

P4-compatible High-level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs

Jeferson Santiago da Silva, François-Raymond Boyer and J.M. Pierre Langlois

Polytechnique Montréal, Canada

{jeferson.silva,francois-r.boyer,pierre.langlois}@polymtl.ca

ABSTRACT

Packet parsing is a key step in SDN-aware devices. Packet parsers in SDN networks need to be both reconfigurable and fast, to support the evolving network protocols and the increasing multi-gigabit data rates. The combination of packet processing languages with FPGAs seems to be the perfect match for these requirements.

In this work, we develop an open-source FPGA-based configurable architecture for arbitrary packet parsing to be used in SDN networks. We generate low latency and high-speed streaming packet parsers directly from a packet processing program. Our architecture is pipelined and entirely modeled using templated C++ classes. The pipeline layout is derived from a parser graph that corresponds to a P4 code after a series of graph transformation rounds. The RTL code is generated from the C++ description using Xilinx Vivado HLS and synthesized with Xilinx Vivado. Our architecture achieves 100 Gb/s data rate in a Xilinx Virtex-7 FPGA while reducing the latency by 45% and the LUT usage by 40% compared to the state-of-the-art.

CCS CONCEPTS

• **Hardware** → **Reconfigurable logic applications**; • **Networks** → **Programming interfaces**;

KEYWORDS

FPGA; packet parsers; HLS; programmable networks; P4

ACM Reference Format:

Jeferson Santiago da Silva, François-Raymond Boyer and J.M. Pierre Langlois. 2018. P4-compatible High-level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

The emergence of recent network applications have opened new doors to FPGA devices. Dataplane realization in Software-defined Networking (SDN) [10] is an example [14] of such applications. In SDN networks, the data and control planes are decoupled, and they can evolve independently of each other. When new protocols are deployed in a centralized intelligent controller, new forwarding rules are compiled to the data plane element without any change

to the underlying hardware. FPGAs, therefore, offer just the right degree of programmability expected by these networks, by offering fine grain programmability with sufficient and power-efficient performance.

A standard SDN forwarding element (FE) is normally implemented in a pipelined-fashion [3]. Incoming packets are parsed in order to extract header fields to be matched in the processing pipelines. These pipelines are organized as a sequence of match-action tables. In SDN FEs, a packet parser is expected to be programmable, and it can be reconfigured at run time whenever new protocols are deployed.

Recent packet processing programming languages, such as POF [9] and P4 [4], allow describing agnostic data plane forwarding behavior. Using such languages, a network programmer can specify a packet parser to indicate which header fields are to be extracted. He can as well define which tables are to be applied, and the correct order in which they will be applied.

The main focus of this work is to propose a high-level and configurable approach for packet parser generation from P4 programs. Our design follows a configurable pipelined architecture described in C++. The pipeline layout and the header layout templates are generated by a script after the P4 compilation.

The contributions of this paper are classified into two classes: architectural and microarchitectural. The summary of the architectural contributions of this work is listed as follows:

- an open-source framework for generation of programmable packet parsers¹ described in a packet processing language;
- a modular and configurable hardware architecture for streaming packet parsing in FPGAs; and
- a graph transformation algorithm to improve the parser pipeline efficiency.

The contributions related to the microarchitectural improvements are as follows:

- a data-bus aligned pipelined architecture for reducing the complexity in the header analysis; and
- a lookup table approach for fast parallel barrel-shifter implementation.

The rest of this paper is organized as follows. Section 2 presents a review of the literature, Section 3 draws the methodology adopted in this work, Section 4 shows the experimental results, and Section 5 draws the conclusions.

2 RELATED WORK

Packet processing languages. The SDN [10] paradigm has brought programmability to the network environment. OpenFlow [11] is the standard protocol to implement the SDN networks. However,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA'18, February 2018, Monterey, California USA

© 2018 Association for Computing Machinery.
ACM ISBN 123-4567-24-567/08/06...\$15.00
https://doi.org/10.475/123_4

¹ Available at https://github.com/engjefersonsantiago/Vivado_HLS

the OpenFlow realization [7] is protocol-dependent, which limits the genericity expected in SDN.

Song [9] presents the POF language. POF is a protocol-agnostic packet processing language, where the user can define the behavior of the network applications. A POF program is composed of a programmable parser and match-action tables.

P4 [4] is an emergent protocol-independent packet processing language. P4 provides a simple network dialect to describe the packet processing. The main components of a P4 program are the header declarations, packet parser state machine, match-action tables, actions, and the control program. Recently, P4 has gained adoption in both academia and industry, and this is why we have chosen P4 as the packet processing language in this work.

Packet parsers design. Gibb *et al.* present in [6] a methodology to design fixed and programmable high-speed packet parsers. However, this work did not show results for FPGA implementation.

Attig and Brebner [1] propose a 400 Gb/s programmable parser targeting a Xilinx Virtex-7 FPGA. Their methodology includes a domain specific language to describe packet parsers, a modular and pipelined hardware architecture, and a parser compiler. The deep pipeline of this architecture allows very high throughput at expense of longer latencies.

Benáček *et al.* [2] present an automatic high-speed P4-to-VHDL packet parser generator targeting FPGA devices. The packet parser hardware architecture is composed of a set of configurable parser engines [8] in a pipelined-fashion. The generated parsers achieve 100 Gb/s for a fairly complex set of headers, however the results showed roughly 100% overhead in terms of latency and resources consumption when compared to a hand-written VHDL implementation.

Recently, Xilinx has released the P4-SDNet translator [13], partially compatible with the P4₁₆ specification, that maps a P4 description to custom Xilinx FPGA logic. One particular limitation of P4-SDNet is the lack of support for variable-sized headers.

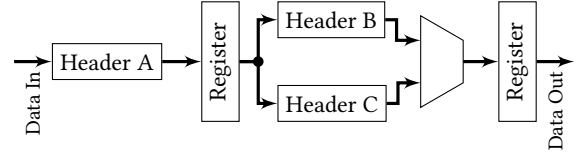
In this work, we deal with some of the pitfalls of previous works [1, 2], trading-off design effort, latency, performance, and resources usage. Our pipeline layout, leads to lower latencies compared to the literature [1, 2]. Moreover, the FPGA resource consumption in terms of lookup tables (LUTs) is reduced compared to [2], since instead of generating each parser code we parametrize generic hand-written templated C++ classes targeted to FPGA implementation.

3 DESIGN METHODOLOGY

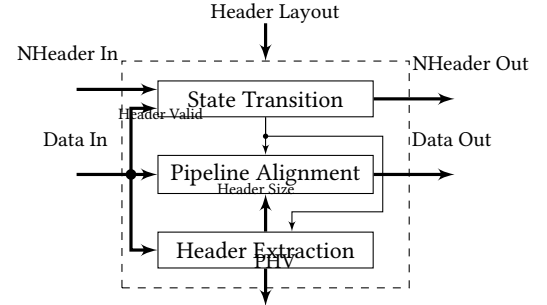
This section presents the methodology followed in this work. Section 3.1 draws the high-level architectural view. Section 3.2 deals with details on microarchitectural aspects. Section 3.3 presents our method to generate the parser pipeline.

3.1 High-Level Architecture

A packet parser can be seen at a high-level as a directed acyclic graph (DAG), where nodes represent protocols and edges are protocol transitions. A packet parser is implemented as an abstract state machine (ASM), performing state transition evaluations at each parser state. States belonging to the path connecting the first state to the last state in the ASM compose the set of supported protocols of an FE.



(a) High-level packet parser pipeline layout



(b) Internal header block architecture

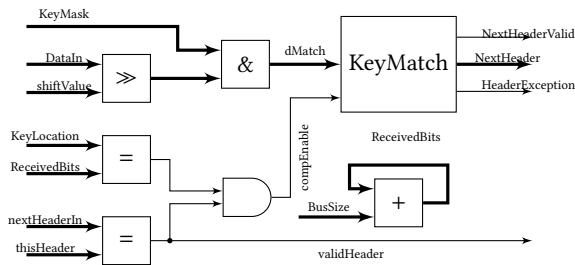
Figure 1: High-level architecture

Figure 1a depicts the high-level view of the packet parser realization proposed in this work. The proposed architecture is a streaming packet parser, requiring no packet storage. Header instances are organized in a pipelined-fashion. Headers that share the same previous states are processed in parallel. Throughout this work, we say that those headers belong to the same parser (graph) level. The depth of the parser pipeline is the length of the longest path in the parser graph. For sake of standardization, thick arrows in the figures throughout this work indicate buses, while thin arrows represent single signals.

The internal header block architecture is shown in Figure 1b. This block was carefully described using templated C++ classes to offer the right degree of configurability required by the most varied set of protocol headers this architecture is intended to support. This design choice was also taken to improve bit-accuracy by accordingly setting arbitrary integer variables, reducing FPGA resources usage.

In Figure 1b, the *Header Layout* is a configuration parameter. It is a set of data structures required to initialize the processing objects. It includes key match offsets and sizes for protocol matching, lookup tables to determine data shift values, expressions to determine the header size, last header indication, and so forth. *Data In* is a data structure that contains the incoming data to be processed in a header instance. It is composed of the data bus to be analyzed and some metadata. These metadata include data start and finish information for a given packet and packet identifier. The packet identifier is used to keep track of the packet throughout the processing pipeline and to identify which headers belong to the same packet. *NHeader In* is assigned by the previous header instance indicating which is the next header to be processed. *PHV* is a data structure containing the extracted fields. It includes the extracted data, number of bits extracted, a data valid information, and header and packet identifier. Signals labelled with *In* and *Out* are mirrored, which means that *In* signals undergo modifications before being forwarded to *Out*.

Internal sub-blocks execute in parallel with minimum data dependency. In fact, only the *Header Valid* information must propagate



among the blocks within the same clock cycle and it is generated from a basic combinational logic. *Header Size* also transits from the *Header Extraction* to the *Pipeline Alignment* module. However, this information is only required in the next cycle, which does not constitute a true data hazard.

3.2 Microarchitectural Aspects

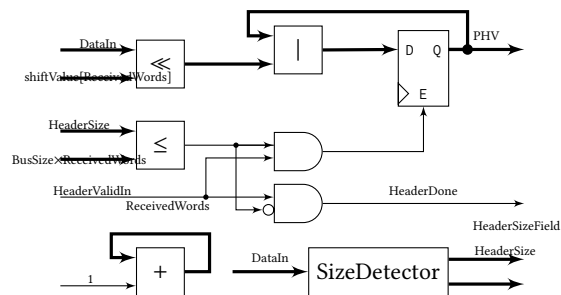
This subsection presents microarchitectural aspects of our proposed method. We start by presenting the state transition block. Details of the header extraction module are drawn followed by the pipeline alignment block. Then, we present the case of variable-sized headers.

3.2.1 State Transition Block. Figure 2 shows the state transition block which implements part of the ASM that represents the whole parser. Each *state* (header) of this ASM performs state transition evaluations by observing a specific field in the header and matching against a table storing the supported next headers for a given state. In this work, this table is filled at compilation time and it is part of what we call *Header Layout*.

The state transition block uses only barrel-shifters, counters, and comparators to perform state evaluations. Such operations can be easily done in an FPGA within a single clock cycle.

In Figure 2, *validHeader* is the result of a comparison between the *nextHeaderIn* and *thisHeader*. *thisHeader* is hardwired and it is part of the header layout. *validHeader* is used as an enable signal for all stateful components in the header instance. *ReceivedBits* is a counter that keeps track of the number of bits received in the same header. This information is used to check if the current data window belongs to the same window in which the *KeyValue* is placed in (*KeyLocation*). A barrel-shifter is used to shift the input data and to align it with the *KeyValue*. The bitwise AND (&) operation after the barrel-shifter guarantees this alignment. Finally, the *KeyMatch* compares the key aligned input data and the key table. If a match is found, the *NextHeader* is assigned to the value corresponding to the match and the *NextHeaderValid* is set. *HeaderException* is asserted otherwise.

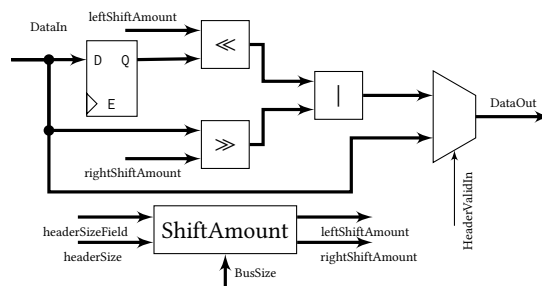
3.2.2 Header Extraction Block. Figure 3 shows the header extraction block which retrieves the header information from a raw input data stream. Similarly to the state transition block, this module is implemented using barrel-shifters, comparators, and counters. Additionally, this module calculates header sizes derived from the raw input data in case of variable-sized headers. For fixed-sized headers, the header size information is hardwired at compile time.



In the header extraction module architecture, the counter *ReceivedWords* is used to delimit the header boundaries for comparison with the *HeaderSize*. It is also used to index a table that stores the shift amounts for the barrel-shifter. This table is fixed and it is filled at compile time. The bitwise OR ($\|$) acts as an accumulator, receiving the current shifted and value accumulating it with the results from previous cycles. *HeaderDone* indicates that a header has been completely extracted.

The *SizeDetector* sub-block is hardwired for fixed-sized headers. For variable-sized headers, this sub-block has a behavior similar to the state transition module, returning the header size and the value of the field corresponding to the header size. More details regarding variable-sized headers are drawn in Section 3.2.4.

3.2.3 Pipeline Alignment Block. Unlike previous works, we opt for a bus-aligned pipeline architecture. That means that each stage in the parser pipeline aligns the incoming data stream before sending it to the next stage. This design choice reduces the complexity of the data offset calculation at the beginning of a stage. The bus alignment is done in parallel with other tasks within a stage and therefore has a low overall performance impact. The pipeline alignment block microarchitecture is depicted in Figure 4.



This block delays the input data and performs bit-shifts to remove the already extracted data at the same parser stage. Shift amounts are functions of the header size and the bus size. In the case of fixed-size headers, these shift amounts are hardwired. For variable-sized headers, they are calculated by the *ShiftAmount*, which is explained in more details in Section 3.2.4.

The output bus is then composed of data belonging to the current input data stream and from the previous cycle. When the current

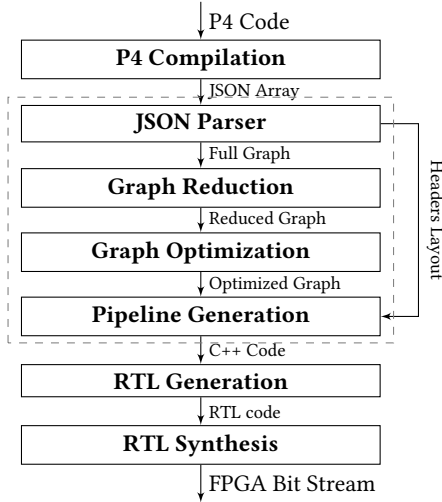


Figure 5: Parser pipeline generation.

header instance is not to be processed, in the case where *HeaderValidIn* is not set, this block just passes the input data to the output bus, playing the role of a bypass unit.

3.2.4 Handling Variable-sized Headers. It is not unusual to have a network protocol in which the header size is unknown until the packet arrives at a network equipment. The header size is inferred from a header field. IPv4 is such an example.

One approach to handle variable-sized headers would be to directly generate the required arithmetic circuit from the high-level packet processing program. However, this is an inefficient option based on our bus-aligned pipeline layout. In our architecture, supporting variable-sized would require dynamic barrel-shifters. Recall that a brute-force approach to design barrel-shifters uses a chain of multiplexers. For a N -bit barrel-shifter, this approach requires $N \log(N)$ multiplexers and introduces $\log(N)$ combinational delay units to the critical path, compromising both FPGA resources and performance.

To get rid of dynamic barrel-shifters, we are inspired by a technique available in modern high-level programming languages known as template metaprogramming. Template metaprogramming uses the compiler capabilities to compute expressions at compilation time, improving the application performance. Based on this technique, during the P4 compilation in our framework, we calculate all valid results of arithmetic expressions storing them into ROM memories. These expressions include header size calculation and shift amount taps for static barrel-shifters. The results for a variable-sized IPv4 header instance showed 13% LUT and 15% FF usage reduction when implementing these ROM memories rather than dynamic barrel-shifters.

3.3 Pipeline Layout Generation

The procedure to generate the parser pipeline is depicted in Figure 5. The input P4 code is compiled using the P4C compiler [12] producing a JSON array. We have chosen to use the result of the P4 back-end compilation (p4c-bm2-ss driver) for sake of simplicity.

Algorithm 1: Graph balancing algorithm

```

input :List of nodes representing a transitive reduced graph
input :Ordered list of nodes belonging to the longest path
output:Optimized balanced graph
Data: A node is a data structure that has pointers to
        successors/predecessors and methods to add/remove them. A node
        level represents the graph level and it is unassigned at the beginning.
1 Function graphBalance(tReducedGraph, longestPath)
   /* Compute the distance of all nodes to the root */
2   computeNodesLevel(tReducedGraph)
   /* Remove edges to successors from nodes not in the longest
   path */
3   for node in tReducedGraph do
4     if node  $\notin$  longestPath then
5       for sucNode in node.successors() do
6         removeEdge(node, sucNode)
   /* Adding spare edges to balance the graph */
7   for node in tReducedGraph do
8     if node  $\notin$  longestPath then
9       addEdge(node, longestPath[node.level + 1])
10  return tReducedGraph

```

Our work is limited to what is enclosed by the dashed rectangle in Figure 5 and it is written in Python. It starts with the parsing of the JSON array file. While parsing, the script extracts the data structures necessary to initialize the multiple C++ *Header* instances that compose the parser pipeline. The JSON parser also extracts the full parser graph. Figure 6a presents a full parser graph generated from a header stack comprising the following protocols: Ethernet, IPv4, IPv6, IPv6 extension header, UDP and, TCP.

For an efficient pipelined design, the graph illustrated in Figure 6a is not suitable. In that representation, almost all pipeline stages need bypass schemes to skip undesired state transitions, introducing combinational delays and increasing the resource usage due to the bypass multiplexers. We propose to simplify the original graph in order to have a more regular pipeline layout.

The graph simplification starts with the graph reduction phase that receives as input the full graph. This step performs a transitive reduction of the original graph in order to eliminate redundant graph edges. This phase also extracts the longest possible path of the parser graph. The result of this phase is shown in Figure 6b.

The graph presented in Figure 6c is an alternative representation for the reduced graph from Figure 6b. In this graph, a dummy node is introduced to offer the same reachability while balancing the graph. This dummy node only acts as a bypass element and therefore has no implementation cost, thus, they can be merged with existent nodes at the same graph level.

We propose a graph balancing algorithm in Algorithm 1 to optimize the reduced graph. It receives as parameters the transitive reduced parser graph and the longest path in the graph. As output, the algorithm returns a balanced graph tailored to our pipelined architecture. The first function call (line 2) in the algorithm executes the node level computation in relation to the root for all nodes. The first loop (lines 3 - 6) iterates over the nodes that are not in the longest path. It deletes the edges from these nodes to their children. The last loop (lines 7 - 9) iterates again over the nodes that are not part of the longest path and assigns a child to them. The chosen child is the first one belonging to the next graph level. Finally, the algorithm returns an optimized graph on line 10. An example of balanced graph is shown in Figure 6d.

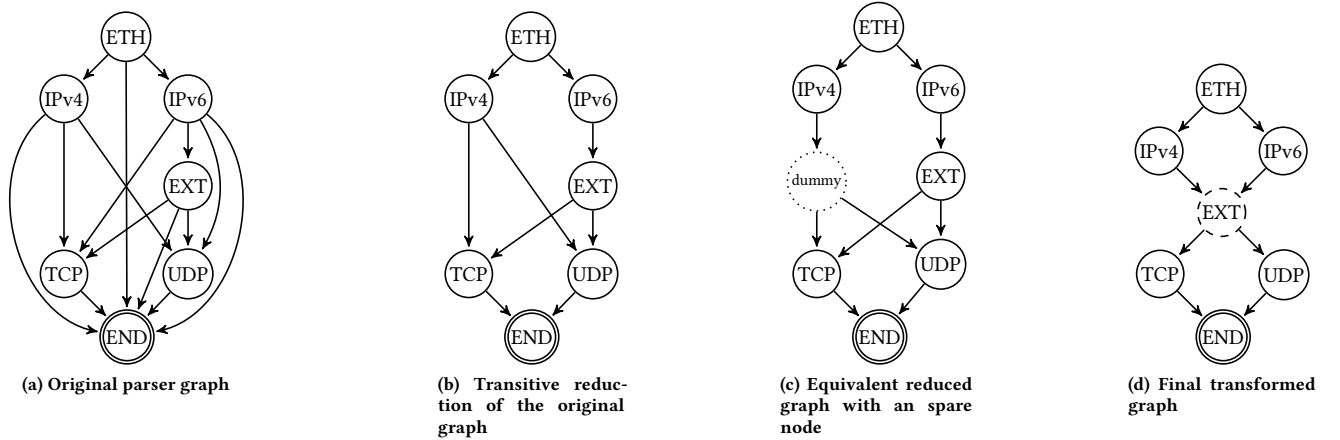


Figure 6: Parser graph transformation

The last step of the proposed approach illustrated in Figure 5 is the code generation. This phase receives as input a set of data structures representing the supported header layouts and the balanced graph. The header layouts are used to initialize both template and construction parameters for the C++ objects. The pipeline layout is drawn based on the balanced graph, with multiplexer insertion when required. The result of this phase is a synthesizable C++ code.

The generated C++ code is tailored for FPGA implementation. The next step in the processing chain is to generate RTL code for FPGA synthesis and place-and-route. Vivado HLS 2015.4 is used in this phase. Then, the generated RTL is synthesized under Vivado, which produces a bit stream file compatible with Xilinx FPGAs.

4 EXPERIMENTAL RESULTS

To demonstrate and evaluate our proposed method, we conducted two classes of experiments, the same ones performed in [2], to simplify comparisons. These two classes are defined as follows:

- **Simple parser:** Ethernet, IPv4/IPv6 (with 2 extensions), UDP, TCP, and ICMP/ICMPv6; and
- **Full parser:** same as simple parser plus MPLS (with two nested headers) and VLAN (inner and outer).

We used Vivado HLS 2015.4 to generate synthesizable RTL code. The RTL code was afterwards synthesized under Vivado 2015.4. The target FPGA device of this work was a Xilinx Virtex-7 FPGA.

Table 1 shows a comparison against others works present in the literature [2, 6] that support fixed- and variable-sized headers. In the case of [6], because they do not provide FPGA results, we reproduced their results based on a framework provided by the authors [5]. For that, we developed a script that converts the P4 code to the data structures needed in the framework.

Analysing the data from Table 1, both this work and [2] outperform [6], which is expected since the framework proposed in that work for automatic parser generation was designed for ASIC implementation and not for FPGA.

We assume as a golden model, labelled as Golden [2] in Table 1, a hand-written VHDL implementation presented in [2], which the authors used to evaluate their method.

Under the same design constraints, our work achieves the same throughput as [2], while not only reducing latency by 45% but also the LUT consumption by 40%. However, our architecture consumes more FFs, which is partially explained by the additional pipeline registers inferred by the Vivado HLS. Nonetheless, we can even have a lower overall slice utilization compared to [2], since in a Virtex-7 each slice has four LUTs and eight FFs, and our architecture does not double the number of used FFs.

Also, a notable resource consumption reduction is noticed when the number of extracted fields are reduced from all fields to 5-tuple, since a large amount of resources is destined to store the extracted fields, which matches with the findings reported in [6].

To compare the impact of our proposed pipelined layout, we implemented the pipeline organization proposed in [2] using the proposed header block architecture illustrated in Figure 1b since their source code was unavailable. This experiment is marked as "Hybrid [2] and this work" in Table 1. For the simple parser, our proposed architecture improves latency by more than 33%, while reducing by 16% and 10% number of used FFs and LUTs, respectively. In the case of the full parser, the latency was reduced by 39%, while the resource consumption follows the results of the simple parser.

Moreover, this hybrid solution also outperforms the original work [2] in both latency and LUT consumption. It shows that our microarchitectural choices are more efficient in these aspects. In addition, these better results can also be related to the language chosen to describe each architecture. In [2], they generated VHDL code from a P4 description. Our design uses templated C++ classes, which can fill the abstraction gap between the high-level packet processing program and the low-level RTL code.

When comparing to the golden model, the results obtained with our architecture are comparable to it in terms of latency. Our design, however, utilizes nearly twice the overall amount of logic resources, following what has been reported in [2]. However, since separate LUTs and FFs, or slice consumption results are unavailable, we cannot fairly compare resources utilization results.

As shown in Table 1, the present work achieves the best maximum frequency comparing to state-of-the-art, which allows scaling

Table 1: Parser results comparison

Work	Performance				Resources			Extracted Fields
	Data Bus [bits]	Frequency [MHz]	Throughput [Gb/s]	Latency [ns]	LUTs	FFs	Slice Logic (LUTs+FFs)	
Simple Parser								
[6]	256	184.1	47	N/A	14 906	2963	17 869	All fields
[6]	256	178.6	46	N/A	6865	1851	8716	TCP/IP 5-tuple
Golden [2]	512	195.3	100	15	N/A	N/A	5000	TCP/IP 5-tuple
[2]	512	195.3	100	29	N/A	N/A	12 000	TCP/IP 5-tuple
Hybrid [2] and this work	320	312.5	100	28.8	4699	7254	11 953	TCP/IP 5-tuple
This work	320	312.5	100	19.2	4270	6163	10 433	TCP/IP 5-tuple
This work	320	312.5	100	19.2	5888	10 448	16 336	All fields
Full Parser								
[6]	64	172.2	11	N/A	6946	2600	9546	All fields
[6]	64	172.2	11	N/A	3789	1425	5214	TCP/IP 5-tuple
Golden [2]	512	195.3	100	27	N/A	N/A	8000	TCP/IP 5-tuple
[2]	512	195.3	100	46.1	10 103	5537	15 640	TCP/IP 5-tuple
Hybrid [2] and this work	320	312.5	100	41.6	6450	10 308	16 758	TCP/IP 5-tuple
This work	320	312.5	100	25.6	6046	8900	14 946	TCP/IP 5-tuple
This work	320	312.5	100	25.6	7831	13 671	21 502	All fields

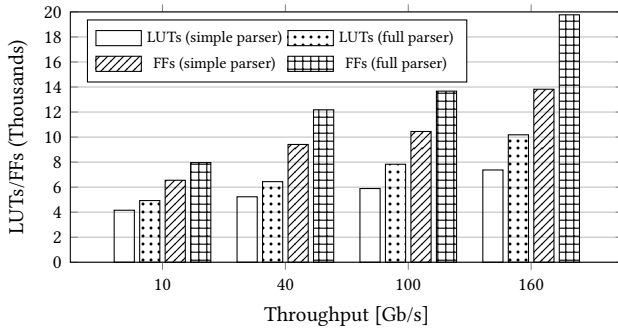


Figure 7: Synthesis results for multiple data rate parsers.

to data rates higher than 100 Gb/s. Figure 7 presents the design scalability results for data rates ranging from 10 Gb/s up to 160 Gb/s. It is worth noting that the data rate scaling causes a non-expressive impact in terms of LUTs, corresponding to an increase of 35 LUTs/Gbps in the case of the full 160 Gb/s parser.

5 CONCLUSION

FPGAs have increasingly gained importance in today's network equipment. FPGAs provide flexibility and programmability required in SDN-based networks. SDN-aware FEs need to be reconfigured to be able to parse new protocols that are constantly being deployed.

In this work, we proposed an FPGA-based architecture for high-speed packet parsing described in P4. Our architecture is completely described in C++ to raise the development abstraction. Our methodology includes a framework for code generation, including a graph reducing algorithm for pipeline simplification. From modern high-level languages, we borrowed the idea of metaprogramming to perform offline expressions calculation, reducing the burden of calculating them at run-time.

Our architecture performs as well as the state-of-the-art while reducing latency and LUT usage. The latency is reduced by 45% and

the LUT consumption is reduced by 40%. Our proposed methodology allows a throughput scalability ranging from 10 Gb/s up to 160 Gb/s, with moderate increasing in logic resources usage.

ACKNOWLEDGMENTS

The authors thank A. Abdelsalam, M. D. Souza Dutra, I. Benacer, T. Stimpfling, and T. Luinaud for their comments. This work is supported by the CNPQ-Brazil.

REFERENCES

- [1] Michael Attig and Gordon Brebner. 2011. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems (ANCS '11)*. IEEE Computer Society, Washington, DC, USA, 12–23. <https://doi.org/10.1109/ANCS.2011.12>
- [2] P. Benáček, V. Pu, 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. <https://doi.org/10.1109/FCCM.2016.46>
- [3] Pat Bosshart et al. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 99–110. <https://doi.org/10.1145/2534169.2486011>
- [4] Pat Bosshart et al. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. <https://doi.org/10.1145/2656877.2656890>
- [5] G. Gibb. 2013. Network Packet Parser Generator. <https://github.com/grg/parser-gen>. (2013).
- [6] G. Gibb et al. 2013. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*. 13–24. <https://doi.org/10.1109/ANCS.2013.6665172>
- [7] N. Gude et al. 2008. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.* 38 (2008), 105–110.
- [8] Viktor Pus, Lukas Kekely, and Jan Korenek. 2012. Low-latency Modular Packet Header Parser for FPGA. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '12)*. ACM, New York, NY, USA, 77–78. <https://doi.org/10.1145/2396556.2396571>
- [9] Haoyu Song. 2013. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. ACM, New York, NY, USA, 127–132. <https://doi.org/10.1145/2491185.2491190>
- [10] The Open Networking Foundation. 2012. Software-Defined Networking: The New Norm for Networks. (April 2012).
- [11] The Open Networking Foundation. 2014. OpenFlow Switch Specification. (Dec. 2014).

- [12] The P4 Language Consortium. 2017. P4 Compiler. <https://github.com/p4lang/p4c>. (2017).
- [13] Xilinx Inc. 2017. P4-SDNet Translator User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1252-p4-sdnet-translator.pdf. (2017).
- [14] S. Zhou, W. Jiang, and V. K. Prasanna. 2014. A flexible and scalable high-performance OpenFlow switch on heterogeneous SoC platforms. In *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*. 1–8. <https://doi.org/10.1109/PCCC.2014.7017053>