Design Methodology of Configurable High Performance Packet Parser for FPGA

Viktor Puš, Lukáš Kekely
CESNET a. l. e.
Zikova 4, 160 00 Prague, Czech Republic
Email: pus,kekely@cesnet.cz

Jan Kořenek
IT4Innovations Centre of Excellence
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
Email: korenek@fit.vutbr.cz

***Abstract*—Packet parsing is among basic operations that are performed at all points of a network infrastructure. Modern networks impose challenging requirements on the performance and configurability of packet parsing modules. However, high-speed parsers often use a significant amount of hardware resources. We propose a novel architecture of a pipelined packet parser for FPGA, which offers low latency in addition to high throughput (over 100 Gb/s). Moreover, the latency, throughput and chip area can be finely tuned to fit the needs of a particular application. The parser is hand-optimized thanks to a direct implementation in VHDL, yet the structure is uniform and easily extensible for new protocols.**

Keywords*—*Packet Parsing; Latency; FPGA

I. Introduction

Since computer networks evolve both in terms of speed and diversity of protocols, there is still a need for packet parsing modules at all points of the infrastructure. This is true not only in the public Internet, but also in closed, application-specific networks. There are very different expectations on packet parsers. For example, consider a multi-million dollar business of low-latency algorithmic trading. In this area, the *latency*, which has long been rather neglected parameter, suddenly becomes more important than the raw throughput. Small embedded devices, on the other hand, often require a parser to be very small (in terms of the memory and chip area), yet still to support a rather extensive set of protocols.

With a rising interest in the Software Defined Networking, it is expected that the ”ossification” of networks will be on decline, and new protocols will appear at even faster rate than before. This expectation handicaps fixed ASIC parsers and favours programmable solutions: CPUs, NPUs and FPGAs. Our work focuses on FPGAs, because of their great potential in high-speed networks.

Current high-speed FPGA-based parsers can achieve a raw throughput of over 400 Gb/s at the cost of the extreme pipelining, which increases both the latency and the chip area (FPGA resources) significantly [1]. Also, the configurability issue is solved only partially. Configuring a set of supported protocols is often addressed by a higher-level protocol description followed by an automatic code generation, but the configuration of the implementation details is left unnoticed.

This paper not only presents a novel packet parser design, but also motivates engineers to create a parametrized solutions, demonstrates the need for a thorough exploration of the space of the solutions and suggests several capabilities that a High-Level Synthesis system should possess to succeed in this area.

The paper is organized as follows: Section II introduces several prior published works in this area, Section III describes our implementation of a modular parser design, Section IV lists all the necessary steps to create own parser in our methodology, Section V presents obtained results and Section VI concludes the work.

II. Related Work

Rather outdated work by Braun et al. [2] uses the onion-like structure of hand-written protocol wrappers to parse packets. However, due to the 32-bits-wide data path and an old FPGA, the parser achieves a throughput of only 2.6 Gb/s. There is no extensive concept of a common interface for module reuse, and it is unclear how the parser scales for a wider data path.

Kobierský et al. [3] describe the packet headers in XML and generate finite state machines, which parse the described protocol stack. However, the number of states in FSMs rises rapidly with the width of the data bus. Also, the crossbar used in the field extraction unit does not scale well.

While not directly related to our work, there has been an extensive research of general High-Level Synthesis systems, usually translating pure or modified imperative languages (such as C, C++, Java) into the hardware. Most of this research aims to find a potential parallelism hidden in the program loops and to make use of it by unrolling the loops and pipelining the computation. However, the *for* or *while* cycle is far from a convenient description of a packet parser, whose most natural model of computation is perhaps a directed graph of mutually dependent memory accesses. That may be the reason why we do not see many results of a general HLS in this area.

There is a general HLS result that was given by Dedek et al. in [4]. Handel-C language is used to describe the design, but the details are not disclosed. The reported speed of 1 454 Mb/s implies that a rather narrow data bus (probably 16 bit) was used. Therefore, the concern is about the scalability in terms of both an effective description in Handel-C and an effective compilation to hardware for much wider data words. This work also demonstrates that using processors for the packet parsing

gives poor results. Compared to the Handel-C implementation, a custom RISC processor designed specifically for the packet parsing yields roughly the same chip area, but achieves only a half of the throughput. Using the MicroBlaze [5] processor (which is not optimized for the packet parsing) requires double resources and brings only 5.7 % throughput compared to the Handel-C solution.

A good example of a domain-specific HLS was given by Attig and Brebner in [1]. They utilize their own Packet Parsing (PP) language to describe (with the syntactic sugar of an object orientation) the structure of packet headers and the methods which define parsing rules. The description is then compiled from PP to the pipeline stages implementation. However, the results indicate that the price for a convenient design entry is the chip area and the latency – most parsers with 1024-bit datapath use over 10 % of the resource-abundant Xilinx Virtex-7 870HT FPGA [6] and the latency varies from 292 to 540 ns.

The Kangaroo system [7] uses RAM to store the packets and employs the on-chip CAM to perform a lookahead. Lookahead is the process of loading several fields from the packet memory at once, allowing to parse several packet headers in a single cycle. The dynamic programming algorithm is used to precompute data structures, so that the parsing of the longest paths in a parse tree is the most accelerated by the lookahead, as it is impractical to perform the lookahead for all the possible protocol combinations. This approach has the architectural limitation of storing the packets in the memory and accessing them afterwards. The memory soon becomes a bottleneck. Our approach, however, parses packets "on the fly", which means that the only packet data storage are the pipeline registers.

III. Modular Parser Design

A. Input Packet Interface

We start with the design of an input packet interface, which conveys packets into (and through) the parser. While the interface design may seem trivial, it becomes very important for high bandwidth applications. This is due to the fact that FPGAs achieve rather low frequency, roughly between 100-400 MHz. To support the bandwidth over 100 Gb/s, we must use a very wide data bus (up to 2048 bits). Since the shortest Ethernet frame is 64 Bytes (512 bits), packet aliasing and aligning become an issue. Therefore, the achievable effective bandwidth is considerably smaller than the theoretical raw bandwidth.

We propose and our packet convey protocol uses two techniques to utilize the raw bandwidth more effectively than the standard approach:

- **Partially aligned start.** The first packet byte may appear at any position aligned to eight bytes. This corresponds to the 40 and 100 Gb/s Ethernet standard. For a data bus wider than 64 bits (8 bytes), this technique allows the packet to start at other positions than the first byte of a data bus word.
- **Shared words.** One data word may contain the last bytes of the packet x and the first bytes of the packet $x+1$. The packets may not overlap within the word and the partially aligned start condition may not be violated.

Examples of both aforementioned techniques are shown in the Fig. 1. Using these techniques we bring the effective throughput for the usual packet length distribution much closer to the theoretical limit.

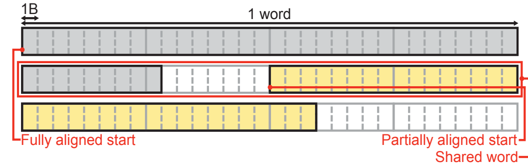

Fig. 1. The example of possible packet positions when using the proposed techniques for the better raw bandwidth utilization.

B. Parser Submodules

Since we realize that the development of VHDL modules is rather low-level and often very time-consuming, we continue with the design of *Generic Protocol Parser Interface* (GPPI). This interface provides the input information necessary to parse a single protocol header: (1) current packet data being transferred at the data bus, (2) current offset of the packet data and (3) offset of the protocol header. GPPI output information includes (4) extracted packet header field values and the information needed to parse the next protocol header: (5) offset and (6) type of the next protocol header. Fig. 2 shows how the modules are connected. By manually adhering to GPPI, we achieve a hint of object orientation in VHDL – all protocol header parsers use the same interface (except for the extracted header fields) and therefore can easily be interchanged if needed. This improves the code maintainability and enables the easy extensibility of the parser: any new protocol header parser is connected just in the same way as the others. This feature also allows an automatic connection of protocol header parsers from the high-level structure description.

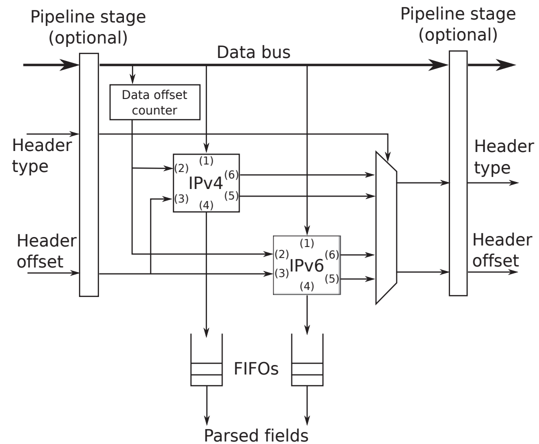

Fig. 2. Example of one pipeline stage.

The inner implementation of each protocol header parser is protocol-specific, but the basic parser block `getbyte` remains the same. This block performs waiting for a specific header field to appear at the data bus, i.e. $po + fo \in \langle do; do + dw \rangle$,

where po is the protocol header offset (module input), fo is the field offset (from the protocol specification), do is the data bus offset (module input), and dw is the data bus width. Once the header field is observed at the data bus, it is stored and can be used to compute the length of the current header, decode the type of the next header, or any other operation. Fig. 3 shows the structure of an IPv4 parser as an example.

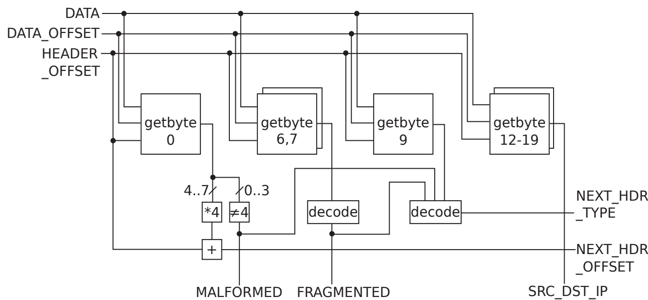

Fig. 3. Example of IPv4 protocol parser.

C. Parser Top Level

Our parser can output the information about types and offsets of protocol headers. This information is more general than just having the parsed header field values. Obtaining the header field values can be done later, externally to the parser. Our parser offers an option to skip the actual multiplexing of header field values from the data stream. This may save considerable amount of logic resources and is particularly useful for applications that read only a small number of header fields, or when packets are modified in a write-only manner.

Similarly to [1], our parser also uses pipelining to achieve high throughput. However, every pipeline step in our design is optional. If many pipelines are enabled, then the frequency (and the throughput) rises, but also the latency and used logic resources increase. By tuning the use of pipelines, designer can find the optimal parameters for the particular use case.

Each protocol parser contains an inner bypass for situations when its protocol is not present in a packet (not shown in Fig. 3). Thanks to this bypass, the protocol parser submodules can be arranged in a simple pipeline with a constant latency. This property also makes adding a support for new protocols into the parser stack very easy, without the requirement for any changes in the existing protocol parsers. Fig. 4 shows the example top level structure of the parser. Note that the inner bypasses allow to skip certain protocol headers (e.g. VLAN, MPLS), if these are not present in the packet.

The data width required for high throughput (over 100 Gb/s) may be 1024 or even more bits. This implies that there may be more packets in one data word. Our parser is able to handle such situation, provided that no two packet headers of the same type from different packets are present in one data word. For example, if the data word contains the IPv4 header (and the following bytes) of the packet A, and a part of the packet B that includes the IPv4 header, then the packet B is delayed by one cycle in our parser. This situation may only occur only for wide data buses (512 bits and more), and short packets (close to minimal length of 64 bytes) with very short inter-packet gap. Our measurements of the real high-speed networks show that it is very rare situation.

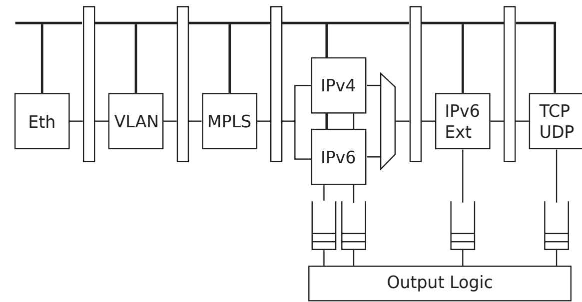

Fig. 4. Example of the parser top level structure.

IV. Creating the Parser According to the Application Requirements

With the description of the parser design, it is rather straightforward to create own, customized parser. We identify three basic steps:

- Parser submodules implementation
- Parser top level connection
- Parser state space search

Parser submodules implementation comprises manual writing of the VHDL code for each supported protocol. However, GPPI enables easy reuse of the submodules – once written, the protocol parser submodule can always be reused. Also, many generic building blocks of the submodules are already available, for example the `getbyte` module, which extracts a single byte at a certain offset from the packet. In general, the parser submodules for the common protocols are very similar and follow the same informal code template. For many today's protocols the parser submodule is only the `getbyte` modules with correctly configured offsets and some protocol-specific logic to compute the information about the next protocol from the extracted fields. Therefore, one can easily create a parser submodule implementation from the protocol structure specification.

There is also a space here for manual optimizations. For example, extracting a byte from the data bus using the getbyte normally requires a full multiplexer, which is able to extract a byte from any byte position in the data word (Fig. 5a). The multiplexer is controlled by the current data bus offset, the offset of a header within the packet, and the offset of the desired field within the header (which is often constant). However, given the fact that a packet may start only at certain positions in the data word, the current data offset may contain only the values with the corresponding resolution. Also, we can often derive all the possible offsets of the packet header from the analysis of all the possible orderings and sizes of the protocol headers appearing in the packet *before* the current protocol header. Combining the three values (data offset resolution, possible header offsets, constant field offset) together, we can use simpler multiplexers, which do not allow to extract fields from impossible positions. The use of simpler multiplexers in getbyte, together with the fact that getbyte

modules form the main core of the protocol analyzing and data extracting, result in significant chip area savings. For example in a classical TCP/IP protocol stack, header lengths are multiples of 4. Therefore, the size of multiplexers can be reduced 4 times and the size of the whole parsing logic by nearly the same amount. This is illustrated in the Fig. 5.

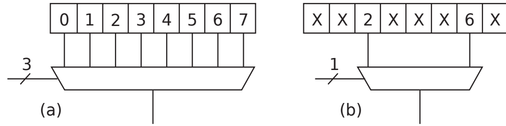

Fig. 5. Example of 64b getbyte multiplexer: full (a) and optimized (b).

Parser top level connection once again requires the designer to write VHDL. In this case, the protocol submodules are connected via GPPI pipelines to the structure corresponding to the expected order of the protocol headers in packets. Extracted header field values can be stored in output FIFOs.

Parser state space search is the final step. It takes into account other parser requirements than a set of supported protocols. The state space is created by the selective bypassing of pipelines and by setting the data width of the packet convey protocol (all easily set by generic parameters).

For example, there is often a requirement on the throughput. In that case, we are looking for a parser with throughput equal or higher than the requirement. By synthetizing a parser with all the possible settings and ruling out those which do not satisfy the throughput requirement, we obtain a set of satisfying solutions. However, the solutions will differ in the size of chip area and in latency. From this set we select a Pareto set, which contains only the dominating solutions (those for which there is no better solution in both chip area and latency). If the Pareto set has more than one member solution, we have to decide which parameter (area or latency) is more important for our application.

Generally, each candidate solution creates one point in the 3-D space with dimensions throughput, area and latency. Each pipeline step and each data width option double this space, possibly ending in a situation when the exhaustive search is no longer possible, taking into account that a single synthesis run takes time in the order of minutes. In that case we suppose that some global optimization algorithm, such as simulated annealing or a genetic algorithm can be used. Good heuristic helping these algorithms could be to rule out some of the pipeline positions, more precisely to place the pipelines evenly in the parser to create evenly long critical paths.

A. Implications for High Level Synthesis

After identifying the steps needed to be performed manually, we can now provide a list of features desirable for a good HLS, general or platform specific:

- Way to describe parser interface and protocols.
- Way to specify header formats and their dependency.
- Automatic inference of logical constraints (for multiplexer simplification etc.).
- Generator of parametrized code.
- Way to describe the design goals (area, latency etc.).
- The best fitting solution finder (exhaustive/heuristic).

Note that these requirements do not imply any particular *type* of a parser. Such HLS may generate pipelined parsers similar to ours, or the parsers based on a completely different paradigm (e.g. FSM or processor+code).

V. Results

We have implemented a parser 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. (see Fig. 6). The parser is able to extract the classical quintuple: IP addresses, protocol, port numbers. Apart from that, it can also provide the information about present protocol headers and their offsets including the payload offset.

We have tested properties of the designed parser with 3 different protocol stacks:

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

For each mentioned protocol stack, one test case is done for the parser with the logic to extract the classical quintuple and one for the same parser without the extraction logic (providing only the offsets).

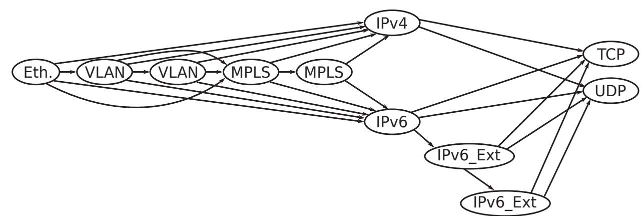

Fig. 6. Structure of supported protocols (full protocol stack).

We provide the results after a synthesis for the Xilinx Virtex-7 870HT FPGA, 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 a large space of solutions, in which the Pareto set can be found and used to pick the best-fitting solution for an application. In each test case, we use 5 different data widths: numbers from 128 to 2 048 bits that are powers of 2. For each data width, every possible placement of pipelines for the tested protocol stack is shown as a point in the graph and the Pareto set is highlighted. Points representing results for each data width are shown in different shapes and colors.

For each test case we provide 2 graphs: the first one shows the relation between throughput and FPGA resources with the Pareto set highlighted, without any regard to latency. The second graph shows the relation between throughput and latency with the Pareto set highlighted, without any regard

to FPGA resources. In the graph with the relation between throughput and FPGA resources, the second Pareto set (the lower, dashed curve) is also shown. This Pareto set shows the best achievable solutions for our parser without the quintuple extraction logic. Similar Pareto set is not shown in the graph with the relation between throughput and latency, because the usage of the quintuple extraction logic affects the latency of the parser only slightly (the critical paths are mostly in the next header computation logic).

Fig. 7 shows the throughput and the FPGA resources and Fig. 8 shows the throughput and the latency for the full protocol stack. There are 9 configurable pipeline positions in the parser implementing the full protocol stack. This leads to 512 different possible placements of pipelines in this parser for each data width. Mentioned graphs therefore show results for 2 560 different solutions with the Pareto sets highlighted.

For a comparison of the achieved Pareto set results for different protocol stacks, we provide graphs in Fig. 9 (throughput and FPGA resources) and the Fig. 10 (throughput and latency). From these figures one can clearly see that the supported protocol stack can rapidly change the parameters of the parser in terms of chip area and latency. Therefore, a careful protocol support selection is very important for the optimal result. For example, just by turning off the IPv6 support we can bring down the resource utilization by almost 50 %. Latency, on the other hand, is sensitive to the depth of the protocol stack, (see Fig. 6) therefore turning off the support for the VLAN and MPLS headers lowers the latency significantly.

A closer look at the Pareto set optimized for latency and throughput (without regard to FPGA resources) from Fig. 8 is presented in Tab. I. The last line of the table is the estimation of the parser from [1] with similar configuration of the supported protocols (TcpIP4andIP6). It is obvious that our parser can achieve much better parameters than the parser from [1].

Data Width	Pipes	Throughput [Gb/s]	Latency [ns]	LUT-FF pairs
256	0	14.5	17.1	3 238
512	0	28.4	18.0	4 053
2 048	0	96.9	21.1	17 685
2 048	1	158.5	25.9	18 547
2 048	2	212.8	28.9	18 317
2 048	4	333.0	30.8	21 775
2 048	5	352.0	34.9	22 373
2 048	7	453.0	36.2	26 728
2 048	8	478.1	38.6	29 301
1 024	?	325	309	67 902

TABLE I. PARETO SET FOR THE BEST THROUGHPUT AND LATENCY OF THE FULL PROTOCOL STACK PARSER

Next, we provide the data for the example from the Section IV: Given a set of supported protocols and the target throughput, find all solutions in the Pareto set. We use three sets of supported protocols mentioned earlier and the target throughputs of 40, 100 and 400 Gb/s. All nine Pareto sets are shown in the Fig. 11. Note that while there are several solutions with the throughput over 400 Gb/s, there is only one 400 Gb/s Pareto solution for each protocol set, which means that the other solutions are not better in terms of FPGA resources nor latency. For the other target throughputs, the designer can choose the appropriate solution according to application priorities.

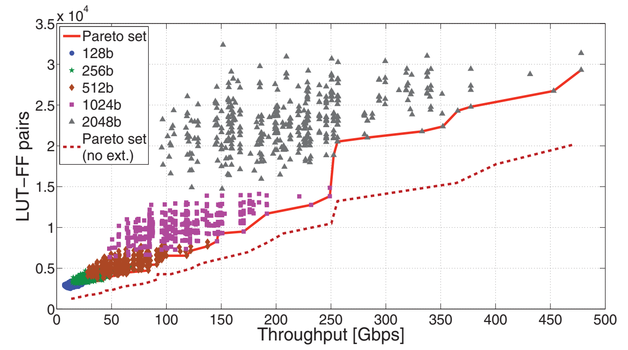

Fig. 7. The FPGA resource utilization for different settings of the full parser.

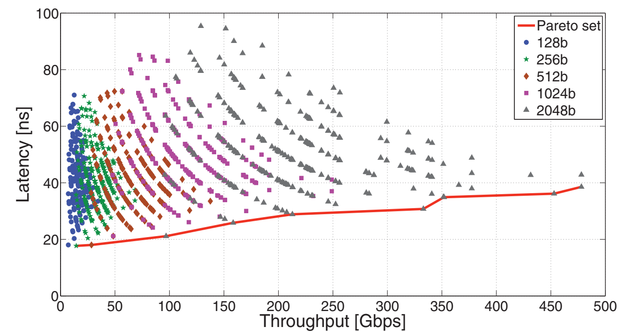

Fig. 8. The latency for different settings of the full parser.

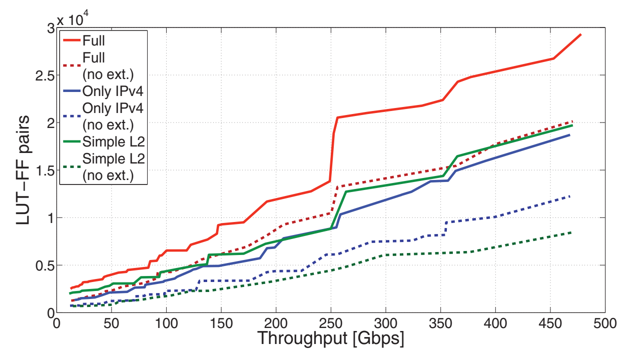

Fig. 9. Comparison of the FPGA resource utilization versus throughput Pareto sets for the tested protocol stacks.

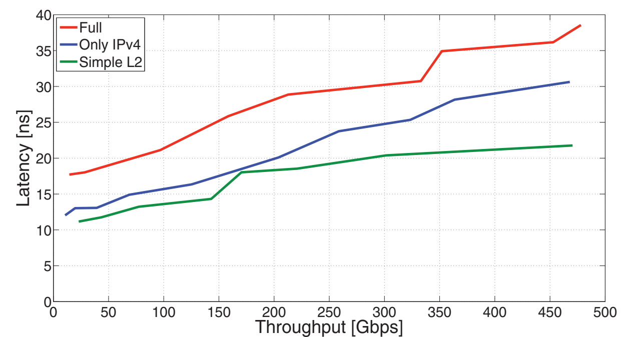

Fig. 10. Comparison of the latency versus throughput Pareto sets for the tested protocol stacks.

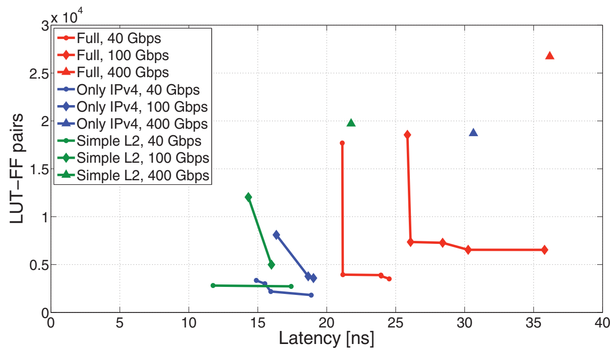

Fig. 11. Pareto sets for three given protocol sets and three target throughputs.

A careful design space exploration is very important for our parser. For example, the parser of the full protocol stack optimized for the latency uses 17 685 LUT-FlipFlop pairs to achieve near 100 Gb/s throughput with the latency of only 21.1 ns (see Tab. I), while the parser optimized for resources uses only 6 536 LUT-FlipFlop pairs to achieve the throughput just over 100 Gb/s, but with the latency of 35.8 ns (see Fig. 11).

Finally, Fig. 12 illustrates the complete Pareto set of solutions in the (latency, throughput, area) space for the full protocol stack. To create the 3D surface in the figure, the bottom (latency, throughput) plane was divided into rectangles of sizes (1 ns $\times$ 10 Gb/s) and the smallest solution that satisfies the required latency and throughput was found for each rectangle. Therefore, each horizontal level of the surface represents one solution from the Pareto set. Finer-grained division of the (latency, throughput) plane would result in more solutions, but also in less readable image.

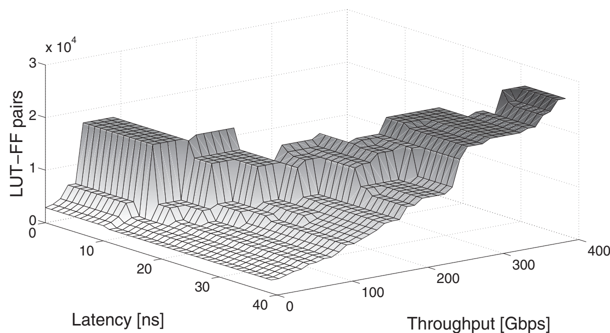

Fig. 12. 3D surface plot of the Pareto set

VI. Conclusion

This paper introduces a highly configurable packet parser for FPGA, which achieves throughput in the range of tens to hundreds of Gb/s and is usable in a variety of applications. The key concept is a selective pipelining, which allows to find the best fitting solution with regards to the requirements. The parser uses only 1.19 % of the Virtex-7 870HT FPGA resources to achieve a throughput over 100 Gb/s and 4.88 % for a throughput over 400 Gb/s, which leaves most of the FPGA resources free for implementing other functions of target applications.

This work also presents the methodology of a modular parser design and demonstrates the need for a thorough exploration of the solution space. Moreover, it suggests several capabilities that a High-Level Synthesis system should possess to succeed in area of packet parsers creation.

Acknowledgement

This research has been supported by the “CESNET Large Infrastructure” project no. LM2010005 funded by the Ministry of Education, Youth and Sports of the Czech Republic, the “DMON100” project no. TA03010561 funded by the Technology Agency of the Czech Republic, BUT project FIT-S-14-2297 and the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070.

References

[1] M. Attig and G. Brebner, “400 gb/s programmable packet parsing on a single fpga,” in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, oct. 2011, pp. 12–23.

[2] F. Braun, J. Lockwood, and M. Waldvogel, “Protocol wrappers for layered network packet processing in reconfigurable hardware,” *Micro, IEEE*, vol. 22, no. 1, pp. 66–74, 2002.

[3] P. Kobierský, J. Kořenek, and L. Polčák, “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*, ser. DDECS. Washington, USA: IEEE Computer Society, 2009, pp. 96–101.

[4] T. Dedek, T. Martínek, and T. Marek, “High level abstraction language as an alternative to embedded processors for internet packet processing in fpga,” in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, aug. 2007, pp. 648–651.

[5] “Xilinx microblaze soft processor,” Xilinx, Inc., http://www.xilinx.com/tools/microblaze.htm.

[6] “Xilinx virtex–7 fpga family,” Xilinx, Inc., http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.

[7] C. Kozanitis, J. Huber, S. Singh, and G. Varghese, “Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system,” in *IEEE INFOCOM*, mar. 2010.