Automated synthesis of FPGA-based packet filters for 100 Gbps network monitoring applications

Jose Fernando Zazo[*], Sergio Lopez-Buedo[*†], Gustavo Sutter[†], Javier Aracil[*†]
[*]NAUDIT HPCN
Calle Faraday 7, 28049 Madrid, Spain
[†]High-Performance Computing and Networking Research Group, Universidad Autonoma de Madrid
Ciudad Universitaria de Cantoblanco, 28049 Madrid, Spain

***Abstract*—Monitoring 100 Gbps network links is a challenging task. Packet filtering allows monitoring applications to focus on the relevant data, discarding packets that do not provide any valuable information. However, such a large line rate calls for custom hardware solutions. This work presents a tool for automatically synthesizing packets filters from a custom grammar, which defines filters in a human-readable format. Thanks to parser generators (Bison) and lexical analyzers (Flex), Verilog code is automatically generated from the filter specification. Rules can be applied over a protocol, a protocol field, the packet payload, or a combination of them. The generated filters use standard AXI4-Stream interfaces, which seamlessly integrate in the packet filtering framework that we have developed for the integrated block for 100G Ethernet available in Xilinx Ultrascale devices. We present the results for two proof-of-concept packet filtering designs. Furthermore, filters are fully pipelined, so the full 100 Gb/s rate is guaranteed. As the framework uses a cut-through approach, latency is kept to a minimum. Finally, the proposed framework allows for the integration of more complex payload-level filters, written in C language with the Vivado-HLS tool.**

I. Introduction

Traffic monitoring is an essential activity for the operation and management of network infrastructure. Moreover, it also allows to assess the quality of service and security of communications. However, the amount of data carried by actual backbone networks is humongous, which makes it very difficult, if not impossible at all, to carry out traffic monitoring using pure software solutions.

The benefits of packet filtering for traffic monitoring have long been known. The objective is to minimize the computational load of the monitoring application by cutting down the number of packets that it needs to process. Berkeley Packet Filter (BPF) is the most popular solution, which has became the de facto standard for “filter machine”. BPF is based on a virtual machine that runs in software. This virtual machine executes custom instructions specially tailored for packet processing. However, it is good only for relatively low-rate links, but not for multi-Gbps links. As a reference figure, in a 100 Gb/s Ethernet link, up to 148809524 packets can arrive per second, that is, one packet every 6.72 ns. Such stringent timescales call for hardware-based solutions.

Porting BPF to FPGA is not a trivial task. For example, in an Ethernet link with two nested VLAN tags, a basic filter that checks the IP source address of a packet is compiled into 31 instructions of the BPF virtual machine, with a critical path of 23 nodes (libpcap BPF compiler 1.5.3). If the clock frequency is 322 MHz (as in the 100 GbE MAC of Xilinx UltraScale devices), the time needed to execute the filter is $23 * 1/322$ MHz $= 71.4$ ns. That is, one order of magnitude higher than the minimum packet time in 100 Gb/s Ethernet. The problem significantly worsens if more complex filtering expressions are being used, or if more nested tags occur. A straightforward solution is to implement multiple BPF processors in parallel, but this is challenging in terms of timing and very costly in terms of FPGA resource utilization.

In this paper we present an alternative architecture for packet filters, completely different from the virtual machine used by BPF. What we propose here is to build filters by chaining blocks that perform different tests to the contents of the packet. Each block performs a certain test and takes one clock cycle to execute. This chain of testing blocks is pipelined. That is, the filter accepts a new data unit every clock cycle, thus guaranteeing that maximum data rate is satisfied. At the end of the chain, the results of the different tests are evaluated, and a decision is made in order to accept or drop the packet.

Although pipelining is a well-known technique for enhancing the performance of hardware implementations, it is not free of difficulties. The problem of custom, pipelined designs of packet filters is that they are very costly to implement if using a conventional approach based on HDL. This is the reason why we developed an alternate methodology, based on the definition of a custom grammar specifically targeted to the design of packet filters. We have developed a compiler that is able to translate code written according to that grammar into a Verilog model of the packet filter. The generated filter follows the proposed architecture, based on a chain of testing blocks, and it uses standard AXI-Stream interfaces.

As proof-of-concept, we present two example designs that integrate a filter generated with our tool with the integrated block for 100G Ethernet of Xilinx UltraScale devices. Such designs demonstrate feasibility of packet filtering in a fully loaded 100 Gb/s link, with rules for protocol, IP address and port. Latency is kept below 20 ns, and the synthesized filters only use a mere 2% of the selected FPGA.

978-1-5090-3707-0/16/$31.00 ©2016 IEEE

II. Related Work

In short, packet filtering facilities apply a predefined set of rules to incoming packets, and use the outcome of these rules in order to take a decision whether to process the packet or not. Packet filtering has a long history: One of the earliest software approaches was the CMU/Stanford Packet Filter (CSPF) [1]. CSPF uses a tree-based model, where each leaf is a condition and the parent nodes are the boolean operators that link conditions in order to construct complex predicates. A well-known problem of tree models is that they cannot represent inter-dependencies among predicates. If a predicate contains the same condition in two different leaves, the condition needs to be evaluated twice.

In 1992, McCanne and Van Jacobson proposed the Berkeley Packet Filter (BPF), which used control flow graphs (CFGs) instead of tree-based models. Rules are modeled as a directed graph, which has two final states: accept or discard packet. Graphs are acyclic, that is, backward branches are not allowed. The advantage of CGFs is that they avoid the problem of evaluating twice the same condition if there are interdependencies in the predicate. BPF filters are implemented in software, but not using native instructions. Filters run in a virtual machine that executes custom instructions tailored towards packet filtering (BPF bytecodes).

In order to reduce the latency of BPF filters, Begel et al. developed BPF+ [2], a JIT (Just In Time) compiler for BPF bytecodes. Another two remarkable developments aimed at increasing the performance of BPF are xPF [3] and Fairly Fast Packet Filters (FFPF) [4]. In xPF, the effort was directed towards minimizing the context switching overhead, whilst FFPF focused in reducing the filter update latency in order to achieve real-time changes.

A more recent approach is to represent filters as FSMs (finite state machines), as pFSA [5] and SPAF [6] do. Both works use a similar approach, which maps each state of the FSM to a network protocol. Transitions between states are triggered by the detection of protocol identifiers (e.g. detection of the Ethertype in an Ethernet network) and the acceptance of a rule is derived from the evaluation of the predicates associated to a particular state. The optimization of the FSM is a concern for both approaches, because the lower the number of states, the lower the number of instructions that the CPU needs to execute.

In order to overcome the limitations of software-based solutions, several works have dealt with the implementation of packet filters in reconfigurable hardware. As opposed to what happens in software, here the tree-based model is the preferred choice, because it is very easy to pipeline and fits specially well the architecture of an FPGA. Gibb et al. [7] present a set of recommendations for the implementation of packet parsers on reconfigurable hardware. Continuing with FPGA implementations, a custom grammar is presented in [8] for the identification of packets and the generation of an unique key (as the flow tuple), but it lacks any support for packet filtering.

The demand for more complex operations is being tackled by the state-of-the-art language P4 [9]. P4 aims to be a generic solution for switching platforms, providing a specification that is not limited to a particular device but is portable to different architectures (CPUs, GPUs, FPGAs, etc.). For instance, Bencek et al. [10] developed a P4-based packet parsing pipeline for FPGAs that achieves a sustained rate of 100 Gb/s. However, this work does not consider traffic filtering and the implementation is only capable of parsing one protocol per clock cycle. If all the same-layer protocols could be parsed at the same clock cycle, the latency would be significantly improved.

P4 can, theoretically, be used to write packet filters. This affirmation is based on the fact that P4 defines 3 different stages: Input parsing, match+action on the ingress port and, finally, an additional match+action on the egress port. The match+action step consists on looking up of the information extracted from the packet in a set of tables. If a match occurs, it triggers a certain action (modification of the origin/destination addresses, decrease of the time-to-live field in IP, recomputation of the CRCs of the packet, etc.). In a packet filter, the action would be to discard the packet. In spite of the portability of the P4 specification, the 3-step model may increase the latency. Therefore, we propose a different approach that does not aim at modifying the contents of a packet, just at filtering. Following this approach, parsing and filtering are not different steps, but actions executed in parallel in order to minimize latency. Additionally, our grammar also supports for nested network protocols. This feature is not yet supported by P4.

III. Design overview

Using a top-down scheme, this section begins by a classification of filtering rules used in real-world scenarios, followed by the definition of the grammar and how the network stack is modeled. Finally, the features of the generated Verilog code and the hardware architecture are discussed and summarized.

A. Packet filtering rules

According to the element to which they apply, the following categories of filtering have been considered:

1) Rules that apply over a protocol: The most basic kind of rule, checking if a packet belongs to a certain protocol. An example of this type of filters could be the expression “arp”. That is, all ARP packets will match this rule.

2) Rules that apply over a field of a protocol: Filtering by destination port or source address is one of the most typical actions carried out by firewalls. For example, the expression “IPv4.ip_dst==10.0.0.1” defines a rule that checks if the IP destination address is 10.0.0.1.

3) Rules that apply over the payload: Such rules are used in DPI (deep packet inspection) applications, that is, higher-layer filtering. They are expressed as a function that is applied over the content of the paper. For instance, a useful use case is discarding ciphered traffic, which is meaningless for monitoring applications. If entropy is used to detect ciphered data, the expression *“entropy(payload)”* will be used for the rule.

4) Combination of rules: These are boolean operations merging two or more expressions of the previous categories.

Rules in the group *3)* are not directly synthesized by the compiling tool. Instead, the generated design provides mechanisms for accessing the payload of the packets. Additional modules can be developed by users for inspecting the payload. Such modules do not necessarily need to be written using HDLs, High Level Synthesis (HLS) frameworks can be used in order to reduce development time and ease debugging.

B. Grammar definition

The most elementary slice of the corpus is a network protocol definition. When representing a protocol in the grammar there are some peculiarities that must be considered, which force the user to provide more details than just a list of fields and their widths. For instance, headers may present either a fixed or a dynamic number of fields. It may also happen that the next protocol in the subsequent layer might be identified by a magic word in the header. This fact motivates that the definition is slightly more complex than just a list of expressions. So, the essential element that integrates the corpus is a list of protocols, which includes a metadata block providing further knowledge about protocol interconnections.

```
VLAN {
        format {
                priority : 3;
                cfi: 1;
                id : 12;
                ethertype : 16;
        }
        metadata {
                protocol_id : 16'h8100;
                header_size: 32;
                level: 1;
                recursivity : 2;
                next_protocol : ethertype;
                key : {id};
        }
        filter {
                filter1: id#1==12'h1234;
                filter2: id#2==12'h5678;
        }
}
```

Listing 1: Representation of VLAN

1) Protocol definition: Each protocol that is going to be processed must be defined in advance at the corpus, providing at least two blocks of information: *format* and *metadata* structure. The first one consists of several identifiers (string of characters) and the associated width, expressed in bits. Using bits as the reference unit instead of bytes presents several advantages in protocols where not all the fields are byte aligned (this is the case of *version* and *header length* in IP). The *metadata* block provides additional information that the compiler requires in order to understand interconnections among protocols. Listing 1 shows a real configuration example for defining the VLAN header.

The fields that compose "*metadata*" are divided into:

- Detection of the current and the next protocol. Network protocols are stacked and the identification of the subsequent layer is generally carried out by a dedicated field in the header. In the case of an VLAN header, the 16b "*ethertype*" field informs what protocol is encapsulated on top of VLAN (e.g. 0x0800 for IPv4). However, this statement remains valid if and only if the VLAN header is detected, which can be done by examining the identifier reported by the previous protocol and verifying that it is equal to the value assigned to VLAN (0x8100).

Within the grammar, this case is handled by inspecting the fields "*protocol_id*" (immediate value) and "*next_protocol*" (immediate value/reference to field) in the metadata block. The "*protocol_id*" provides the constant that unequivocally defines the current protocol; "*next_protocol*" the identifier that is going to be passed to the following protocol parsing unit.

- Header size (immediate value/reference to field and/or arithmetic operations over fields and immediate values). Some protocols have the possibility of adding optional entries in the header, which entails that the payload offset is variable. This field helps the code generator to correctly parse protocols that do not present fixed boundaries.
- Key elements (list of fields that are eligible for filtering). Filters exclusively operate over the exported information through the keys of the protocol.
- Level of the protocol (immediate value). This feature allows the compiler to group protocols that are suitable to be processed concurrently because they are conceptually similar. For instance: Two protocols at the transport layer such as TCP and UDP. Note that in our grammar this field does not necessarily correspond to the layer in the OSI model stack.
- Recursivity (immediate value). Number of times that the current protocol can be consecutively nested in the network stack. Note that protocols at the same level must present the same value for the recursivity field.

Finally, the optional block named "filter" can be used to define rules that apply over the fields in a protocol. Such rules are preceded by a name (that enhances the readability of the Verilog code and lets the filter to be referenced) and a boolean or arithmetic operation that involves one, or multiple, key elements of the protocol. For instance, "*flag_fin||flag_rst*" or "*dst_port*<1024" are rules for the TCP protocol. Nested protocols and fields can be dereferenced by appending the character # and the level (e.g. id#1 in the presented VLAN example refers to the first VLAN tag, whilst id#2 points to the second VLAN tag).

2) Corpus definition: the representation of the network stack is built by the concatenation of multiple protocols, as depicted in listing 2. The block "*match*" concludes the corpus, and is used to define the complete predicates for the rules. Once again, each rule is preceded by an identifier (for the sake of Verilog readability) and a series of filters by protocol and/or filters by key elements. For instance, "dns" would be an example of a filter by protocol whilst "IPv4.filter1 & dns" involves a filter by key element and a filter by protocol.

```
protocol_1 {
        format { ... } metadata { ... }
        filter { /* Filters at protocol_1 */ }
}
...
protocol_n {
        format { ... } metadata { ... }
        filter { /* Filters at protocol_n */ }
}
match {
        rule1: protocol_1.filter1 & protocol_n;
        rule2: protocol_1.filter2 & protocol_n;
}
```

Listing 2: Corpus example

The grammar supports more than one rule at the *"match"* statement. This feature allows the generated designs to go beyond simple discard/accept packet filters. For example, consider the case where different monitoring applications exist, each dedicated to a certain set of protocols. Creating different rules will allow to classify packets in different queues, one for each monitoring application.

C. Autogenerating code: Analysis and data structures

Lexical analyzers (such as *Flex* [11]) are tools for automatically identifying tokens and relevant blocks, abstracting the programmer from the necessity of parsing expressions manually. Meanwhile, the semantic information is completed with tools such as *Bison* [12]. These tools provide basic mechanisms for the generation of an intermediate representation of the parsed code.

Such representation is interpreted in a second step by the Verilog compiler, turning the definition of the grammar and the generation of the code two independent processes. This intermediate structure is composed by a vector of protocols and a list of filters ("*match*" block) which are implemented using standard C types. At the same type, a protocol is subdivided into a set of fields (which are described by the "*format*" block), pointers to fields (elements that integrate the "*metadata*" block) and, optionally, a list of filters which just involves the fields of a certain protocol.

Filters are always modeled as expression trees where the leaves are references to fields, references to the existence of a protocol or immediate values. The parents simply indicate how the leaves are linked. The interconnection possibilities are: unary (logical not, bitwise not) or binary (arithmetic and Boolean operators).

D. Autogenerated code: Hardware architecture

Using the corpus definition as a starting point, a hierarchical Verilog design is generated for the filter machine. When taking a look to the internals of the generated code, it can be seen that three main components are created, as depicted in Fig. 1.

1) Parser logic: It extracts protocols from a packet, as well as the associated keys. The process is pipelined, having the same number of stages as protocol levels are. Every protocol level is evaluated in just one clock cycle, unless the header spans several data words. The parser instantiates as many modules as levels are, and every level, at the same time, instantiates as many entities as protocols are in that level. Each protocol parser has a slave AXI4-Stream interface as input to the module (the packet stream to process), and its outputs are: A bit to signal a match, the offset for the next protocol, the next protocol identifier, and the entire packet as a master AXI4-Stream interface.

The component that groups protocols that can be processed in parallel only operates as a multiplexer. It selects the output of the first protocol in the level that returns a coincidence. If no coincidence is found, the parser at that level will just propagate the registered input of the component. Suppose a firewall that only allows TCP packets with destination port 80. The VLAN header might be missing in a packet but, if it is the case, the rest of the protocols must be checked anyway. This action is accomplished by propagating the information reported by the previous level (e.g. Ethernet), to the next level of protocols (e.g. IPv4) so the absence of the VLAN does not cause an interference to the analysis of the packet.

2) Filter inference: This module instantiates as many filtering instances as the number of levels. For each level there are as many instances as protocols to be filtered. Note that the number of instances in the parser and in the filter logic is not necessarily the same, because certain protocols may not be used for rule matching. Consider for example a rule for UDP port 53 (DNS): the IP protocol header must be parsed in order to reach the UDP header, even though the information in the IP header is not used for filtering.

The inputs to the filtering modules for each level are the keys coming form the parser module at the same level, and the outcome of the rules from the previous level. The filtering module adds the outcome of the rules for the current level, and outputs them. That is, the outcome of the rules propagate from level to level, each level adds its own results. At the output of the filtering module for the last level, the outcome of the rules for all levels is available.

3) Filter application: For each packet, the parser logic module generates a bitmask named Protocol Bit Array (PBA). A '1' in a position of this bitmask means that a certain protocol has been detected. Similarly, the filter inference module generates a bitmask with all the rules: A '1' in a position of this bitmask means that a certain rule has been matched. These two bitmasks are the inputs to the filter application module, which implements the predicates defined in the the block *"match"* of the grammar.

E. Integration in an Virtex UltraScale platform

A complete design has been developed by leveraging the Xilinx IP core *Ultrascale 100G Ethernet Subsystem* [13]. This block supports CAUI-4 (25.78125 Gb/s per line) and CAUI-10 (10.3125 Gb/s per line) 100 Gb/s Ethernet interfaces. On the programmable logic side, the connections to the IP core are respectively accomplished by means of the *RX segmented local bus* (LBUS) and the *TX segmented LBUS*. Conceptually, the *segmented LBUS* can be considered as a 512-bit streaming bus where network packets are only allowed to start at the

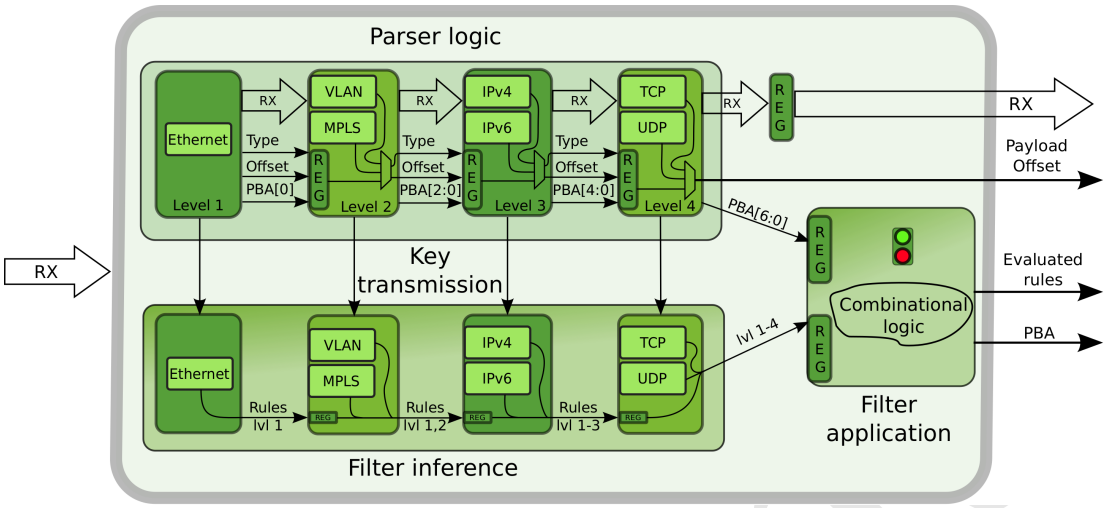

Fig. 1: Architecture of the generated filter

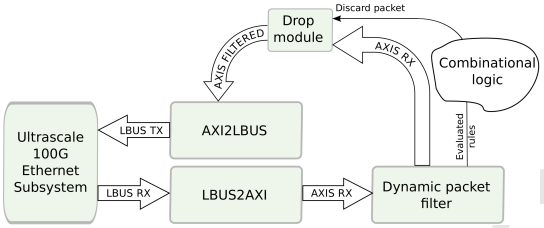

Fig. 2: Integration in Xilinx Virtex UltraScale

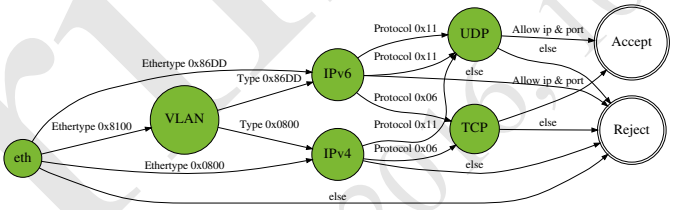

Fig. 3: Port filter automaton

beginning of a 128-bit segment. That is, the 512-bit bus is divided into 4 narrower 128-bit segments, and each network packet is forced to be aligned to a 128-bit position. This behavior is aimed at reducing the wasted space between packets in order to make good use of the bus capacity. The nominal frequency of the bus is 322.265625 MHz.

In Fig. 2, a block diagram of the simulated design is depicted. Traffic starts at the *100G Ethernet Subsystem*, and, in a first stage, it goes through a LBUS to AXI4-Stream converter. As it was detailed in the previous section, the generated code uses AXI4-Stream to transmit packets. AXI-4 Stream is a standard bus, widely used by many IP cores and development tools, as opposed to the proprietary LBUS. The most straightforward alternative for packet transmission in AXI4-Stream is that they begin on a data word boundary. But this mode of operation creates unused spaces between packets, that potentially reduce the available bandwidth of the bus. However, since the raw data rate (322.27 MHz * 512 bit = 165 Gb/s) is far higher than 100 Gb/s, we have verified that the LBUS to AXI4-Stream is safe for all packet sizes.

The incoming AXI-4 Stream then goes to the packet filter module. The latter module outputs the packets, once again in AXI4-Stream format, the outcome of the *match* rules of the filter that has been compiled, and the PBA bitmask. Actually, the outcome of the evaluated rules and the PBA is the information required to take a decision whether to discard or drop the packet. This decision is taken in a small combinatorial logic, which sends the result to Drop module.

The drop module receives the packet stream and a signal to indicate whether the packet should be dropped or forwarded. Such module uses a cut-through approach in order to minimize latency, and its output is an AXI4-Stream containing just the packets that the filter wants to be forwarded. Finally, an AXI-4 Stream to LBUS converter translates the stream format again in order to make it compatible to the LBUS interface of the *100G Ethernet Subsystem*, where packets will be transmitted.

IV. Results

This section presents the results for two proof-of-concept filters that have been synthesized with the tool presented in this paper: A DNS filter, and a firewall that only allows traffic from certain IP addresses and a given destination port. Both are examples of real-world applications; in particular, DNS monitoring is very useful to learn what webpages are being visited by the users of a network.

The first case is a simple filter where the conditions are translated into allow UDP traffic (encapsulated under IPv4 or IPv6) with destination port 53. The associated FSM for the second filter is illustrated in Fig. 3. It involves multiple combinations (TCP/UDP over IPv4/IPv6). Additionally, the corpus has been designed in this second case to also support one VLAN tag.

Resources	Utilization DNS	Utilization Firewall
LUT	5504 (1.023%)	6798 (1.264%)
LUTRAM	0	136 (0.177%)
FF	16155 (1.502%)	17488 (1.626%)
BRAM	8 (0.462%)	8 (0.462%)
IO	11 (1.322%)	11 (1.322%)
BUFG	3 (0.275%)	3 (0.275%)

TABLE I: Categorized FPGA occupation

In order to verify the two filters, the described architecture on Subsection III has been tailored for the Xilinx Virtex Ultrascale VCU108 board [14] using *Vivado 2016.2*. The occupation of the FPGA resources does not excess 2% (see Table I). Being the filters so small paves the way for a partial reconfiguration approach for dynamic filtering. That is, using a ping-pong scheme, it would be possible to achieve a hitless change of filters. While one filter is working, a new filter could be programmed in the FPGA by means of partial reconfiguration. After programming the new filter, moving from old one to the new one could be as simple as changing the control input of an AXI4-Stream bus multiplexer.

The latency of the DNS filter is 4 clock cycles: The pipeline has 3 stages (Ethernet, IPv4/IPv6 and UDP), and one extra clock cycle for matching key filters and protocol filters. The latency of the firewall example is 5 clock pulses: 4 stages of the pipeline: (Ethernet, VLAN, IPv4/IPv6 and UDP/TCP), and the additional clock cycle to match key filters and protocol filters. The clock frequency is 322.265625 MHz, so the latency is respectively 12.4 ns and 15.5 ns.

V. Conclusions

Nowadays, backbone networks are rapidly evolving to 100 Gb/s. However, network monitoring at such speeds is very challenging. One alternative to make monitoring at 100 Gb/s possible is to trim down the traffic that needs to be processed. The goal of this paper, precisely, is to use packet filtering to reduce the input traffic load to monitoring applications. We present an architecture for FPGA-based packet filters, and we demonstrate its operation in a Xilinx UltraScale device.

Nevertheless, the major contribution of the paper is the definition of a new network grammar. This grammar allows the specification of protocols and their interactions. Moreover, the grammar can be used to define filtering rules that apply to a continuous stream of data, even when nested protocols are considered. We have developed a compiler that automatically generates Verilog code for packet filters specified with our grammar.

Although filters are automatically synthesized without needing to write HDL code, there is no shortage in performance: 100 Gb/s links can be processed even if they are fully loaded with minimum size packets. The synthesized architecture is designed to work at a clock rate of 322.265625 MHz and a data width of 512 bits. The architecture is based on a pipelined chain of rule-checking blocks, that is able to process a new data unit every clock cycle. This architecture provides significantly better results than a straightforward porting of the BPF virtual machine to FPGA.

In order to test the proposed approach, we developed two proof-of-concept designs: a DNS filter and a simple firewall, based on IP source address and port. As the generated architecture is a deterministic pipeline, the designs showed a fixed latency of respectively 12.4 ns and 15.5 ns, independently of the traffic conditions. The more levels the filter had, the bigger the latency. The designs were integrated with the 100G Ethernet block of a Xilinx Virtex UltraScale device, using the VCU108 board as reference platform. The proposed filtering system uses standard AXI-Stream buses, so that the effort to integrate the filter was minimal. The resources used by the proof-of-concept designs were below 2% of the total available in the FPGA.

Acknowledgment

This work was partially supported by the Spanish Ministry of Economy and Competitiveness under the project TRÁFICA (MINECO/FEDER TEC2015-69417-C2-1-R), and by the European Union through the dReDBox project (grant agreement No. 687632) of the H2020 programme.

References

[1] J. Mogul, R. Rashid, and M. Accetta, "The packer filter: An efficient mechanism for user-level network code," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, (New York, NY, USA), pp. 39–51, ACM, 1987.

[2] A. Begel, S. McCanne, and S. L. Graham, "Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, (New York, NY, USA), pp. 123–134, ACM, 1999.

[3] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis, "xpf: packet filtering for low-cost network monitoring," in *High Performance Switching and Routing, 2002. Merging Optical and IP Technologies. Workshop on*, pp. 116–120, IEEE, 2002.

[4] H. Bos, W. De Bruijn, M.-L. Cristea, T. Nguyen, and G. Portokalidis, "Ffpf: Fairly fast packet filters.," in *OSDI*, vol. 4, pp. 24–24, 2004.

[5] M. Leogrande, F. Risso, and L. Ciminiera, "Modeling complex packet filters with finite state automata," *IEEE/ACM Trans. Netw.*, vol. 23, pp. 42–55, Feb. 2015.

[6] P. Rolando, R. Sisto, and F. Risso, "Spaf: stateless fsa-based packet filters," *IEEE/ACM Transactions on Networking (TON)*, vol. 19, no. 1, pp. 14–27, 2011.

[7] 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*, pp. 13–24, IEEE, 2013.

[8] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," pp. 12–23, IEEE, Oct. 2011.

[9] P. Bosshart, G. Varghese, D. Walker, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, and A. Vahdat, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 87–95, July 2014.

[10] P. Bencek, V. Pu, and H. Kubtov, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 148–155, May 2016.

[11] V. Paxson *et al.*, "Flex–fast lexical analyzer generator," *Lawrence Berkeley Laboratory*, 1995.

[12] C. Donnelly and R. Stallman, *Bison: the yacc-compatible parser generator*. Free Software Foundation, 1991.

[13] Xilinx, "UltraScale Architecture Integrated Block for 100G Ethernet v1.10," tech. rep., 06 2016.

[14] Xilinx, "VCU108 Evaluation Board, User Guide UG1066," tech. rep., 07 2016.