Low-Latency Modular Packet Header Parser for FPGA

Viktor Puš, Lukáš Kekely
CESNET a. l. e.
Zikova 4, 160 00 Prague, Czech Republic
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
korenek@fit.vutbr.cz

ABSTRACT

Packet parsing is the basic operation performed at all points of the network infrastructure. Modern networks impose challenging requirements on the performance and configurability of packet parsing modules, however the high-speed parsers often use very large chip area. We propose novel architecture of pipelined packet parser, which in addition to high throughput (over 100 Gb/s) offers also low latency. Moreover, the latency to throughput ratio can be finely tuned to fit the particular application. The parser is hand-optimized thanks to the direct implementation in VHDL, yet the structure is very uniform and easily extensible for new protocols.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays, Algorithms implemented in hardware*

Keywords

Packet Parsing, Latency, FPGA

1. INTRODUCTION

As the computer networks evolve both in terms of speed and diversity of protocols, there is still 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 may be different expectations on packet parsers. Consider for example the multi-million dollar business of fast algorithmic trading, where the parsing latency is often more important than the raw throughput.

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

2. MODULAR PARSER DESIGN

Our modular packet parser is designed to meet the demands rising from various applications. Since we realize that the development of VHDL modules is very low-level and often slow, we start with the design of *Generic Protocol Parser Interface* (GPPI). This interface provides the input information necessary to parse a single protocol header: current data being transferred at the data bus, offset of these data, offset of the protocol header. GPPI also contains output information needed to parse the next protocol header or to read or modify packet header fields. The GPPI output information includes the type and offset of the next protocol header. Fig. 1 shows how modules are connected. By manually adhering to the GPPI, we achieve a hint of object orientation in VHDL – all protocol header parsers are derived from the same "class". This improves the code maintainability and enable easy extensibility of the parser: new protocol header parser is connected just in the same way as the others.

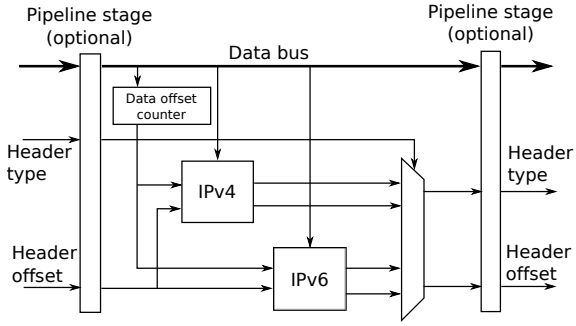

Figure 1: Example of one pipeline stage. Protocol header parsers share inputs, their outputs are selected based on the input type of protocol.

The inner implementation of each protocol header parser is protocol-specific, but the basic parser block is usually the 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 protocol offset (module input), fo is field offset (from protocol specification), do is data bus offset (module input), and dw is data bus width. Once the header field is observed at the data bus, it can be used to compute the length of current header, decode the type of the next header, or any other operation.

The output information about types and offsets of protocol headers is more general than having already parsed header field values: The offset is needed for packet editing, and obtaining the header field values can be done simply by multiplexers, with the knowledge of offsets. Our parser offers the option to skip the actual multiplexing of header

ANCS'12, October 29–30, 2012, Austin, Texas, USA.
ACM 978-1-4503-1685-9/12/10.

field values from the data stream. This may save considerable amount of logic resources and is particularly useful for applications where only small number of header fields must be read, or when the 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 throughput) rises, but also latency and logic resources increase. By tuning the use of pipelines, designer can find the optimal parameters for the particular use case.

3. RESULTS

We have implemented the parser supporting the following protocol stack: Ethernet, up to two MPLS headers, up to two VLAN headers, IPv4 or IPv6 (with up to two extension headers), TCP or UDP. The header field extraction module (if present) extracts the classical 5-tuple: IP addresses, protocol, TCP or UDP ports. We provide results after synthesis for Xilinx Virtex-7 870HT FPGA, with the different settings of data width, number of pipeline stages and the use of extraction module.

These settings, together with the resulting frequency, latency and resource usage generate a large space of solutions, where the Pareto set can be found and used to pick the best-fitting solution for the application. Fig. 2 shows the throughput and FPGA resources for data widths from 128 to 2 048 bits. For each data width, each possible placement of pipelines is shown as a point in the graph and the Pareto set (finding the best throughput and resource utilization, without regard to latency) is highlighted. The lower curve is the Pareto set for the parser without the extraction module. Tab. 1 shows the Pareto set optimized for latency and throughput, without regard to chip area. The last line of Tab. 1 is the estimation of parser from [1] with similar configuration of supported protocols (TcpIP4andIP6).

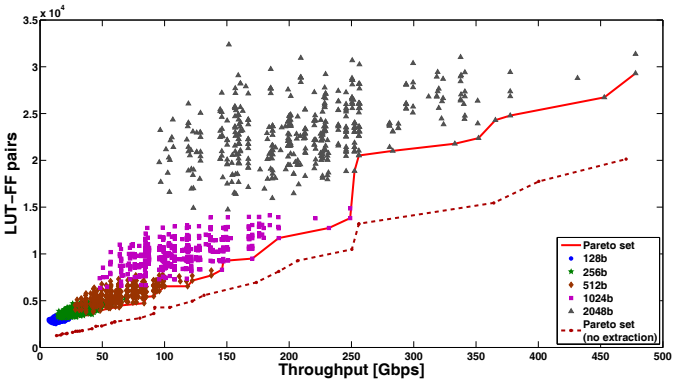

Figure 2: FPGA resource utilization for different parser settings.

Careful design space exploration is very important for our parser. For example, parser optimized for latency uses 17 685 LUT-FlipFlop pairs to achieve near 100 Gb/s throughput with latency only 21.1 ns (see Tab. 1), while parser optimized for chip area uses only 6 536 LUT-FF pairs to achieve throughput just over 100 Gb/s (but with the latency of 35.8 ns).

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 1: Pareto set for best throughput and latency

4. CONCLUSION

Our low-latency modular packet parser for FPGA uses only 1.19 % of the Virtex-7 870HT FPGA to achieve throughput over 100 Gb/s and 4.88 % for throughput over 400 Gb/s with reasonable latency, while most of the FPGA logic is kept for application. These results, together with the latency below 40 ns are better than previous solutions (compare to FPGA utilization over 10 % and latency over 300 ns at 300+ Gb/s throuput in [1]). Our parser uses 68 % less FPGA resources and has 90 % smaller latency than [1] for throughput over 300 Gb/s. Moreover, the parser can be finely tuned trough design space exploration to meet the demands of the particular application.

Acknowledgment

This research has been partially supported by the "CESNET Large Infrastructure" project no. LM2010005 funded by the Ministry of Education, Youth and Sports of the Czech Republic, the research programme MSM 0021630528, the grant BUT FIT-S-11-1 and the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070.

5. 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*, pages 12 –23, oct. 2011.