An FPGA-Based Approach for Packet Deduplication in 100 Gigabit-per-Second Networks

Mario Ruiz*, Gustavo Sutter*, Sergio López-Buedo*†, Jose Fernando Zazo†, and 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, josefernando.zazo, jorge}@naudit.es

***Abstract*—Network traffic monitoring usually faces the problem of packet duplication, which arises when port mirroring is being used. That is, when traffic is copied from the ports of a switch or a router that are being monitored, to a mirror port where a monitoring probe is attached. Thus, a packet can be copied twice, both at the ingress and egress ports, therefore generating duplicates. Information redundancy caused by packet duplication not only leads to increased workloads at the monitoring probes, but also calls for more disk space to store the network traces. Actually, packet duplication may increase 100% the monitoring load. There are different sorts of packet duplication; in this paper we focus on switching duplication, because it is the most common in a network monitoring scenario, where the network probe is attached to a core switch. We present a high performance FPGA architecture that is able to detect and remove duplicated packets in 100 Gbit/s networks. It is based on a 64-bit key and a BRAM-based shift register that allows us to build an element-based sliding window of size up to 79,872 elements. The design targets the Xilinx Virtex UltraScale family, using the integrated 100G Ethernet Subsystem available in such devices, and it has been tested on a VCU108 evaluation kit.**

I. Introduction

Traffic monitoring plays a fundamental role in the activities of network administrators: it is the cornerstone of forensic network traffic analysis. Unfortunately, packet duplication has usually been an undesirable side-effect of traffic monitoring. Packet duplication produces redundant information in the monitoring probe, which leads to an increased workload, and calls for more disk space to store the network traces.

The most common way to capture traffic from a network is using the port mirroring feature of the switching devices, which is also known as Switched Port Analyzer (SPAN). This feature, included in most enterprise-grade switches and routers, consists in making a copy to a mirror port of packets traversing monitored ports. The mirror port is then connected to a monitoring probe running a capture traffic engine such as `tcpdump`, `tshark`, a DPDK-based tool, etc. Port mirroring unavoidably creates packet duplication. For example, in Fig. 1, when computer A sends data to computer B, packets pass through port 1 (ingress port) and port 2 (egress port). If both port 1 and port 2 are configured to be mirrored, packet appears twice in the mirroring port. In the worst scenario, all traffic could be duplicated, which implies a large resource wasting. It is a challenge to know the interval between copies of the same packet, because such lapse of time is function of switching time, queuing delay and traffic load at that moment.

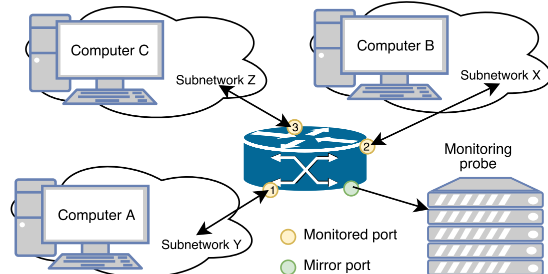


Fig. 1: Simple monitoring set up.

Packet deduplication has had little relevance in the academia, because off-the-shelf hardware could, in the past, deal with this problem. But nowadays, with the advent of 100 Gbit/s networks, this problem is becoming increasingly challenging, since time between packets could be as small as 6.7 ns. But packet deduplication is key for monitoring 100 Gbit/s networks, because at such speeds, processing unnecessary information causes a penalty that cannot be afforded. Additionally, packet deduplication has additional benefits, as authors in [1] discuss.

In this paper, we explore the possibility of using an FPGA to detect and discard duplicate packets caused by port mirroring at 100 Gbit/s speeds. We use a novel architecture to detect and remove duplicated packets, which is based on using an element-based (not time-based) sliding window. For this, a k-bit key represents each packet. There is a total of N sliding windows, which are addressed using $n = log_2 N$ bits of the key, and each window contains M elements whose size is $k-n$ bits.

The rest of this paper is organized as follows: First of all, section II presents the related work. Then, packet hashing is explained in section III. Next, section IV describes the proposed architecture to detect and drop duplicated packets, with FPGA details being given in section V. Later, section VI presents the implementation results of the proof-of-concept design. Finally, section VII summarizes our contributions, conclusions and future work.

II. Related Work

Authors in [2] introduced the packet duplication problem. They studied the different types of duplicates that can be generated when a mirror port is used to capture traffic. The main problem that packet duplication brings is an increase in the amount of information to handle, which makes more difficult the network monitoring process. We took this work as a starting point to create an FPGA architecture that deals with the problem of packet duplication in 100 Gbit/s networks. Packet duplication means that two or more packets contains the same information. However, this definition does not imply that some fields inside protocol headers could change —*e.g.*, such as IP addresses or ports. Anyway, in this work we focus on packets that look exactly the same, which is the common behavior for switching duplicates.

In addition, authors in [2] recommended to use a time-sliding window of 15 ms to detect packet duplication in 100 Mbit/s networks. According to that work, it is expected that such time window is going to be several times shorter at 100 Gbit/s, due to faster switching times —for instance, some switch vendors claim that their equipment can switch frames in the order of μs. Nevertheless, we decided instead to use an element-based sliding window, because time management in an FPGA would need extra bits to store timestamps and a mechanism to remove expired packets.

Comparing each byte of the payload, as it is mentioned in [2], is a complex task, because it implies parsing each packet header in order to get a pointer to the first byte of the payload. Some works about parsing packets in FPGA have been done [3], [4], but we consider that adding this complexity to the architecture is not necessary for a proof-of-concept, although this approach could be taken into account in the future.

Thus, we are looking for a solution that is able to find if a value exists inside a memory of S elements, and that also allows a very fast insertion and deletion of elements. Currently, there are different technological choices to satisfy these requirements, following we summarize the most popular options. Content-Addressable Memory (CAM) [5] is a memory that compares input data against a set of stored values, and returns the address of the matching data, if the data is present in the memory. Usually, CAMs feature a single-cycle latency, thus making them a really fast option, although the speed of a CAM comes at the cost of a huge resource utilization. Authors in [6] introduce the architecture of CAMs and its features. As a generalization of CAMs, Ternary CAM (TCAM) [7], [8] is one of the most popular methods for packet classification. Using the concept of "don't care" in the comparison, it allows searching for not exact matches. This sort of memory is widely used in network equipment to route traffic, because it is very useful to compare subnetworks. All the same as in conventional CAMs, latency is one clock cycle but resource utilization is significant. Additionally, exporting the stored elements in a TCAM is a difficult task, as it takes longer than comparison [7].

Furthermore, another popular alternative is Bloom Filters [9], this data structure is widely used to test if a element is within a set. Due to its functionality, false positive matches are possible, however false negative matches are not. While it is straightforward to insert elements, removing a single element is not possible for the simple reason that it could remove other elements as well.

Finally, there is a commercial solution in the market [10], which claims to be capable to detect all packet duplicates and remove them at 100 Gbit/s speed. However, they do not provide any details on how they achieve this functionality, or what is the size of the sliding window in terms of time or number of elements.

III. Hashing a Packet

Storing and comparing variable-length packets is neither efficient nor deterministic, because the Ethernet frame length could range between 60 and 1518 bytes (not counting the Frame Check Sequence). In order to solve this problem, hash functions are commonly used to reduce the contents of a packet to a fixed-length key (digest). There are different sorts of hash functions, some better than other [11]. In the context of networking, a very important metric of hash functions is how the packets are distributed along the addressable area of the hash function [12]. A hash function is judged as good if it produces a uniform distribution of packets into the addressable area. However, achieving this goal can sometimes be complex because each network link has its particular traffic patterns.

Hash functions suffer from a problem called hash collision, that is, that two different inputs to the hash function produce exactly the same hash. In our case, this means that several packets can have exactly the same key. Software implementations typically deal with this problem using either linked lists or a fixed array of M possible collisions. In both cases, it is necessary to store the original data along with the hash in order to solve the collision. The difference is that the former uses dynamic structures, while the latter uses static ones. In principle, the latter is more suitable for FPGA implementation, but some type of insertion and replace policy has to be implemented when no more space is available in the fixed array of M possible collisions.

Hash collisions are particularly adverse to this application. In our design, two equal hashes mean a packet duplicate. But in case of collision, two different packets will have the same key, thus generating a false positive for duplicate. This is an undesirable condition because it will remove a non-duplicated packet, thus potentially losing valuable information. In order to minimize the probability of a false positive occurring, we analyze the relationship between the key size and number of packets in the element-based sliding window that we want to keep. Equation 1 is a fair approximation of the probability that two different elements in a window of S elements have the same key in a addressable area of size K. More details about this equation can be found in [13].

$$P_{collision} = \frac{S^2}{2K} \tag{1}$$

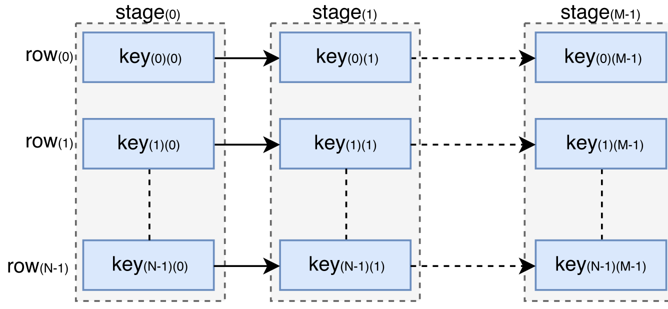


Fig. 2: Architecture of the Memory with N rows and M stages.

Given the equation above, we face a conflict in the number of elements in the window. The more elements we have, the bigger the window is, and the bigger the window is, the more delay that can exists between duplicates. However, the probability of collision rises quadratically with the number of elements, so there exists a trade-off between window size and probability of a false positive. In our case, we chose as a starting point a 64-bit key ($K = 2^{64}$) and a 65,536-element window ($S = 65,536$), which gives a probability of collision of approximately $1.164 \cdot 10^{-10}$, which is low enough for our problem.

In the bibliography [14]–[16] there are many suitable implementations of different hash functions in FPGA, and also more complex schemes, such as sketch tables [17], but none of them operates at 100 Gbit/s.

IV. Architecture Overview

The trivial FPGA implementation of an element-based sliding window consists in using shift registers with M stages, and comparing each stored key against the current key in parallel. This solution could be easily implemented in LUTRAMs, but it is only valid for a few keys. If the number of stages of the shift register grows, the resource utilization grows too, thus adding excessive complexity to FPGA design. As a result, the design frequency has to be decreased in order to pass time constrains, and a 100 Gbit/s data rate cannot not be achieved. Actually, it is absolutely impossible in current FPGAs to compare 65,536 elements in parallel at 322.265625 MHz, which is the frequency of operation of the 100 G Ethernet Subsystem.

We developed instead a mixed alternative. On the one hand, it takes advantage of BRAMs and their true dual port capabilities. On the other hand, we use a scheme similar to that of a shift register, to emulate the behavior of an element-based sliding window. The main idea is to connect the data output of $BRAM(i)$ to the data input of $BRAM(i+1)$, so that when the write enable signal is asserted, data from the i-th stage passes to the $(i+1)$-th stage and so on, producing a shifting of the elements, and also dropping the oldest element. Now, instead of only one shift register, we have N shift registers selectable by the address of memories. We use the n most significant bits of the key to address the memory and the remaining bits of the key are stored in the memory.

Fig. 2 shows a high-level overview of this architecture. First, each row behaves as a different shift register, thus increasing the number of the elements in the window. Second, it allows sharing the comparison logic to detect duplicates among all rows (that is, all shift registers), because only one row will be active at a given time. However, it should be noted that this scheme does not guarantee an even access to all rows. Depending on the traffic patterns, the hash function might tend to access some rows more frequently than the others. Thus, we cannot state that 100 % of the elements are going to be always used. Anyway, this solution offers a larger capacity than a simple shift register.

V. FPGA Architecture

Figure 3 shows a block design of the deduplicate module and the connection between its submodules. As the hash function is computed for the whole packet contents, the design follows a store and forward approach.

The **Hash Table** module computes the hash function for the packet contents. It is straightforward task because we have chosen a naïve hash function: it simply divides the packet contents in 64-bit chunks, and does a bit-by-bit XOR of all the chunks in order to obtain a 64-bits key. Though this hash is not as good as others are, it is enough for proof-of-concept purposes. But any other hash function could be implemented as long as it has a pipeline implementation. The resulting 64-bit key is outputted one clock cycle after *tlast* is asserted in the incoming AXI4-Stream interface. Meanwhile, while the hash function is computed, the packet is being queued in the FIFO, waiting for the drop or forward decision depending whether it is a duplicate or not.

The **Port Arbiter** module selects what port of the BRAMs is going to be used. Usually, it works as a Round Robin mechanism, that intersperses each key to one of the two ports of the BRAMs. The reason for using the two ports is doubling the performance. However, if two consecutive keys point to the same address (the n most significant bits are equal), the both keys will be sequentially sent to the same port, to prevent data corruption. If both ports use the same address simultaneously, data integrity cannot be guaranteed.

The **Deduplicate** module is composed of M BRAMs (stages). The n most significant bits of the key are used to address the memories, while the remaining $k - n$ bits are compared with the data output of the M memories. If there

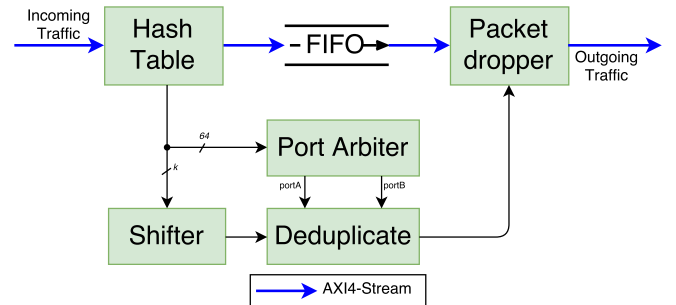


Fig. 3: Global Architecture of the deduplicate module.

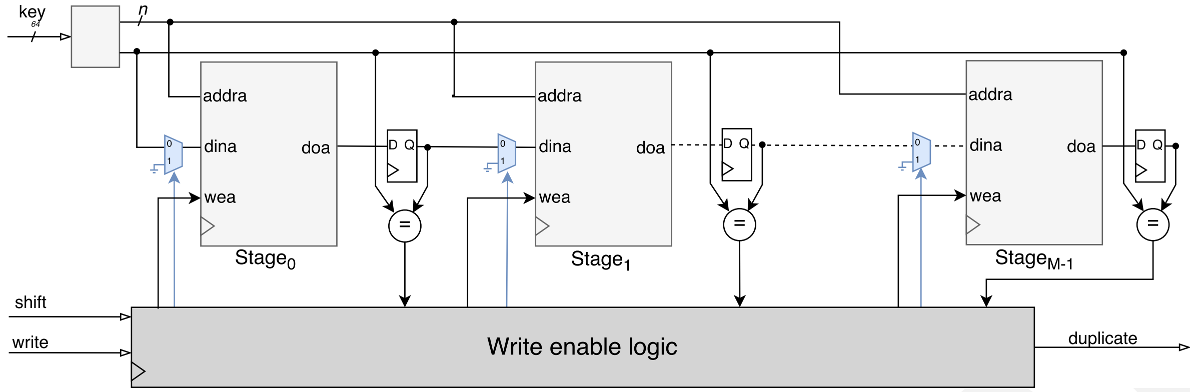


Fig. 4: Deduplicate Architecture based on M stages.

is a match, the current key will not be written, and there are two possible actions regarding the element that matches: **A)** do nothing and keep that element in the memories; or **B)** remove it and create a bubble (gap) in the memories. Additionally, this module informs that the current packet is a duplicate and needs to be dropped. Later we will explain the implication of choosing **A)** or **B)** options. On the contrary, if there is no match, the $k - n$ bits of the current key are written to the BRAM at stage 0, and data from BRAM at stage i is written to BRAM at stage $i + 1$, as it was detailed in section IV.

The **Packet Dropper** is a simple module, which waits for the decision coming from the Deduplicate module. When the decision is made, it starts to read the packet stored in the FIFO. If the decision was to drop the packet, the packet will be dequeued, but not sent. Note that it is mandatory to dequeue the packet in order to empty the FIFO. On the contrary, if the decision was to forward the packet, the entire packet will be dequeued and sent to outgoing traffic.

Finally, the **Shifter** module is a low priority process that is in charge of generating dummy elements to force a shifting of elements in the sliding window. This process is designed to remove old packets from the BRAM-based shift register. It uses a BRAM memory where each position contains the timestamp of the last access to each row. If we assume that the hash produces a uniform distribution, the mean access time to each row will be the maximum time of a packet times the number of rows ($pkttime_{max} \cdot N$ = $1230.4ns \cdot N$). The checker process inserts a bubble when a row has not been accessed for this time. Therefore, this module helps to reduce the false positive ratio by removing the oldest key.

A. Low Level Architecture of Deduplicate module

Figure 4 shows in more detail the low level architecture of the BRAM-based shift register. As it was explained above, the current key is latched and separated in two parts, where the n most significant bits are used as address to select one of the N shift registers. Additional registers have been placed at the output of BRAMs due to the relatively high clock-to-output time of these memories ($T_{RCKO_DO} \approx 1.35ns$ in the Xilinx UltraScale Family). Without these registers it would be very difficult to meet the timings, but the drawback if that read latency is increased to 2 clock cycles. Once that the output data is at the registers (two clock cycles after setting the address), its values are compared with the remaining $k - n$ bits of the current key. If there is a match in any of the comparisons, it means that a duplicate has been found. That is, as a result of the parallel comparison, we have a vector of M bits pointing (one-hot encoding) at the position of the element that matches (if it was any). This vector is registered, and in the next step a bitwise OR reduction is made. This OR reduction is critical in terms of timing, and its results is the duplicate found flag. It this flag is true, the current key will not be inserted in the first element of the shift register —$wea_0 = 0$.

As it was explained before, there are two options when a duplicate is detected. **A)** Do nothing with the element that is stored in the memory: In this case the blue multiplexer in Figure 4 is not implemented. **B)** Overwriting the position where the match was found, creating a bubble. The comparison bit of the i-th stage is connected to the select pin of the i-th multiplexer. When a match happens, the element in the i-th stage will be deleted. This feature is a good idea when we are sure that there is only one duplicate, as it creates space for other elements.

When there is no match, there are two possibilities as well. If we have decided to use alternative A, the write enable signals for all M BRAMs are asserted, thus producing a shift of the elements from the i-th stage to the $(i + 1)$-th stage, and also eliminating the oldest element. However, if we chose alternative B, the row could include bubbles (each element has a valid flag). If we assert the all write enable signals, a shift will be produced removing the oldest element, but this has no sense, because we are wasting space. Instead of asserting all write enable signals, we propose using a more smart logic — *e.g.*, a priority encoder. When a new key arrives and there is not match, we use the valid flag to calculate the value of the write enable vector. For example, if the j-th position is empty, the priority encoder asserts the write enable signal only for stages 0 to the j-th position. This pokes the first bubble and does not remove the oldest element. Although this solution is straightforward, it however requires for each stage an OR reduction of the valid flags of the previous stages. That is, that for the worst case ($Stage_{M-1}$) we need an OR reduction of $M - 1$ bits, which might be challenging considering the reduced clock period.

Finally, the shift bit connects to the Shifter module and it is used to insert bubbles in case that the incoming traffic pattern is such that certain rows are not updated. It takes only one clock cycle to insert a bubble in a given row. All write enable signals are asserted, and the select pin of the multiplexer for $Stage_0$ is set to '1' so that a bubble is inserted in the first stage, while the select pin of the multiplexers of the remaining stages are set to '0' in order to shift the elements.

As a summary, the deduplication process takes 4 clock cycles. Anyway, if two consecutive addresses are the same, then we can save one clock cycle at the fetching process, because the address has already been sent. Due to this latency, we decided to use both ports of the BRAMs in order to get the maximum throughput, though this decision implied additional logic in order to avoid conflicts, as it has been explained in the previous section.

B. Targeting into an FPGA

We targeted this architecture in the VCU108 [18] evaluation kit from Xilinx, which has a Virtex UltraScale xcvu095-ffva2104-2-e FPGA with over a million flip-flops, over a half million of LUTs, and 1728 BRAMs. We used the integrated 100G Ethernet subsystem [19] to capture incoming traffic. Its output is a 512-bits LBUS interface, that we translated into 512-bit AXI-Stream interface clocked at 322.265625 MHz (clock period 3.103 ns).

As the minimum Ethernet frame size is 60 bytes, then the minimum time between packets in 100 Gbit/s is 6.72 ns. Considering the clock period, this means that at least there are 2.16 clock cycles between minimum-size packets. The architecture described in V-A can process one packet every two clock cycles, because it leverages the dual port capabilities of BRAMs. As a conclusion, our architecture can handle 100 Gbit/s Ethernet at line rate without any packet loses.

VI. Experimental Results

We firstly simulated the behavior of the naïve XOR hash function. We used five different traces to carry out this simulation. In Table I we describe their main features. Trace 1 was captured at a student laboratory from our university, and packets are capped to a maximum of 300 bytes; trace 2 is a capture from the internal network of a big Spanish company, and packets are not capped; finally, traces 3, 4 and 5 were downloaded from CAIDA [20]. CAIDA provides anonymized Internet traces captured at the Equinix Chicago and San Jose monitors. Due to privacy policies, CAIDA traces do not have payload. However, they can help us to understand how our hash table works. Fig. 5 shows how the packets are distributed along the BRAM rows when we respectively use 8, 9 and 10 bits to address the memories. We also plot in the same graphic the results of using a 64-bits Spooky hash [21], [22] . There is a big difference on how hashes distributed the packets, as the naïve XOR hash introduces pronounced spikes and the Spooky hash yields to much more uniform distributions (with $P(n) \approx 2^{-k}$).

TABLE I: Features of used traces.

Trace	File Size (GB)	Data size (GB)	Millions of Packets	Duration (Minutes)	Packet size mean (Bytes)
1	351	1,406	1,378	35,947	1,019.87
2	415	406	539	11,472	754.32
3	85	663	1,165	62	569.78
4	272	2,443	3,688	62	662.58
5	39	513	531	9	965.03

TABLE II: Summary of the schemes of Depth and Stages that meet timing and its maximum sliding window as a function of the packet size.

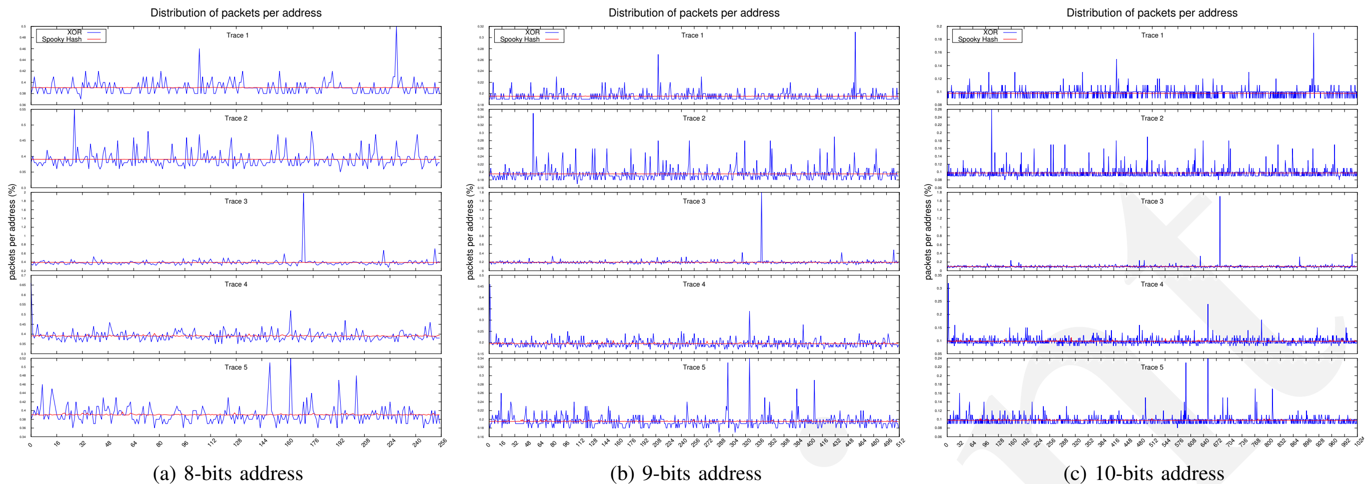
BRAM-based Shift register			packet size (bytes) 60	760	1514	Used BRAMs
Depth	Stages	Elements	Sliding Window (ms)			
256	64	16,384	0.11	1.02	2.01	128
512	64	32,768	0.22	2.05	4.03	128
1024	64	65,535	0.44	1.92	3.77	128
1024	75	76,800	0.51	4.81	9.44	150
1024	78	79,872	0.53	5	9.82	156

Besides, we completed several implementations of our architecture, varying the Depth and Stages parameters. Table II presents the results for five different combinations of Depth and Stages parameters for which timing constraints could be met. We have highlighted our initial goal of 65,536 elements, but a larger number of elements is also possible if more stages are used. The selected configuration of BRAMs is 10 address bits and 72 data bits, so memories are underutilized if the Depth parameter is less than 1024. Additionally, 17 bits out of the total 72 bits of data are not used; these bits could be use to store an extra hash in order to reduce the false positive probability. FFPGA resource utilization is for all combinations under 10 % except for BRAMs. Therefore, there are enough resources to include additional features, such as a traffic filters [23].

VII. Conclusion and Discussion

In this paper, we have introduced a new FPGA-based architecture capable of detecting and removing duplicated packets from a mirror port in a 100 Gbit/s Ethernet network traffic monitoring system. Deduplication is important, because it means less CPU load and less storage requirements at the network monitoring equipment. The architecture uses packet hashing and a element-based sliding window. For the largest studied scenario, the size of the sliding window corresponds to of 0.53 ms with minimum-size packets, or 9.82 ms with maximum-size packets. According to the bibliography, such times are big enough to successfully remove switching-based duplicates. We presented a novel scheme of memory that we have named BRAM-based shift register. This memory scheme allows us to keep N shift registers of M stages with M elements simultaneously available. Additionally, we used the two ports of BRAMs in order to access two different shift registers in parallel and thus double performance.

Overall, both theoretical and experimental results confirm that the proposed architecture can be used to detect and remove packet duplicates. According to the conclusions obtained from



(a) 8-bits address (b) 9-bits address (c) 10-bits address

Fig. 5: Packet distribution in the rows using different hash functions and address sizes in used network traces. Blue lines show the distribution with the naïve XOR hash; red lines show the distribution with Spooky hash.

Fig. 5, our future work will focus on finding a better hash table, which should be as well able to work at 100 Gbit/s. Additionally, we think that we could use a chain of Deduplicate modules to increase the number of stored elements without jeopardizing the performance of the architecture.

ACKNOWLEDGMENT

This work was partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under the project TRÁFICA (MINECO/FEDER TEC2015-69417-C2-1-R), and by the European Commission, under the dReDBox (grant agreement No. 687632) and METRO-HAUL (grant agreement No. 761727) projects, both of the H2020 programme.

REFERENCES

[1] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 219–230.

[2] I. Ucar, D. Morato, E. Magana, and M. Izal, "Duplicate Detection Methodology for IP Network Traffic Analysis," in *Measurements and Networking Proceedings (M&N), 2013 IEEE International Workshop on*. IEEE, 2013, pp. 161–166.

[3] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," in *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*. IEEE Computer Society, 2011, pp. 12–23.

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

[5] Z. Qian and M. Margala, "Low power RAM-based hierarchical CAM on FPGA," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE, 2014, pp. 1–4.

[6] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[7] W. Jiang, "Scalable Ternary Content Addressable Memory implementation using FPGAs," in *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*. IEEE Press, 2013, pp. 71–82.

[8] C. A. Zerbini and J. M. Finochietto, "Performance Evaluation of Packet Classification on FPGA-based TCAM Emulation Architectures," in *Global Communications Conference (GLOBECOM), 2012 IEEE*. IEEE, 2012, pp. 2766–2771.

[9] M. J. Lyons and D. Brooks, "The design of a bloom filter hardware accelerator for ultra low power systems," in *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*. ACM, 2009, pp. 371–376.

[10] A. Technology, "ANIC Features Overview," https://accoladetechnology.com/whitepapers/ANIC-Features-Overview.pdf, Tech. Rep., 2017, accessed: 2017-08-02.

[11] M. Molina, S. Niccolini, and N. Duffield, "A Comparative Experimental Study of Hash Functions Applied to Packet Sampling," in *International Teletraffic Congress (ITC-19), Beijing*, 2005.

[12] W. Shi, M. H. MacGregor, and P. Gburzynski, "An Adaptive Load Balancer for Multiprocessor Routers," *Simulation*, vol. 82, no. 3, pp. 173–192, 2006.

[13] "Hash Collision Probabilities," http://preshing.com/20110504/hash-collision-probabilities/, accessed: 2017-07-19.

[14] Z. István, G. Alonso, M. Blott, and K. Vissers, "A Hash Table for Line-Rate Data Processing," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 2, p. 13, 2015.

[15] M. Ahmadi and S. Wong, "Hashing Functions Performance in Packet Classification," in *Proceedings of the International Conference on the Latest Advances in Networks (ICLAN07)*, 2007, pp. 127–132.

[16] A. Fiessler, D. Loebenberger, S. Hager, and B. Scheuermann, "On the Use of (Non-) Cryptographic Hashes on FPGAs," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2017, pp. 72–80.

[17] S. Pati, R. Narayanan, G. Memik, A. Choudhary, and J. Zambreno, "Design and Implementation of an FPGA Architecture for High-Speed Network Feature Extraction," in *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*. IEEE, 2007, pp. 49–56.

[18] Xilinx All Programmable, "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

[19] Xilinx All Programmable, "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

[20] "Trace Statistics for CAIDA Passive OC48 and OC192 Traces," https://www.caida.org/data/passive/trace_stats/, accessed: 2017-08-02.

[21] B. Jenkins, "Hash Functions," *Dr Dobbs Journal*, vol. 22, no. 9, pp. 107–+, 1997.

[22] A. Kleen and J. Layton, "Spooky-c," https://github.com/andikleen/spooky-c.git, 2017.

[23] M. Ruiz, G. Sutter, S. López-Buedo, and J. E. López de Vergara, "FPGA-based Encrypted Network Traffic Identification at 100 Gbit/s," in *ReConFigurable Computing and FPGAs (ReConFig), 2016 International Conference on*. IEEE, 2016.