2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines

P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers

Pavel Benáček, Viktor Puš
CESNET a.l.e.
Zikova 4,160 00 Prague
Czech Republic
Email: benacek,pus@cesnet.cz

Hana Kubátová
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9,16000 Prague 6
Czech Republic
Email: hana.kubatova@fit.cvut.cz

***Abstract*—Software Defined Networking and OpenFlow offer an elegant way to decouple network control plane from data plane. This decoupling has led to great innovation in the control plane, yet the data plane changes come at much slower pace, mainly due to the hard-wired implementation of network switches. The P4 language aims to overcome this obstacle by providing a description of a customized packet processing functionality for configurable switches. That enables a new generation of possibly heterogeneous networking hardware that can be run-time tailored for the needs of particular applications from various domains.**

In this paper we contribute to the idea of P4 by presenting design, analysis and experimental results of our packet parser generator. The generator converts a parse graph description of P4 to a synthetizable VHDL code suitable for FPGA implementation. Our results show that the generated circuit is able to parse 100 Gbps traffic with fairly complex protocol structure at line rate on a Xilinx Virtex-7 FPGA. The approach can be used not only in switches, but also in other appliances, such as application accelerators and smart NICs. We compare the generated output to a hand-written parser to show that the price for configurability is only a slightly larger and slower circuit.

I. Introduction

OpenFlow [1], as the most popular embodiment of the ideas of Software Defined Networking, provides a way to fill the match tables of switches at runtime. The OpenFlow specification defines exactly which protocols must be supported by the switches to support all matching features. At the same time, it is not possible to change that set of protocols – switches' packet parsers are fixed. (Or at least they appear so to the OpenFlow controller.)

P4: Programming Protocol-independent Packet Processors [2], [3] makes a step forward and evades the limitation of fixed protocol set. It defines a way to configure the packet parsing functionality of switches at runtime. It is envisioned that a P4-capable switch translates this platform-independent parser description into a representation that fits the actual hardware resources of the switch. Such representation may include a code for a general or specialized processor, an FPGA firmware, an advanced ASIC configuration or other computational platforms.

Since P4 is a visionary and relatively new concept, there are no commercially available switches supporting it at the time of writing this paper (December 2015). We contribute towards the vision of P4 by designing and evaluating a generator of high-speed packet parser suitable for FPGAs. The generator's output is a synthetizable VHDL code that performs packet parsing as defined by the P4 program. Its internal structure is inspired by that of a parser hand-written by a skilled HDL programmer and therefore there is only a small difference in chip area and frequency. Parser's data width and number of internal pipeline stages are configurable parameters, so that the parser can be tuned for particular throughput, area and latency constraints. Since we envision that P4 may span applications besides network switches, we believe that our work can be used in other appliances, such as application accelerators, smart NICs and various network security devices.

The rest of the paper is organized as follows: Section II presents other papers related to our work. Section III provides more details about the P4 language aspects that are relevant for this work. Section IV describes the design of the generator as well as the structure of parsers it generates. Section V provides results of our generator and compares them to a hand-written parser. Finally, section VI draws conclusions from the results.

II. Related Work

NetPDL [4] is an XML-based language for packet header description. Its capabilities are similar to header formats and packet parser description in P4. Similarly, Attig and Brebner in [5] present the PP language, which serves for the description of packet headers and parse graphs. They also demonstrate a compiler from PP to FPGA representation.

Both approaches lack description of the packet processing functionality that follows after packet parsing. In order to allow seamless continuation of our work in that direction, and due to recent good reception of P4 in the SDN community, we have chosen P4 as our starting point.

Gibb et al. in [6] present a detailed design of fixed and configurable packet parsers for ASICs. Reconfigurable Match Tables (RMT) [7] provide a design of a complete packet processing functionality similar to P4, including packet parsing. The design of packet parser in RMT is rather simple and relies heavily on associative memories.

Both of these works assume ASICs as the target implementation platform. Our work aims at exploiting the on-field

978-1-5090-2356-1/16 $31.00 © 2016 IEEE
DOI 10.1109/FCCM.2016.46

structural programmability of FPGAs, avoiding overheads of general-purpose architecture.

The P4 Language Consortium provides some basic tools which can be used in 3rd party projects. Source codes of these tools are publicly available under open source license. The main project is P4-HLIR [8] which is the front end of the P4 compiler, creating Python object model of the P4 program. It becomes useful for other projects because one can easily continue with implementation of compiler's back end from this object representation. P4C-BEHAVIORAL [9] and P4C-GRAPHS [10] are examples of back ends for two different targets - C language in case of P4C-BEHAVIORAL

III. The P4 Language

P4: Programming Protocol-independent Packet Processors [2], [3] 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 in the field, protocol independence and target (platform) independence. Using relatively simple syntax, P4 allows to define five basic aspects of packet processing:

- **Header Formats** describe protocol headers recognized by the device.
- **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).
- **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, the first two aspects of P4 are relevant. Header format description may look like this:

```
header ethernet {
   fields {
      dst_addr : 48; // width in bits
      src_addr : 48;
      ethertype : 16;
   }
}
```

The description simply lists fields of the packet header and their width in bits. The example above shows the situation with static header where the header length is the sum of lengths of all fields. This can't be done for protocols with variable header length. 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_t {
    fields {
        nextHdr     : 8;
        totalLen    : 8;
        frag        : 12;
        padding     : 3;
        fragLast    : 1;
    }

    length : (totalLen + 1) * 8;
    max_length : 1024; // Bytes
}
```

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;
      case 0xA100 mask 0xF100 : myProto;
   }
}
```

The provided example uses `switch` and `extract` statements. The `extract` statement instructs the parser to examine input packets and look for data defined in the header. Parsed data is then used in the `switch` statement to determine the next state (protocol) to process. There are also situations when we don't want to use the whole value from the protocol field. The P4 language solves this with the `mask` statement which is used in the `case` statement together with mask value. In our example, the `mask` statement instructs the P4 parser to take the `ethertype` field, perform logical and operation between the value and mask and compare the result to `0xA100`.

IV. Parser Generator Design

This section introduces the basic architecture (Section IV-A) of firmware parser module called HFE M2 [11] and a transformation algorithm (Section IV-B) from a P4 description to the VHDL architecture of HFE M2. Section IV-C describes how to infer special parameters of HFE M2 architecture which allow us to optimize consumed resources of FPGA chip. Finally, section IV-D analyzes time complexity of the transformation algorithm.

A. Overview of HFE M2

The main idea of automatic generation of 100 Gbps parser comes from the architecture of HFE M2. The architecture consists of two main block types - protocol analyzers and pipelines. Protocol analyzers are the heart of the whole approach. A generic interface is used for connection between the protocol analyzers. There is an optional pipeline block between each two protocol analyzers. The pipeline blocks can be individually enabled/disabled at compile time to tune the

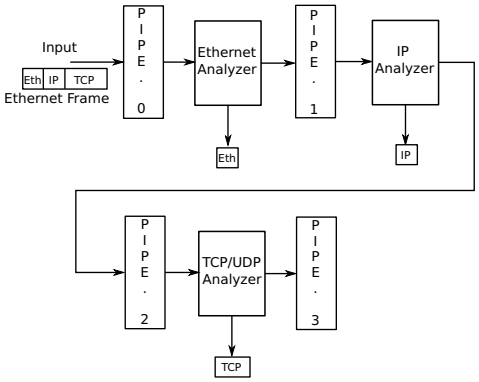

Fig. 1. HFE M2 architecture

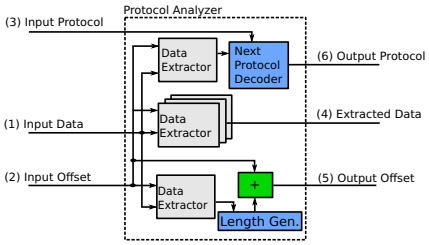

Fig. 2. Protocol Analyzer Architecture

final frequency, latency and chip area. Protocol analyzers and pipeline blocks are connected to the processing chain which represents the protocol stack of incoming network packet. No memory (DDR or similar) is used in the HFE M2 - we are working with the pure pipeline architecture which is based on exchange of incoming and control data. An example of the HFE M2 processing chain is shown in Fig. 1.

Protocol analyzer uses *Generic Protocol Parser Interface* (GPPI) for connection between modules. This interface provides the input information necessary to parse a single protocol header. That is: (1) current packet data being transferred at the data bus, (2) current header offset within the packet and (3) expected protocol to parse. GPPI output information includes (4) extracted packet header field values, plus the information needed to parse the next protocol header: (5) next header offset and (6) type of the next protocol header. More details about the GPPI can be found in [11]. Brief architecture of each protocol analyzer block is shown in Fig. 2.

Protocol Analyzer architecture contains four basic block types: (1) Data Extractors, (2) Next Protocol Decoder, (3) Length Generator and (4) Adder. Data Extractors are used to extract packet data from a given offset. Data Extractors are configured with two parameters - *Extract Length* and *Extract Offset*. The first parameter defines the number of extracted bytes from packet data. The offset of data within the packet is computed as the sum of current header offset (a value from GPPI interface) and the Extract Offset parameter. Note that Data Extractor blocks contain multiplexers which allow to extract data from any byte position. These multiplexers can be configured with some additional optimization parameters which have an influence on consumed resources. We describe these parameters in Sec. IV-C.

Next Protocol Decoder is used to compute the next expected protocol. Its structure is fully dependent on the protocol header format. Generally, it is a function converting some extracted packet header bytes into an internal number representing the protocol type.

Length Generator block is used to compute the length of current protocol header, so that it can be added to the Input Offset signal to obtain the Output Offset signal, which represents the start offset of the next protocol header. The added offset value can be a constant or a result of an equation (see the header format specification in previous section).

From the perspective of parser generation, we can identify three basic types of blocks in the Protocol Analyzer structure (see Fig. 2): (1) Static (green color), (2) Configured (grey color), (3) Fully protocol-specific (blue color). The static block is used in every Protocol Analyzer without any change. The protocol analyzer architecture contains only one static block, which is the adder. This block is used to compute the next protocol offset from current Input Offset and Length Generator output. The second group of blocks is general enough for usage in all protocol analyzers, only with different parameters settings. The architecture contains several Data Extractor blocks which are instantiated (from hand optimized template) and configured regarding to Header Format specification. There are two blocks marked blue. These blocks are entirely protocol specific, so that every protocol analyzer needs custom implementation of them. That means that Protocol Decoder and Offset Generator must be uniquely generated for each protocol analyzer from P4 Header Description.

This architecture of Protocol Analyzer is general enough for processing of most protocols. For this target block structure, we can generate, configure and connect all described blocks in automatic way from a P4 program. Details of this transformation process are discussed in the next section.

B. Transformation from P4 to HFE M2

Transformation algorithm is one of the key problems addressed in this paper. As we note in Sec. IV-A, HFE M2 architecture consists of protocol analyzers and pipeline modules which are connected to processing chain. The transformation from P4 to HFE M2 can be divided into two basic problems - (1) Generating the protocol analyzers and (2) Generating the processing chain. Inputs of the transformation process are Header Format and *Parser Graph Representation*.

We define the Parser Graph Representation (PGR) as an oriented graph which is generated from the P4 Packet Parser description. Each node (or state) represents one packet header and each transition represents the next parsed protocol header. Each transition is taken based on the parsed data. Condition of a transition is inferred from the P4 Packet Parser description. Loop edge represents situation when we want to support more protocols of the same type in protocol stack (like VLAN or MPLS stacking, for example). Each non-finite state contains

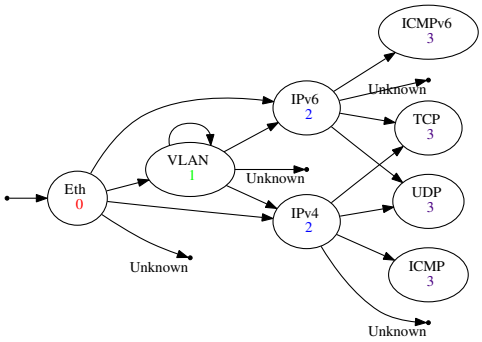

Fig. 3. Parser Graph Representation; supports 2xVLAN, IPv4, IPv6, TCP, UDP, ICMP, ICMPv6

additional transition to the *Unknown* state. This state is not explicitly described in P4 program but it is implicitly required by the parser. It represents the situation when no value matches the actual set of transition conditions (e.g. we cannot continue in the parsing of next protocol header). Each PGR node also contains a pointer to Header Format specification which is needed during generation of individual Protocol Analyzers. The PGR representation is built from a P4 Packer Parser description using the *depth-first search* (DFS) algorithm. We introduce more details about this structure in the following text. An example of this generated representation is in Fig. 3. The figure doesn't show transition conditions in order to keep it well arranged.

As we note in Sec. IV-A, Protocol Analyzer consists of four basic types of blocks. We now describe how each Protocol Analyzer block is generated.

The Length Generator block is derived directly from P4 Header Description. It can be either a constant in case of constant length header, or a (usually simple) formula in case of variable length header.

The Next Protocol Decoder is also generated from the P4 Packet Parser description. Each transition from the parser state is described in the P4 `switch` statement by the tuple (*value*, *next state*). Therefore we can implement Protocol Decoder by a multiplexer which selects the next protocol based on currently parsed values. The protocol headers that follow the currently parsed header are found in PGR.

Extracted Data of Packet Analyzer can be inferred from the Header Format specification because we know the structure of protocol fields in the parsed protocol. Therefore, we can extract protocol fields from extracted data using the list of protocol fields and their sizes.

Adder is a static block common to all Protocol Analyzers.

Finally, Data Extractor blocks are parameterizable units which can be used in all Protocol Analyzers without any change, only by setting the parameters to match the target protocol. When generating the Next Protocol Decoder and Length Generator blocks and Extracted Data outputs, Data Extractors are instantiated and parametrized as needed. Both Extract Length and Extract Offset parameters are directly derived from the P4 description.

Algorithm 1: Recursive algorithm for identification of node levels

```
Function FindNodeLevels(node, curr_level)
    Data: node = actual node to process
    Data: curr_level = actual level of the node
    Result: Node with updated maximal level
    begin
        if node.fresh == False then
            return;
        end
        /* Mark the node as not fresh and
           update the level                    */
        node.fresh = False;
        act_level = node.get_level();
        if act_level ¡ curr_level then
            node.set_level(curr_level);
        end
        /* For all fresh successors, update
           the level and call the same
           function                            */
        node_successors = node.get_next_states();
        for next_node in node_successors do
            /* Don't call the node if the
               longest path already exists     */
            if next_node.get_level() - node.get_level() ¡ 1 then
                FindNodeLevels(next_node,curr_level+1);
            end
        end
        /* Mark the node as not visited        */
        node.fresh = True;
        return;
    end
```

Generation of the processing chain uses PGR as an input. The key problem is to identify a place for insertion of each protocol analyzer in the processing chain. This chain of protocol analyzers represents processed protocol stack as we describe in Sec. IV-A. Our task is to identify the longest paths to each node in a PGR. Length of the longest path from root to a node represents a position in the processing chain. If we have several nodes on the same level, we connect protocol analyzers in series with arbitrary ordering. While same-level analyzers could be connected in parallel, the serial approach allows us to keep the homogeneous structure of processing chain. The longest paths in PGR is found using the Alg. 1. The algorithm recursively traverses and identifies node levels in inspected PGR. The result of this algorithm is shown in Fig. 3, where each node contains a number which represents the length of the longest path from root (e.g. the latest possible use of a protocol header in a packet).

Finally, we introduce a brief Alg. 2 which is used for generation of complete HFE M2 architecture from a P4 description. We have implemented this complete transformation algorithm in Python language with usage of P4-HLIR [8] project.

The result of Alg. 2 can be seen in Fig. 4 which represents the processing chain generated from the PGR in Fig. 3. Fig. 4

Algorithm 2: Brief transformation algorithm from P4 to HFE M2

```
Procedure TransformationToHFEM2(prog)
    Data: prog = P4 Program
    Result: VHDL code of the HFE M2 architecture
    begin
        /* 1) Identify the Parser Graph
           Representation                    */
        parser_graph =
        GetParserGraphRepresentation(prog);
        /* 2) Mark all nodes as Fresh. After
           that, traverse through the graph
           and identify level of each node   */
        MarkFresh(parser_graph);
        FindNodeLevels(parser_graph.root,0);
        /* 3) Generate Protocol Analyzers and
           processing chain                  */
        GenerateProcessingPipeline(parser_graph);
    end
```

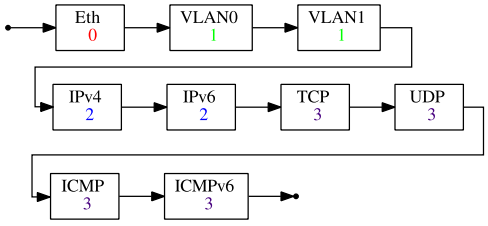

Fig. 4. Generated processing chain; pipeline modules are omitted for brevity

doesn't contain any pipeline modules for brevity, but the real hardware implementation contains a pipeline module between each two adjacent protocol analyzers. The figure also shows expanded VLAN modules because our input PGR supports two VLAN headers. This figure also shows the situation when two or more different nodes are situated on the same level. Such nodes are connected in series and their relative position doesn't matter. The only rule is to keep them together (i.e. to connect modules which belong to the same level in series).

As we note in III, we use the subset of P4 language which is related to packet parsing description. The P4's parser statement supports another constructs which are related to setting of metadata fields (intrinsic or user-defined). This functionality is not supported by our tool because it is not related to data extraction from ingress data. However, this functionality can be easily implemented into existing HFE M2 architecture.

C. Optimizations

The original hand-written HFE M2 parser supports two optimizations which save significant amount of chip resources. Therefore, we have decided to support some of these optimizations in our generator as well.

The first optimization is related to protocol analyzers' GPPI interface. The key idea is to optimize width of the offset bus which is used for signalization of protocol header start (Input Offset and Output Offset signals). In another words, Input Offset and Output Offset signals are *pointers* into the packet.

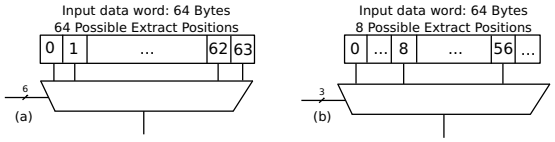

Fig. 5. Example of data extraction multiplexer: full (a), optimized (b)

The reason for this optimization is that protocol stack is being analyzed in a sequential manner within the packet. Therefore, the width of offset bus can increase in a sequential manner too – it is unnecessary to implement all logic (adders etc.) at full data width. The bus width parameter is inferred from maximal protocol header lengths during translation. The bus width on each level is computed as a binary logarithm of the sum of all preceding protocol header lengths. Narrower offset bus leads to smaller modules for computation of next header offset and other values and thus saves chip resources.

The second optimization is related to data extraction which is performed by multiplexers within Data Extractor blocks. Generally, each Data Extractor is able to extract data from any byte position in the packet bus data word. The multiplexer is controlled by current data bus offset and the offset of the desired field within the header. However, given the fact that the packet header may start only at certain positions on the data bus, current offset can contain only values with the corresponding resolution. This resolution is computed from Header Format and Packet Parser description. Using these two specifications, we identify a list of all possible starting positions of each analyzed header in the processing chain. This knowledge is built from a protocol header length and relations between protocol headers by simulation of data transfer on data bus. Computed lists are used for identification of required multiplexer's parameters. By making Data Extractors less general, we simplify the structure of each extraction multiplexer and save a significant amount of chip resources. An example of this optimization is shown in Fig. 5.

There is also a place for optimizations of P4 program which cannot be automatically generated. Instead, it is required to optimize the program during design time. The generator can then benefit from more efficient input, which results in better design in terms of latency and consumed resources. In this text, we introduce one such optimization of P4 program which leads to less protocol analyzer blocks being generated. The basic idea is to merge protocol headers that are compatible in term of extracted protocol header fields. As an example, we want to extract source and destination ports of TCP and UDP protocol. Therefore, we create a new fake protocol header which describes export of interesting fields. We don't have to worry about incomplete protocol header specification because our replaced protocol headers are leaves of PGR (see Fig. 3), so that there is no further processing after this merged header. We define the new protocol header like this:

```
header tcp_udp_t {
    fields {
```

```
        src_port : 16; // width in bits
        dst_port : 16;
    }
}
```

D. Time complexity

Time complexity of the proposed transformation Alg. 2 consists of following components:

1) `GetParserGraphRepresentation`'s time complexity is equal to $\mathcal{O}(V + E)$ (DFS algorithm), where V is the number of nodes and E is the number of edges. Our transformation algorithm requires PGR with no cycles (loop edges are ignored because they means a protocol repetition). In general, maximal number of edges in acyclic graph is equal to $\frac{n}{2} * (n-1)$ where n is the number of protocol analyzers (e.g. nodes of our graph). Total time complexity of DFS is $\mathcal{O}(n + \frac{n}{2} * (n-1)) \sim \mathcal{O}(n^2)$
2) `FindLongestPaths`'s time complexity is equal to $\mathcal{O}(n^2)$ (DFS algorithm), where n is the number of protocol analyzers
3) `MarkFresh`'s time complexity is equal to $\mathcal{O}(n)$, where n is a number of protocols (e.g. nodes of PGR)
4) `GenerateProcessingPipeline`'s time complexity is equal to $\mathcal{O}(n)$ because we are generating a processing chain with n protocol analyzers

Total time complexity of the transformation is $\mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(2 * n^2) + \mathcal{O}(2 * n) \sim \mathcal{O}(n^2)$.

V. Results

We have generated parsers supporting the following protocol stack: Ethernet, up to two VLAN headers, up to two MPLS headers, IPv4 or IPv6 (with up to two extension headers), TCP or UDP, ICMP or ICMPv6. The parser is able to extract the classical quintuple: (*IP addresses*, *protocol*, *port numbers*).

We have tested properties of generated parsers with two different protocol stacks:

- **full** - Ethernet, 2×VLAN, 2×MPLS, IPv4/IPv6 (with 2×extension headers), TCP/UDP, ICMP/ICMPv6
- **simple L2** - Ethernet, IPv4/IPv6 (with 2×extension headers), TCP/UDP, ICMP/ICMPv6

For each mentioned protocol stack, we compare the manually optimized HFE M2 parser and the generated parser with all optimizations enabled. All tested designs implement parsing engines without any additional logic like outputs FIFOs for parsed data etc. We use the Slice Logic (number of used LUTs and REGs) as a metric of resource utilization because these parts are the most utilized in most FPGA designs.

We provide results after synthesis for the Xilinx Virtex-7 XCVH580T FPGA using Xilinx Vivado 2015.1 design tool. All designs were synthesized with different settings of the data width and the number of pipeline stages. These settings, together with the resulting frequency, latency and resource usage, generate the large space of possible solutions for each P4 program. These solutions were searched for Pareto set which allows us to pick the best-fitting solution for an application.

TABLE I
Comparison of different optimization methods with resource consumption for the Xilinx Virtex-7 XCVH580T FPGA

Opt.	Pipes	Latency [ns]	Thr. [Gbps]	Slice LUT [-]	Slice REG [-]
O0	8	39.8	102.7	25335 (6.98%)	5055 (0.69%)
O1	14	75.3	101.9	21477 (5.91%)	8930 (1.23%)
O2	**8**	**46.1**	**100.0**	**10103 (2.78%)**	**5537 (0.76%)**
O3	9	44.5	115.1	14270 (3.93%)	5427 (0.74%)
O4	7	40.7	100.7	8314 (2.29%)	4795 (0.66%)

In each **hand optimized** design testcase, we use two different data widths: 256 and 512 bits. For each data width, every possible placement of pipelines was tested: 2^9 possible combinations in the case of the full protocol stack and 2^5 combinations in the case of simple L2 protocol stack. (Because there are 9 and 5 configurable pipeline stages, respectively, in those parsers.)

In each **generated** parser testcase, we use two different data widths: 256 and 512 bits. For each simple L2 parser, every possible placement of pipelines was tested: 2^{10} possible combinations. In the case of full parser, there are 2^{14} possible solutions. We have randomly selected 20% of all possible solutions. This approach allows us to briefly inspect properties of generated processing chain in a reasonable compile time.

For each test case, we provide two graphs: the first graph shows the relation between throughput and FPGA resources. The second graph shows the relation between throughput and latency of the tested parser. Finally, we provide two more graphs with Pareto sets: one showing Pareto sets optimized for throughput and chip area without any regard to latency, and second optimized for throughput and latency without any regard to FPGA resources. We also introduce results for optimizations which are described in Sec. IV-C because they have an influence on latency and used resources. We define the following notation:

- **No optimizations (O0)** - no optimizations are used
- **Offset optimization (O1)** - optimization of the offset width
- **Offset + multiplexer optimization (O2)** - offset and multiplexer optimizations
- **Optimized P4 program (O3)** - O0 version with effectively written P4 program (manual optimization)
- **All optimizations (O4)** - all proposed optimizations are used

Resource utilization (values in parentheses represent the percentual usage of available FPGA's resources) and latency for parsers generated with different optimizations are shown in Tab. I. The table shows the influence of proposed optimizations on similar hardware configurations with throughput around 100 Gbps. For all of the following results we use O2 optimization, since O3 and O4 require manual modifications to the original P4 program, which we consider highly non-standard scenario.

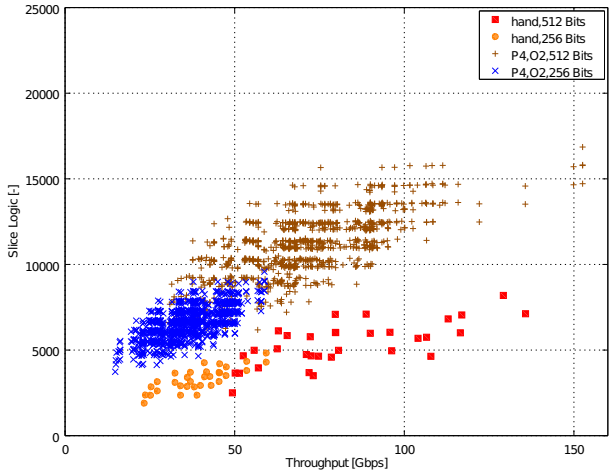

Fig. 6. The FPGA resource utilization for different settings of the simple L2 parser

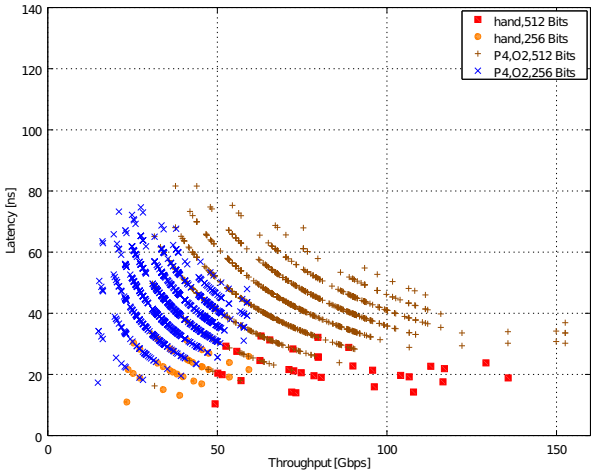

Fig. 7. The latency for different settings of the simple L2 parser

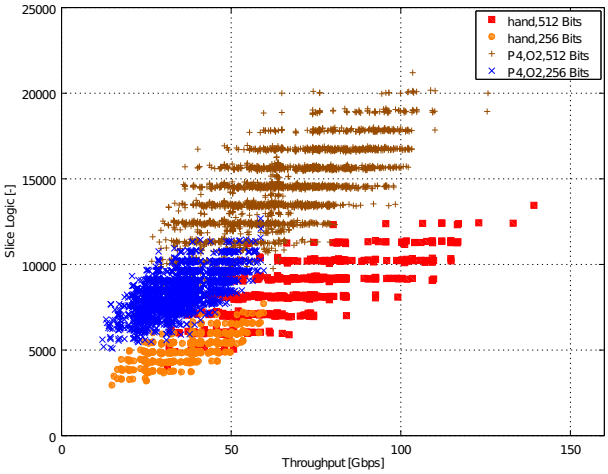

Fig. 8. The FPGA resource utilization for different settings of the full parser

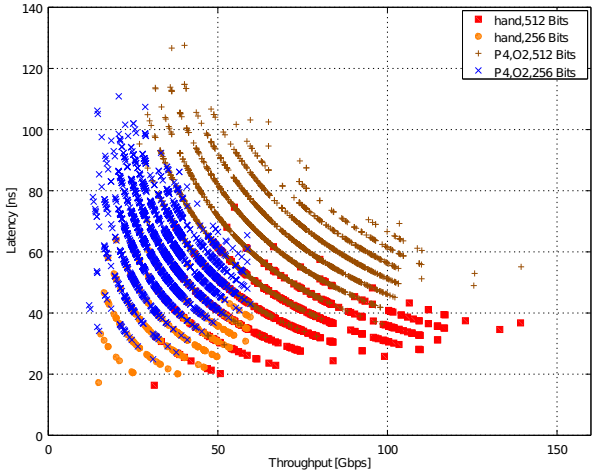

Fig. 9. The latency for different settings of the full parser

Fig. 6 shows throughput and FPGA resources and Fig. 7 shows throughput and latency for simple L2 protocol stack. Each point in both figures represents one tested solution for hand optimized and P4 based HFE M2 parsers. Fig. 8 shows throughput and FPGA resources and Fig. 9 shows throughput and latency for full protocol stack. It can be seen that the data width of 256 bits is suitable for throughput up to 60 Gbps, while for higher throughput a 512b pipeline is needed.

For comparison of the achieved Pareto set results for different protocol stacks, we provide graphs in Fig. 10 (throughput and FPGA resources) and Fig. 11 (throughput and latency). The Pareto sets show the best achievable solutions for our parsers. From these figures, we can see that supported protocol stack can significantly change parameters of the parser in terms of FPGA resources and latency. We can also see that P4 based parsers are approximately two times worse in terms of both latency and consumed resources. The average price (in Slice Logic per Gbps) of hand-optimized parsers from the Pareto set are 61 and 102 for the simple L2 and full variants respectively, while the generated parsers cost 125 and 168 Slice Logic/Gbps. This is because manually created parsers include some even more advanced optimizations, which are highly protocol-specific and could not be included in our generator for the sake of universality. The highly protocol-specific optimization of hand written modules is similar to node merging which was introduced in section IV-C. We assume that the difference between P4-generated parsers (with O2 optimization) and hand optimized parsers is small enough to justify the added flexibility of P4 in many cases.

From presented results, we can infer following important conclusions:

1) We are able to generate parsers with equal functionality in shorter time.
2) Generated parsers aren't significantly worse than hand optimized versions created by a professional with many years of experience in HDL coding.

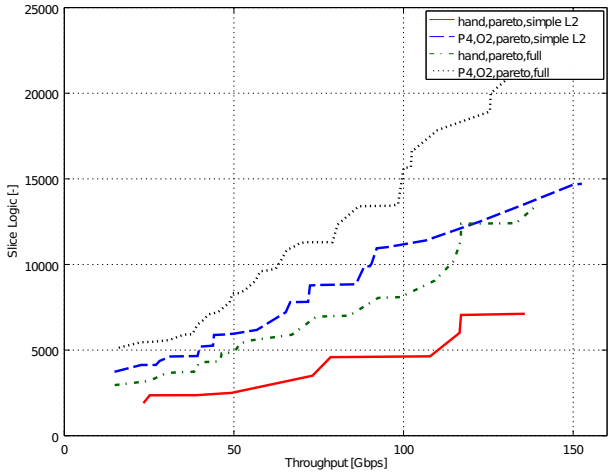

Fig. 10. Comparison of the FPGA resource utilization versus throughput Pareto sets for the tested protocol stacks

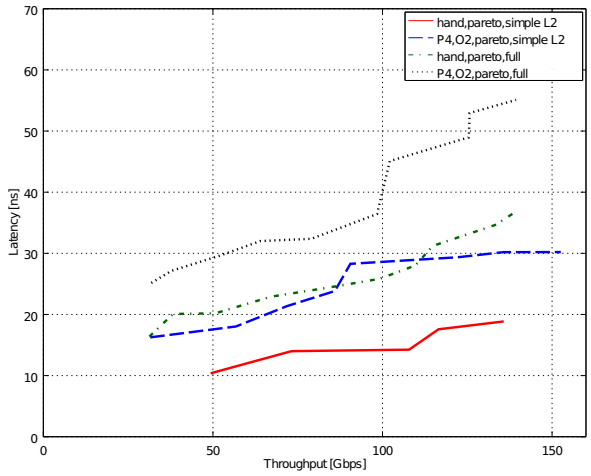

Fig. 11. Comparison of the latency versus throughput Pareto sets for the tested protocol stacks

VI. Conclusion

Each networking device needs to parse incoming data in order to perform subsequent actions. This paper presents the transformation tool from P4 language to a highly configurable packet parser for FPGA, which achieves throughput above 100 Gbps and is usable in a variety of application. We present the details of the transformation algorithm which directly generates HDL code from a given P4 description.

One can see from the presented results that we can generate a parser from P4 description with throughput, latency and chip area parameters similar to a hand optimized solution. This parser can be generated from a P4 program without any knowledge of hardware description language. That makes this approach more feasible for networking experts without deep knowledge of FPGA programming. The second advantage is related to the time of development. By using our P4-to-VHDL generator, the development time of a networking device can be shortened significantly. Once the high-level P4 description is ready, it takes only a couple of seconds to generate a VHDL parser. Moreover, the advantage of a growing P4 ecosystem is that additional tools are available. For example, one can first generate a C program from P4, perform software conformance test, and only then easily switch to hardware, having part of the system already tested and verified.

The full version of generated parser consumes only 2.78% Slice LUTs and 0.76% Slice REGs of the Xilinx Virtex-7 XCVH580T FPGA to achieve throughput of 100 Gbps (in the case of O2 optimization). This result leaves most of the FPGA resources free for other functions of a target application. Moreover, we can reach even smaller resource consumption in the case of O4 optimization. However, this requires manual modifications to the original P4 program, which we consider rather non-standard approach.

Acknowledgment

This research has been partially supported by the "CESNET E-infrastructure" project no. LM2015042 funded by the Ministry of Education, Youth and Sports of the Czech Republic, the grant SGS15/122/OHK3/1T/18 and by the European Union in the context of the "BEBA" project (Grant Agreement: 644122).

References

[1] Open Networking Foundation, "Open Flow," https://www.opennetworking.org/sdn-resources/openflow.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[3] P4 Language Consortium, "P4," http://p4.org/.

[4] F. Risso and M. Baldi, "Netpdl: An extensible xml-based language for packet header description," *Comput. Netw.*, vol. 50, no. 5, pp. 688–706, Apr. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2005.05.029

[5] M. Attig and G. Brebner, "400 gb/s programmable packet parsing on a single fpga," in *In Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, ANCS '11*. IEEE Computer Society, 2011, pp. 12–23.

[6] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, Oct 2013, pp. 13–24.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486011

[8] P4 Language Consortium, "P4-HLIR," https://github.com/p4lang/p4-hlir.

[9] P4 Language Consortium, "P4C-BEHAVIORAL," https://github.com/p4lang/p4c-behavioral.

[10] P4 Language Consortium, "P4-GRAPHS," https://github.com/p4lang/p4c-graphs.

[11] V. Pus, L. Kekely, and J. Korenek, "Low-latency modular packet header parser for fpga," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12. New York, NY, USA: ACM, 2012, pp. 77–78. [Online]. Available: http://doi.acm.org/10.1145/2396556.2396571