Towards 100 GbE FPGA-based Flow Monitoring

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

***Abstract*—This paper explores the problem of flow metering in 100 GbE links, presenting a flow exporter architecture based on a FPGA acceleration card using only on-chip memory. Peak performance without packet sampling even at the maximum packet rate is assured and means to avoid data loss are provided, since a low level of aggregation is achieved. This is the first approach in a series of architectures that are built upon the previous one, where the resources of the custom hardware are gradually increased, improving the aggregation level, while the required commodity hardware resources for subsequent stages are consequently lowered. We consider that FPGA-fabric offers adequate flexibility and performance for this task and is capable of reducing overall system cost. A functional prototype of the system has been implemented on the Xilinx VCU118 development board configured to export TCP sessions records. This achievement represents a cornerstone of a 100 GbE FPGA flow exporter design, that aims for supporting in the order of tens of millions concurrent flows.**

***Index Terms*—networking, packet processing, TCP flows.**

I. Introduction

Network traffic monitoring is required for performance assessment, traffic classification and the detection of problems, such as congested or broken links, as well as Distributed Denial-of-Service (DDoS) attacks. This information, in turn, is used in the high-level management of routers, anti-virus and firewalls.

The question is whether, in the near future, commodity hardware will still be suitable or not for tracking traffic in a highly aggregated node of the network. This doubt arises when we observe that, for example, high-speed traffic recording and subsequent packet-by-packet analysis is not practical any more, whereas it was a common practice in the past.

Alarmingly, this situation is likely to worsen in the coming years taking into account that Nielsen's law has held steady for more than 30 years now, in opposition to the rate of improvement of processor performance, which, according to David Patterson and John Hennessy, has been decreasing over the last years [1].

Others are more optimistic, quoting Peter Denning and Ted Lewis [2], "*Data parallelism further assures us we can grow systems performance as long as the workloads have sufficient parallelism*". Unfortunately, there are on-line traffic analyzes in which the workload balance across cores cannot be assured [3]. Besides, for this type of analysis, a high-bandwidth, low-latency memory system is mandatory, so by adding more cores we might not only reach the point of diminishing returns, but also see a detriment in performance because of memory access overhead.

To address this problem, we consider that we should focus on both the deployment platforms and the analysis algorithms. Keeping this in mind, we consider systems that aggregate traffic in flows, from which it is possible to gather information such as the status of connections, the bandwidth utilization, Round-Trip Time (RTT) or analyze attacks on a server [4]–[6]. This technique, at the expense of losing per packet information, has the advantage of requiring considerably less memory resources. After detecting the characteristics of the anomaly, for example, we can filter the desired traffic for in-depth investigation.

To aggregate flows in high-speed links we still need to have a great computational power and a high-bandwidth, low-latency memory. Likewise, the flexibility is an important factor these days, due to the rate at which new systems and protocols are developed. Therefore, and in line with David Patterson who states: "*For general-purpose applications, we have run out of ideas for making them faster. The path forward is domain-specific architecture*"[1], taking that into account, we delve deep into FPGA-based architectures for flow metering.

FPGAs enable the use custom-designed hardware in cases in which would be inviable with Application Specific Circuit Technologies (ASICs) due to their high costs, and, at the same time, they provide a greater flexibility given the possibility of updating the implemented hardware. We estimate that an implementation using a XCVU9P FPGA and DRAM memory can support in the order of tens of millions concurrent flows. More expensive, but already available FPGA with integrated High Bandwidth Memory (HBM) can further increase performance, due to massive increase in bandwidth.

The rest of this article is structured as follows: In section II an introduction to flow monitoring is presented, followed by a discussion of the state-of-the-art of flow export systems in section III. Then, in section IV, the flow exporter hardware constrains are analyzed, while the description of the proposed architecture is in section V, and section VI details the proof of concept specifications and results. Finally, in section VII, the contributions of this paper are summarized and future work is proposed.

[1]https://www.computerworld.com/article/3209724/computer-processors/cpu-architecture-after-moores-law.html

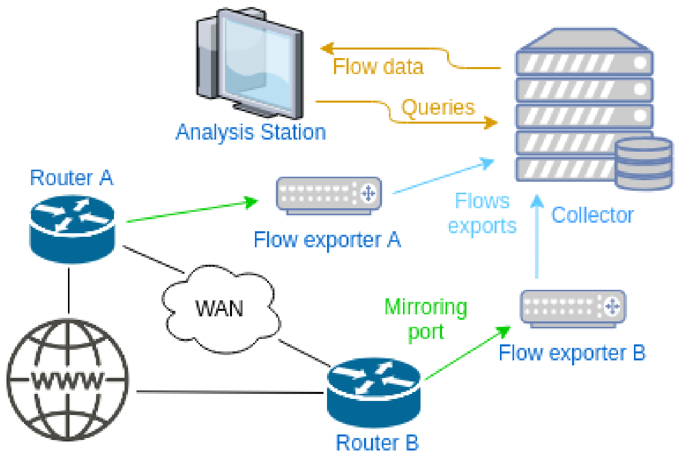


Fig. 1: Typical flow monitoring system.

II. Flow monitoring

According to the IPFIX standard [7], a flow is defined as "*a set of packets or frames passing an observation Point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties*". Typically, IP addresses, source and destination ports and transport protocol among others are chosen as part of the key that identifies a flow. This is how Netflow v5 defined it [8], the first widespread protocol for flow export, although since 2013 IPFIX has become the Internet standard for this purpose. In the latter, bidirectional flows (biflow) [9] are contemplated, comprised of those streams that connect the same points but have opposite direction.

A flow monitoring system consist of three components, which are shown in Fig. 1. The flow exporter creates flow records from the traffic going through the observation point and then sends these records to one or more flow collectors, where they are stored and processed. Finally, an analysis application inquiries the collectors and analyzes the data. Probes can be placed in different nodes of the network, normally in high-aggregation links are chosen.

The flow table is the memory that contains the records of active flows. Every time a packet is received from a new flow, an entry is created in the table, which will be released when the flow record is transmitted, that is, the flow is exported. Most commonly, flows are exported for the following reasons [10]:

- *Inactive timeout*: A flow is said to be active if at least one packet belonging to it is received in the last T_i seconds. When a flow turns inactive, it is exported. T_i is usually set by default to 15 seconds.
- *Active timeout*: If a flow remains active for more than T_a seconds, it is exported.
- *Resource constrains*: For example, if a new flow entry needs to be created and there is no place in the memory to store it, one of the flows that generate the conflict is exported. This situation is called a collision.

In practice, other reasons are also used to export flows, such us:

- *TCP flow control*: If a FIN flag in each direction or an RST flag in any of them is received, the flow is exported.

To go deeper into the topic of flow monitoring, there is a tutorial in [11] that describes the topic in depth.

III. State of the Art

The aggregation of traffic in flows is usually carry out in routers and switches, taking advantage of the resources they already have, as is the case of Cisco's NetFlow [12]. However, they usually use packet-sampling techniques, which are a function of their level of congestion. This leads to data loss, however several researchers are looking for ways to reduce it [13].

As mentioned before, higher speed networks (10 Gbit/s and beyond) are becoming more frequent, especially in data centers, making this task even more challenging. Therefore, if a greater precision in the analysis is desired, a dedicated system is required. Several implementations of flow exporters have been proposed using CPUs, GPUs, FPGAs and hybrid versions, distributing the load of the system in different ways among these devices. Down below are mention some of this implementations.

Marco Forconesi *et al.* [14] presented an architecture to export flows in 10 GbE networks based on the NetFPGA-10G platform, which is capable of handling the maximum packet rate (14.88 Mpps) without sampling and up to 786,432 concurrent flows using a SRAM memory for the flow cache.

Paula Roquero *et al.* in [15] proposed a CPU-GPU flow generator for 10 GbE links system capable of obtaining complex information related to the TCP protocol, such as the detection of retransmissions with memory at the packet level, achieving a processing rate of up to 4.4M packets per second (Mpps).

Viktor Puš *et al.* in [16] deployed an exporter running on a 20-core server with a custom FPGA (Virtex-7 H580T) based Network Interface Card (NIC), capable of supporting the export of flows from 100 GbE links. Here, according to the result of a hash function applied to header fields of the packets, the NIC injects the packets into 16 queues allocated in main memory (64 GB DDR4) of the two E5-2660v3 CPUs.

In the latter case, the offloading provided by FPGA enables the system, which makes us consider whether the CPU continues to perform tasks for which the FPGA is more suitable, and hence reducing the cost of the system (since a lower performance server would be required) or enabling support of higher speed links.

For this reason, we decided to explore the offloading of the packet aggregation using FPGAs. This work is the continuation of the previous work [17], where we presented a 40 GbE flow exporter. Here, a 100 GbE version is described, which is implemented in the Xilinx VCU118 development board, which underneath uses a XCVU9P FPGA.

IV. Problem statement

A. Scenario

The hardware system for flow aggregation offloading may look like fig. 2. The pre-processing comprises the packet level operations, while in the flow metering, the flows are created. Optionally, flow inspection and filtering might take place before they are send to the output, where, depending on the destination of the exported flows, they have to be processed accordingly. Also, mains to control the hardware might be desired (dashed arrows).

If the flows are sent over the network, packetization should be compliant with a flow export protocol such as the IPFIX standard. Another option is to implement the system as a NIC, similarly to [16], in which case a DMA controller could be used to write in the main memory of the server, distributing the load between the cores properly.

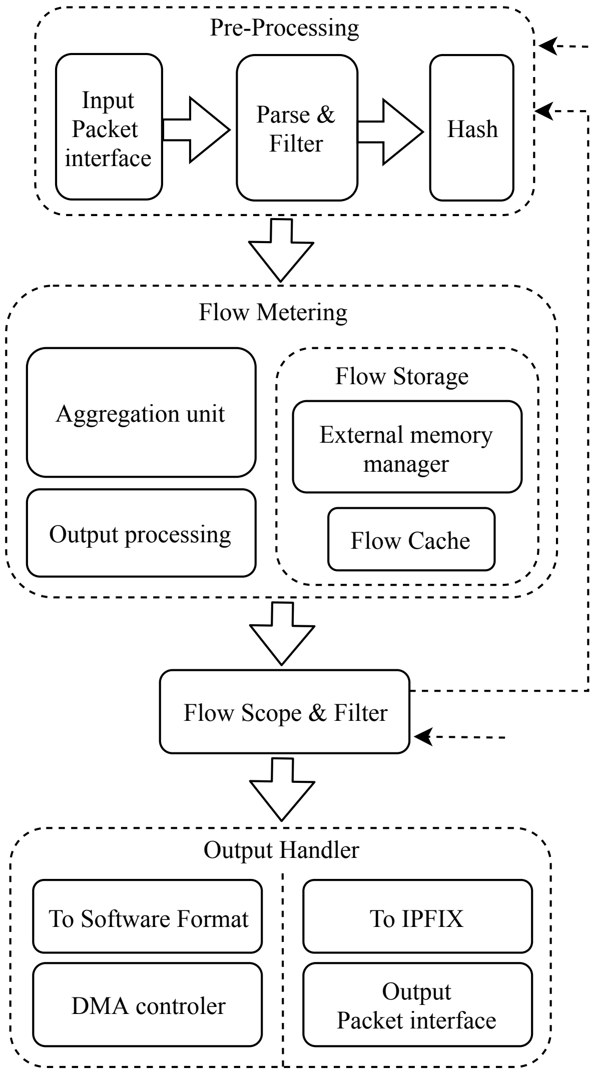


Fig. 2: Hardware accelerator

B. Timing constrains

Inside the FPGA, a transaction with the input packet interface may carry data from more than one packet since they do not need to be align to the bus width. Considering that processing this transaction would increase the complexity of the hardware, as a first approach, we will consider that a module care of this, presenting a new packet in a different transaction. However, by doing this, we are reducing the bus throughput, having the worst case scenario when it is required the minimum possible transactions for a packet with just one byte of data in the last transaction.

Because of this alignment plus the fact that there is a minimum packet size, the trade-off between bus size and clock frequency is not as straightforward as a simple serial to parallel conversion. Taking into account the specifications of the Ethernet protocol, we arrive at the following formula, which relates the maximum link speed supported by a 512 bits wide bus with the frequency F at which it is clocked:

$$F = \textit{Link speed}/356 \tag{1}$$

If a 100 GbE link is to be supported, the formula shows that the internal bus must be clocked at least at 280.9 MHz, which is a feasible frequency for the state-of-the-art FPGAs.

Moreover, it is necessary to know the maximum packet rate. Considering the same assumptions as before, we arrive to:

$$Min\left[\frac{\textit{clock cycles}}{\textit{packet}}\right] = \left\lfloor 672 \cdot \frac{F}{\textit{Link speed}} \right\rfloor \tag{2}$$

Replacing F by 280.9 MHz and a link speed of 100 Gbit/s $min[cycles/packet]$ is equal to $\lfloor 1.88 \rfloor = 1$. Particularly, we decided to increase the clock frequency of the bus to 300 MHz in order to get in the worst case one packet every two clock cycles since the frequency is still adequate and this will ease restrictions such as the number of memory ports available.

C. Memory constrains

Let us first assume that it is desired to offload completely the flow metering process. In this case, the memory system should have enough capacity and flexibility to host the maximum amount of concurrent flows. At the same time, we should be able to read and update it every 6.72 *ns* in the worst case. This case presents when all the packets in the link have the minimum size and all of them belong to different flow, reaching over 2.23 Giga concurrent flows for an idle timeout of 15 seconds.

However, [18] indicates that the ratio of number of flows to the volume of the traffic in Mb/s tends to be between 100 and 1000 when the idle timeout is 15 seconds. For a full saturated 100 GbE link, this would imply between 5.5 Mega and 100 Mega concurrent flows. In order to obtain a ball-pack number, let the flow entry be 128 bytes wide. In this case, the capacity of the memory system should be around 7 GiB.

Regarding bandwidth, if it is assumed that just one entry is read and updated every time a packet arrives, it would be required to read at a rate of 17.73 GiB/s and write at 17.73

GiB/s. Additionally, it has to be added the bandwidth required for the exporting process, which is dependent on the flow table structure. Assuming a trivial memory scan every second, it would require to read at a rate of 7 GiB/s.

Taking this into account, DDR4 memories seem to be an adequate alternative. Yet, the latency of this memories plus the latency of the memory controller is in the order of hundreds of nanoseconds according to our benchmarks. Nevertheless, they may still be suitable if we design a way to mask this latency, for example, using a cache memory and pre-fetching.

For the cache to make sense, the probability distribution of the memory address required needs to be different to the uniform distribution, and it needs to be known to a certain degree to achieve the appropriate cache hit rate. Considering how the stack protocols work, it is expected to see temporal locality of flows and between opposite flows, that is, with in biflows. For this reason, we contemplate using on-chip memory as a N-way cache and DDR4 for the main memory.

To know to degree of the temporal locality, if any, we conducted a series of cache memory simulations ranging from a capacity of 1 KiFlows (2^{10} flows) to 1 MiFlows (maximum cache size of Cisco's Netflow [19]), contrasting different levels of cache associativity and uniflows versus biflows. Every time a new flow was observed, an entry was created in the cache. If there was a collision, the flow closest to be expired was removed, as if it was displaced to main memory. In this way, for each collision we would have a write and a read to this memory.

Regarding the configuration of the simulated system, a hash function was used to map flow to cache addresses, employing Jenkins's one at a time for flow uniflows and xor operations for biflows. The in active timeout was set to 15 seconds and there was no active timeout.

As we did not have access to any 100 GbE trace, we created one by mixing ten traces provided by CAIDA, captured during 2018 from a 10 GbE link [20]. Traces of both directions of five months were merged after editing their timestamps in order to appear that the captures started at the same time. Figure 3 shows the results of these simulations, where the vertical axis is the percentage of packets that produce a collision and the horizontal is the cache capacity in number of flows.

It is observed that even for the highest capacities there is a collision approximately every 5 packets, which is not enough to obtain the necessary average access time. To achieve this, we might use other latency masking techniques such as out of order processing.

However, another approach can be followed noticing that, although the cache memories that can be fit in the FPGA are not capable by themselves to enable the use of the DDR4 memories, with a capacity of a few tens of Kiflows they achieve an aggregation of two, that is, on average the entries taken out of the cache have information of two packets. Therefore, instead of completely offloading the metering process, as first approach, we could simply send the collided flows to main memory of a server, where a bigger flow table will be maintained. In this way, the server load would be greatly

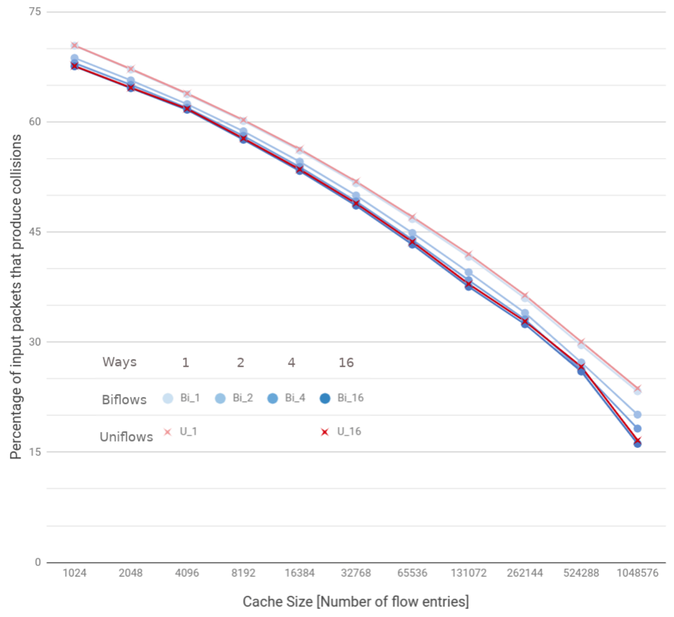


Fig. 3: Percentage of collisions vs cache size

reduced compared to the case of having to process each packet directly since it would have to process on average half the number of elements, and no packet parsing would be required.

As for the temporal locality of response flows in the cache, for this type of link, uniflows should be preferred over biflows, when the entry size for the latter are approximately 15% bigger than the former's. After analyzing the traces, it was observed that the opposite flow seemed to be routed differently, which may not happen to the same extent in other nodes of the net.

Regarding the level of associativity of the cache, it is observed that, on average, the collision percentage decreases in 1.8% when a 2-way cache is used instead of a direct mapped. Then, from 2 to 4 ways we there is a 0.9% increase, from 4 to 8 a 0.4% and from 8 to 16 a 0.2%.

Concerning flow tables with more than 64 MiFlow entries, the ratio converges to 7% and 7.16% for biflows and uniflows, respectively. For this sizes and the described set up, there are almost no collisions when the level of associativity of the memory is equal or greater than 4, confirming the hypothesis on the number of concurrent flows. Further experiments should take place to be able to extrapolate these results.

D. First Approach

Along what has been exposed, hereafter is presented a flow exporter architecture implemented on an FPGA-fabric using only on-chip memory able to achieve a coarse aggregation in 100 GbE links, which is meant to be completed in commodity hardware. It was design as a point in a series of architectures that are built incrementally upon the previous one — making the project more manageable—, and looking to explore different degrees of offloading of the flow metering process with the aim of reducing overall system cost and, possibly, enabling this processing in future high-speed links.

V. Proposed Architecture

A. High Level Description

The system has a pipeline architecture to maximize the processing rate and take advantage of FPGA capabilities. Additionally, this enables to achieve a great degree of decoupling between the different modules of the system, which facilitates the independent development of each of them, provided that the protocols and interfaces are respected.

The first stage is the *Parser*, which filters packets and extracts information from them. Then, the *Hash* function computes the address of the cache flow, where the records are stored. The *Data Updater* module is responsible for updating the information of a flow entry each time a new packet that belongs to it arrives. The *Exporter* continually examines the table for expired flows, which will be passed to the output interface.

B. Pipelined vs Multicycle Stage

Throughout the design, when the input rate of a module was equal or lower that one packet every two cycles, multi-cycle paths (MCPs) were preferred over pipelined stages for several reasons:

- MCPs allow higher frequencies even if the pipeline is perfectly balanced because there is no time spent in the set up and propagation times of the pipeline registers.
- Since pipelines are rarely perfectly balanced, the maximum allowed frequency of a design is further reduced.
- There is no need to perform the task of balancing the pipeline, which can be a non-obvious procedure and is technology dependent, hence making harder the porting of the design.
- MCPs have lower resource requirements, since no additional registers are needed
- MCPs allow some logic optimizations. Synthesis tools are able simplify the logic that would be otherwise divided by the registers, which leads to further increase in clock frequency and area reduction.

C. Parser

As stated previously, this module analyzes the incoming frame, decides to accept it according to the protocols it uses and extracts or computes the necessary fields for the chosen flow records. Above 40 GbE traffic processing, we were not able to close timing in a single stage parser with features such as: multiple VLAN tags support and trivial computations.

In order to obtain a parser clocked at 300 MHz, a pipelined version was designed, where the alignment and dissection tasks were separated into different stages. The proposed transfer protocol between the parser's stages is as follows: Each stage appends the extracted data to the received from the previous one — in AXI4-Stream this would be done in the TUSER field —, and setting to not valid the analyzed bytes of the packet when it is presented to the output interface. Then, if these bytes are no longer required, a packet cutter might be inserted. In AXI4-Stream TKEEP and TSTRB may be used to control this operation.

If after each stage the packet is aligned, cutting the previous header, non-optional fields will be found in fixed positions, hence the necessary hardware to extract information is considerably reduced. For our needs, we need a module that aligns the parsed packets by cutting the first $32 \cdot n$ bits, were n is an integer. Thus, again, aiming to support 300 MHz transactions, a barrel-shifter alike method is used, needing $log_2(bw/32) = log_2(512/32) = 4$ stages, where bw stands for bus width in bits.

D. Hash

The *Hash* module is responsible for mapping flows ID to addresses of the flow cache. The choice of the function to be implemented mainly pursues two objectives:

- Obtain an approximately uniform distribution of addresses, reducing collisions.
- In the case of using biflows, obtain the same address from both tuple IDs that belong to the flow

Additionally, for memory systems with hierarchy, it is desirable to have a low latency function, since the hash is required for data management and, in this way, it would avoid the need to store it.

E. Flow Cache

The system uses on-chip true dual port memories to implement the flow cache, one port for the *Data Updater* module and another for the *Exporter*. Each memory address can store more than one flow entry, distinguishing each entry by ways as a N-way associative cache memory. Thus, the effective memory capacity increases and the probability of collisions decreases, as it was observed in the simulations.

F. Data Updater

This module condenses all the information coming from packets of a flow in a set of registers — entry of the flow table—, thus reducing its output information rate. It is capable of processing a new packet every two clock cycles as long as the flow cache has an access time of one clock cycle or less. To obtain the adequate performance, two clock cycles are allocated for updating an entry, using MCPs to increase the maximum length allowed for combinational paths. This enables the ALU to perform more complex computations.

Figure 4 shows the relationship between the *Exporter* and the *Output Interface* with this module, as well as the two stages that comprised it: *Pre-fetch* and *Actualization*. Despite of being a non-linear pipeline, these stages have a large degree of decoupling thanks to the defined communication protocols.

For instance, the memory is configured in "no change" mode, which means that the output information of the memory does not change when performing a write operation. Taking into account that the actualization stage only performs write operations, the pre-fetch stage is the only one capable of changing the data output of that port, thus behaving just like another output register of this latter stage. This fact is important since, thanks to this, the necessary coordination —

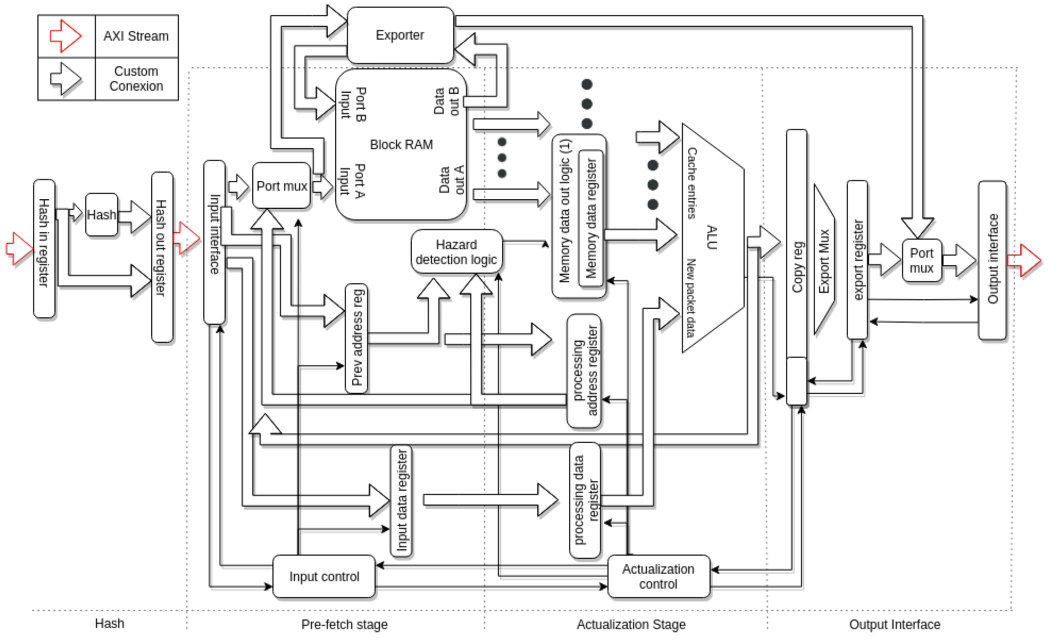
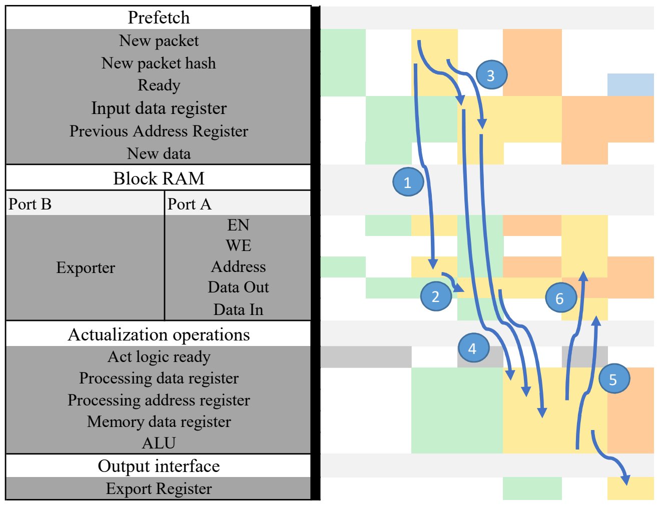


Fig. 4: Data updater architecture.



The data movements for each packet are distinguished by color.

Fig. 5: Data updater timing.

and therefore the degree of coupling — between these two modules decreases considerably.

Another notable point of this design is the low level coupling between the control and the data path, enabling this module to handle any database just by changing the ALU since the control is agnostic of the stored data.

Timing: Figure 5 shows how packets are processed by this module. When new valid data is presented to its input, the pre-fetch stage uses the hash to address the memory (arrow 1 in Figure 5), obtaining the entries stored in table in the next cycle (arrow 2). The parsed information of the packet and the hash outputted synchronously with the table entries (arrow 3).

When the actualization stage accepts the information, and as stated before, two clock cycles are assigned to the creation or update of the entry despite being a combinational process, given the great number of computations performed in parallel, each with several levels of logic. In addition to this, a large routing delay has to be taken into account, since this stage ends updating the flow table (arrows 5 and 6), which is implemented in BRAM, which will be disaggregated over the FPGA fabric.

Hazard of data corruption: As it can be seen in figure 4, the *Data updater* comprises a non-linear pipeline (one port of the cache memory it is used in two stages of the pipeline). Consequently, there is a read after write (RAW) data hazard, which is detected and corrective measures are applied.

G. Exporter

As its name suggest, this module scans the memory using the second port of BRAMs, comparing the timestamps against the current time looking for expired flows and exporting them. Since the timestamps will eventually overflow, the comparison for serial numbers defined in the RFC 1982 section 3.2 was used. The task of exporting is less critical than the actualization of an entry, so to improve the performance of the system, data is addressed and gathered using multi-cycle paths to decrease congestion around the memories and to enable the *Data updater* to be closer to the memory tiles, and thus, run at higher frequencies.

It is important to highlight that the exporter needs to monitor the operations performed in the port used by the *Data updater* to avoid data corruption. When the exporter reads a memory entry and decides to export it, this entry is erased. Since this operation is not atomic, if the *Data updater* reads that memory entry between the exporter read and write operations, the memory retrieves the expired flow data, which should be an empty entry. There is also a similar hazard if the exporter reads a memory entry while the *Data updater* is processing it.

For this reason, if this situation is detected, the exporter waits until there is no more hazard.

H. Timestamp Expander

The *Timestamp Expander* allows using less bits internally while exporting long timestamps, increasing the amount of flows entries that can be fit in memory and reducing the required memory bandwidth. Therefore, its task is to extend the timestamps, which can be performed provided that the internal timestamp is not too small, so it would be possible to confuse an expired flow (because of the active timeout) with one that is not because the timestamp clock overflowed more than once.

VI. Proof of concept

A. Methodology

To develop the proof of concept, VHDL was used to describe the system's modules, primarily aiming for performance through the employment of techniques, such as, long pipelines, multi-cycle paths and combinational path decoupling when it was feasible.

Additionally, it was avoided the redesign of already available and verified IPs provided by the FPGA vendor, in this case, Xilinx. As for the interfaces protocol, AXI4-Stream was chosen to intercommunicate the modules for compatibility reasons with Xilinx's IP library and, since AXI4 is a well-spread standard, this choice increases the possibilities of reusing our IPs in new systems.

Recognizing the need for greater flexibility, as stated earlier, a high level of decoupling between the control and the data path was sought, requiring only alterations in the packet parser and in the processor's ALU to change the computed information, paving the way for future projects where these operations will be defined through higher level languages, as proposed in [21].

B. Features of the proof of concept

In this proof of concept, only IPv4 packets with TCP transport protocol were accepted. Bidirectional flows were aggregated, which were defined by the IP addresses and source and destination ports, conforming a 4-tuple ID. The prototype implemented for the testing phase was configured to store up to 16,384 flows concurrently, distributed in a four-way cache.

a) Information Elements: The chosen information elements (IEs) per direction were the number of IP payload bytes, the number of packets, the or-reduction of RST, SYN and FIN TCP flags, an estimation of the number of retransmissions and the timestamp of the first packet. For both directions, the timestamp of the last packet received was introduced. Given the possibility of collisions, the exported record contains the necessary information to complete the aggregation process in the subsequent stages.

b) Parser: In conformity with the IEs, the parser extracts the 4-tuple, the sequence number and the flags of the TCP header. Additionally, the module applies a timestamp, computes number of bytes of IP payload and the sequence number of the last byte.

c) Hash: The flow to memory mapping, is performed computing an XOR to the 4-tuple, with a symmetry such that the result is independent of the stream direction. We notice that this hash does not distribute as well as others in the state-of-the-art. The study and implementation of better hashes is left for future work.

d) Timestamps: The timestamps used internally by the system are 32 bits wide, 20 bits for microseconds and 12 bits for seconds. Although it would be interesting to reach a greater precision than μs, this magnitude has been chosen given the limitations of available memory, and the fact that the network delay is mostly above 100 μs [22].

e) Retransmission detection: An indicator that there is a problem in the network is that there are excessive retransmissions in a TCP connection, reason why it was proposed to implement an algorithm that detects them. For this, a simple method was used, used in [23], which is based on the hypothesis that the TCP segments arrive in an orderly manner (according to the order of delivery). This hypothesis is not necessarily true, given that not all packets travel through same path. However, according to [24], the number of packets arriving disorderly is usually very low (less than 0.07%). To detect a retransmission, the sequence number of the last byte received is compared with the first byte of the new segment.

C. System debugging and verification

To verify the correctness of the system, simulations and field tests were carried out. Due to the difficulty to obtain patterns to compare the results, tests were first carried out with synthetic traffic aiming to check specific parts of the system, and then, a more realistic work regimen was tested using real traces and comparing the results with those provided by a well-known tool for traffic analysis, Wireshark [2].

D. Field tests

A test system was implemented on the VCU118 Evaluation Board [25] (FPGA: Xilinx Virtex UltraScale+ 9P). Given that 100 Gbit/s traffic generation was not available at the time of the tests, the system was verified at 10 Gbit/s, and this work is left for when the appropriate equipment is available to us. We employed a server with two 10 GbE interfaces, one for traffic generation and other for output gathering. Traces of different characteristics were used, first simple ones to verify specific behaviors and then, traces of greater volume and level of traffic aggregation.

After analyzing the behavior of the system in simulation and field tests, we can say that the system works according to specifications. Small errors were observed in the number of retransmissions, due to some disorder of the packets, and some differences in exported flows were spot, due to collisions (which can be rectified by adding partial exports, since as it was mentioned, there is not any information lost). Nonetheless, this behavior was as expected.

[2]https://www.wireshark.org/

TABLE I: Resource usage.

Module	LUT	FF	BRAM
Parser	107	422	0
Data Updater	6897	4925	228
flow cache	0	0	228
Exporter	1002	912	0
Total hardware accelerator	8063	7405	228
Total hardware accelerator (%)	0.7	0.3	10.6
Total testing system (%)	3.9	4.1	14.4

Yet, it is considered that more rigorous tests should be carried out regarding the marking of times, given that the pattern that was available is not sufficiently precise.

E. Specifications of implemented hardware

As mentioned before, the designed system was implemented with a clock of 300 MHz, which (based on equations 1 and 2) enables to support the speed of links of 100 GbE, given its capacity to process a new packet every two clock cycles. Regarding the resources used in the implementation of the designed system and the complete test system, they are shown in Table I. It is noted that sufficient resources, such as 270 Mb of URAM, are available to create a hierarchical memory system for the flow table.

VII. Conclusions and future work

This work has presented a hardware accelerator for a flow exporter system, which is a cornerstone of a standalone flow exporter system based on FPGA, providing an efficient architecture in terms of performance to obtain a series of statistics of flows established through an Ethernet link at 100 GbE. Thanks to the decoupling achieved between the data path and the control logic, this architecture can be reused for the computation of different statistics to those implemented and even under a different flow of information without making structural changes.

It is considered that there is still a long way to go, leaving for future work the field verification at 100 GbE, the study of optimal hash functions. Likewise, we will seek to create a hierarchical memory system for this architecture to be capable of supporting the order of tens of million concurrent flows.

Acknowledgments

This work has been partially funded by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund with the TRÁFICA project (MINECO/FEDER TEC2015-69417-C2-1-R), and by the European Commission with the H2020 METRO-HAUL project (Grant Agreement No. 761727).

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann, 2016.

[2] P. J. Denning and T. G. Lewis, "Exponential laws of computing growth," 2017.

[3] C. Vega, P. Roquero, and J. Aracil, "Multi-gbps http traffic analysis in commodity hardware based on local knowledge of tcp streams," *Computer Networks*, vol. 113, pp. 258–268, 2017.

[4] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of ip flow-based intrusion detection." *IEEE Communications Surveys and Tutorials*, vol. 12, no. 3, pp. 343–356, 2010.

[5] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational ip networks: Methodology and experience," *IEEE/ACM Transactions on Networking (ToN)*, vol. 9, no. 3, pp. 265–280, 2001.

[6] A. Callado, C. Kamienski, G. Szabó, B. P. Gero, J. Kelner, S. Fernandes, and D. Sadok, "A survey on internet traffic identification," *IEEE communications surveys & tutorials*, vol. 11, no. 3, 2009.

[7] B. Claise, B. Trammell, and P. Aitken, "Specification of the ip flow information export (ipfix) protocol for the exchange of flow information," Tech. Rep., 2013.

[8] (2019, February) Cisco Netflow Collection Engine. [Online]. Available: https://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/products_implementation_design_guide09186a00800d6a11.html

[9] B. Trammell and E. Boschi, "Bidirectional flow export using ip flow information export (ipfix)," Tech. Rep., 2008.

[10] G. Sadasivan, N. Brownlee, B. Claise, and J. Quittek, "Architecture for ip flow information export," Tech. Rep., 2009.

[11] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.

[12] (2019, February) Flexible NetFlow Configuration Guide . [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/fnetflow/configuration/xe-16/fnf-xe-16-book.html

[13] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 270–313, 2003.

[14] M. Forconesi, G. Sutter, S. Lopez-Buedo, and J. Aracil, "Accurate and Flexible Flow-based Monitoring for High-speed Networks," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–4.

[15] P. Roquero, J. Ramos, V. Moreno, I. González, and J. Aracil, "High-speed TCP Flow Record Extraction Using GPUs," *The Journal of Supercomputing*, vol. 71, no. 10, pp. 3851–3876, 2015.

[16] V. Puš, P. Velan, L. Kekely, J. Kořenek, and P. Minařík, "Hardware Accelerated Flow Measurement of 100 Gb Ethernet," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE, 2015, pp. 1147–1148.

[17] T. Alonso, M. Ruiz, G. Sutter, C. Sisterna, S. López-Buedo, and López, "Monitorización con FPGAs de flujos y sesiones TCP en enlaces de 40 Gbit/s."

[18] J. L. García-Dorado and J. Aracil, "Flow-concurrence and bandwidth ratio on the internet," *Computer Communications, 2019. To appear.*

[19] (2019, February) Cisco IOS Flexible NetFlow Command Reference. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios/fnetflow/command/reference/fnf_book/fnf_01.html

[20] "The CAIDA UCSD Anonymized Internet Traces - 2018," http://www.caida.org/data/passive/passive_dataset.xml, accessed: 2019-02-06.

[21] J. F. Zazo, S. Lopez-Buedo, G. Sutter, and J. Aracil, "Automated Synthesis of FPGA-based Packet Filters for 100 Gbps Network Monitoring Applications," in *ReConFigurable Computing and FPGAs (ReConFig), 2016 International Conference on*. IEEE, 2016, pp. 1–6.

[22] M. Trevisan, D. Giordano, I. Drago, M. Mellia, and M. Munafo, "Five years at the edge: watching internet from the isp network," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. ACM, 2018, pp. 1–12.

[23] E. Miravalls-Sierra, D. Muelas, J. Ramos, J. E. López de Vergara, D. Morató, and J. Aracil, "Online detection of pathological TCP flows with retransmissions in high-speed networks," *Computer Communications*, vol. 127, pp. 95–104, Sept. 2018.

[24] D. Murray and T. Koziniec, "The state of enterprise network traffic in 2012," in *Proc. 18th Asia-Pacific Conference on Communications (APCC)*, 2012.

[25] (2018, October) Xilinx virtex ultrascale+ fpga vcu118 evaluation board. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/vcu118.html