Flexible, extensible, open-source and affordable FPGA-based traffic generator

Tristan Groleat, Matthieu Arzel, Sandrine Vaton, Alban Bourge, Yannick Le Balch, Hicham Bougdal

Flexible, extensible, open-source and affordable FPGA-based traffic generator

Tristan Groléat
Télécom Bretagne
tristan.groleat@telecom-bretagne.eu

Matthieu Arzel
Télécom Bretagne
matthieu.arzel@telecom-bretagne.eu

Sandrine Vaton
Télécom Bretagne
sandrine.vaton@telecom-bretagne.eu

Alban Bourge
Télécom Bretagne
alban.bourge@telecom-bretagne.eu

Yannick Le Balch
Télécom Bretagne
yannick.lebalch@telecom-bretagne.eu

Hicham Bougdal
Télécom Bretagne
hicham.bougdal@telecom-bretagne.eu

ABSTRACT

As high-speed links become ubiquitous in current networks, testing new algorithms at high-speed is essential for researchers. This task often requires traffic to be generated with some specified features : distribution of packet sizes, payload content, number of TCP or UDP flows... When targeting a data rate of many Gb/s, this cannot be done with commodity computers. Commercial traffic generators exist for this task, but they are expensive and do not fit the precise needs of researchers. In this paper, we describe an open-source implementation of a traffic generator capable of filling a 10 Gb/s Ethernet link, with traffic features specified in software. The implementation works on a board including an FPGA and a 10 Gb/s network interface, like the Combo from INVEA-TECH or the NetFPGA 10G. These boards are affordable for research and can provide a configurable and easily extensible traffic generator.

1. INTRODUCTION

Traffic generation is the task of sending network traffic on a link to simulate the use of this link in an actual network of computers. It is used mainly by researchers to assess the performance of new algorithms, and by deployment teams to test new equipments.

Two methods exist to generate traffic:

- Replaying traffic: it requires a probe to be installed on a commercial network, save a trace of the received packets, and send the packets to the equipment under test, exactly as they were received.
- Generating synthetic traffic: it consists in sending invented packets respecting some specified criteria (data rate, number of packets, distribution of packet sizes, headers of the packets).

The first method has the advantage of being very realistic, but the characteristics of the traffic are unknown. The second method is very useful to test algorithms in edge cases.

For example, algorithms that process packet headers one by one will have much more problems with small packets than with big packets, as the number of packets received per second will be much higher for the same data rate. Performance of such algorithms should be tested in the worst case, that is to say with the link filled with the smallest packets the protocol allows. An other example would be an algorithm that only considers TCP traffic on port 80: sending packets using other protocols or ports would only give the algorithm more time for its processing, so the performance results using a replayed trace would be difficult to interpret.

This article presents a generator of synthetic traffic. Its design started with the need to test a hardware traffic classifier meant to work at 10 Gb/s [11].

The first idea was to use a computer equipped with a 10 Gb/s network card to generate traffic. But filling a network link at 10 Gb/s with some specific traffic is very challenging in software, and not possible on a commodity computer. And controlling precisely the inter-packet time is impossible without some very low-level and hardware-dependent implementation.

An other affordable possibility would be to put 10 computers with 1 GB/s network cards on a switch, and let them all send traffic together. But controlling the characteristics of the resulting traffic would be impossible as packets ordering cannot be guaranteed by the switch.

Many commercial solutions are also available, but they are very expensive. And although they allow the properties of the generated flows to be precisely configured, they are never fully extensible (for example, generating long flows of packets of a newly-designed protocol would be impossible).

FPGAs are chips that can be configured at a hardware level.

Their main interest is that they provide massive parallelism, and a very precise control of what is executed at each clock cycle. Cards exist that include an FPGA and are designed for network processing, like the NetFPGA 10G [2], with 4 interfaces at 10 Gb/s, which is very affordable for academics. The traffic generator presented in this article works on a Combov2 board from INVEA TECH [12], which has a smaller FPGA than the NetFPGA 10G and 2 interfaces at 10 Gb/s. But it would be fairly easy to adapt to the NetFPGA 10G.

Section 2 will describe existing generators and their limits for research, and derive requirements for a new open-source generator. Then Section 3 will propose a software and hardware architecture. The way to use and extend the generator will be described in Section 4, along with performance results on a simple example.

2. TOWARDS AN OPEN-SOURCE TRAFFIC GENERATOR

2.1 Existing solutions

As traffic generators are widely used for Research and Development, many solutions exist. Different companies sell their own solutions, like Ixia [1] or Xena [3]. These tools work out-of-the-box, some software is usually provided with the machine to configure the parameters of the traffic to be generated using a simple interface. They are also often capable of receiving the traffic after it went through the device under test, and providing statistics like the rate of dropped packets.

Sadly these solutions are all expensive. And the way they are implemented is kept secret, making it impossible to extend their functions when the configuration options are not enough. As traffic is usually defined by flow, defining your own time-varying alteration of each packet would be impossible for instance.

An important number of generators with some specific features have also been developed in software, to run on commodity hardware. For example, an implementation focuses on generating flows that have the same distribution of packet length as some well-known application types (HTTP, FTP, SMTP, etc.) [5]. The traffic composition is configured using a graphical interface and could be used to test the behavior of a device under some realistic traffic. But it supports a data rate of only 1 Gb/s, and specific fields (like IP addresses or payload) cannot be specified.

The Harpoon [16] traffic generator goes even further in the idea of generating realistic traffic. It can generate TCP and UDP flows with realistic distributions of packet sizes, flow sizes and inter-packet delays. The goal is to recreate the load of a real network on the device under test. Its capacity in terms of data rate is not specified, but it is developed in software at a high level, which would not allow to reach capacities like 10 Gb/s.

An other drawback of this kind of software implementation is that standard operating systems do not permit to control timing precisely enough for traffic generation. This results in imprecise data rates and inter-packet delays in most software traffic generators [9].

To circumvent the problems due to the operating system, N. Bonelli et al. [8] implemented a solution that uses a custom socket called PF_DIRECT. It is a very low-level development that permits to communicate more directly from software to the network card. With a powerful computer and an adapted network card, they succeed to send packets at almost 10 Gbit/s. But they reach the limits of what is currently possible to do, and still have problems with the smallest packets. This is a drawback to stress-test an equipment, because small packets often constitute the most difficult traffic to handle.

As it is very challenging to obtain precise timing and high data rate in software, hardware acceleration is often required. Network processors represent a compromise between software flexibility and hardware acceleration. They are specialized processors, which are tailored for packet processing, and benefit from a direct access to a network interface. In [7], a network processor is used to build a generator, which can send traffic up to 1 Gb/s. A limited number of flows with identical data are configured using a graphical user interface and then sent at full speed. BRUNO [6] is another approach, which uses the same strategy but supports an unlimited number of flows, because each software generator instance can send a packet from any flow.

To get more flexibility than a network processor, Caliper [10] uses a custom network processor implemented on the NetFPGA 1G. This implementation focuses on precise timing and works at 1 Gb/s. Scaling it to 10 Gb/s would be difficult because it relies on the computer to generate the traffic, so the computer would act as a bottleneck.

Using an FPGA with a network interface at 1 Gb/s too, A. Tockhorn et al. [17] have a more hardware-focused approach: they stream headers with all data about one packet from the computer to the FPGA, which transforms this header into a full packet. As the header is smaller than the packet, the FPGA is able to reach a higher data rate than the computer. This approach would show its limits when trying to send small packets, as the header and packet would be of similar sizes.

FPGEN [15] is a traffic generator based on an FPGA, which does not require a computer to stream traffic to it. It stores configuration in RAM, and can reach up to 5 Gb/s. It is focused on packets size and arrival time statistical distributions, and does not allow to specify some fields like IP addresses, ports.

All these existing solutions do not fulfill our needs of an affordable and configurable 10Gb/s traffic generator to test the classifier we developed. So Section 2.2 describes in more details the specifications of a new open-source traffic generator.

2.2 Requirements

The main purpose of this generator is to be able to push the limits of a device under test. This means that it should not attempt to generate realistic traffic, but traffic designed to be difficult to handle. The definition of such traffic depends

on the functions of the device under test.

For example for the traffic classifier [11] we wanted to test, two factors are important: the number of packets per second for flow reconstruction, and the number of flows per second for classification. Packets should be UDP or TCP because others are ignored. As some caching mechanisms are implemented, it is much less challenging to always send the same packet, than to send packets with changing flow identifiers (source/destination IP addresses, UDP or TCP protocol, source/destination ports). The payload of the UDP or TCP packets is of no interest to the classifier.

As we want to use the generator to test all kinds of algorithms, the generated traffic has to be highly configurable: number of packets, size of the packets, content of the headers and payloads, etc. For ease of use, this configuration should be done on a graphical user interface on a computer, which will then send it to the FPGA.

In terms of protocols, the generator will be based on Ethernet, as this is the most common protocol and it is implemented on all platforms. But the protocols on top of Ethernet should be completely configurable: IPv4, IPv6, ICMP or others, and higher-level protocols UDP, TCP or even HTTP, SSH, etc should all be supported.

The generator should fill a 10 Gbit/s Ethernet link, even with the most challenging traffic (smallest packets). This goal is just a start, and the implementation should be able to scale to higher data rates, even if it may require more powerful FPGA chips.

To make the traffic as configurable as possible, the best way would be to specify each packet separately. But this would require to send configuration data to the FPGA for each packet. As explained is Section 2.1, this would create a bottleneck between the computer and the FPGA.

To avoid this problem, we only want to be able to configure flows of packets. This is the same approach as [7, 6], but to provide a maximum of flexibility, each flow is defined by:

- a skeleton: list of bytes defining the default packet data;
- a number of packets, which may be infinite;
- a list of modifiers, which modify the packets before they are sent.

Modifiers should be able to change the size of the packets, the time between each packet, or any bytes of the packets. This will allow to make packets vary by introducing, for example, incrementing fields, random fields and computed checksum fields. For instance, to define a flow of 50 ICMP packets that ping successively the IP addresses from 192.168.0.1 to 192.168.0.50, the configuration should be:

- Skeleton: the bytes of a valid ICMP/IP ping packet with destination address set to 192.168.0.1, and IP header checksum bytes set to 0
- Number of packets: 50
- Modifiers:
  - Increment: for bytes 16 to 19 (destination address), with a step of 1
  - Checksum: for bytes 7 to 8 (IP header checksum), computed on bytes 0 to 19 (IP header)

The increment modifier makes the destination IP address vary, which forces to compute the checksum of the IP header for each packet. Otherwise they would risk to be rejected as invalid.

To make the flows totally configurable would require an infinite number of types of modifiers. First, only the most commonly-used modifiers should be available. But developing a new modifier should be as easy as possible.

The generated traffic should consist of configured flows mixed together, the data rate of each flow being modulated by modifiers. Ordering of packets in a flow should be guaranteed, but no specific order is required for packets of different flows.

3. PROPOSED ARCHITECTURE

3.1 Global components

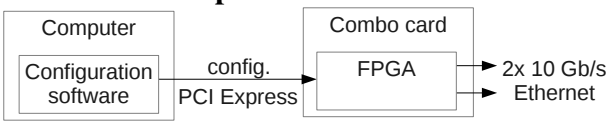

Figure 1: Global components of the generator

Figure 1 shows the global architecture of the generator. It is made of a computer that hosts configuration software and the Combo board. Configuration is sent through a PCI Express bus.

The Combo board is a COMBO-LXT model, with a COMBOI-10G2 interface board, which means that it has two 10 Gb/s ports. The included FPGA is a Xilinx Virtex 5 XC5VLX155T, which is much smaller than the FPGA included on the NetFPGA 10G. This ensures that transitioning to a NetFPGA 10G will only improve performance.

Development on the Combo board is done in VHDL, a hardware description language, using the NetCOPE platform [14]. This platform makes development more independent of the board architecture. Requiring only to develop a hardware block that receives incoming traffic through a specific bus type called FrameLink, and sends outgoing traffic through an other bus of the same type.

For the hardware generator block, incoming traffic will be the configuration data sent by the computer. NetCOPE provides a specific library to send data from the computer through PCI Express, that will be seen as incoming traffic by the block. Outgoing traffic will be the generated traffic.

An other advantage of the NetCOPE platform is that it has been made compatible with the NetFPGA 10G [13], which

means that the code should be able to run without modification on this board. But the platform is not open-source. So the best option might be to develop an adaptation block around the open-source generator block, that will adapt it directly to the NetFPGA 10G.

When the user starts a traffic generation, execution is done in two steps: first the computer sends configuration data to the FPGA (configuration phase), then once the FPGA is fully configured and ready, the generation of packets starts (generation phase). It stops only once all packets of all flows have been sent, or when the user requests it.

The hardware generator block and configuration software are open-source and available online [4].

3.2 Hardware design

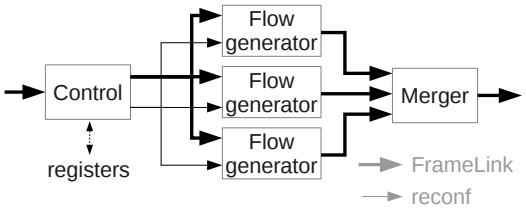

Figure 2: Architecture of the hardware generator block

Figure 2 details the design of the hardware generator block:

- The *control* block works mainly during the configuration phase. It receives incoming configuration data for all flows, and distributes one flow per flow generator. The maximum number of supported simultaneous flows is thus limited by the number of flow generators implemented. Once all *flow generators* are configured, the *control* block sends a specific *start* configuration word, which means that flow generators should start sending flows. During the generation phase, the control block can set the *reconf* signal to 1 at any time, meaning that *flow generators* should stop sending and get ready to receive a new configuration.
- Each *flow generator* manages only one flow during a whole execution. It receives the configuration of the flow during the configuration phase, and then waits for the *start* configuration word to start sending. If the *reconf* signal is set to 1 at any time, it goes back to the configuration phase.
- The *merger* receives packets generated by all flows and stores them in FIFOs. It has one FIFO per flow generator. Each FIFO is checked successively in round-robin: if a packet is ready, it is sent, otherwise the next FIFO is checked.

Many flow generators are implemented in parallel so as to be able to handle simultaneously flows with different configurations. But each flow generator should be able to fill the 10 Gbit/s link even with small packets, so that the link can be filled without losing control of the ordering of the packets. Configuration makes sure that the total rate is not over the link capacity. Ordering of packets from different flows is not guaranteed. The only guarantee is that each flow starts being generated at the same time because the *start* command is sent to all flow generators during the same clock cycle.

The FrameLink bus is designed so that no clock cycle is lost when using it. It carries 64 bits of data at each clock cycle. Both the receiver and the sender declare if they are ready, which allows to use it to synchronize the blocks in the pipeline. It transports both configuration data and generated packets. When sending packets, data on the bus represents the whole Ethernet frame, excluding preamble and start frame delimiter, and starting with a NetCOPE-specific header of 16 bytes. Only the *reconf* signal is used in addition to the bus as a synchronous reset.

Registers that are read and written by the *control* are used to communicate status and commands with the computer. The computer is capable of reading and writing to these registers to read the status of the generator, or send commands like "start sending" or "reset the configuration". This feature is provided by NetCOPE.

3.3 Inside the flow generator

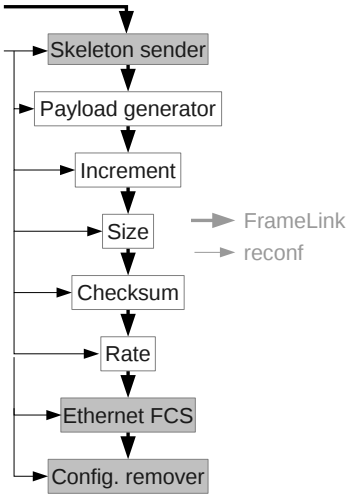

Figure 3: Example of blocks composing a flow generator

The flow generator is where the packets are actually created. An example of composition of a flow generator is visible in Figure 3. It is a long pipeline where blocks are synchronized through the FrameLink bus. The interest is that all blocks in the pipeline are constantly working: while the *skeleton sender* sends the last bytes of a packet, the *rate* block may already be working on the first bytes, wasting no clock cycle.

Blocks with a gray background in Figure 3 are essential. The *skeleton sender* receives and stores the skeleton associated to its flow during the configuration phase. The skeleton is a piece of data from 64 to 1500 bytes, which contains the basic packet that will then be modified by the next blocks. This block also stores the number of packets that should be sent for this flow. Then when the *start* configuration

word is received, the block starts sending the same skeleton continuously, until it has reached the number of packets to send, or the *reconf* signal is set to 1. The *Ethernet FCS* block computes the Frame Check Sequence (FCS) field of each generated packet and sets it, to conform to the Ethernet specification. The FCS field depends on all bytes of the frame. This block may only be omitted if the goal is to generate packets with wrong FCSs or if no change is made to the skeleton generated by the *skeleton sender*. The role of the *config. remover* is to prepare generated data to be sent to the network interface, which essentially consists in dropping the configuration frame.

Each block with a white background in Figure 3 is associated to a modifier:

- The *payload generator* enables to append a random payload of a fixed or random size to the skeleton, so that only headers have to be specified in the skeleton.
- The *increment* block allows to modify up to 4 fields of 2 bytes of the skeleton, setting them to incrementing values. The increment value and period are configurable. This may be used to make an IP address or UDP port vary.
- The *size* block allows to modify up to 4 fields of 2 bytes of the skeleton, setting them to the length in bytes of the whole Ethernet frame, minus a configured value. This may be used in conjunction with the payload generator to set the size fields in the packet header to the proper value when generating payloads of random sizes.
- The *checksum* block allows to modify one field of 2 bytes of the skeleton, setting it to the value of the checksum computed on a configurable range of bytes of the skeleton. This may be used to set the checksum in the IP header after some fields of this header have been modified. This block is detailed in Section 4.1.
- The *rate* block allows to limit the data rate of this particular flow. If it is not present, the flow will be sent at maximum speed. It works by setting a constant inter-frame gap as an integer number of clock cycles (5.33 ns on the Combo board) between each packet.

None of these blocks is mandatory, and depending on the flow configuration, some blocks may remain inactive during the generation. In this situation, they will not be configured during the configuration phase, and will just copy their input to their output during the execution phase.

This architecture allows to add new modifiers very easily: develop a modifier block, taking one of the existing blocks as example, and add it wherever it is needed in the pipeline. Some modifier blocks may also be duplicated. This can be useful for the *checksum* block, which handles only one checksum field. If more than one checksum have to be computed, just implement enough checksum blocks in the pipeline.

This flexibility has a cost: each time a modifier block is added/removed, the bitfile that configures the FPGA has to be regenerated, which takes about one hour. This is why it is best to generate a bitfile with a pipeline with all the blocks that could be needed, and then disable the useless blocks using the configuration.

3.4 The configuration mechanism

Data (64)		**Structure (4)**			
Id. (8)	*Config. (56)*	*SoF*	*EoF*	*SoP*	*EoP*
8	Rate	Yes		Yes	Yes
9	Checksum			Yes	Yes
10	Size			Yes	Yes
11	Increment			Yes	Yes
12	Payload generator			Yes	Yes
1	Skeleton (word 1)			Yes	
...					
Skeleton (word N)			Yes		Yes

Table 1: Sample configuration structure for a flow sender

Configuration is generated by the computer, and interpreted by the FPGA. Like the other components, it is designed to be easy to extend, especially by adding new modifier types. Its structure is made to send it easily through the FrameLink bus.

The FrameLink bus transmits 64 bits of data each clock cycle. But it also transmits some structure information: data is divided into frames, which are themselves divided into parts. This is done using 4 additional bits as flags signaling a start of frame (SoF), end of frame (EoF), start of part (SoP) or end of part (EoP). This structure is used in the configuration.

Table 1 shows the configuration of the flow generator described in Figure 3. One flow corresponds to one configuration frame. Each modifier is configured by one part of the frame. Parts start with an identifier, which is an 8-bit field used by blocks to determine if they are interested by this configuration part or not. Some parts, like the *skeleton* configuration do not concern modifier blocks, but special mandatory blocks. They have a reserved identifier comprised between 0 and 7. This means that at most $2^8 - 8 = 248$ modifier blocks can be configured on the same flow generator. Obviously this would take a lot of space on the FPGA, and there is no reason to need that many modifiers.

Each modifier block defines the structure of data inside its part, only the identifier is required, so that other blocks ignore this part. Most parts are only 64 bits long because the SoP and EoP flags are set on the same data word. This leaves 56 bits for configuration data. But a part can be as long as necessary, by using multiple 64-bit words. A part of 2 words leaves $56+64 = 120$ bits for configuration data, and so on. An example of long part is the *skeleton*.

During the configuration phase, the *controller* receives many configuration frames from the computer. It forwards one frame to each flow generator. If it does not receive enough frames, some *flow generators* will remain inactive during the generation. Only one configuration frame can be sent to one *generator*. Once the whole frame has been received, the

Data (64)		Structure (4)			
Id. (8)	*Config. (56)*	*SoF*	*EoF*	*SoP*	*EoP*
0	0	Yes	Yes	Yes	Yes

Table 2: Special configuration frame: *start*

generator stops listening to incoming data until it is ready to generate traffic. Once all *generators* are ready, the controller sends a small special frame: the *start* configuration word, visible on table 2. This part uses the reserved identifier 0, it is received only by the *skeleton sender*, which immediately starts to send packets. It is important to synchronize the start time of all flows.

3.5 Configuration software

The program used to configure the traffic generator is still in a very early stage of development. For now, it can only send configuration data written manually in hexadecimal format in text files. It is useful to test the traffic generator, but not really user-friendly for end-users who only want to generate traffic without having to understand the whole underlying architecture.

This is why a more flexible program with a graphical user interface will be developed. Its main features will be:

- Enable or disable all available *flow generators*.
- Configure the skeleton of each flow from some well-known protocols: it should be possible for example to add an IP header by entering only the source/destination addresses, and the underlying protocol.
- Enable the available modifiers and configure them simply by entering the required values. For example for the checksum modifier, the *start* and *end* values could be entered as text or directly by selecting the proper bytes in the skeleton.
- Monitor and control the state of the generator.

It should also be very easy to add new modifier types to the program, and to indicate a different hardware layout: number of *flow generators* and modifiers available. This could be done by using text files in a well-known and simple format like JSON for example:

- one directory will contain one file for each known protocol to fill the skeleton,
- one directory will contain one file for each modifier type,
- one file will contain the description of the current hardware layout.

4. USING AND EXTENDING THE GENERATOR

An important aspect of the generator is that it is extensible: adding a new way to modify traffic before sending it only requires to develop a new modifier block. These blocks are described in a Hardware Description Language like VHDL and they share a common design. Section 4.1 describes the design process of the checksum modifier as an example of the way to develop a new modifier.

Information needed to start using the generator is available online [4]. Section 4.2 describes an example use case and shows the performance of the generator.

4.1 Developing a modifier block

The first thing to do when defining a new modifier is to specify its action and how it can be configured. In this example, the developed block is the *checksum* modifier. Its role is to set checksums in IPv4, TCP or UDP headers after data changes. To do this, the block should read the proper fields, compute the checksum, and set it at the proper location in headers. The checksum value is always 16 bits long. The TCP and UDP protocols add a difficulty because a pseudo-header derived from the IP header has to be included to compute the CRC. So necessary configuration variables are:

- *start*: the byte offset at which computation should start (0 to 1517),
- *end*: the byte offset at which computation should end (0 to 1517),
- *value*: the byte offset at which the result should be inserted (0 to 1517),
- *ip*: if a pseudo-header has to be computed, the byte offset of the IP header (0 to 1517),
- *type*: a flag indicating if a pseudo-header should be computed, and if the IP version is 4 or 6 (none, 4 or 6).

The limit of all offsets at 1517 bytes is due to the Ethernet protocol, which states that an Ethernet frame can handle up to 1500 bytes of data (to which 18 bytes of header are added). Jumbo frames are currently not supported by the traffic generator.

Config. (56)					
start (11)	end (11)	value (11)	ip (11)	type (2)	0

Table 3: Checksum modifier configuration

Table 3 specifies the structure of the configuration part for the checksum modifier. It is only one word long. For the *type field*, possible values are 0 for no pseudo-header, 1 for an IPv4 pseudo-header and 2 for an IPv6 pseudo-header. The *identifier* from Table 2 is chosen among free identifiers, here 9 has been chosen.

Now that the configuration is specified, the *checksum* block should be designed. The configuration phase is simple: wait for a data word with the SoP flag set and an identifier of 9, when this word is found, store all configuration data in registers. When data is received with the EoF flag set, go to the generation phase.

The generation phase can be divided into 2 steps:

- compute the checksum by adding interesting bytes according to the configuration at each clock cycle,
- set the checksum value at the proper position in the packet.

But processing speed is critical when generating packets, so a pipeline will have to be used, so that data is never stopped during the process.

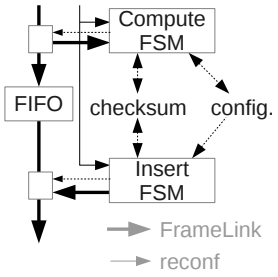

Figure 4: Architecture of the *checksum* modifier block

Figure 4 describes the architecture used for this block. Two Finite State Machines (FSMs) are used. The *compute* FSM reads received data without ever modifying it. It manages configuration storage and checksum computation. The *insert* FSM lets data flow until it finds the location where the checksum should be inserted. Then it waits until the checksum is ready. During this time, received data is stored in the FIFO. When the checksum is ready, it sends it along with the received data, and lets data flow again until a new checksum has to be written in the next packet. The FIFO has to be big enough to store a whole packet, to be able to handle the case when the checksum is at the start of the packet, and is computed on the whole packet.

This process incurs a delay when the first packet is sent, but then while the *insert* FSM sends data stored in the FIFO, the *compute* FSM computes the next checksum. So the bit rate is not decreased by this block.

The source of this block is available [4]. Note that although it is possible to specify in the configuration that the pseudo-header has to be created from an IPv6 packet, the function is not implemented and the *checksum* modifier block only supports UDP and TCP over IPv4.

4.2 Generator use case

To illustrate how the generator works, the goal will be to flood a device under test with ICMP ping packets with varying source IP addresses. The flow generator used is composed of:

- a *skeleton sender* (required): sends skeleton ICMP packets with a source IP address set at 192.168.0.1 and a checksum set at 0,
- an *increment modifier*: increments the source IP address of 1 at each sent packet,
- a *checksum modifier*: computes the new checksum (changed because of the IP address) and set it,
- an *Ethernet FCS computer* (required): computes the FCS field of the Ethernet protocol,
- a *configuration remover* (required): removes the configuration from the data bus.

In addition to the modules of the flow generator, the *control* and *merger* blocks are always present in the generator.

In terms of data speed, the only possible limits are due to the block designs. All blocks in the flow generator have been designed and checked so that they never stop traffic on the FrameLink bus. This means that the bus can work at its maximum speed. As it is 64 bits wide and works with a period of 187.5 MHz, its maximum rate is 12 Gb/s. But 16 bytes are used on each frame for the NetCOPE header. Ethernet specifies that the minimum payload of a frame is 46 bytes (including the FCS). So the minimum payload data rate is $12 \times 46/(46+16) = 8.9$ Gb/s. But Ethernet adds the equivalent of 34 bytes to each frame, so the maximum data rate is 5.7 Gb/s. As Ethernet adds much more headers than NetCOPE, it is guaranteed that data for a 10 Gb/s link can be transmitted on the bus.

So data speed is not limited by the traffic generator, but by the board hosting it. The Combo and NetFPGA boards are both designed to handle the full link speed, even with small packets. We tested this limit on the Combo board and reached the maximum speed of the 10 Gb/s link without problems.

Number of flow generators	1	10
Maximum frequency	201 MHz	201 MHz
Number of slice registers	600	5 950
Number of slice LUTs	953	8 611
Number of slices	440	3 644
Occupation	1.8 %	15 %

Table 4: Synthesis results of the sample traffic generator on the FPGA of the COMBO-LXT board

An other interesting metric is the surface taken by the traffic generator on the FPGA. This indicates the maximum degree of parallelism that can be reached. Table 4 shows the number of basic elements (registers and Look-Up Tables, LUTs) used on the FPGA by the generator described above with 1 and 10 parallel flow generators. The maximum frequency is superior to the frequency used by the Combo board (187.5 MHz), which ensures that there are no timing problems. The global occupation is very low: 15 % of the FPGA are used with 10 parallel flow generators. This represents $10 \times 5 = 50$ modules in the flow generators.

Although some space is used by the NetCOPE platform itself, it means that there is a lot of space available on the FPGA to add you own modifier modules, to make very configurable flow generators with a lot of modifiers, or to generate multiple flows in parallel.

5. CONCLUSION

This article presents an open-source traffic generator based on FPGA. Its use of the Combo board enables it to fill a 10 Gbit/s link easily, even with the smallest packets the Ethernet protocol allows. As the board has a second interface, and each flow generator is able to produce 10 Gbit/s, we could imagine an architecture like the one in Figure 2 but with two *merger* blocks. Each block would be linked to an interface, and the generator would be able to deliver 20 Gbit/s by configuring at least 2 flows.

Although the configuration program is not ready yet, it will make configuring the generator through the graphical user interface extremely easy. The ability to enable and configure each modifier for each flow makes it possible to send traffic adapted to the device under test, without having to know anything of the internal architecture of the generator, and without having to synthesize new bitfiles for the FPGA.

And if the existing modifiers are not enough to generate the wanted traffic, the generator has been designed from the start to make it easy to add new modifiers. It only requires to develop one VHDL block, following the process described in Section 4.1.

Although the generator currently works on the Combo board from Invea-Tech, it is very similar to the NetFPGA 10G, which is well-known and very affordable for academics. The NetCOPE platform used by the generator is compatible with the NetFPGA 10G, and even porting the generator to the NetFPGA platform should be fairly easy.

Finally, this generator is an open-source project. The source code is available online [4]. If you want updates about the development of the configuration program, if you can help porting the generator to NetFPGA, or if you have ideas on how to make it better, do not hesitate to get involved.

6. ACKNOWLEDGMENTS

The authors would like to thank INVEA-TECH for its support during the implementation of the traffic generator on the Combo board.

7. ADDITIONAL AUTHORS

Additional author: Manuel Aranaz Padrón (Télécom Bretagne, email: `manuel.aranazpadron@telecom-bretagne.eu`).

8. REFERENCES

[1] 10g ethernet test solution. `http://www.ixiacom.com/products/interfaces/display?skey=in_10g_universal`, 2012. [Online; accessed 6-February-2013].

[2] NetFPGA 10G. `http://netfpga.org/10G_specs.html`, 2012. [Online].

[3] Xenacompact. `http://www.xenanetworks.com/html/xenacompact.html`, 2012. [Online; accessed 6-February-2013].

[4] Open-source hardware generator. `https://github.com/tristan-TB/hardware-traffic-generator`, 2013. [Online].

[5] C. Albrecht, C. Osterloh, T. Pionteck, R. Koch, and E. Maehle. An application-oriented synthetic network traffic generator. In *22nd European Conference on Modelling and Simulation.*, 2008.

[6] G. Antichi, A. Di Pietro, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Design of a high performance traffic generator on network processor. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 438–441, 2008.

[7] R. Bolla, R. Bruschi, M. Canini, and M. Repetto. A high performance ip traffic generation tool based on the intel ixp2400 network processor. In *Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements*, pages 127–142. Springer, 2006.

[8] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. Flexible high performance traffic generation on commodity multi–core platforms. *Traffic Monitoring and Analysis*, pages 157–170, 2012.

[9] A. Botta, A. Dainotti, and A. Pescapé. Do you trust your software-based traffic generator? *Communications Magazine, IEEE*, 48(9):158–165, sept. 2010.

[10] M. Ghobadi, G. Salmon, Y. Ganjali, M. Labrecque, and J. Steffan. Caliper: Precise and responsive traffic generator. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*, pages 25–32, aug. 2012.

[11] T. Groleat, M. Arzel, and S. Vaton. Hardware acceleration of SVM-based traffic classification on FPGA. In *IWCMC*, pages 443–449. IEEE, 2012.

[12] Invea-Tech. COMBO cards. `http://www.liberouter.org/hardware.php?flag=2`, May 2012. [Online].

[13] P. Korcek, V. Kosar, M. Zadnik, K. Koranda, and P. Kastovsky. Hacking netcope to run on netfpga-10g. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, pages 217–218, oct. 2011.

[14] T. Martmek and M. Kosek. Netcope: Platform for rapid development of network applications. In *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*, pages 1–6. IEEE, 2008.

[15] M. Sanlı, E. Schmidt, and H. Güran. Fpgen: A fast, scalable and programmable traffic generator for the performance evaluation of high-speed computer networks. *Performance Evaluation*, 68(12):1276–1290, 2011.

[16] J. Sommers, H. Kim, and P. Barford. Harpoon: a flow-level traffic generator for router and network tests. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 392–392. ACM, 2004.

[17] A. Tockhorn, P. Danielis, and D. Timmermann. A configurable fpga-based traffic generator for high-performance tests of packet processing systems. In *ICIMP 2011, The Sixth International Conference on Internet Monitoring and Protection*, pages 14–19, 2011.