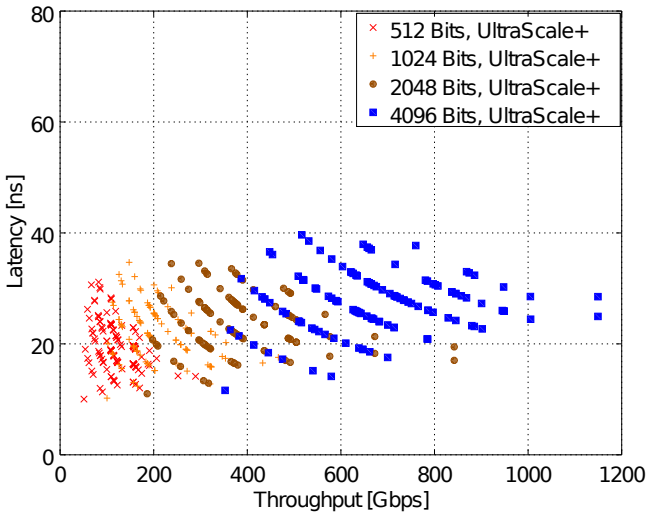

Figure 9: The relation between throughput and processing latency of the simple L2 parsers on the UltraScale+ FPGA.

in the graphs represents one parser with a different combination of parameters. The FPGA resource utilization linearly increases with the achieved throughput in both graphs. In the case of the UltraScale+ FPGA, we are able to reach the effective throughput of well over 1 Tbps. Achieved throughputs for the Virtex7 FPGA are notably worse while the used resources remain very similar. This is because the same parsers reach lower frequencies when implemented on the Virtex-7 compared to the UltraScale+ FPGA – our results show the frequency to be 1.5 to 2 times lower.

Fig. 9 and Fig. 10 show the latency and throughput of different settings of the simple L2 parser on the UltraScale+ and the Virtex-7 FPGA. Generally, the latency depends on the configured number of enabled registers in pipeline stages and the working frequency.

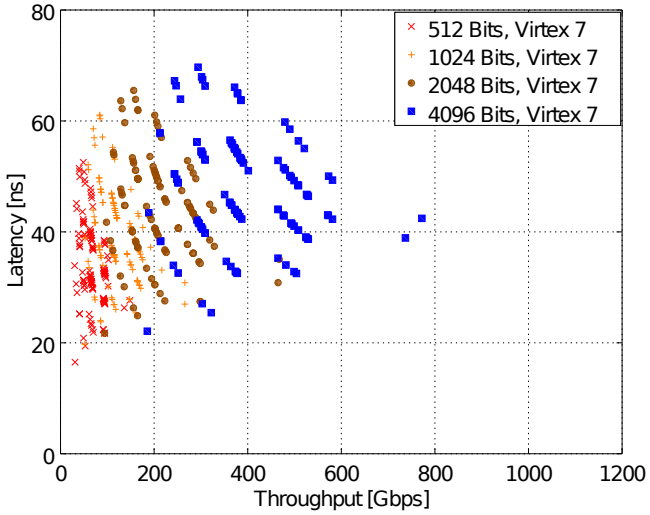

Figure 10: The relation between throughput and processing latency of the simple L2 parsers on the Virtex-7 FPGA.

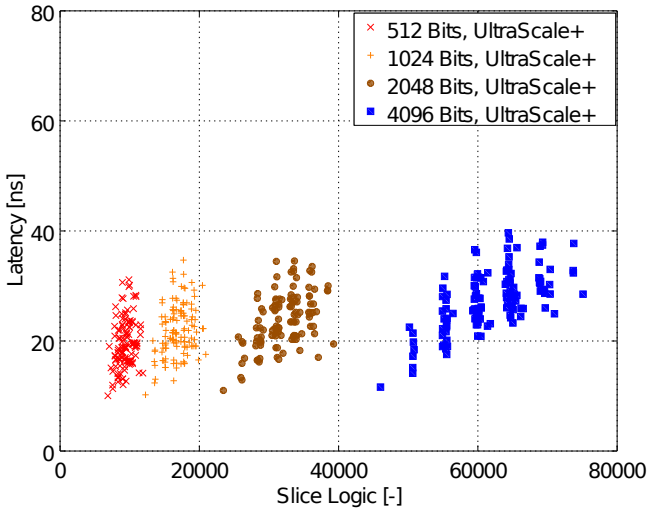

Figure 11: The relation between resources and latency of the simple L2 parsers on the UltraScale+ FPGA.

From the graphs, we can see that the latencies of our parsers are increasing as the achieved throughput is rising. This is because higher throughput is achieved by more extensive registering (i.e., more clock cycles between start and end of parsing). The latency is again generally from 1.5 to 2 times better on the UltraScale+ because of the higher frequencies achieved.

Finally, Fig. 11 and 12 show the third point of view on the results – the relation between latency and resource utilization of the simple L2 parser on the UltraScale+ and the Virtex 7 FPGA. We can see that although resources utilization rises considerably with data bus width, the latency pretty much stays in the same boundaries.

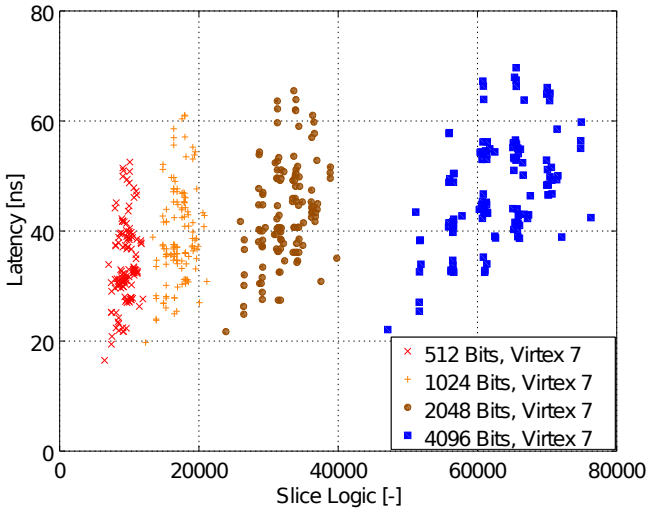

Figure 12: The relation between resources and latency of the simple L2 parsers on the Virtex-7 FPGA.

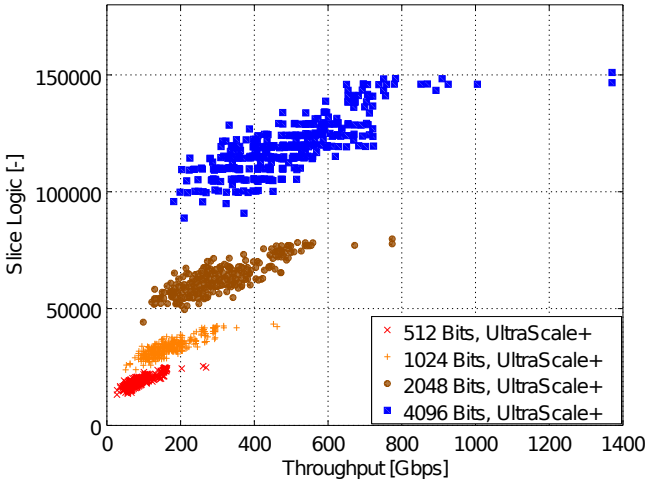

Figure 13: The relation between throughput and used resources of the full parsers on the UltraScale+ FPGA.

5.2 Full Protocol Stack

The state space of the full protocol stack parser parameter combinations is huge – 2^{15} configurations for every bus width. Therefore we synthesized only some hand-picked and randomly selected configurations that account for about 1 % of all the possibilities.

Fig. 13 and Fig. 14 show the resource utilization and effective throughput of the synthesized full parsers on the UltraScale+ FPGA and the Virtex 7 FPGA. The full parsers are much larger because they support more protocols than the simple L2 parsers. Therefore, the resource utilization reaches nearly 2 times higher values here. However, we still managed to achieve effective throughput of well over 1 Tbps on the UltraScale+ FPGA. The results again show the 1.5 to 2 times lower achieved throughputs for the Virtex-7.

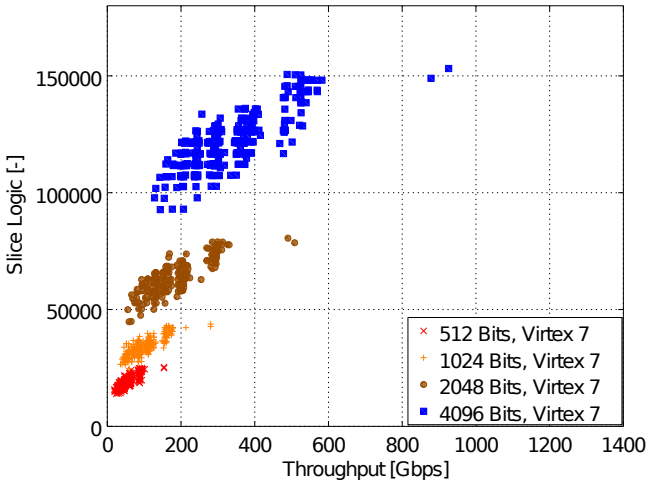

Figure 14: The relation between throughput and used resources of the full parsers on the Virtex-7 FPGA.

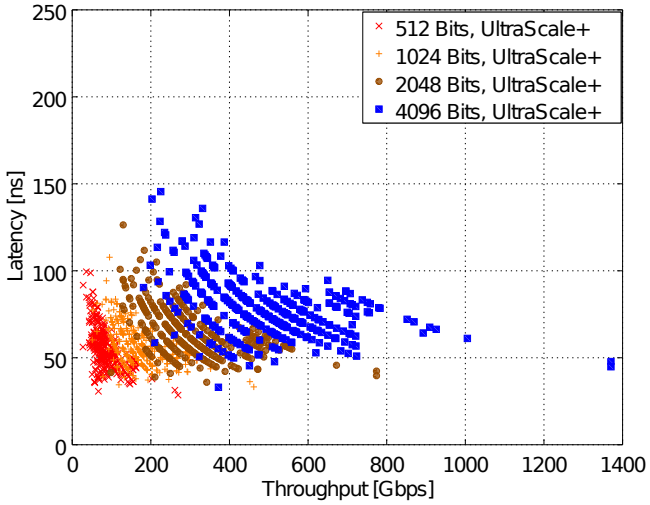

Figure 15: The relation between throughput and processing latency of the full parsers on the UltraScale+ FPGA.

Fig. 15 and Fig. 16 show the the latency and throughput relation of different configurations of full parsers implemented on the UltraScale+ and the Virtex-7 FPGA. The full parsers have considerably more pipeline stages, and therefore their latency is much higher – nearly 4 times in some cases. Apart from that, we can see the same trends as for the simple L2 parser. This is also true for the relationship between latency and resource utilization of the full parser on the UltraScale+ (Fig. 17) and the Virtex-7 FPGA (Fig. 18).

5.3 Summary

Fig. 19 shows sets of tested parsers with Pareto optimal results of resource utilization to achieved throughput. From the graph, we can clearly see the difference between the simple L2 parsers (full line) and the full parsers (dashed line) in resource utilization – the

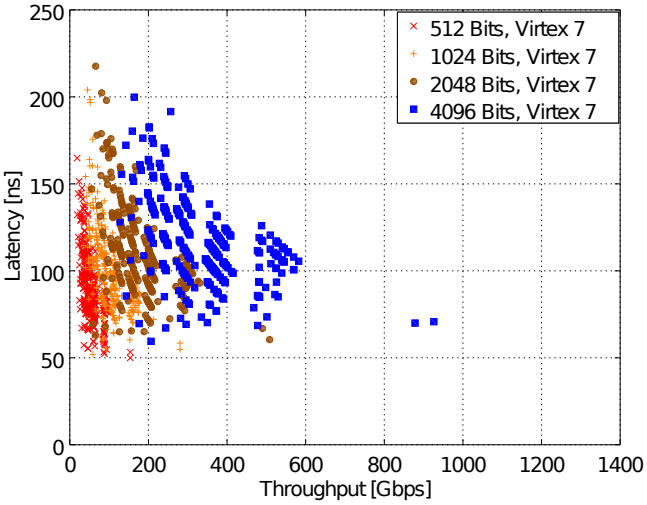

Figure 16: The relation between throughput and processing latency of the full parsers on the Virtex-7 FPGA.

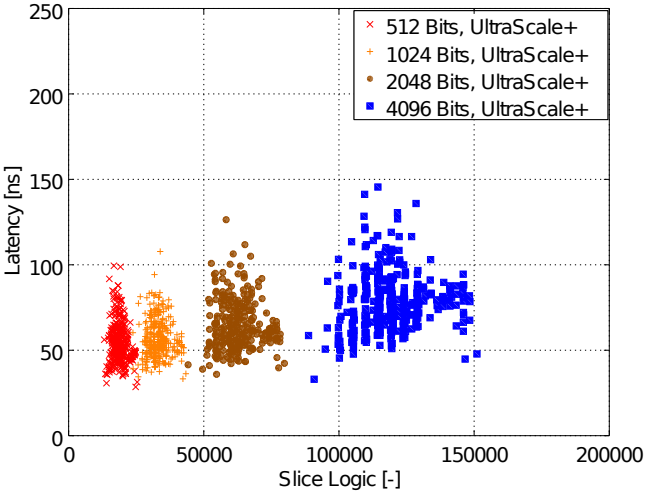

Figure 17: The relation between resources and latency of the full parsers on the UltraScale+ FPGA.

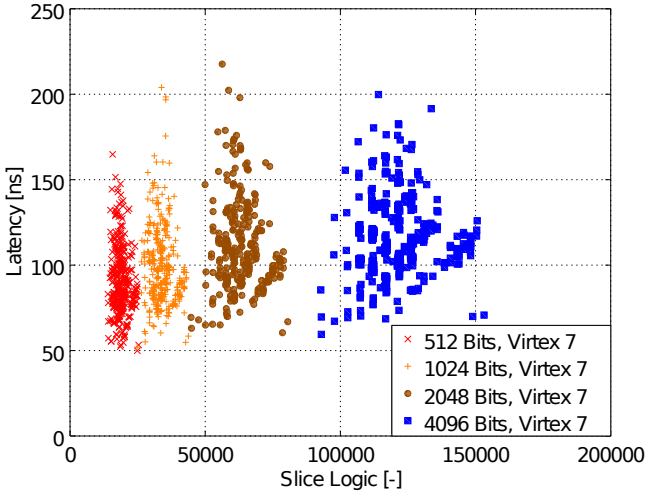

Figure 18: The relation between resources and latency of the full parsers on the Virtex-7 FPGA.

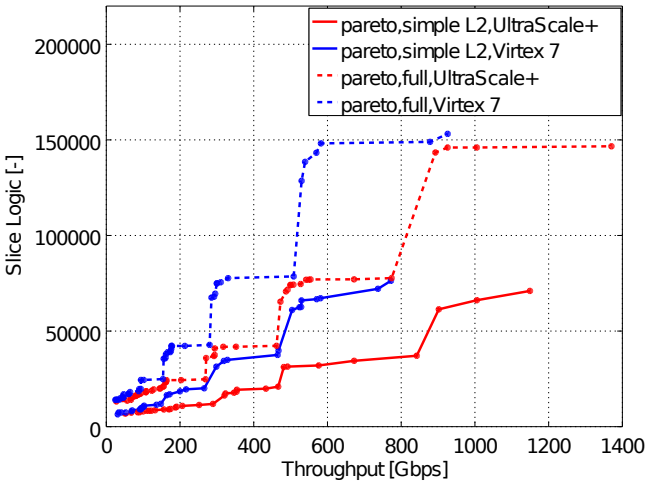

Figure 19: Pareto optimal sets of parsers in throughput × used resources space.

full parsers are up to 2 times larger. Also, the difference between target platforms (red for the UltraScale+ and blue for the Virtex-7 FPGA) is apparent. To compensate for the 1.5-2 times lower achieved frequencies on Virtex-7 and to achieve the same throughputs, nearly 2 times larger parsers must be used compared to UltraScale+. The stairs in the graphs are caused by the changing data bus width.

Fig. 20 shows Pareto optimal sets of parsers configurations in latency to throughput space. In all cases, the latency increases with the throughput. Again, we can see the positive effect of the higher frequencies on the UltraScale+, where the latency is only 10 to 30 ns for the simple L2 parsers and 30 to 50 ns for the full parsers.

To summarize, the measured results clearly show that the newer family of Xilinx FPGAs enable to achieve considerably better results of packet parsing implementation. Up to 2 times higher achieved frequencies allow selection of parsers with a narrower data bus to conserve a significant portion of FPGA resources for a given throughput requirement. Or, on the other hand, allow our parser architecture to achieve effective throughput of over 1 Tbps using wider buses. As the throughput holds even in the worst case scenario, we are able to achieve an incredible processing rate of over 1 500 Mpps when parsing the shortest Ethernet frames. Furthermore, we can see that the number of supported protocols does not negatively affect the throughput (nor packet rate) of our parsers thanks to the proposed pipelined architecture.

The effective throughput of our parser architecture is better than any of the so far presented approaches to parsing in FPGAs. The highest raw throughputs have been presented by Attig and Brebner in [1] (AB parser) and by Kekely et al. in [3, 4] extended with automatic parser generation by Benáček et al. in [2] (KB parser). Their

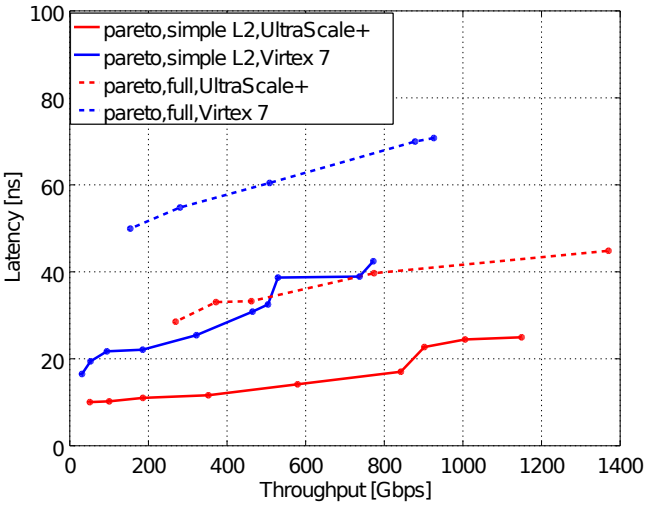

Figure 20: Pareto optimal sets of parsers in throughput × latency space.

Throughput	Parser	Utilization	Latency
over 100 Gbps	*our*	*2.05 %*	*69 ns*
	AB [1]	9.50 %	320 ns
	KB [2]	1.94 %	45 ns
over 400 Gbps	*our*	*6.38 %*	*67 ns*
	AB [1]	22.70 %	365 ns
	KB [2]	5.87 %	56 ns

Table 2: Comparison with other state of the art approaches.

approaches are discussed at the end of the section 2. Results of both of these parsers are presented for the Xilinx Virtex-7 XCVH870T FPGA – the same as we used for our Virtex-7 measurements. Also, both show results for similarly complex protocol stack as our full version of the parser (AllStack for AB and full stack for KB parser). Unfortunately, they use different metrics to describe the utilization of FPGA resources (percentage of the FPGA vs. LUT-FF pairs). Therefore, we transformed the utilization notation in the following paragraph to represent the percentage of used slices from the total available on the XCVH870T FPGA.

Under the specified common conditions, the AB parser is shown to achieve raw throughput of up to 578 Gbps and the KB parser of up to 478 Gbps. Our parser notably surpasses them both with the maximum of 926 Gbps. Furthermore, unlike the other approaches, the high throughput of our parser is retained even when processing the shortest packets. A comparison in terms of FPGA resources utilization and latency for specific throughput requirements (100 and 400 Gbps) is provided in the table 2. Compared to the AB parser, our approach (highlighted) requires several times fewer FPGA resources and operates with considerably smaller processing latency. The KB parser is only a little bit better in both metrics than our approach, but again, our approach guarantees sufficient throughput even in the worst case while the KB parser does not. To overcome this throughput limitation, the whole KB parser can be replicated (e.g., 4 × for 400 Gbps) and traffic distributed between the copies. But of course, this would considerably increase its utilization of FPGA logic and processing latency (well above our results).

6 CONCLUSIONS

This paper introduces and elaborates a novel parser architecture that enables processing of network traffic in current FPGAs at very high throughput. Unlike virtually all other approaches in this area, we do not focus solely on the raw achievable throughput but pay increased attention to sustainment of the performance even in the worst case – when parsing long bursts of very short packets. This way, the proposed parser architecture is able to guarantee wire-speed processing of network traffic at given link speed without any packet losses. Furthermore, HDL implementation of the parser can be automatically generated from a high-level description of a protocol stack in P4 language.

Our measurements show, that even for rather complex protocol stack the proposed parser concept enables to achieve high effective throughput at a cost of just a few percent of resources available in a single current FPGA. The achieved throughput is as high as 1.37 Tbps on the Xilinx UltraScale+ FPGAs and 926 Gbps on the Xilinx Virtex-7 FPGAs. Thanks to sustainment of the performance even for the shortest 64 B Ethernet frames, a huge packet rate is also reached – up to 2 038 Mpps on the UltraScale+ resp. 1 377 Mpps on the Virtex-7. The achieved throughput and, more notably, packet rate are considerably higher than in other published works. Moreover, they are well above the requirements for lossless wire-speed processing of 1 Tbps Ethernet traffic.

ACKNOWLEDGMENTS

This research has been partially supported by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 co-funded by the MEYS of the Czech Republic, IT4Innovations excellence in science project IT4I XS – LQ1602, and by the Technology Agency of the Czech Republic project TH02010214.

REFERENCES

[1] M. Attig and G. Brebner. 2011. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*. 12–23. https://doi.org/10.1109/ANCS.2011.12
[2] P. Benáček, V. Puš, and H. Kubátová. 2016. P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 148–155.
[3] L. Kekely, J. Kořenek, and V. Puš. 2012. Low-latency Modular Packet Header Parser for FPGA. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, New York, NY, USA, 77–78.
[4] L. Kekely, V. Puš, and J. Kořenek. 2014. Design Methodology of Configurable High Performance Packet Parser for FPGA. In *17th IEEE Symposium on Design and Diagnostics of Electronic Circuits&Systems*. IEEE Computer Society, 189–194.
[5] P. Kobierský, J. Kořenek, and L. Polčák. 2009. Packet header analysis and field extraction for multigigabit networks. In *Proceedings of the 2009 12th International Symposium on Design and Diagnostics of Electronic Circuits&Systems (DDECS)*. IEEE Computer Society, Washington, USA, 96–101.
[6] C. Kozanitis, J. Huber, S. Singh, and G. Varghese. 2010. Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System. In *Proceedings of the 29th Conference on Information Communications*.
[7] P. Bosshart et al. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review* 44, 3 (July 2014), 87–95.
[8] The P4 Language Consortium. 2017. The P4 Language Specification. (24 May 2017). https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf
[9] The P4 Language Consortium. 2017. $P4_{16}$ Language Specification. (22 May 2017). https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf
[10] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 122–135.