Master Thesis

Accurate and Flexible Flow-Based Monitoring for High-Speed Networks

Author:
Marco Forconesi

Supervisor:
Gustavo Sutter

High-Performance Computing and Networking Research Group
Escuela Politécnica Superior, Universidad Autónoma de Madrid
(Spain)

September, 2013

Acknowledgment

To the HPCN staff, specially G. Sutter, S. López B. and J. López V.

'An absence of fear of the future and of veneration for the past. One who fears the future, who fears failure, limits his activities. Failure is only the opportunity more intelligently to begin again. There is no disgrace in honest failure; there is disgrace in fearing to fail. What is past is useful only as it suggests ways and means for progress.'

Ford H., *"My Life and Work"*, 1922.

Declaration of Authorship

I,Marco Forconesi, declare that this thesis titled, 'Accurate and Flexible Flow-Based Monitoring for High-Speed Networks' and the work presented in it are my own. I confirm that:

- This work was done mainly while in candidature for a Master degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

Contents

Summary 1

Introduction 2

Published Paper 3

Paper Revisions 11

Application Note: Integration on NetFPGA-10G . . 17

Thesis Conclusion and Future Work 30

Summary

This Master's Thesis[1] reports the project developed for the 'Trabajo de Fin de Máster' subject of the Master's Degree in ICT Research and Innovation (I^2-ICT). The project consists of a hardware prototype implementation that classifies 10 Gbps IP network traffic. The work is novel since it can monitor a higher number of concurrent communications than all previous implementations and also it analyzes every packet on the monitored link, i.e. it does not perform packet sampling.

[1]This project has been developed with the support of the High Performance Computing and Networking Research Group.

Introduction

As opposed to a monograph style, this thesis is based on a research publication submitted for the *23rd International Conference on Field Programmable Logic and Applications* that took place in Porto, Portugal last September, 2-4, 2013. The reasons why this work has not been reformatted to be presented in the traditional style are the following:

- A paper accepted to be presented at an international conference reflects the fact that it has a research and innovation component, since it is of interest to the modern research community[2].
- The writing quality of the documentation related to the work, e.g. the paper, is as good as possible since it will have more opportunities to be accepted.
- The publication is subject to international and independent reviewers who accept or reject the work depending on the work itself.
- The student agrees with the use of the modern fashion of a *'thesis by publication'* due its quality, economical and efficient advantages.

In the following the publication mentioned above, titled "Accurate and Flexible Flow-Based Monitoring for High-Speed Networks", is presented in its original full-version (7 pages). Although the paper has been accepted to be presented as a short paper (4 pages) plus a poster presentation at the conference, the full-version provides a more complete documentation of the developed work.

As a final complement to the developed project, an application document explains how to integrate and test the prototype in the NetFPGA-10G platform. This document was specifically written for the NetFPGA-10G community and for that matter the authors gave their best in order to obtain the best possible quality, as well as for the paper.

[2]This is a mandatory requirement for the 'Trabajo de Fin de Máster' subject.

Published Paper

The following document is the paper submitted to the 23rd International Conference on Field Programmable Logic and Applications. The article explains why network monitor is a need, the state-of-the-art in the topic, the reason to do the packet processing in hardware, the description of the developed architectures and the obtained results after the testing phase.

ACCURATE AND FLEXIBLE FLOW-BASED MONITORING FOR HIGH-SPEED NETWORKS

Marco Forconesi, Gustavo Sutter, Sergio Lopez-Buedo, Javier Aracil

High-Performance Computing and Networking Research Group
Escuela Politécnica Superior, Universidad Autónoma de Madrid (Spain)
email: {marco.forconesi, gustavo.sutter, sergio.lopez-buedo, javier.aracil}@uam.es

ABSTRACT

In this paper we present an architecture to classify 10 Gbps network traffic based on FPGAs using the NetFPGA platform. Flow-based network monitoring is today the most spread technology employed by engineers and administrators to analyze and detect network issues.

The main improvements of our design, compared to the state-of-the-art flow analyzer tools, come from: (i) the architecture allows to process full rate 10 Gbps links without sampling even with the shortest packets (14.88 Mpps - Million packets per second); (ii) it is possible to manage up to 786,432 concurrent flows; (iii) the project is developed in an open-source hardware platform and the code is opened to the community; (iv) it relieves the networks and its devices from the overload of monitoring process.

1. INTRODUCTION

Flow-based monitoring is the most common practice used to analyze network traffic in order to track information on resources and bandwidth utilization as well as network dysfunctionalities and attacks. The infrastructure consists mainly of two parts: the data source that analyzes the packets in the network and creates the flows, and the data collector which receives the exported flows from the data sources and stores them for future processing. A data source analyzer is typically implemented inside the network devices, such as routers and switches, and it takes advantage of its resources to analyze the network packets and store the flows' information while those are active on the link. Periodically, the removed flows from the data source are sent to the data collector through the same network that is being monitored. The communication between the data source and data collector is typically done by the use of some specific protocol like NetFlow [1] by Cisco®, sFlow [2] by Foundry®, Jflow [3] by Juniper® or IPFIX [4] an IETF standard. Those Layer 3 protocols are carried on a UDP stream from the data source to the data collector. The main advantage of monitoring a network using these devices is that is not necessary to add any extra component, since the entire packet processing is done inside the routers or switches and the communication to a data collector is accomplished by the use of the same network. Unfortunately, in the context of high-speed networks, they exhibit the following drawbacks that are not possible to mitigate:

- In highly loaded networks, the extra processing performed by routers and switches could result in too much overload. When this occurs, the flow monitoring is skipped in order to dedicate all the device's resources to route packets, thus losing all the flow-based information [5].
- In high-speed networks, the flow-based monitoring performed by routers and switches, is accomplished in the basis of packet sampling, i.e. not all the packets on the network are analyzed. This could result in poor accuracy of the data delivered to the collector.
- As stated earlier, communication between the data source and the data collector is done by the use of the same network that is being monitored. This could lead to network overloading and also to incorrect information since production traffic is mixed with network measurement traffic.
- The flows' information sent on the UDP packets to the data collector, are deleted forever from the data sources. Since UDP is not a reliable transport, if the packets are dropped somewhere in the network, then it is not possible to recover the flow-based information in anyway.
- Routers and switches have limited resources so they cannot scale to higher link-rates or larger memory capacity to store the active flows. Moreover, these network devices are closed platforms, so even if they perform flow-based monitoring, network engineers are not free to modify how flows are defined or what type of information is collected.

In order to overcome the limitations listed above, a flexible open-source platform to collect the flow-based information suitable for high-speed networks is needed. It is also desirable that the measurement traffic technique does not become intrusive on the network that is being monitored and does not impact on the network devices either. Our goal

is to improve the decision-making process of network administrators with the aid of a high-quality flow monitor tool which will supply analysis algorithms with accurate inputs.

2. RELATED WORK

Besides of commercial routers, there are many proposed probes that generate network flows at 10 Gbps and beyond. Many attempts are made based on software that use commodity hardware while others are based on reconfigurable hardware.

Software based probes: Although modern NICs (Network Interface Cards) hardware can handle high packet rates, standard operating system network stacks are still affected by a several software bottlenecks. The most important is the per-packet operations like buffer allocation and transfer to user-space. To overcome such issues, several approaches in order to bypass standard stacks are used. For example, processing multiple packets in batch to limit the number of interruption and DMA transactions, or exposing memory of packet buffers to the user-space for "zero-copy" access. The state-of-the-art in software implementations requires multicore implementations and a careful balance between cores. nProbe [6] and softflowd [7] are two very well-known and popular open-source approaches. Danelutto *et al.* [8] reports the use of several advanced parallel programming techniques in the implementation of ffProbe and compares against nprobe, sofflowd. They achieve up to near 10 Mpps for ffProbe and poorer results for the others.

Reconfigurable Hardware Probes: Žádník [9] makes a first implementation on NetFPGA-1G (a Virtex®-2 platform). This version is publicly available in the NetFPGA-1G repository and supports only 1 Gbps link-rate. The project evolved to FlowMon [10] using Virtex®-5 in the contest of liberouter project [11]. This report describes two versions of FlowMon: the first one uses the FPGA to receive the packets while the flow processing and storage is made in the host PC, the second one uses on board QDR memory for the flows' storage and the processing is done in the FPGA. The implementation is limited to 3 Mpps. Yusuf *et al.* [12] proposed an architecture for network flow analysis on FPGA. They use a Virtex®-2 device and are able to store up to 65,536 concurrent flows and a maximum below 3 Mpps. Rajeswari *et al.* [13] give some results for Virtex®-5, they claim superior speed but only for 500 concurrent flows.

We present in this paper two architectures, one using only internal BlockRAMs of the FPGA to store the flows and a more complex one that uses external QDR-II memory. Our architecture distinguishes itself from previous works in the following main aspects:

1. It can store up to $^{3}/_{4}$ million concurrent flows.
2. It allows out-of-band transport of the exported flows from the data source to the data collector for the sake of freeing the monitored network of flow-based information traffic. It can be done via PCIe or another network interface instead.
3. Communication between the data source and data collector can be implemented using any format and with a reliable transport mechanism.
4. It can analyze 10 Gbps links even in the worst-case scenario (smallest packets with shortest inter-packet gap), i.e. 14.88 Mpps with 64-bytes packets.
5. The architecture could scale up to 40 Gbps networks with the use of state-of-the-art FPGA devices.
6. The project is implemented in an open-source hardware [14] and the code is opened to the community, thus it is completely customizable depending on the network engineer's needs.

Both proposed architectures are available throughout a public repository [15] and will be discussed further.

3. FLOW-BASED MONITORING TECHNIQUE AND DEVELOPMENT PLATFORM

In this section we review the key concepts of flow-based analysis, that led to the developed architectures. Afterwards we briefly present the NetFPGA-10G platform as the hardware resource where the designs have been implemented.

3.1. Flow Cache and Network Flow as the Traffic Analysis Unit

A flow, according to Cisco's® definition, is a unidirectional stream of packets between a given source and destination [1]. Each flow is identified by five key fields, which are listed below:

- Source IP address
- Destination IP address
- Source port number
- Destination port number
- Layer 3 protocol type

Those packets with the same elements above (5-tuple henceforth) belong to the same flow. Every Ethernet frame received by the application is parsed to extract its 5-tuple and additional information, such as: timestamp, TCP flags and bytes within the frame. A fast local memory inside the core, known as flow cache, is used to store the active flows of the link that is being monitored. The data structure on the flow cache is called flow table and it consists of a list of flow records, one for each active flow. The information contained for each flow record: number of packets, total number of transmitted bytes, timestamp of the flow creation/expiration and TCP flags, is used for a later traffic analysis, once the flow is purged from the flow cache. Every time a packet is received, the memory is polled to determine if the extracted 5-tuple matches an active flow, if not, a new flow entry is

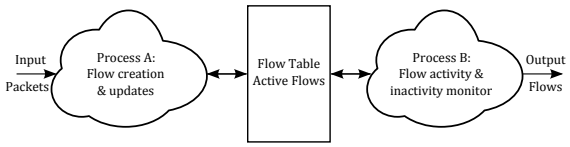

Fig. 1. Concurrent Process Running on the Flow Table.

created. Otherwise the active flow in the flow table is updated.

Parallel to the flow creation and updates, there is a mechanism that is in charge of removing the flow records from the flow table once it is no longer on the link. The rules for expiring the flows, i.e. purge them from the flow table, are the following:

- If a flow shows no activity for a certain time (no new packets belonging to the flow are received), then it is removed from the flow table since the connection no longer exists on the link. Inactivity timeout is typically set to 15 seconds.
- The long lived flows, those that keep receiving packets with a certain regularity, are removed after an activity timeout has elapsed. This condition is useful to avoid counters' overflow. Activity timeout is typically set to 30 minutes.
- Flows with TCP protocol that have reached the end of the connection (FIN flag set) or have been reset (RST flag set) are purged from the flow cache.

As can be inferred, two concurrent processes are accessing the flow table (the memory) as depicted in Fig. 1.

After the flows are purged from the flow cache, normally they are packed in bundles and delivered to some kind of computing device (data collector device) for its storage and/or high-level analysis. This task is performed by another unit, known as flow export, and the implementation can vary depending on the needs. Since the construction of the flows acts much as a filter, the amount of information at the input of flow cache is several orders of magnitude bigger than its output, when dealing with regular network traffic. For this reason the focus of this implementation is on flow cache rather than flow export, since the latter can be implemented in many ways without a hard challenge.

3.2. Development Platform

The design has been implemented and tested on the open-source NetFPGA-10G project [14]. NetFPGA-10G is the second release of the NetFPGA project and refers to an effort to develop an open source hardware and software platform for enabling rapid prototyping of networking devices [16]. It was developed at the Standford University together with Xilinx® Research Labs to help researchers quickly build fast, complex designs, mainly in hardware. The platform is intended to provide everything necessary to get end users off the ground faster, while the open-source community allows researchers to leverage each other's work. The platform has a Virtex®-5 TX240T FPGA [17] interfaced with five SFP+ cages trough AEL2005 PHY chips that provides four independent 10 Gbps Ethernet ports. The board is also populated with three external QDR-II and four RLDRAM memory modules that give 27MB and 288MB respectively.

4. PROPOSED ARCHITECTURES

In this section we will examine the two implementations that were developed: NF_BRAM and NF_QDR. The former uses internal BlockRAMs of the FPGA to store the active flows, while the latter uses external QDR-II memory instead. The NF_BRAM implementation supports up to 16,384 concurrent active flows and the NF_QDR supports up to 786,432. Both of them were designed to work with no-sampling over the received packets in order to implement a precise flow analyzer tool. Before we introduce the details of each architecture, we will analyze the timing requirements needed for being able to deal with 10 Gbps Ethernet traffic, even in the worst-case scenario.

4.1. Temporal Restrictions

According to the IEEE 802.3 standard [18], the minimum Ethernet frame is 576-bits long. That includes: preamble, start of frame delimiter, source and destination MACs, type/length, data and frame check sequence. On the other hand, the IEEE 802.3ae-2002 [19] defines the minimum interframe gap to be 40-bits (5 octects) in the receiving MAC. That sums the total of 616-bits i.e. 61.6 ns between packets. The user side of the MAC core in the NetFPGA-10G is a 64-bits wide interface running at 200 MHz in single-data rate (SDR). That means we have a total of 12 clock cycles to process every packet in the worst-case scenario.

4.2. Architecture of NF_BRAM

This implementation of the flow cache uses BlockRAMs available on the FPGA for the flow table. Since the BlockRAMs of the FPGA are true dual port, the connection of the two concurrent processes, depicted in Fig. 1, is as simple as connecting the logic of Process A to one port of BlockRAMs and Process B to the other. In the following we will describe the internal modules of the NF_BRAM architecture shown in Fig. 2.

Packet Parser. This module receives all the Ethernet frames from the link that is being monitored. It extracts its 5-tuple, described in Sec. 3.1, plus the information needed to create a new flow or update an existing one. That information is comprised of: the timestamp aligned to the start

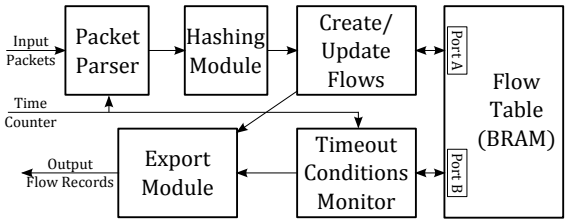

Fig. 2. NF_BRAM Block Diagram.

of the frame reception, the TCP flags (if the packet has that Layer 3 protocol) and finally the number of bytes within the frame. If the frame the interface is receiving is not TCP or UDP or if its FCS (Ethernet CRC) is corrupted (signaled from the MAC user interface), then the frame is discarded by this module. An external GPS module provides a time counter to precisely timestamp the incoming frames. The GPS module is a different project and will not be addressed here.

Hashing Module. When a 5-tuple is received from the packet parser, this unit calculates a 14-bit wide hash code to obtain an address where the flow record will be stored. A hash function is needed since a memory with 2^{104} locations, one for each possible 5-tuple, is not feasible. In addition, a full memory look-up would neither meet timing nor be scalable to bigger flow tables. Note although, that we have at least 12 clock cycles to calculate a hash code for each packet, so a relative complex hash function is implemented (using multipliers and more) in order to minimize the probability of collision.

Create/Update Flows. Is the name given to Process A in Fig. 1. With the previously calculated hash code, the flow table is addressed and its content analyzed. If the busy flag is set, it means that an active flow is on that location, so the received 5-tuple is compared to the stored one. If they match, the flow is updated with the information of the received packet, if they do not match there is a collision and the received packet is discarded. On the other hand, if the busy flag is clear then a new flow record is created in that position of the flow table. Note that in case of a collision we could decide to export the old flow and create a new flow record with the new packet, instead of throwing it. Both approaches do not achieve the goal of the tool. The third condition, listed in Sec. 3.1, to purge a flow record from the memory is performed by this unit. If a TCP packet with a RST or FIN flag set to one is received, the memory is polled to check if there is an active flow to which the packet belongs to. In that case the flow is updated and exported immediately. If the memory in the corresponding location is empty or other flow record is on it, the received TCP packet is exported as flow record with only one packet on it.

Flow Table. As stated earlier, this module is implemented with BlockRAMs of the FPGA. It is codified to allow each flow record to be stored and read back in one memory address as shown in Fig. 3. The access of the two processes to the memory is completely in parallel and independent. With this configuration, the design could run at line rates above 10 Gbps.

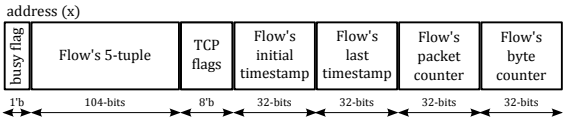

Fig. 3. Memory Organization on NF_BRAM.

Timeout Conditions Monitor. Is the name given to Process B in Fig. 1. This unit uses a linear counter to address the flow table. For each entry the busy flag is checked. If it is clear, the linear search continues, if it is set instead then two operations are performed in parallel. The first one checks that the time elapsed since the last packet of the flow arrived is less than the *prefixed inactive timeout*. This is done by subtracting the flow's last timestamp to the current time counter (from the external GPS module). The second operation consists of checking that the time elapsed since the flow record was created is less than the *prefixed active timeout* and is done by the subtraction of the flow's initial timestamp from the current time counter. If either of the two conditions are satisfied, the flow record is exported outside the core and removed from the flow table by clearing the busy flag.

Export Module. This module receives the flow records that were purged from the flow table and exports them out of the flow cache core. It implements the necessary logic to generate AXI4-Stream [20] transactions in order to communicate with different types of external modules. As an example, a NetFlow v5, v9 or IPFIX core could receive the removed flows and send them out with the corresponding protocol. It can also be connected to a PCIe DMA engine in order to send the flow records to the host, or it can send the flows out with the use of an Ethernet interface in a plain format (one Ethernet frame for every exported flow record).

4.3. Architecture of NF_QDR

This second architecture boosts the flow-based monitoring on FPGA in three manners. First, it uses external QDR-II memory to implement a much bigger flow table, compared with the previous one, and also it does not consume the FPGA's BlockRAMs. Second, it gives a flexible way for the network engineer to configure the flow cache according to his/her needs, with the aid of two-addresses per flow record scheme. Third, it reduces the flow drops caused by the flaws of the hash function.

Since the functionality of both NF_QDR and NF_BRAM remains the same, only the differences between them and the

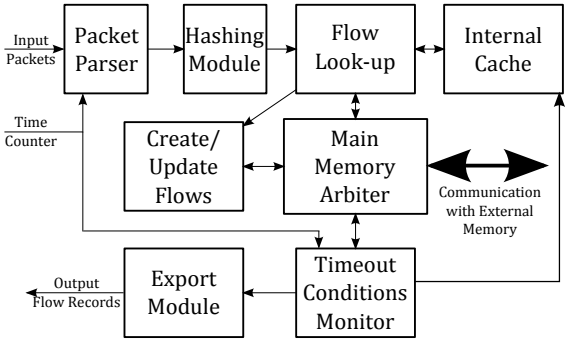

Fig. 4. NF_QDR Block Diagram.

enhancements will be addressed next.

4.3.1. Architectures' Details

A top level block diagram of the NF_QDR implementation is shown in Fig. 4. The main difference with the NF_BRAM architecture is due to the memory access of the two concurrent processes. Since there is only one available port in these external memories, we must provide a multiplexing mechanism for both processes to share the communication with the memory. We will describe the following internal modules: **Flow Look-up, Internal Cache** and **Main Memory Arbiter** of the NF_QDR since all the others have the same functionality as described in 4.2.

Flow Look-up. When a hash code is calculated from an extracted 5-tuple, this module first looks-up if the active flow record is in the internal cache module. If the flow is in cache, it is updated and the update is written back to the external main memory. If the flow is not found in the cache, this module performs a read operation on the main memory. The information about the received packet and its hash code is passed to the create/update flows module and the process of creating a new flow record or updating an existing one is almost the same than in the other architecture.

Internal Cache. In order to reduce the read operations to main memory, this cache module is used to store the most recently created flows. Burst of packets that belong to the same flow do not poll the memory every time they are received to check if the flow is active. If the flow to be updated upon a packet reception is in the cache, then the external memory is only addressed to write the updated flow back so an exact copy of the information in the cache is present in the main memory. This module increases the bandwidth available for Process B that must share the communication to the memory with Process A, as depicted on Fig. 1.

Main Memory Arbiter. As stated earlier, the two concurrent processes must share the communication with the main memory, since no true dual ports are available in this

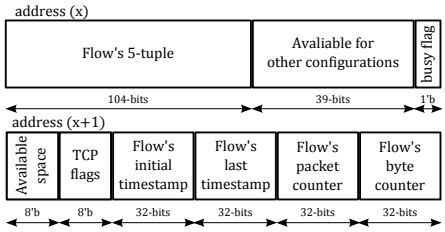

Fig. 5. Memory Organization on NF_QDR.

case. The reception of a packet is an asynchronous event so a predefined scheduling is not possible. The dispatcher maintains a number of read operations on queue that maximize the memory throughput, but the Process B's pending tasks are limited to a small number so a good response time for the creation and updates of the flows (Process A's tasks) is achieved. This module also implements all the necessary logic to interface the external memories with the aid of the Xilinx® MIG (Memory Interface Generator) [21].

4.3.2. User Defined Flows

The main memory organization of the NF_QDR is based on a two-addresses per flow scheme and is shown in Fig. 5. Every flow record is split on the flow table in the flow's ID (5-tuple) and the flow's information. What uniquely identifies the flow (e.g. its 5-tuple) is stored in address x with its LSB (Least Significant Bit) set to one, while the flow's information is stored in the contiguous address $x+1$. With this scheme, the architecture provides the flexibility needed for the network engineer to define the flows in whatever he/she understand by a flow, having only to modify the packet parser module and leaving the more complex memory management as it is. For example, a researcher that is interested in analyzing how many concurrent TCP connections over each VLAN are active in a particular link, would have to modify the packet parser module extracting the VLAN ID field from each Ethernet frame, and drop all the non-TCP packets.

4.3.3. Collision Treatment

A good design of the hash function is crucial in order to minimize the probability of collisions, i.e. two or more input elements that get the same hash code. Normally whenever a collision takes place, the colliding flow must be discarded and the performance of the tool is compromised since it is not capable of doing what it was designed for: monitor the flows on a link. The construction of a well-balanced hash function is not an easy task since the normal assumption that all the input elements have the same probability of occurrence is not true. For example, the Layer 3 Protocol type

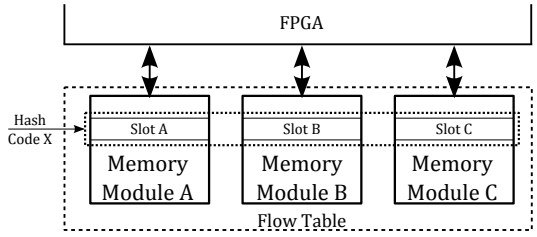

Fig. 6. Slots for Collision Treatment on NF_QDR.

field is likely to be: TCP (0x06), UDP (0x11) or some more few values but not any arbitrary value out of the 256 possible. The IP addresses as well as the Layer 3 Ports also share this behavior. To reduce the possible flaws of the hash function, in the NF_QDR implementation, the three available QDR-II memory modules available on the NetFPGA-10G board are used to provide three slots for each hash code as shown in Fig. 6. When a new flow record with a particular hash code is about to be created, an empty slot is fetched from module A to module C. If all three slots are busy with other flow records, it means that at least four elements (5-tuples) generated the same hash code and the fourth new flow record has to be dropped. Fortunately, the likelihood of occurrence of a level 3 collision is less than the occurrence of a level 2 and even less than a level 1 collision, both of which are not an issue in this implementation.

5. IMPLEMETATION RESULTS AND VALIDATION

In this section we firstly report the total amount of FPGA resources the architectures use. Then we describe the procedures followed to test the performance of the described designs, covering hardware, software and traffic details. Finally we present a comparison between our architectures and the ones introduced in Sec. 2.

5.1. Hardware Resource Utilization

The designs were coded in VHDL and synthesized using Xilinx EDK/XST v13.4 [22] [23]. The code for NF_BRAM and a simplified version of NF_QDR are publicly available [15]. As mentioned earlier the target platform is NetFPGA-10G which contains a Xilinx® Virtex®-5 TX240T FPGA. Table 1 shows the main resource utilization of the FPGA for each design. The clock frequency for both architectures is 200 MHz i.e. the same of the user side of the MAC core, as stated in Sec. 4.1. A careful design methodology was taken in order to meet all timing constraints.

Table 1. Resource Utilization of the FPGA

	#Luts	#Flip Flops	#BRAMs
NF_BRAM	897	1642	121
NF_QDR	5296	7050	7

Table 2. Performance Comparison

	Link (Gbps)	Dev	Max. con. Flows	Max. Mpps
Žádník [9]	1	V2	60K	1.4
Rajeswari [13]	10	V5	0.5K	-
Yusuf [12]	10	V2	64K	2.9
FlowMon [10]	10	V5	500K	3.0
NF_BRAM	10	V5	16K	14.8
NF_QDR	10	V5	768K	14.8

5.2. Experimental Testbed

The verification and stress test of the designs were carried out in our laboratories with the aid of software tools [24]. In the following, we describe the tests in more detail. Two general-purpose PCs containing 10 Gbps Ethernet interfaces were connected to the NetFPGA-10G platform. The first PC was used as traffic generator, running a high-performance network driver [24] that is capable of 10 Gbps link saturation. The second machine captured in a file the output flow records exported by the design under test. Each Ethernet frame sent from the design carried one exported flow record in a non-standard plain format. The same input traffic was processed offline with a well-known flow capturing software [25]. Both output files were compared to extract the differences. From a detailed analysis of the results, the observed differences were due to the collisions occurred in the hardware implementations. All the Ethernet connectivity, between the NetFPGA-10G and the PCs, is by means of multi-mode fiber optics, with SFP+ transceivers using 850 nm wavelength i.e. 10GBASE-SR. We utilized both, synthetic traffic and real traces for the tests. With the first we tested the worst case scenario using a loop generator of minimum size packets with minimum interframe gaps during a 100-second run. With the real traffic captures, we tested the flow creation in a real scenario and checked the output flows against the output of the software tools mentioned above.

5.3. Performance Comparisons

In Table 2 we compare the published hardware implementations refered in Sec. 2 against the two introduced here. We evaluate the performance of each architecture based on: the maximum supported number of concurrent flows, the link-rate, the maximum amount of the smallest packets (as de-

scribed in Sec. 4.1) expressed in millions, and finally the FPGA device used for the implementation (V5 stands for Xilinx® Virtex®-5 and V2 for Virtex®-2).

6. CONCLUSION

We have developed an accurate and flexible flow-based classifier tool for 10 Gbps networks. The main characteristics that outperform the previous published works comes from the possibility to process saturated 10 Gbps links without packet sampling even with the shortest packets, i.e. 14.88 Mpps, and the possibility to manage up to $^{3}/_{4}$ concurrent flows. We proposed two architectures, one that uses internal BlockRAMs of the FPGA (NF_BRAM) and a second that uses external QDR-II memory (NF_QDR), which both are public open-source hardware projects. We encourage network engineers to create specific flow-based tools to aid the detection of network issues and attacks on the basis of our work. We hope that new network monitoring techniques will rapidly emerge with the fact of building in the work of others.

7. REFERENCES

[1] Cisco Inc., "Netflow services solutions guide," Jan. 2007. [Online]. Available: http://www.cisco.com/en/US/docs/ios/solutions_docs/netflow/nfwhite.html

[2] M. Lavine and P. Phaal, "sFlow Version 5," Foundry Networks and InMon Corp., Tech. Rep., July 2004. [Online]. Available: http://www.sflow.org/sflow_version_5.txt

[3] Juniper Networks Inc., "Junos OS, Flow Monitoring," Nov. 2012. [Online]. Available: http://www.juniper.net/

[4] B. Claise, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," RFC 5101 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5101.txt

[5] WildPackets Inc., "Beyond Monitoring – Root-Cause Analysis," Tech. Rep., Dec. 2009. [Online]. Available: http://www.wildpackets.com/resources/whitepapers

[6] L. Deri, "nProbe: an Open Source NetFlow Probe for Gigabit Networks," in *Proceeding of Terena TNC*, May 2003.

[7] D. Miller, "softflowd, a software NetFlow probe," Sept. 2004. [Online]. Available: http://code.google.com/p/softflowd/

[8] M. Danelutto, D. Deri, and D. De Sensi, "Network monitoring on multicores with algorithmic skeletons," *Advances in Parallel Computing*, vol. 22, pp. 519 – 526, 2012.

[9] M. Žádník, "NetFlowProbe on NetFPGA-1G," May 2010. [Online]. Available: http://wiki.netfpga.org/foswiki/bin/view/NetFPGA/OneGig/NetFlowProbe

[10] M. Žádník, L. Polčák, O. Lengál, M. Elich, and P. Kramoliš, "FlowMon for Network Monitoring," in *Networking Studies V : Selected Technical Reports*. CESNET, z.s.p.o., 2011, pp. 135–153.

[11] Liberouter. [Online]. Available: http://www.liberouter.org/

[12] S. Yusuf, W. Luk, M. Sloman, N. Dulay, E. C. Lupu, and G. Brown, "Reconfigurable architecture for network flow analysis," *IEEE Trans. VLSI Syst.*, vol. 16, no. 1, pp. 57–65, 2008. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4407545

[13] P. Rajeswari and N. Nagarajan, "An fpga based hardware architecture for network flow analysis," *European Journal of Scientific Research*, vol. 83, no. 3, pp. 338–337, Aug. 2012.

[14] "NetFPGA-10G board description," 2012. [Online]. Available: http://netfpga.org/10G_specs.html

[15] M. Forconesi, G. Sutter, and S. Lopez-Buedo, "Open source code of nf_bram and nf_qdr," 2013. [Online]. Available: https://github.com/forconesi/HW-Flow-Based-Monitoring

[16] Wikipedia, the free encyclopedia, "NetFPGA." [Online]. Available: http://en.wikipedia.org/wiki/NetFPGA

[17] Xilinx Inc., *Virtex-5 FPGA Data Sheets*, March 2010. [Online]. Available: http://www.xilinx.com/support/

[18] IEEE Standard Association, *IEEE Standard for Ethernet - Section 1*, IEEE Std. 802.3-2012 (Revision to IEEE Std 802.3-2008), Dec. 2012.

[19] ——, *IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks- Specific Requirements Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/S Operation*, IEEE Std. 802.3ae-2002 (Amendment to IEEE Std 802.3-2002), Dec. 2002.

[20] ARM Inc., "AMBA AXI Protocol v2.0," Tech. Rep., 2010. [Online]. Available: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[21] Xilinx Inc., *Memory Interface Solutions, User Guide UG086 (v3.6)*, Sept. 2010. [Online]. Available: http://www.xilinx.com/support/

[22] ——, *Embedded System Tools Reference Manual EDK (v13.4). UG111*, Jan. 2012. [Online]. Available: http://www.xilinx.com/support/

[23] ——, *XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices (v12.4). UG627*, Dec. 2010. [Online]. Available: http://www.xilinx.com/support/

[24] P. S. del Río, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil, "Wire-speed statistical classification of network traffic on commodity hardware," in *Internet Measurement Conference*, 2012, pp. 65–72.

[25] P. S. del Río, D. Corral, J. García-Dorado, and J. Aracil, "On the impact of packet sampling on skype traffic classification," *IFIP/IEEE International Symposium on Integrated Network Management (IM 0213)*, 2013.

Paper Revisions

In the following are given the four revisions the above paper had. It is worthy to note that these reviewers are independent of the department and the university where the work was developed.

```
============================================================================
                            REVIEWER #1
============================================================================


----------------------------------------------------------------------------
Reviewer's Scores
----------------------------------------------------------------------------

Question 1: Technical Contribution and Quality: 4
                Question 2: Originality: 4
              Question 3: Overall Rating: 4
   Question 4: Presentation and Language: 4
             Question 5: In Scope of FPL: 4
    Question 6: Reviewer Expertise Level: 3
        Nominate for FPL Community Award: Yes
             Suggested Presentation Type: Regular paper


----------------------------------------------------------------------------
Comments
----------------------------------------------------------------------------

This is a fine paper about flow monitoring in FPGAs. The presented work is new
in that it supports a higher number of flows then previously published work.
This was achieved by deploying a hash table for the flow lookup.
The code is contributed to the open source community through NetFPGA.

The paper could be further improved by including more specifics. For example,
the SRAM-based architecture contains a cache. I'm missing the argumentation of
why a cache is necessary. How much access bandwidth to SRAM is available, how
much is needed, and how effective does the cache needs to be. Also what is the
performance without the cache (included in table 2) Can you not meet the 10G
line rate any more?
Also nice to have would be a discussion on
* deployment models for this flow monitor
* discussion of the design's limitations (for example a simplistic hash
implementation)
* how the priority handling is done in the main memory arbiter (time slots?)
```

```
============================================================================
                            REVIEWER #2
============================================================================


----------------------------------------------------------------------------
Reviewer's Scores
----------------------------------------------------------------------------

Question 1: Technical Contribution and Quality: 4
                 Question 2: Originality: 3
              Question 3: Overall Rating: 4
   Question 4: Presentation and Language: 4
             Question 5: In Scope of FPL: 4
    Question 6: Reviewer Expertise Level: 4
        Nominate for FPL Community Award: No
             Suggested Presentation Type: Poster


----------------------------------------------------------------------------
Comments
----------------------------------------------------------------------------

The paper describes a hardware design implemented in the NetFPGa platform for
flow-based network processing. The proposed design can handle 750k concurrent
flows and supports a 10Gbit line.

The paper describes two proposed designs and reports an improvement in terms of
# of flows and processing throughput, however there are several concerns:

- most of the related works the authors compare to, use older FPGA devices
(Virtex 2 instead of Virtex 5) so the comparison is not very fair. One would
expect the same design implemented in V2 to be at least 2-4 times faster in a
V5.

- Authors do not explain what is the main idea behind their design that make
their solution more efficient (faster, better)

- It appears that the use of external memory in the NF_QDR case increases the
number of flows handled, do the other works use external memory e.g. papers [9,
12]?

Apparently the increase in # of flows comes from the use of  external memory
and the improvement in performance comes from the use of better FPGA device
(compared to [12] or even [9]).

The writing quality of the paper is fine.
```

```
============================================================================
                            REVIEWER #3
============================================================================


----------------------------------------------------------------------------
Reviewer's Scores
----------------------------------------------------------------------------

Question 1: Technical Contribution and Quality: 2
                 Question 2: Originality: 1
              Question 3: Overall Rating: 2
   Question 4: Presentation and Language: 4
             Question 5: In Scope of FPL: 3
    Question 6: Reviewer Expertise Level: 4
        Nominate for FPL Community Award: No
             Suggested Presentation Type: Poster


----------------------------------------------------------------------------
Comments
----------------------------------------------------------------------------

I have two main concerns with the paper:

1) The novelty of the paper is very low. The paper presents a hardware
accelerated platform for NetFlow monitoring for 10 Gbps half-duplex line.
However, full-duplex monitoring of 10 Gbps were already achieved in [10] which
is even referenced in the paper. As all Ethernet 10 Gbps lines are full-duplex,
it does not make much sense to monitor half-duplex lines.

2) The authors probably did not read [10] carefully enough, becauseu they list
its performance as 3 Mpps but the correct performance is almost 30 Mpps (i.e.
fully loaded full-duplex 10 Gbps line with 64 bytes-long frames). Name of the
author si also misspelled (dnic instead of dnk).

I also have some other minor comments:

* In 4.2, the authors state that "Since the BlockRAMs of the FPGA are true dual
port, the connection of the two concurrent processes, depicted in Fig. 1, is as
simple as connecting the logic of Process A to one port of BlockRAMs and
Process
B to the other." However, both processes can access the same flow record at the
same time, e.g. A updates the flow record while active timeout (B) fires. The
authors does not address such problem.

* The authors does not elaborate more about the hash function used for flow
record addressing. What hash function is used? Are there any references of
```

its evaluation? What is the likelihood for a collision occurring?

* In 4.2, the authors state that "Both approaches do not achieve the goal of the
tool." It is not clear what this sentence mean?

* The duration of tests is not clear. How many flows and/or packets were evaluated? I also suggest to include established tests like RFC 2544. What is the performance of the tool for other packet sizes (e.g. 65B)?

Typography and language issues:

* Data are in a cache, not on a cache (section 4.3.1).

* Processes read from a memory, not on a memory (section 4.3.1).

* Conclusion states "and the possibility to manage up to 3/4 concurrent flows." instead of 768,000 flows.

```
============================================================================
                            REVIEWER #4
============================================================================


----------------------------------------------------------------------------
Reviewer's Scores
----------------------------------------------------------------------------

Question 1: Technical Contribution and Quality: 5
                 Question 2: Originality: 3
              Question 3: Overall Rating: 5
   Question 4: Presentation and Language: 5
             Question 5: In Scope of FPL: 4
    Question 6: Reviewer Expertise Level: 4
        Nominate for FPL Community Award: No
             Suggested Presentation Type: Regular paper


----------------------------------------------------------------------------
Comments
----------------------------------------------------------------------------

This is a good paper: clearly explained, implemented, and open sourced.  The
work will form a useful contribution to the NetFPGA 10G community.

A weak point was the treatment of hashing.  There is only vague discussion of
the issues involved in choosing a hash function, and no description of the
actual hashing function used and its statistical worthiness.  There was no
attempt to deal with hash overflows in any subtle kind of way, which adds
concern given the lack of evidence that such collisions might be minimized.
```

Application Note: Integration on NetFPGA-10G

The document below details the steps to integrate the open source architecture presented here, in the NetFPGA-10G platform. This guide was prepared for the NetFPGA-10G community and is publicly available in the NetFPGA-10G repository[3].

[3]https://github.com/forconesi/NetFPGA-10G-live/wiki/NetFlow-simple-10G-Bram

NETFLOW SIMPLE 10G BRAM ON NETFPGA-10G

Design Document

Revision 1.0

October, 2012

Marco Forconesi, Gustavo Sutter, Sergio Lopez-Buedo
High Performance Computing and Networking (HPCN) research group
Universidad Autónoma de Madrid, Spain

Table of Contents

1 Introduction....1
2 Description....1
3 Internal Architecture....2
3.1 NetFlow Cache Pcore....2
3.1.1 Packet Classification....2
3.1.2 Create or Update Flows....3
3.1.3 Export Expired Flows from Memory....4
3.1.4 Export Flows to NetFlow Export & Export Flows to 10G interface....4
3.1.5 Time Stamp Counter Generator....4
3.2 NetFlow Export Pcore....5
3.2.1 Flow Encoding....5
3.2.2 NetFlow v5 Header....6
3.2.3 UDP Header....6
3.2.4 IP Header....6
3.2.5 General Control Process....6
3.2.6 Ethernet Frame Sender....6
4 Building the Project with and without NetFlow Export Pcore....7
5 Testing the Design....9
5.1 An example for sending traffic and capturing NetFlow v5 packets....9
6 References....10

Table of Figures

Figure 1: NetFlow_Simple_10G_BRAM. Pcore block diagram....1
Figure 2: NetFlow Cache Pcore's internal architecture....3
Figure 3: Composition of the 5-tuple....3
Figure 4: NetFlow Export Pcore's internal architecture....5
Figure 5: Building the project with NetFlow Export Pcore....7
Figure 6: Building the project without NetFlow Export Pcore....8
Figure 7: Using one 10 Gbps Interface to test the design....8
Figure 8: An example for testing NetFlow_Simple_10G_Bram....9

1 Introduction

This guide describes the NetFlow_Simple_10G_Bram project on NetFPGA-10G platform [4]. It captures the active flows received on a 10 Gbps Ethernet interface and exports them via other 10 Gbps Ethernet interface using the NetFlow v5 protocol.

It is the first prototype of NetFlow at 10 Gbps. It stores up to 4096 concurrent flows using internal FPGA's Block Rams (BRAM). It is able to process every packet the interface receives even in the worst case (shorter Ethernet frames with minimum interframe gap), i.e. without sampling.

A second prototype of NetFlow that uses the external QDRII memory is being developed right now (NetFlow_Simple_10G_QDR). It will store up to 786,432 concurrent flows. Shortly, this design will be available at NetFPGA-10G community projects.

2 Description

NetFlow_Simple_10G_Bram project processes every packet the interface 0 receives and classifies them in the different active flows. It discards the unsupported frames and builds the

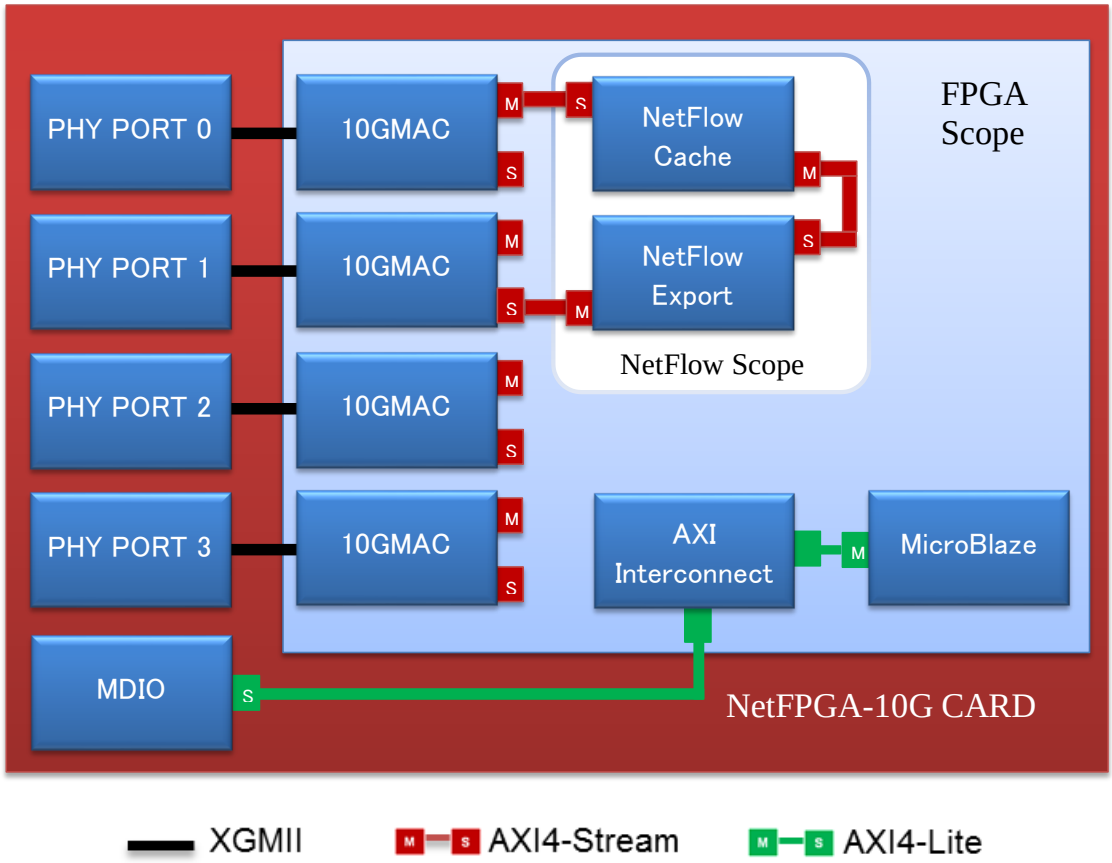

Figure 1: NetFlow_Simple_10G_BRAM. Pcore block diagram

flow entries with those packets that form the active TCP and UDP flows. Once these flows expire, they are grouped together in NetFlows v5 packets that are sent through interface 1 to a remote NetFlow controller. Figure 1 shows a block diagram of the design at the EDK Pcore level.

This project was created using the "10G Ethernet Interface Loopback Test" project as a reference design. The configuration program of the MicroBlaze that configures the MDIO registers was maintained. The "Packet Generators/Checker" Pcores and the internal connections between the 10GMAC Pcores were removed. To build the NetFlow_Simple_10G_Bram project it is necessary to follow the same steps as for the reference loopback test project.

3 Internal Architecture

The project is organized in two main cores: *Netflow Cache* and *NetFlow Export*. This section describes these cores. The reason why NetFlow is split in these two parts is mainly due to:

1. Cisco [3] (who invented NetFlow) makes this conceptual division, described in NetFlow Services Solutions Guide. Technical report [2].
2. To implement others exports protocols rather than NetFlow v5 (e.g.: NetFlow v9, IPFIX), it is only necessary to replace (or modify) the *NetFlow Export* Pcore. *NetFlow Cache* remains the same.

In what follows, *Netflow Cache* and *NetFlow Export* Pcore's internal architectures are depicted. Both Pcores are coded in VHDL.

3.1 NetFlow Cache Pcore

As shown in Figure 1, *NetFlow Cache* has two AXI4-Stream interfaces. The one on the left is a 64-bits width slave interface and is connected to the 10GMAC user interface to receive the Ethernet frames. The other one on the right is a master interface and is also 64-bits width. It exports the expired flows out of the Pcore.

A top level diagram of *NetFlow Cache* Pcore's internal architecture is shown on Figure 2. The VHDL modules are explained below, following the data path sequence. The *Flow Table* is implemented using dual port Block Rams of the FPGA, as well as the output FIFO.

3.1.1 Packet Classification

This module extracts the 5-tuple of each Ethernet frame and timestamps it using the time information. It also extracts the next items to keep track of the flows statistics:

- Number of bytes in the IP Total Length field of IPv4 packet's header.
- TCP flags, if protocol is TCP.

If the Ethernet frame that the interface is receiving is not valid, the frame is discarded. This occurs when the packet is neither TCP nor UDP.

After a valid frame is received, this module sends the 5-tuple plus the information listed above to the *Create or Update Flows* module. The composition of the 5-tuple is shown at Figure 3.

If there were more types of flows of interest, this is the unit to be modified. Also, it is very easy to do that, because it is only necessary to add if-then-else line codes with the fields that must be checked. The rest of the design remains the same.

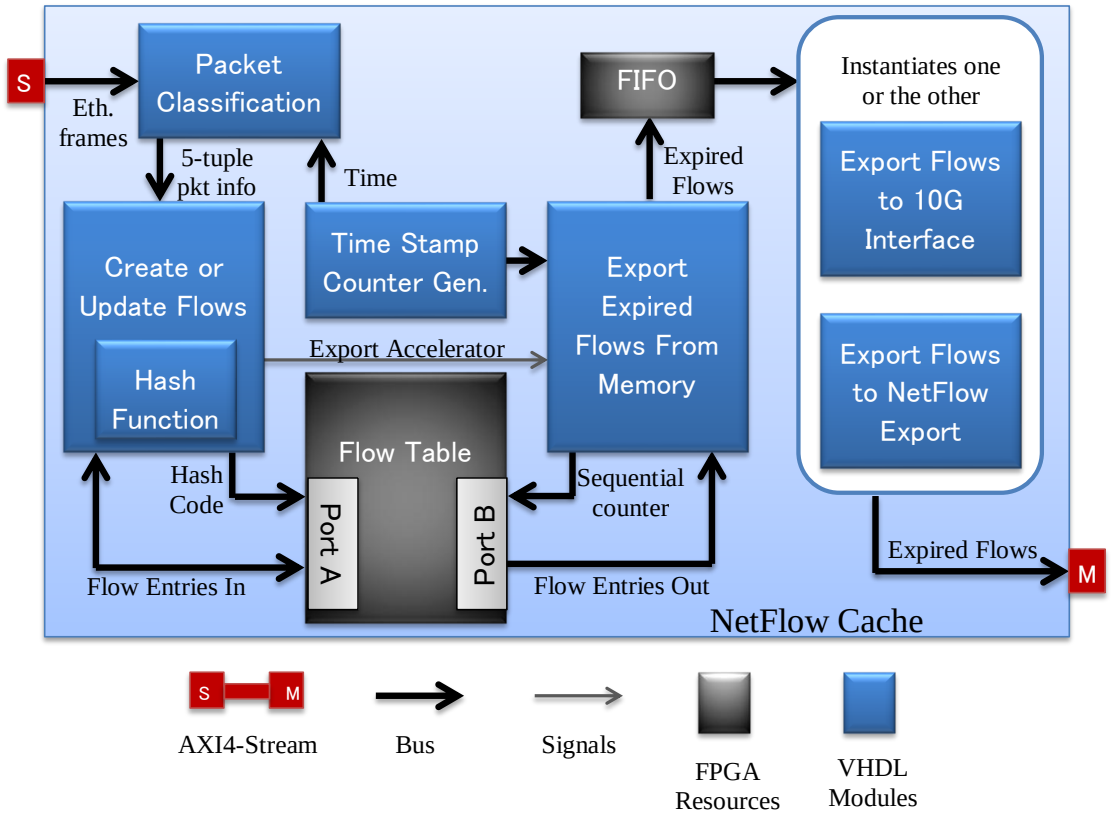

Figure 2: NetFlow Cache Pcore's internal architecture

3.1.2 Create or Update Flows

When a new frame arrives, this unit checks if it belongs to an existing flow in the *Flow Table*. If so, the flow statistics (i.e. number of bytes, number of packets and time stamp since the last frame matched the flow) are updated. If the flow does not exist, i.e. the received frame is the first of a new flow, then a new flow entry is created in the *Flow Table*.

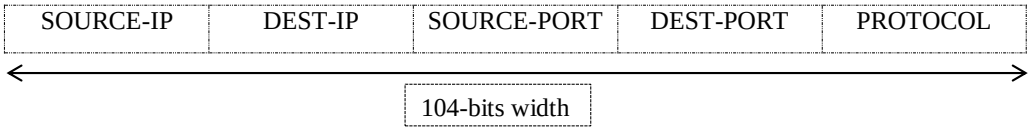

Figure 3: Composition of the 5-tuple

This module receives the 5-tuple plus the frame information provided by the *Packet Classification* module described in 3.1.1. With the 5-tuple it computes a hash code using the *Hash Function* module. After this, the *Flow Table* is addressed using the hash code as the address, and if the flow exists it is updated, otherwise it is created.

The reason of the hash function is because it is not possible to use the full 104-bits to address a memory, since 2^{104} x 136-bits is an unreal memory size. Using smaller memories (2^{12} x 136-bits in this design) a hash function is implemented in order to reduce the probability of collisions (two or more different flows trying to occupy the same flow entry). The other approach is to put the flows in contiguous *Flow Table's* entries as they are created. The problem with this would be

that a sequential look up will have to be done each time a frame is received and thus the response time will increase. On the other hand, if a parallel look up is implemented the amount of hardware will not be attainable with larger memories (e.g.:1Mflows flow tables).

If a collision takes place, this is, a new flow is to be created on a busy flow entry, then the new flow is discarded and the old one is preserved. Every time there is a collision and a flow is lost it is a bad performance by the application. The idea is to use a large memory size for the *Flow Table* and also the best possible hash function. The hash function of this design was created with polynomial division using primitive polynomial.

3.1.3 Export Expired Flows from Memory

This module checks if a flow has expired and if it has, it removes the flow entry from the *Flow Table* and exports the flow to a FIFO. The expiration conditions (see also [2]) are:

- Flows which were idle for a specified time (by default 15 sec, A.K.A. *Inactive Timeout*) are expired and removed from cache.
- Long lived flows are expired and removed from cache. Flows are not allowed to live more than a specified time (30 minutes by default, A.K.A. *Active Timeout*). The reason of this condition is to avoid counters overflow.
- TCP connections which have reached the end of byte stream (FIN) or which have been reset (RST) (RST or FIN flag set to '1') will be exported.

The constants *Active Timeout* and *Inactive Timeout* can be setup in the Pcore instantiation in EDK.

Since the *Flow Table* is implemented using dual port BRAMs, through the memory's Port B, a sequential counter addresses the *Flow Table* as shown in Figure 2. At each entry the expiration conditions are evaluated. Additionally, to accelerate the export process, when a TCP packet containing a FIN or RST flag set to '1' is received, the *Create or Update Flows* module signals to *Export Expired Flows from Memory* module to remove the flow entry immediately. This is also shown in Figure 2.

3.1.4 Export Flows to NetFlow Export & Export Flows to 10G interface

Expired flows are read from FIFO and exported via AXI4-Stream transactions out of the Pcore. Normally, an instance of *NetFlow Export* Pcore exists in the EDK project, so *Export Flows to NetFlow Export* VHDL module is generated. If you wish to see the expired flows immediately after they leave the *NetFlow Cache* Pcore, for example to check the expiration conditions, then *Export Flows to 10G interface* has to be generated.

Whichever module, *Export Flows to NetFlow Export* or *Export Flows to 10G interface*, is instantiated in the project, is controlled from the *NetFlow Cache* Pcore instance in the EDK. Setting the parameter "NetFlow Export Present = YES" generates the former and "NetFlow Export Present = NO" generates the latter.

3.1.5 Time Stamp Counter Generator

This module generates a counter with the number of milliseconds elapsed since the device booted (the FPGA was configured). This counter is used to timestamp the frames the interface receives, and also to determine in the *Export Expired Flows from Memory* module, how long has a flow been active/inactive.

In the future this module could be connected to a GPS receiver to synchronize with absolute time.

3.2 NetFlow Export Pcore

Expired flows from the *NetFlow Cache* Pcore are transmitted via AXI4-Stream transactions to this Pcore. Once it has N expired flows, where N is 30 for NetFlow v5 [2], it sends a NetFlow v5 packet through a 10GMAC interface to a NetFlow Collector. If it has one or more flows but less than N it waits one minute and then sends the NetFlow v5 packet with the number of flows received until that moment. Figure 4 shows the internal architecture of this Pcore, after it all the VHDL modules are explained.

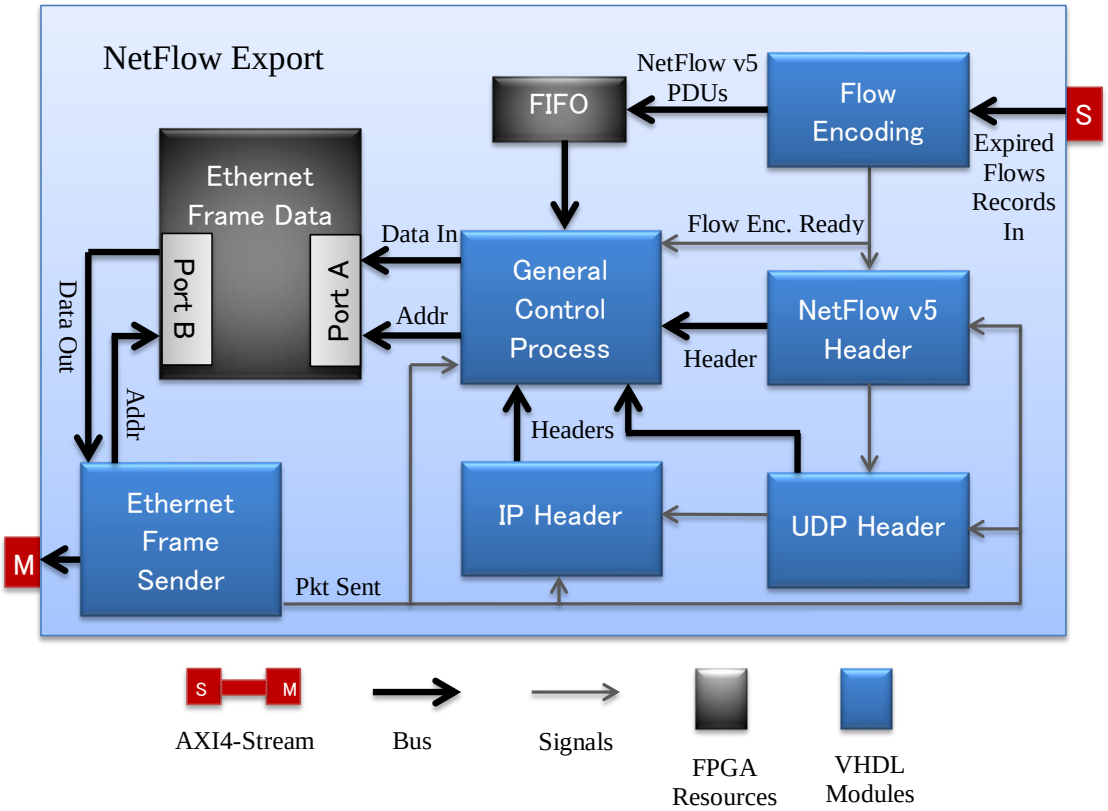

Figure 4: NetFlow Export Pcore's internal architecture

3.2.1 Flow Encoding

This module receives the flows exported by *NetFlow Cache* Pcore via the AXI4-Stream master interface and writes them in a FIFO. When it has up to N flows (see 3.2) or it has one or more flows and the minute waiting for more flows to fulfill the N flows has elapsed, it then signals the *General Control Process* that all PDUs in the NetFlow packet are in the FIFO. After that, it waits for an ACK signal from *General Control Process* to start over the encoding process.

Flow Encoding, as well as *NetFlow v5 Header* module, calculates the UDP checksum over the data while it is arriving. In this module, the partial UDP checksum calculated over the flow records (PDUs) is delivered to *NetFlow v5 Header* module when the encoding process finishes. See [5] for UDP checksum calculation procedure.

3.2.2 NetFlow v5 Header

Once the flow encoding process is ready, this module calculates the header of the NetFlow v5 packet according to the payload. It also captures time information, necessary for the header, from a *Sys Time Generator* module (not shown on Figure 4) similar to the one present in *NetFlow Cache* Pcore. For more information about the NetFlow v5 Header format see [1].

Additionally, this module calculates a partial UPD checksum over the generated header and adds it to the partial UDP checksum provided by the *Flow Encoding* module.

When the NetFlow v5 header and the partial UDP checksum are ready, *NetFlow v5 Header* module signals the next module, *UDP Header*, to start operating.

3.2.3 UDP Header

Once the NetFlow v5 header is ready the UDP header calculation starts. It calculates the final UDP checksum, using the partial checksum provided, as well as the others UDP header's fields.

3.2.4 IP Header

Like *UDP Header* and *NetFlow v5 Header* modules, this one waits for the UDP header to be calculated before it starts the IP header calculation. This is due to the fact that the fields in the header depend on the payload (UDP packet is the payload of IP packet).

When the IP header has been calculated, the *General Control Process* is signaled to continue its operation.

3.2.5 General Control Process

This is the main module of this Pcore. Its function consists in fulfilling a memory which has the same width of the master AXI4-Stream tdata vector. The information in the memory is the Ethernet frame, with all the high level packets, that will be sent through the 10GMAC Pcore. The memory is implemented in Block Rams of the FPGA and its size is as big as the maximum NetFlow v5 packet that can be sent. This includes packets' headers plus the maximum number N of NetFlow v5 PDUs (N = 30 for NetFlow v5 [2]). Once all the data is on the memory, *Ethernet Frame Sender* module is signaled to start reading this memory, using the other port, and generating the proper AXI4-Stream transactions to the 10GMAC Pcore.

The process of fulfilling the memory starts by saving the flow records in the right position on the memory when the *Flow Encoding* module finishes. After this, all the packets' headers are calculated, so the *General Control Process* reads those headers and adds this information on the memory's lower addresses. When the operation is completed, the *Ethernet Frame Sender* module starts reading the memory from address zero to the maximum address recorded.

3.2.6 Ethernet Frame Sender

As it was explained in 3.2.5, this module waits for the Ethernet frame's data to be written on the *Ethernet Frame Data* memory and then it starts reading this memory from address zero to the maximum indicated. Each memory entry has the same width as the master AXI4-Stream tdata vector. Since AXI4 provides a mechanism for flow control (this means the slave can prevent the master of sending data) the completion of *Ethernet Frame Sender's* task is non-deterministic. This is why this module signals when all the other modules can start over and the *Ethernet Frame Data* memory can be rewritten.

The FIFO shown in Figure 4 was implemented to avoid the situation where no new flow records can be received from the *NetFlow Cache* Pcore until the end of the transmission of the current NetFlow v5 packet. In this manner when the *Flow Encoding* module operation has finished,

General Control Process starts reading the FIFO and in that moment it signals back to *Flow Encoding* module that it can start over. Having two memory storages makes it possible for the reception of the flows and the transmission of the previous encoded NetFlow v5 packet to be performing at the same time.

4 Building the Project with and without NetFlow Export Pcore

Normally *NetFlow Export* Pcore is connected to *NetFlow Cache* Pcore as shown in Figure 5. With this configuration, Port 1 is going to send the expired flows on NetFlow v5 packets as explained in 3.2.

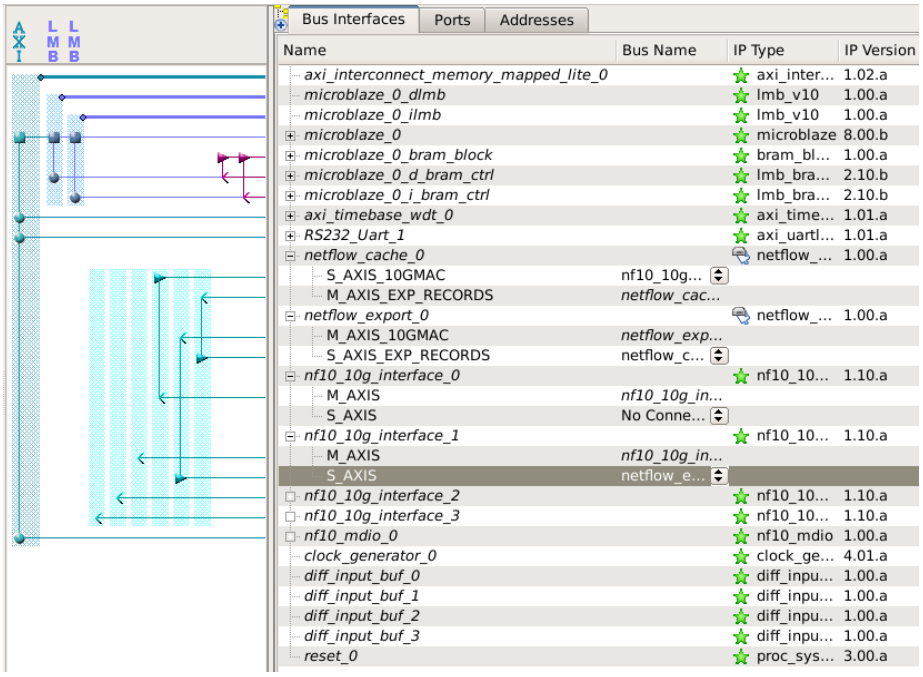

Figure 5: Building the project with NetFlow Export Pcore

In case you want the NetFPGA-10G to output the expired flows directly to a 10 Gbps Ethernet interface instead of NetFlow v5 packets, build the project with the option "NetFlow Export present = NO" in the instantiation of the *NetFlow Cache* Pcore in the EDK. Then, connect the output of *NetFlow Cache* to a 10GMAC slave interface where you want the expired flows out. Figure 6 shows the connection of the *NetFlow Cache* Pcore without *NetFlow Export* Pcore.

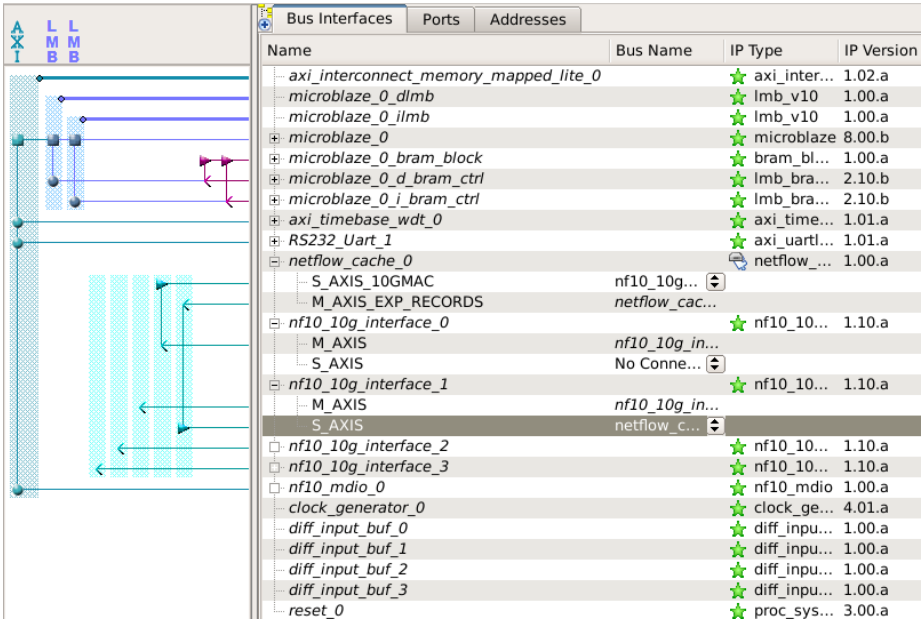

Figure 6: Building the project without NetFlow Export Pcore

If you only have one computer with a 10 Gbps interface, you can still run the design connecting the input traffic and the output expired flows to one Port as illustrated in Figure 7.

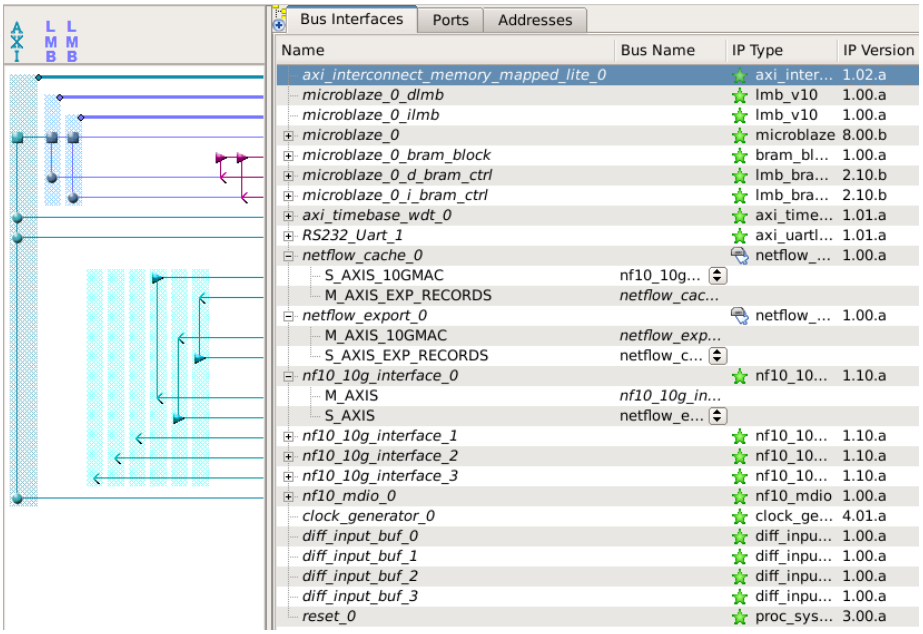

Figure 7: Using one 10 Gbps Interface to test the design

5 Testing the Design

Connect Port 0 to a 10 Gbps traffic generator (or splitter) and Port 1 to a 10 Gbps NIC (NetFlow collector) to receive the NetFlow v5 packets which contain the expired flows.

Once the bitstream configuration file is ready, following the steps mentioned in 4, configure the FPGA. It is also possible to use the pre-built bitstream file provided with the code.

Send traffic to Port 0 and listen to Port 1 to receive NetFlow v5 packets sent by the NetFPGA-10G.

It is possible to send 10 Gbps traffic to Port 0 in different ways. Here is a very simple way to do that.

5.1 An example for sending traffic and capturing NetFlow v5 packets

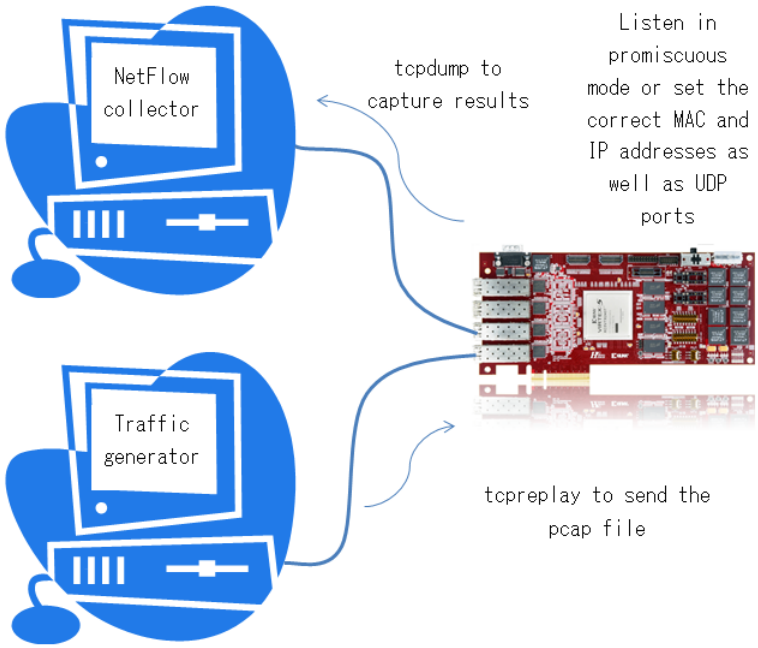

Figure 8: An example for testing NetFlow_Simple_10G_Bram

Figure 8 is a picture that shows a possible scenario. Use a pcap file with TCP, UDP and all kinds of packets as the input traffic. In the computer connected to Port 1 run the bash command:

```
# tcpdump -i eth1 -w output_file.pcap
```

in order to start listening to the interface. In the other computer, connected to Port 0, run the command:

```
# tcpreplay -t -i eth0 input_pcap_file.pcap
```

Wait enough time to make sure all the flows have expired and the encoding timeout in the *Netflow Export* Pcore has elapsed. It is necessary to provide the correct flags to both `tcpreplay` and `tcpdump` commands. See [7] [6] to specify the correct ones for your system.

Finish the `tcpdump` process (Ctrl-C). Read the `output_file.pcap` with Wireshark program. Check the flows in the NetFlow v5 packet(s) and all the other packets' headers.

6 References

[1] Cisco Inc. NetFlow Export Datagram Formats. Technical report.

[2] Cisco Inc. NetFlow Services Solutions Guide. Technical report, 2007.

[3] Cisco Inc. http://www.cisco.com/. 2012.

[4] NetFPGA Project. *NetFPGA-10G board description*. http://netfpga.org/, 2011.

[5] J. Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980.

[6] Tcpdump.org. *Tcpdump & Libpcap*. http://www.tcpdump.org/.

[7] Aaron Turner. *The Tcpreplay suite*. http://tcpreplay.synfin.net/.

Conclusions and Future Work

Thesis Conclusions

As the paper conclusion states, we have developed, implemented and tested a set of hardware tools for monitoring IP networks. For analysis processes, the more accurate the input information is, the more precise are the conclusions, given that the methodology is correct.

Network monitoring is a useful approach to detect issues and diagnose problems, but for high speed networks the measure technique becomes a challenge. In order to mitigate such limitations we proposed two hardware architectures that performs all the time-critical tasks and supplies the analysis algorithm with the flow information needed.

The presented architectures are capable of monitoring 10 Gbps saturated links without packet sampling. Besides, up to 786,432 concurrent flows can be monitored and a flexible flow definition is possible. The project is open source code and available to be used as a starting point for further advanced monitoring techniques.

Research Publications

The presented architectures as well as the published paper are part of a bigger research work. In chronological order, the following papers have been published:

- Forconesi M., Sutter G., Lopez-Buedo S. and Sisterna C., "Clasificación de Flujos de Comunicación en Redes de 10 Gbps con FPGAs", in *Jornadas de Computación Reconfigurable y Aplicaciones*, Sept. 2012.
- Forconesi M., Sutter G., Lopez-Buedo S. and Aracil J., "Accurate and Flexible Flow-Based Monitoring for high-Speed Networks", in *International Conference on Field Programmable Logic and Applications*, Sept. 2013.
- Forconesi M., Sutter G., Lopez-Buedo S. and Gomez-Arribas F., "Clasificación de paquetes IP a 10 Gbps descripto desde lenguajes de alto nivel", in *Jornadas de Computación Reconfigurable y Aplicaciones*, Sept. 2013.

Future Works

The last paper is the future work of this Thesis. Despite of the advantages of using hardware to solve high speed processing problems, it lacks of the software low time development time. New techniques to design hardware from high level languages (C/C++) are currently the target of our research. They will provide lower time to market and a broader community of developers rather than the Verilog/VHDL; in other words this new design flow will aid to achieve low-cost high-performance hardware implementations.