FPGA-based encrypted network traffic identification at 100 Gbit/s

Mario Ruiz*, Gustavo Sutter*, Sergio López-Buedo*†, Jorge E. López de Vergara*†
* High-Performance Computing and Networking Research Group,
Escuela Politécnica Superior, Universidad Autónoma de Madrid, Spain
{mario.ruiz, gustavo.sutter, sergio.lopez-buedo, jorge.lopez_vergara}@uam.es
† NAUDIT HPCN, Spain
{sergio, jorge}@naudit.es

***Abstract*—Network traffic monitoring is becoming increasingly hard to manage due to the ever-growing speed of network links. At 100 Gbit/s, the huge volume of data makes it very difficult to perform online analyses or to store traffic for subsequent forensic investigations. It is therefore mandatory to carry out some kind of filtering and/or capping in the network traffic to be analyzed. Additionally, the fraction of encrypted traffic is relentlessly increasing. For such encrypted traffic, storing the payload is most times useless. In this paper we present an FPGA implementation of a method to identify plain text (that is, human readable) in the network packet payload. The method is based on both detecting bursts of printable ASCII characters and calculating the fraction of these printable characters in the packet payload. This method has proven to be very effective in reducing the amount of information used in traffic analysis, by saving only the headers of packets with encrypted payloads. We leveraged the advantages of high-level languages to reduce development time, though traditional HDL languages were also used to optimize critical areas of the design. The design targets the 100 Gbit/s Ethernet interfaces of Xilinx Virtex UltraScale devices and it is able to detect human-readable packet payloads at line rate, with a high accuracy.**

***Index Terms*—FPGA, Network Traffic Filter, Deep Packet Inspection, Real Time Analysis, 100 Gbit/s Ethernet**

I. Introduction

Capturing traffic for real-time or offline forensic analyses is widely used to determine the healthiness of networks. The exponential growth of network speeds has caused that the real-time analysis and storage of traffic is becoming more and more complex and expensive. It is also increasingly more common that network traffic is encrypted, and therefore the information that can be extracted from the packet payload is limited. At this point it is mandatory to implement mechanisms to significantly reduce the amount of information used in analysis without losing any relevant details.

In this paper we present the implementation in Field-Programmable Gate Array (FPGA) of a method able to identify plain text (human readable) in the packet payload, which is extremely useful for forensic and real-time analyses. The method to identify plain text is based on the detection of bursts of ASCII printable characters, and also on the percentage of total bytes of the payload containing these printable characters [1]. This method has proven to be very effective to reduce the amount of information used for network traffic analysis. We leveraged the advantages of high-level languages [2] to cut down development time. However, we also used a traditional approach based on Hardware Description Languages (HDLs) to optimize critical areas of the design, which targeted the integrated block for 100G Ethernet of Xilinx UltraScale devices. Thus, we have achieved a design able to work at 100 Gbit/s, reducing the costs and computational load of the host associated to the network analysis.

In the 100 Gbit/s interface of Xilinx UltraScale devices, data is received in a 512-bit (64-byte) interface at 322.265625 MHz (that is, a new data word every 3.103 ns [3]). The maximum packet rate, considering minimum-size packets and minimum interframe gap, is 148.8 millions packets per second. At this rate, it becomes a serious challenge to implement the selected algorithm for discriminating ciphered traffic. In order to achieve this challenge, a low-level, handmade architecture was developed using VHDL and integrated in a High-Level Synthesis (HLS) flow.

Previous efforts on encrypted traffic discrimination can be found in the literature. The main contributions and distinguishing features of our work can be summarized as:

- This architecture is able to deal with fully loaded 100 Gbit/s Ethernet links, guaranteeing that the complete payload of packets can be processed at line rate.
- The presented approach is able to be extended to filtering by other different criteria, while maintaining the line rate capability.
- The work shows the benefits of using a mixed approach combining high-level synthesis and low level design for critical parts, which reduces development effort and increases performance.

The rest of this paper is organized as follows. First of all, section II presents the related work. Next, the method used to recognize relevant packets is described in section III. Section IV describes the proposed architecture to discriminate encrypted traffic. In addition, in subsection IV-D the key component to achieve line rate is discussed. Section V presents the implementation results of the proof-of-concept design. Finally, section VI summarizes our contributions, conclusions and future work.

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

Preprint
December 15, 2016

II. Related Work

With respect to traffic capture, the authors in [4] studied the hardware that is necessary to store 10 Gbit/s network traffic at line rate. Trying to scale the problem at 100 Gbit/s makes the task extremely difficult and costly. Thus, the need of reducing the traffic to be analyzed and eventually stored is evident. The approach to recognize encrypted traffic and capping it, since it is mostly irrelevant for further analysis, has already been explored. We found some relevant works in the literature with different approaches to identify encrypted traffic; in the following paragraphs we summarize them.

For instance, the authors in [5] present a method to detect encrypted traffic in real-time, evaluating only the first packet on the flow. There, 94 % of encrypted traffic is detected as encrypted. Regrettably, that solution is impracticable in FPGA at 100 Gbit/s because it needs a large memory to save the information of flows —on-chip memory is a limited resource in FPGA—, even with an efficient implementation of a hash table, because the collisions will reduce the performance. Similar to the previous work, in [6] a mechanism is implemented to recognize applications encapsulated in Secure Sockets Layer (SSL) connections, which is based on the size of the first packet in the connection. Such method achieved more than 85 % accuracy, but it needed again a large memory to identify the flows, which makes it less suitable for a hardware implementation. In [7] an FPGA-based solution is implemented that achieves 786,432 concurrent flows using an external QDR-II memory at 10 Gbit/s, a small number of flows compared with the amount of concurrent flows in a real core network. This result discards the idea of saving flow information to detect encrypted traffic, given that in other software-based works [8] authors deal with about 10 Million of flows at 10 Gbit/s.

Additionally, in [9] a hybrid method to identify encrypted application traffic is presented, combining a signature-based method and a statistical analysis method. The method achieves a classification accuracy above 99%. But that work does not consider the speed at which they can classify the traffic, and unfortunately it seems to be impossible to achieve line rate if using this method. On the other hand, other paper [10] presents a folded pipeline architecture for 100 Gbit/s Ethernet packet processing. It consists of several parallel mini-pipelines, which process labels belonging to different packets in parallel. This architecture was designed for Multi-Protocol Label Switching - Transport Profile (MPLS-TP) packets and Provider Backbone Bridge Traffic Engineering (PBB-TE) frames.

Also, the patent [11] describes an encrypted-traffic discrimination device, but the description about the architecture is very vague and generalist. In fact, the speed that can be achieved by the proposed system is not provided.

Our approach is based on previous works [1], [12]. These works use bursts of consecutive printable ASCII characters and the percentage of printable ASCII present in a packet as the basis to recognize relevant packets that need to be further analyzed (*i.e.* not encrypted packets). The output of this algorithm can be a whole packet without encrypted traffic or a capped packet with just protocol headers (*e.g.* IP or TCP) for encrypted traffic. For example, this implementation can keep some content of the handshake of SSL flows such as the X.509 certificate, because of the amount of ASCII text in the payload, and discard the payload in the rest of the packets of the encrypted flow. Given the good results provided in [1], [12], we exploit the stateless characteristics of this method and leverage the massive parallelism of FPGA to make an architecture able to filter encrypted network traffic at 100 Gbit/s.

III. Method to identify relevant packets

Currently, over 70 % of the information present in packets payload is encrypted [13]. However, some applications continue using plain-text in the payload. We have focused our work on finding these packets that carry information in plain text, specifically those encoded in ASCII and the subset of variable 8-bit Unicode Transformation Format (UTF-8).

According to their payload, different types of packets can be defined: (a) completely binary (encrypted), (b) with printable ASCII at the beginning of payload, (c) a mix of printable ASCII and binary and (d) pure plain-text ASCII. In [1], [12] the authors studied statistically the relation of consecutive printable ASCII characters and the percentage of printable ASCII characters in the payload to determine the usefulness of the payload of packet for the network analyst. Additionally, they proposed two software-based solutions: the first one inspected the whole packet but did not achieve 10 Gbit/s, and the second one implemented an algorithm able to achieve line rate at 10 Gbit/s but at the expense of not scanning the whole payload.

In our implementation, based on such previous works, we scale the speed to 100 Gbit/s and furthermore, we inspect all bytes in the payload in order to increment the granularity and precision in the decision. Moreover, in our implementation we use fixed parameters to simplify the design: the packet is accepted as non-encrypted when we detect 12 consecutive printable ASCII characters or the percentage of printable ASCII is greater or equal than 50% of the packet size. If the packet does not meet these rules, it is considered binary and then, just the first 64 bytes are kept (headers containing information of Ethernet, IP, and transport protocols), which is the only part of the packet that will be useful for a network analyst.

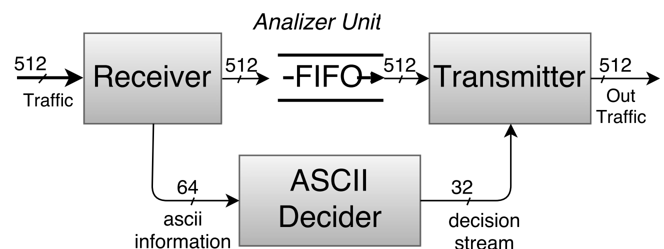

Fig. 1: Analyzer Unit architecture

Preprint
December 15, 2016

IV. Architecture

The basic structure to filter encrypted traffic is the so-called *Analyzer Unit* depicted in Fig. 1. The architecture is composed of three main blocks that will be explained later on. Almost the whole structure and submodules are described in C/C++ using Vivado-HLS (High-Level Synthesis) tool [14], and using directives as described in [15], [16]. The proposed architecture can be easily extended to filtering any kind of traffic by simply modifying the *decider* module, as long as the filtering criteria are stateless. The big challenge here is to meet the timing constraints: At 322.265625 MHz, we only have a few nanoseconds in each clock period to do the job.

A. Receiver

The receiver is connected to the incoming 100Gbit/s Ethernet interface (source of traffic to be filtered) through a 512-bit (64-byte) width AXI4-Stream interface. Its function is to receive Ethernet traffic and split it into two streams: One is an exact copy of the input, which is sent to a First-In First-Out (FIFO) to be stored. The other is a byte-by-byte reduction to a 64-bit boolean vector that represents if the corresponding byte is a printable character (byte decimal value between 32 and 126) or not. This vector will be called from now on the *vector of ASCII information*, which is later sent to the decider to be analyzed. It is obvious to mention that the reduction from 512-bit to 64-bit is trivial. It corresponds to detecting for each octet if it is between the hexadecimal values 0x20 and 0x7E, both inclusive. A simplification is to consider also `DEL` (0x7F) as printable ASCII. If so, the comparative is simpler: It is only needed to check the three most significant bits of each byte. This module was developed in C using Vivado HLS.

B. Transmitter

This module is in charge of generating the output stream. It receives two input streams: The *decision stream* indicates the amount of bytes to be transferred, and it comes from the *ASCII decider* module through a 32-bit stream. The other stream is coming from a FIFO and contains the data back-up. When a valid decision arrives, the logic starts to read data from the FIFO and sends the previously stored bytes. If the decision has less bytes than the packet, the rest of data are discarded in order to empty the FIFO for the next computation. This module was developed in C using Vivado HLS.

C. ASCII Decider

The ASCII decider receives the 64-bit *vector of ASCII information* indicating which characters are printable. This core uses that information to count the amount of printable characters and to detect sequences as explained in section III. At each clock cycle, two operations should be done: i) the 64-bit vector is reduced to a 7-bits number that represents the amount of printable characters in current transaction; and ii) using the last 11 bits from previous transaction it tries to find sequences of 12 consecutive printable characters.

The second operation is computationally easy, it implies to test 64 times in parallel the presence of 12 consecutive '1'.

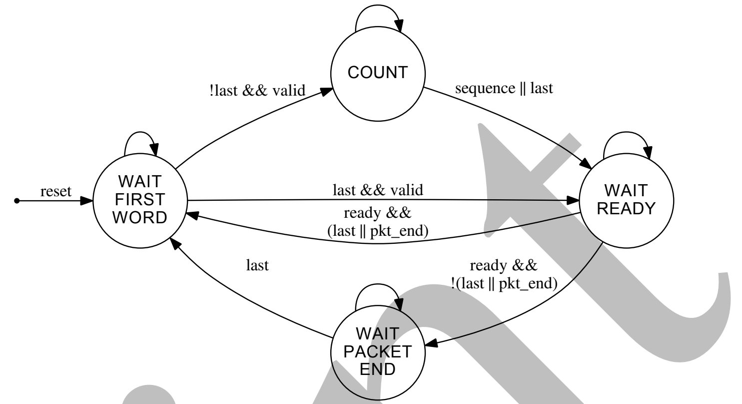

Fig. 2: Finite State Machine implementation

In a Xilinx UltraScale device, it implies 44 LUTs and less than 2 ns using a behavioral VHDL description. Nevertheless, counting the amount of '1' is a much harder problem, and it is deeply analyzed in section IV-D.

Finally, in order to take the decision of how many bytes are to be transferred, a 4-state Finite State Machine (FSM) as shown in Fig. 2 is implemented. Each state and the transition between states are described below.

- WAIT_FIRST_WORD: This is the first state after reset. When the packet has 64 bytes or less, the packet must be sent completely, then decision is 64, and next state is WAIT_READY, else packet is greater than 64 bytes and the next state is COUNT. While there is not any packet, state does not change.
- COUNT: This state sums printable ASCII characters and bytes received. When a sequence of twelve printable ASCII characters is detected, the packet must be sent completely; else, if the packet ends, the amount of printable ASCII characters is compared with bytes received: if is greater or equal than 50 % the packet must be sent completely, else only first 64 bytes are sent, then next state is WAIT_READY. If no sequence or packet ends are detected, state does not change.
- WAIT_READY: This state waits for handshake (assert ready signal) from transmitter. When ready is valid and packet ends the next state is WAIT_FIRST_WORD, else if packet has not ended yet, the next state is WAIT_PACKET_END. If ready is not asserted, the value on decision does not change and state does not change, also decider ready is set to '0'.
- WAIT_PACKET_END: If we arrive at this state we detected a sequence and we sent the decision, but the packet has not ended yet, then this state waits until signal last or pkt_end is asserted to change state to WAIT_FIRST_WORD, else state does not change.

D. Counting the amount of ones in a vector

The problem to count the printable ASCII characters in a 512 bits transaction is reduced to the problem of counting the amount of "1" present in a 64 bit *vector of ASCII*

Preprint
December 15, 2016

information. In the work [17] the author studied different implementations of Hamming Weight and compared with other implementations, but with a vector of width 64 the latency is 5.2 ns, almost doubling the maximum latency required in our architecture. Then, we decided to study different alternatives to achieve the timings.

A naïve implementation in VHDL gives poor results, then we have studied different alternatives that fit better in a 6-LUTs architecture present in Xilinx devices. The idea is the use of *n*-to-*k* reducer (or counters), also a kind of Carry Save Adder (CSA) than counts *n* bits giving *k* bits results. We have evaluated different alternatives:

- **Using 7-to-3 reduction trees (V1)**: The main building block is a 7 to 3 reducer since it can be efficiently implemented using three times two 6-LUTs and a muxF7. Starting with 9 7-to-3 reducers — S(0) to S(8) in Fig. 3 — we have as result 9 3-bit numbers plus one bit (Fig. 4.A) to obtain 3-bit results. Then we reduce again as shown in Fig. 4.A obtaining 3 3-bit number plus a 4-bit number that can be added using a ripple carry adder tree.
- **Using 7-to-3 and 8-to-4 reduction tree (V2)**: In this approach, after the first 7 to 3 reduction we apply an 8 to 4 reduction (using 4 6-LUTs, 2 muxF7 and a muxF8 per bit) with the aim to reduce the logic depth (Fig. 4.B). This approach increases area, and worsens the delay (due to more fan-out and network congestion).
- **Using reduction trees 6-to-3 (V3)**: Since a 6 to 3 reducer can fit in 3 parallel 6-LUTs we expect to reduce net congestion and improve delay. The first state reduces from 64 bits to 11 3-bit numbers — T(0) to T(10). Then, again a second level reduces to six 3-bit numbers and a third level to 3 4-bit that can be added with a 3 input ripple carry adder (Fig. 4.C). The code of this reduction is shown in Fig. 5.

For the two first proposed architectures (V1 and V2), a first step reduces the 64-bits vector to nine 3-bits numbers plus an additional bit that should be added later. The array of reducers is shown at Fig. 3, the resulting dot graphic to be added is depicted at Fig. 4.A and Fig. 4.B respectively for V1 and V2 architectures. In case of V1 a second level composed by three parallel 7-to-3 reducers and *ad-hoc* reducer that outputs 4 bits as shown at Fig. 4.A produces three 3-bits and a 4-bit number that are added by an addition tree. On the other hand, V2 uses three parallel 8-to-4 reducers as proposed in Fig. 4.B, generating a number than can be added using ternary adders. Finally, Fig. 3.C shows for V3 the first step of 6-to-3 reducer, followed by the second level of reduction, composed by a 3 times 6-to-3 reduction and 3 times 5-to-3 reduction. The third logic level finally reduces to three 4-bits numbers to be added by a ternary adder. The code of counter of ones in a 64-bits vector is shown in Fig. 7.

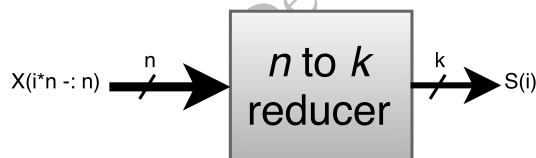

Fig. 3: Generic *n*-to-*k* reducer.

The implementation details of these alternatives are summarized at table I, where the 6-to-3 reduction clearly gives better results both in area and delay.

E. Architecture to achieve 100 Gbit/s

The previously presented architecture does not support pipelined operation, because each packet needs to be stored until a decision is taken. Then, in order to support line rate at

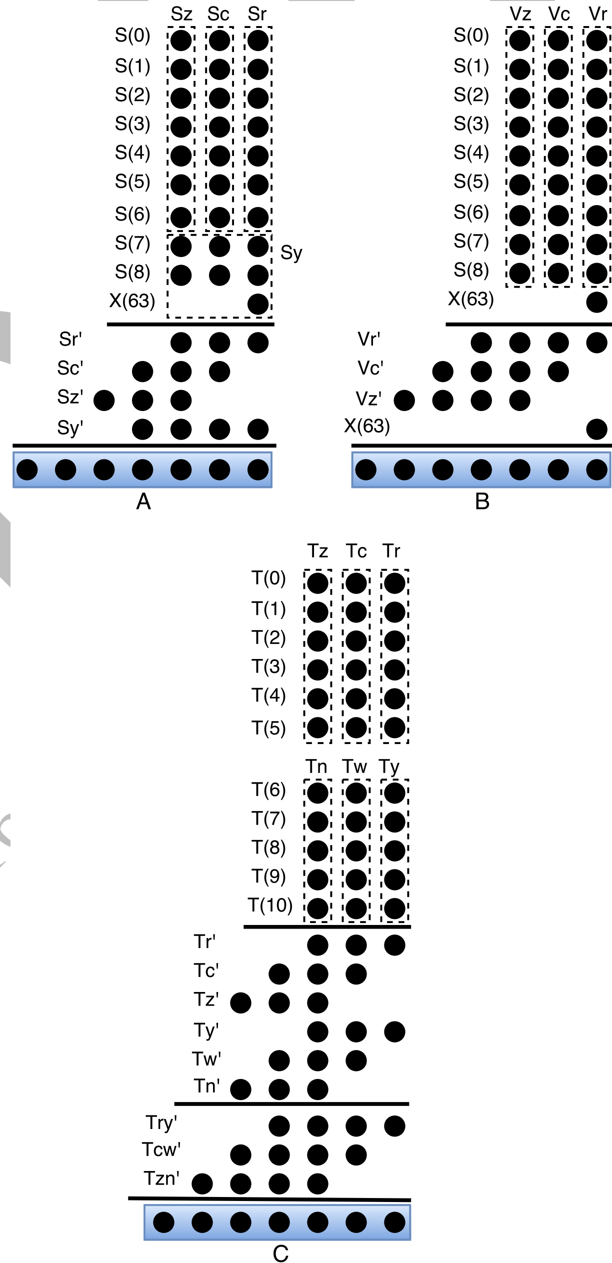

Fig. 4: Dot Graph for different reduction trees: A) on two levels of 7-3 reduction and finally quaternary adder. B) on 7-3 and 8-4 reduction and finally ternary adder. C) on two levels of 6-3 reduction and a level of two input adders and a final ternary adder

December 15, 2016

TABLE I: Critical path and resources utilization for different tree reduction.

Version	Critical Path (ns)	LUTs
naïve	2.99	93
7-to-3 (V1)	2.89	92
7-to3 & 8-to-4 (V2)	2.98	80
6-to-3 (V3)	2.65	65

the worst scenario (minimum-sized packets), it is necessary to instantiate more *Analyzer Units* in parallel.

The *Analyzer Unit* has a latency of 5 cycles, for packets equal or less than 64 bytes (the whole packet fits in a 512-bits transaction). Then, the worst case is to have a new packet at each cycle, which implies a minimum of five *Analyzer Units* working in parallel to support 100 Gbit/s in the worst scenario. The resulting architecture is used and shown in Fig. 6, where input traffic is split in sequential Round Robin to different instances of the *Analyzer Unit*. In order to avoid packet disorder, the Round Robin collector reads sequentially the output of the analyzer units. In this way we ensure that packets in filtered traffic keep the same order as in the incoming Ethernet interface.

V. Implementation Results

The implementation targeted the Xilinx VCU108 board [18], and was made using the Vivado Design Suite 2016.2. The parallel implementation was simulated with the Integrated Block for 100G Ethernet [3] obtaining successful results, achieving line rate. The clock timing constraint of a 3.103 ns period (322.265625 MHz) is satisfied and the resource usage for a Xilinx Virtex UltraScale XCVU095-FFVA2104-2-E device is provided in Table. II. It should be noted the small footprint of the design, which uses less than 4% of the total area. Thus, a smaller and cheaper device such as an XCVU065 could be used in a commercial implementation of the design.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity reducer_6to3 is port (
  x: in std_logic_vector (5 downto 0);
  s: out std_logic_vector (2 downto 0)
);
end reducer_6to3 ;
architecture rtl of reducer_6to3 is
  type memrom is array (0 to 63) of STD_LOGIC;
  signal sum_0: memrom := x"6996_9669_9669_6996";
  signal sum_1: memrom := x"177E_7EE8_7EE8_E881";
  signal sum_2: memrom := x"0001_0117_0117_177F";
begin
  s(2) <= sum_2(conv_integer(x));
  s(1) <= sum_1(conv_integer(x));
  s(0) <= sum_0(conv_integer(x));
end rtl;
```

Fig. 5: HDL code of 6 to 3 reducer

TABLE II: Resource usage in a Xilinx UltraScale xcvu095

Resource	used	max available	% usage
LUT:	11761	537600	2.19
FF:	31016	1075200	2.88
BRAM:	53	1728	3.04
GT:	10	52	19.23
BUFG:	3	960	0.31

VI. Conclusion and Discussion

In this paper, a stateless architecture to identify and filter encrypted traffic at 100 Gbit/s was presented. The algorithm is based on the recognition of sequences of consecutive characters or the presence of a percentage of readable characters in the payload of a packet. We are able to scan the complete payload, obtaining a higher accuracy than previously published results, which only analyzed partially the payload.

The architecture is designed to work at line rate in 100 Gbit/s Ethernet links (up to 148.8 million packets per second), using a 512-bit AXI4-Stream interface clocked at 322.265625 MHz. Such data rate calls for a careful design of the critical parts of the algorithm. We showed how a mixed methodology that uses High Level Synthesis (HLS) for non-critical parts and VHDL for time-critical regions is able to cope with that data rate, but at the same time it provides a significantly better productivity than a conventional, HDL-only approach.

The present architecture can be easily extended to discriminate any kind of traffic by simply modifying the decider module of Fig. 1. Additionally, it would be possible to add more stateless filters (such as IP/port origin/destination, protocol, etc.) in parallel to filter presented here, but maintaining the line rate operation.

Although we have used this architecture for network monitoring purposes (capturing and storing unencrypted traffic), it can also be used to divert binary traffic to network nodes with decryption capabilities for further deep packet inspection.

A simple extension under development is to support standard network capture rules such as BPF (Berkeley Packet Filters) or similar with minor changes and the possibility of hot-plugging different filters using dynamic reconfiguration.

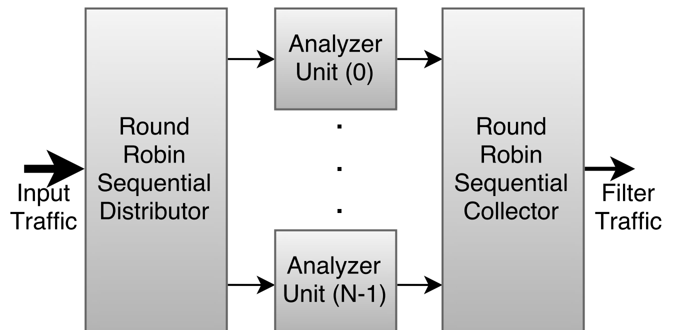

Fig. 6: Several Analyzer Units connected in parallel to Improve Throughput

Preprint
December 15, 2016

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity counter64_7_v3 is port (
  x: in std_logic_vector (63 downto 0);
  s: out std_logic_vector (6 downto 0)
);
end counter64_7_v3 ;
architecture rtl_3 of counter64_7_v3 is
  COMPONENT reducer_6to3 is port (
      x: in std_logic_vector (5 downto 0);
      s: out std_logic_vector (2 downto 0)
    );
  END COMPONENT;
type sums_L1 is array (0 to 10) of
                        STD_LOGIC_VECTOR(2 downto 0);
signal sum_L1: sums_L1;
signal L1_vert0: STD_LOGIC_VECTOR(10 downto 0);
signal L1_vert1: STD_LOGIC_VECTOR(10 downto 0);
signal L1_vert2: STD_LOGIC_VECTOR(10 downto 0);
signal sum_L2_0, sum_L2_1 : STD_LOGIC_VECTOR(2 downto 0);
signal sum_L2_2, sum_L2_3 : STD_LOGIC_VECTOR(2 downto 0);
signal sum_L2_4, sum_L2_5 : STD_LOGIC_VECTOR(2 downto 0);
signal sum_L3_0: STD_LOGIC_VECTOR(3 downto 0);
signal sum_L3_1: STD_LOGIC_VECTOR(3 downto 0);
signal sum_L3_2: STD_LOGIC_VECTOR(3 downto 0);
begin
  -- First level of reduction
  L_1: for i in 0 to 9 generate
  reduc: reducer_6to3 port map(x => x(i*6+5 downto i*6),
                               s => sum_L1(i) );
  end generate;
  reduc10: reducer_6to3 port map(x =>x(63 downto 60)&"00",
                                 s => sum_L1(10));
  -- grouped vertically result of first level of reduction
  L_1a: for i in 0 to 10 generate
    L1_vert0(i) <= sum_L1(i)(0);
    L1_vert1(i) <= sum_L1(i)(1);
    L1_vert2(i) <= sum_L1(i)(2);
  end generate;
  -- Second level of reduction
  L_2b: reducer_6to3 port map(x => L1_vert0(5 downto 0),
                              s => sum_L2_0 );
  L_2c: reducer_6to3 port map(x => L1_vert1(5 downto 0),
                              s => sum_L2_1 );
  L_2d: reducer_6to3 port map(x => L1_vert2(5 downto 0),
                              s => sum_L2_2 );
  -- reduce partial 5 to 3
  L_2e: reducer_6to3 port map(x=>L1_vert0(10 downto 6)&'0'
                              ,s => sum_L2_3 );
  L_2f: reducer_6to3 port map(x=>L1_vert1(10 downto 6)&'0'
                              ,s => sum_L2_4 );
  L_2g: reducer_6to3 port map(x=>L1_vert2(10 downto 6)&'0'
                              ,s => sum_L2_5 );
  -- sum result of second level reduction
  sum_L3_0 <= sum_L2_0 + ('0' & sum_L2_3);
  sum_L3_1 <= sum_L2_1 + ('0' & sum_L2_4);
  sum_L3_2 <= sum_L2_2 + ('0' & sum_L2_5);
  --L4
  s <= ('0' & sum_L3_2 & "00") + (sum_L3_1 & '0')
      + sum_L3_0;
end rtl_3;
```

Fig. 7: Counter of ones, final integration

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] V. Uceda, M. Rodríguez, J. Ramos, J. L. García-Dorado, and J. Aracil, "Selective Capping of Packet Payloads at Multi-Gb/s Rates," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 6, pp. 1807–1818, June 2016.

[2] M. Forconesi, G. Sutter, S. López-Buedo, J. E. López de Vergara, and J. Aracil, "Bridging the Gap between Hardware and Software Open Source Network Developments," *Network, IEEE*, vol. 28, no. 5, pp. 13–19, 2014.

[3] Xilinx, "UltraScale Architecture Integrated Block for 100G Ethernet v1.10," Tech. Rep., 06 2016. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/cmac/v1_10/pg165-cmac.pdf

[4] V. Moreno, J. Ramos, J. L. García-Dorado, I. González, F. J. Gómez-Arribas, and J. Aracil, "Testing the Capacity of Off-the-Shelf Systems to Store 10GbE Traffic," *IEEE Communications Magazine*, vol. 53, no. 9, pp. 118–125, September 2015.

[5] P. Dorfinger, G. Panholzer, and W. John, "Entropy Estimation for Real-Time Encrypted Traffic Identification," in *International Workshop on Traffic Monitoring and Analysis*. Springer, 2011, pp. 164–171.

[6] L. Bernaille and R. Teixeira, "Early Recognition of Encrypted Applications," in *Proceedings of the 8th International Conference on Passive and Active Network Measurement*, ser. PAM'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 165–175. [Online]. Available: http://dl.acm.org/citation.cfm?id=1762888.1762911

[7] M. Forconesi, G. Sutter, S. Lopez-Buedo, and J. Aracil, "Accurate and Flexible flow-based Monitoring for High-Speed Networks," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–4.

[8] G. Nassopulos, D. Rossi, F. Gringoli, L. Nava, M. Dusi, and P. M. S. del Rio, "Flow management at multi-gbps: tradeoffs and lessons learned," in *International Workshop on Traffic Monitoring and Analysis*. Springer, 2014, pp. 1–14.

[9] G.-L. Sun, Y. Xue, Y. Dong, D. Wang, and C. Li, "An novel hybrid method for effectively classifying encrypted traffic," in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*. IEEE, Dec 2010, pp. 1–5.

[10] K. Karras, T. Wild, and A. Herkersdorf, "A folded pipeline network processor architecture for 100 Gbit/s networks," in *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on*, Oct 2010, pp. 1–11.

[11] S. Ata, G. Hasegawa, Y. Nakahira, and N. Nakamura, "Encrypted-traffic discrimination device and encrypted-traffic discrimination system," Apr. 28 2015, uS Patent 9,021,252. [Online]. Available: https://www.google.com/patents/US9021252

[12] V. Uceda, M. Rodríguez, J. Ramos, J. L. García-Dorado, and J. Aracil, "Selective capping of packet payloads for network analysis and management," in *International Workshop on Traffic Monitoring and Analysis*. Springer, 2015, pp. 3–16.

[13] Sandvine, "Global Internet Phenomena Spotlight: Encrypted Internet Traffic," Tech. Rep., 05 2015. [Online]. Available: https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/encrypted-internet-traffic.pdf

[14] Xilinx Inc. Vivado High-Level Synthesis (Vivado-HLS). [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[15] K. Karras and J. Hrica, "Protocol-Processing System Thrive with Vivado HLS," *Xcell Journal*, vol. Third Quarter 2014, no. 88, pp. 45–51, 2014.

[16] ——. (2014) Designing Protocol Processing Systems with Vivado High-Level Synthesis. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1209-designing-protocol-processing-systems-hls.pdf

[17] V. Sklyarov, I. Skliarova, A. Sudnitson, and M. Kruus, "Fpga-based time and cost effective hamming weight comparators for binary vectors," in *EUROCON 2015 - International Conference on Computer as a Tool (EUROCON), IEEE*, Sept 2015, pp. 1–6.

[18] Xilinx, "VCU108 Evaluation Board, User Guide UG1066," Tech. Rep., 07 2016. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/vcu108/ug1066-vcu108-eval-bd.pdf

Preprint
December 15, 2016