Bridging the Gap Between Hardware and Software Open-Source Network Developments

Marco Forconesi, Gustavo Sutter, Sergio López-Buedo, Jorge E. López de Vergara, Javier Aracil
High Performance Computing and Networking Research Group,
Department of Electronics and Communication Technologies,
Escuela Politécnica Superior, Universidad Autónoma de Madrid
Campus de Cantoblanco, 28049 Madrid, Spain
{marco.forconesi, gustavo.sutter, sergio.lopez-buedo, jorge.lopez_vergara, javier.aracil}@uam.es

Abstract—The rise of network speeds to tens of Gigabit per second poses a challenge to develop packet processing applications that can cope with such bit rates. Therefore, the need for a suitable open-source system that can be used as a prototype platform to test new network functionality while assuring line-rate processing, accurate timestamping, or reduced power consumption, becomes evident. All these requirements cannot be achieved by using software-only solutions but rather with hardware-based platforms such as NetFPGA. The main obstacle when using this type of open-source FPGA-based solution is the cost of development, both in time and hardware development skills required. The spread of new circuit synthesis tools using High-Level Languages opens the door for the development of hardware-based networking applications with reasonable development effort, compared to the use of traditional Hardware Description Languages. In this paper, we describe how existing open-source hardware-based platforms for networking applications will be fueled with the change in the programming model of FPGAs, provided by the modern High-Level Synthesis tools. For this, we implemented a network flow monitor using High-Level Languages and compared the effort spent with respect to a traditional hardware development cycle. Preliminary results are very promising, given that the development time is reduced from months to weeks.

Index Terms—NetFPGA, High-Level Language, Hardware Description Language, Packet Processing, High-Speed Networks, High-Level Synthesis, Network Flow Monitor.

1 Introduction

DATA LINK technologies are providing an ever-growing capacity in communication networks. Current deployments include 10 Gbps interfaces, whereas 40 and even 100 Gbps speeds are also starting to be available in the marketplace. Inside such high-speed networks, it is necessary to deploy packet processing applications for many network tasks. For instance, security (firewalls, intrusion detection/protection systems, or lawful interception) or network performance (to analyze delay, jitter, loss, or throughput). The processing systems must be flexible enough to allow the modification of the applications within a reasonable period of time. Software running on standard x86 processors is currently the most convenient approach because of the number of available software engineers, the ease of use, the reduced development cycles, and the inherent flexibility of software. Unfortunately, software-only solutions cannot achieve the performance requirements imposed by the latest network bit rates.

Open-source software solutions for high-speed networks come from the use of high performance network drivers (*e.g.* PacketShader, PFRing, or Intel DPDK). They work well on current high-end off-the-shelf hardware at 10 Gbps. However, the access speed between applications and network devices is currently limited, so it is very difficult to achieve throughputs above 10 Gbps steadily without packet losses. Due to the different layers the packets have to pass through, latency could be high and unpredictable, making it unsuitable for certain applications such as high frequency trading. Lastly, timestamping performed by software drivers is done in a packet batch basis, introducing inaccuracy and jitter. Thus, these drivers cannot assure line-rate low-latency processing for higher speeds nor accurately timestamp the packets when needed [1].

When software-based solutions do not achieve the desired performance, it is necessary to use a specific hardware solution in order to offload part or all of the packet processing application down to the network device. Observing the history of network equipment, it is easy to see that it has been common practice to use custom hardware in high-end packet processing devices. In fact, many network interface controllers already offload protocol tasks (*e.g.* TCP, IPSec) to reduce the work that is usually done by software, improving the overall performance of the system. Unfortunately these systems are closed platforms that have almost no flexibility to modify its behaviour. NetFPGA [2] offers the possibility to develop open-source high-performance

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness under the project PackTrack (TEC2012-33754).
Manuscript received August 15, 2013. Received in revised form January 31, 2014. Accepted April 14, 2014.

hardware, leaving the non-critical tasks to the software running on an x86 processor.

There are several advantages when using a Field Programmable Gate Array (FPGA) based hardware platform such as the NetFPGA: (i) the line-rate processing capability can analytically be assured, due to a finite state machine with a fixed frequency, which allows determining the cycles it takes to perform a given task; (ii) the hardware modules can be replicated for each network interface without a loss in performance, since they use different hardware resources that run in parallel; (iii) packets can be accurately timestamped directly in the hardware when received, avoiding the inaccuracies of software-based solutions; (iv) finally, FPGA solutions reduce the power consumption compared with a general purpose computer, being more environmentally friendly. Nevertheless, as stated before, there are also some drawbacks: it is tipycally necessary to develop the application with a Hardware Description Language (HDL), usually VHDL or Verilog [3]. HDLs describe the circuits in a precise and formal way that allow automated analysis, simulation, and translation to a lower level specification (*i.e.* synthesis) able to program an FPGA. Thus, it is more difficult and costly to develop an application for such platforms.

Several strategies may be introduced to reduce the cost of development. The first is to send only the components of the network application which require more computational power to the hardware. Another possibility is to develop the whole application with High-Level Languages (HLL) and then translate it into hardware with the aid of a High-Level Synthesis (HLS) tool. This possibility is gaining momentum, as it allows using well-known programming languages such as C/C++ instead of an HDL. This last solution reduces the development cost, given that there are more skilled C programmers already trained [4].

In this paper we address the implementation of packet processing applications on hardware using HLL, and compare it to the traditional HDL approach. HLL presents the advantage of a reduced codification time while retaining the assured line-rate speed, low-latency, and timestamping precision of hardware architectures. In order to demonstrate a specific example, we have developed the same packet processing application for the NetFPGA 10G platform, using the two mentioned approaches. Our results show that the development time can be reduced by roughly one order of magnitude when using an HLL design methodology. Based on our experience, we will also provide hints on what has to be improved for this to be a realistic solution.

The rest of the paper is structured as follows. First of all, we describe the classical HDL development flow of packet processing applications, targeted for the NetFPGA-10G platform. Then, HLL designs are introduced as a way to reduce the complexity of firmware development. Next, a comparison of both development approaches to implement a network flow monitor is provided. Finally, conclusions are made and potential prospects discussed.

2 Development on NetFPGA-10G

The interest of FPGA-based platforms for networking packet processing applications becomes evident with the development of the open-source NetFPGA project. Despite the long duration of design cycles of hardware prototypes, the NetFPGA project was conceived as a research and teaching tool and was rapidly adopted by the academia for prototyping new ideas for future networks. NetFPGA is a community contributed project developed by Stanford University and Xilinx Research Labs [2]. NetFPGA-10G [5] is the second generation release and it consists of a PCI Express board populated with a Xilinx Virtex-5 FPGA and four full-duplex 10 Gbps Ethernet interfaces, as the main components. Besides the FPGA internal memory (known as Block RAMs, with 18 Kbits each block), the platform also includes two types of memory banks in order to support a large number of network applications. The first type provides 27 Mbytes of high-speed static RAM, targeted for fast lookup tables, whereas the second type has 288 Mbytes of low-latency dynamic RAM, which is aimed for packet buffering. The board has a PCI Express Gen 1 communication channel to connect with a host computer. However, the NetFPGA 10G platform can run completely standalone (i.e. not connected to any PCI Express socket) since it only needs a 12 V power supply; this possibility may be the best choice to run line-rate applications at a very low power consumption. Fig. 1 outlines the structure of the hardware platform.

Researchers willing to develop FPGA-based network applications can benefit from using the NetFPGA-10G platform due to its open-source nature. There exists a public repository for the community of developers where they can share specific hardware modules, software drivers, and development tools.

NetFPGA-10G projects are developed using the Xilinx Embedded Development Kit (EDK). EDK projects are divided into a) development of the hardware platform that is implemented on the FPGA and b) development of the software that is executed by an embedded processor on the FPGA. The hardware platform is composed by cores such as: Ethernet MAC, PCI Express interface (PCIe), embedded soft processor (MicroBlaze), direct memory access (DMA) and user custom modules. The designer chooses which hardware cores will be part of the platform that is going be implemented on the FPGA. The software part of an EDK project is executed by the embedded processor; on the NetFPGA-10G it plays an important role since this processor sets up the Ethernet interfaces.

Although the hardware and software targeted to the FPGA are developed by means of an EDK project, a NetFPGA-10G project is more than that and also includes software code (*e.g.* drivers and user applications) to be

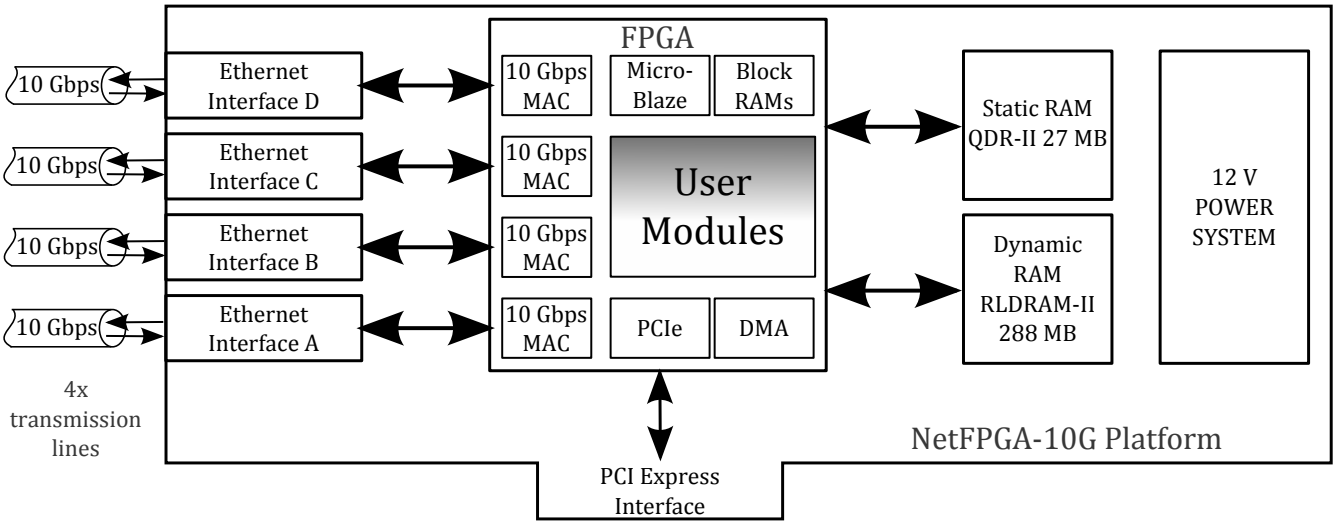

Fig. 1: NetFPGA-10G structure.

executed on a x86 processor. The interaction between the processes running on the FPGA and the ones on a x86 host computer is provided by the PCI Express communication channel. All these components make up the necessary framework to develop high performance hardware-based open-source network applications.

As shown in Fig. 2, a typical design cycle of network applications on the NetFPGA-10G starts with the download of the latest releases of the available projects from the public repository (task 1). The developers have to select the one that suits them best as the basis for the new design. The main idea is to reuse all the functionality provided in order to spend most of the development time on the implementation of the new project. First, one has to choose which software runs in the host computer (if any) and in the FPGA, respectively. Such a choice is a trade-off between the development time and the certainty in the performance (number of clock cyles it takes to execute) of each task. Then, the developers will create their own hardware modules (task 2.a), if they are not already available on the repository, following an HDL design flow that comprises Verilog or VHDL codification and validation; we note that this is the most time-consuming step in the design flow. These hardware modules can be replicated as much as needed to handle each Ethernet interface. Once all the modules have been designed and/or adapted, the integration process takes place (task 2.b) by interconnecting the modules using on-chip communication protocols. The final step in the FPGA embedded system design is to modify the program that the embedded processor executes, if needed (task 2.c). The next design phase, after the hardware and embedded software are ready to run on the FPGA, consists of the development of the host computer processes that are not time critical (task 3). It is also possible to use or modify a Linux PCI Express driver available on the NetFPGA-10G repository. Besides, applications at the user level can run in conjunction with the mentioned driver to solve relatively low-speed tasks, employing the classical C/C++ software development cycle. By the time the design functionality has been verified and tested, developers are free to upload the project to the public repository for the sake of the community (task 4).

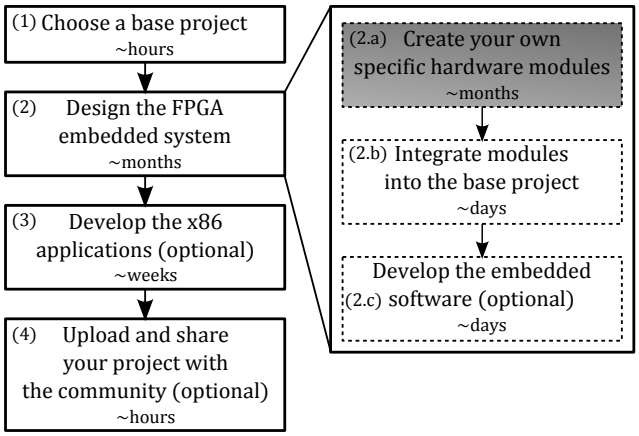

Fig. 2: Typical design flow in the NetFPGA platform.

The development time is broken down as follows: the most time-consuming task is task 2.a which typically takes several months (70%-90% of the total development time). Then, task 2.c, which, depending on the application, requires a few days' effort, followed by task 3, which, when used, will require a few weeks' work. The rest of the tasks demand just a few hours worth of work of a skilled engineer. Then, the cost in person-months is much higher with respect to software-based developments. The reason why the specific hardware modules creation (task 2.a) is the most time-consuming step is because of the FPGA programming model. HDLs are used for designing the circuit with a model which contains information about the flow data and timing, the so-called Register Transfer Level (RTL). The designer benefits from the fact that HDLs provide a higher level of

abstraction than the circuit that finally runs on the FPGA. Synthesis tools for HDLs translate the transfer functions between registers of the RTL model to logic gates, but the hardware registers keep a one-to-one correspondence with those of the HDL RTL model. For this reason, HDL codification implies fixing a-priori the architecture of the hardware being implemented, and the development time of HDL designs is well beyond that needed for software solutions. This is the reason why FPGAs are not so popular for networking applications. In order to bridge the gap between software and hardware network developments and have the best of both worlds, the development time spent on task 2.a must be reduced.

3 High-Level Languages to the Rescue

Modern High-Level Synthesis (HLS) can be used to reduce hardware development time. HLS tools change the programming model of the FPGAs, making it possible to use High-Level Languages (HLL) in the design capture phase. Therefore, they dilute the difference in the programming model between a processor and an FPGA [6]. High-Level Languages range from graphical descriptions to *ad hoc* languages using extensions to traditional languages.

The concept of HLS has been evolved over several decades [7], but only in the last few years new promising and successful tools have appeared. The electronic industry is moving fast in the adoption of these tools based on HLS. Most of them support synthesis from a set of ANSI-C, C++ or SystemC, producing HDL code as a result.

The success of using C/C++ as a design entry has many driving forces. Firstly, almost all computer and electrical engineers are familiar with them, with a great deal of code already available. C/C++ is the *de facto* language for prototyping and developing in many application fields including networking. Additionally, it is a natural way for Hardware/Software co-design, starting with a software application and moving to the hardware those pieces that need more performance, but using the same language.

The most popular tools include C-to-Silicon from Cadence, Synphony C Compiler from Synopsys, Catapult C from Calypto, CoDeveloper from Impulse, Vivado-HLS from Xilinx, BSC (Bluespec Compiler) from Bluespec and ROCCC 2.0 (Riverside Optimizing Compiler for Configurable Computing) from Jacquard computing, being this last one an open-source project.

3.1 Why HLL help on the hardware design process

The use of HLL makes the initial implementation steps go much faster and allows the exploration of a broader design space in less time. As suggested by Fig. 3, software simulation is fast and it suffices to quickly jump to the next design process iteration. Accordingly, the cumbersome and slow HDL simulation phase is avoided.

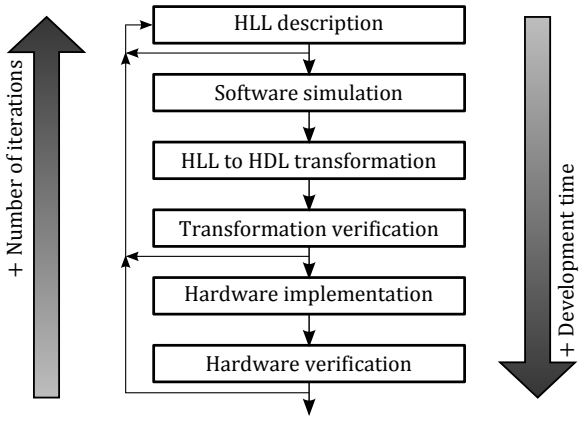

Fig. 3: Hardware design flow using High-Level Languages.

Similarly, the HLL-to-HDL translation (also known as C-to-hardware) is very efficient and provides additional performance information about the hardware (number of cycles spent in the execution, maximum frequency, area usage, etc.). Such methodology allows for a fast evaluation of the proposed solution and allows the designer to try several design options or add extra features which are indeed more costly with HDLs.

An archetypal claim against HLS tools is that the reduction of design time comes at the expense of a loss of performance, since these tools prevent designers from optimizing low-level details of the architecture. As suggested by [6], [8], these claims are arguable thanks to the expressivity of HLL and the faster development time. The designers can effectively explore a much larger design space in comparison to the HDL approach. Even though HDLs can also offer such optimizations, it turns out that the programming model, which is centered on RTL description, involves the implementation of a fixed architecture and prevents further optimizations without re-writting the code. Furthermore, HLLs hide all non-critical implementation details (*e.g.* optimizing state machines, timing closure issues, resource allocation and scheduling, etc.); instead, they allow the designer to focus on performance critical problems at the system level (processes intercommunication, storage issues, etc.), that have more impact in the overall performance.

3.2 How to create hardware for networking applications using HLL

The C/C++ language itself does not provide the necessary parallelism, communication, timing restrictions or other hardware features to develop NetFPGA applications. The inclusion of these extra features complicates matters. Regrettably, the way the HLS tools deal with these issues is far from being standardized. Up to date, a successful hardware implementation does not come from a generic C/C++ code, but from a code that has specifically been tailored towards a given architecture.

In this work we have used the Xilinx Vivado-HLS tool [9]. Such tool takes a standard C/C++ code that models the algorithm and synthesizes it in an HDL description that can be used in Xilinx FPGAs (such as the one present on the NetFPGA-10G). Besides, the Vivado-HLS tool generates suitable hardware cores that can be integrated in an EDK project without modifying a single line of HDL code.

Since the tool is aware of the targeted device and the clock frequency, it generates circuits that meet the timing requirements. It reports the amount of clock cycles each task consumes and thus, the latency involved. As a consequence, likewise HDL-designed hardware, there is no jitter per task at all (for example, when timestamping packets). In case the processing requirements are not satisfied by the initial code, the so-called *directives* (`#pragma` statements) can also be used in order to exploit parallelism, pipelines, control latency, define interfaces and other hardware features.

In presence of multiple clock domains, even though the tool generates a module for each clock domain, it is possible to use dual-clock FIFOs at the EDK level to glue the generated cores for each domain. As a result, the development of the processing logic performed on each packet (*i.e.* the most complex, time-consuming step) is simplified by the use of HLL. For instance, if an application must process all the Ethernet packets it receives and must send aggregated information to a software layer on the x86 computer, a dual-clock FIFO could be used to interface the 10G-MAC clock domain to the DMA domain. All of the user-added intelligence (*i.e.* processing and communication) will be designed and validated using an HLL model capture.

4 Experimental Comparison of Developments

Once the NetFPGA and HLL advantages have been shown, we present a comparison to HDLs with a use case packet processing application.

4.1 The application: "A simplified 10 Gbps network flow monitor without packet sampling"

Flow-based monitoring is common practice to analyze network traffic. High-end routers and switches provide this functionality (*e.g.* NetFlow in Cisco or Jflow in Juniper) with packet sampling, because they cannot operate at line-rate, thus compromising accuracy [10].

Using both development approaches (HDL and HLL) we have implemented a flow monitor for the NetFPGA-10G with the following design characteristics:

1) It works at a 10 Gbps line-rate even with the smallest packet sizes (14.88 Mpps - Millions of packets per second).
2) It uses internal FPGA memory only (Block RAMs), which allows for a 4K concurrent flows cache.
3) Flows are removed from memory and exported either when an inactivity timeout (15 seconds), or an activity timeout (30 minutes) expire, or the TCP connection ends.

The designs are open and available for downloading at [11]. Additionally, a more complete and functional design including the use of external memories (to handle up to $^3/_4$ million concurrent flows), is described in [12]. Fig. 4 presents the block diagram of the implemented circuit into the FPGA. A brief explanation of the functionality of each block is provided below:

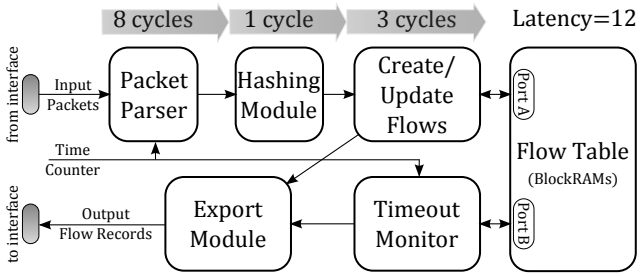

Fig. 4: Block diagram of the implemented design.

Packet Parser. Analyzes the bytes within each Ethernet frame and extracts its 5-tuple (source and destination IP addresses and ports as well as the protocol) plus the information needed to create a new flow or update an existing one: timestamp, TCP flags, and the number of bytes within the packet.

Hashing Module. Calculates a hash code from the 5-tuple, which is used as the memory address where the flow will be stored.

Create/Update Flows. Updates the flow table according to the previous hash code. If the flow was previously active in the memory it will be updated, otherwise a new flow will be created.

Flow Table. Stores the active network flows on the monitored link. As Fig. 4 shows, the flow table is accessed by two processes. The first one creates and updates the active flows; the second one removes the expired flows and exports them according to the inactivity and activity timeouts. The flow table is implemented with FPGA double-ported Block RAMs.

Timeout Monitor. Checks whether the active flows in the flow table do not exceed the activity and inactivity timeouts. Namely, a flow will be removed from memory either if it has been on the flow table for too long or if no more packets that belong to that flow are received.

Export Module. Receives the flow records that were purged from the flow table and sends them out of the board through one of the 10 Gbps Ethernet ports. This block could also be connected to the PCI Express DMA engine in order to send expired flows directly to the host computer.

4.2 Development of solutions

Implementing the packet processing application described above using the HDL design cycle implies two

major steps: firstly, the architecture and interface of each module is coded and validated separately; secondly, all the modules are interconnected and the overall performance evaluation is carried out. With HLL, one has to code each module in a separate C/C++ function. The communication between the functions is performed by argument passing and the complexity of some hardware modules is completely hidden. For example, the *flow table* module is reduced to the declaration of a C/C++ global variable. An example of such simplicity is provided in Fig. 5, which shows, the *create/update flows* function in HLL.

```
/*
 * This function process the received packet,
 * registering it in the flow table and
 * exporting the flow when appropriate
 */
void create_update_flow(
     flow_info_t flow_table[4096],
     pkt_info_t pkt_info,
     ap_uint<10> hash_code,
     flow_info_t *exported_flow) {
#pragma HLS LATENCY max=3

//read the flow from BRAM memory
flow_info_t cur_flow = flow_table[hash_code];

if (cur_flow.location_busy){//location in use
  if (match_xtuples(&pkt_info,&cur_flow)){
    //the flow has been previously created
    cur_flow.byte_counter +=
                      pkt_info.byte_counter;
    cur_flow.pkt_counter++;
    cur_flow.tcp_flags|=pkt_info.tcp_flags;
    flow_table[hash_code] = cur_flow;
  } else *collision_cntr++; //a collision
} else{//location available. 1st packet in flow
   cur_flow.byte_cntr = pkt_info.byte_counter;
   cur_flow.pkt_counter = 1;
   cur_flow.tcp_flags = pkt_info.tcp_flags;
   flow_table[hash_code] = current_flow;
}
//check fin or reset TCP flags
if ((pkt_info.tcp_flags & 0x01) ||
              (pkt_info.tcp_flags & 0x04)){
  *exported_flow = current_flow;}
  flow_table[hash_code].location_busy = 0;}
}
```

Fig. 5: An HLL code that implements *create/update flows* module.

To operate at 10 Gbps line-rate, knowledge of the processing time per packet, from its arrival to the flow table update, is required. The smallest packet size, including the interframe gap is 616-bits long, *i.e.* 61.6 ns at 10 Gbps is the minimum period of time between packets. The user side of the MAC core in the NetFPGA-10G is a 64-bits wide interface running at 200 MHz. This means that a total of 12 clock cycles are available to process every packet in the worst-case scenario. In an HDL design, the latency and the total processing time per packet is determined by the number of intermediate registers in the flow data of the RTL model. Changes in these properties imply the manual re-codification of the model which can lead to unwanted changes in the semantic of the algorithm. On the other hand, the latency can be controlled with the LATENCY pragma statement in HLL, as shown in Fig. 5, which does not change the model semantics.

The total number of clock cycles per algorithm is reported by the HLS tool when synthesis is executed. For optimizations, some `#pragma` statements can be applied to allow the HLS synthesizer to use more hardware resources and reduce the processing time. Once again the algorithm semantics remains unchanged.

4.3 Comparison of results

HDL and HLL designs were both coded by the same developer (electronic engineer, PhD student). His background includes 3 years of experience in HDL (VHDL) and FPGA design, good knowledge of FPGA design tools, basic knowledge of the NetFPGA-10G platform, and some background in HLL design flow. The designer has had full support of experts in FPGA design, networking applications and HLL design.

First, the HDL design was coded in VHDL whereas the HLL was implemented later on in plain C using `#pragma` statements for Vivado-HLS.

In order to compare the design effort, the number of files, code lines, size of files, and development time of both approaches are presented in Table 1a. The number of files include header and code files; the number of code lines does not take into account neither comments nor blank spaces; and the development time includes testing and validation design times. The rows "HDL" and "HLL" report the results of each design flow, whereas the row "Synth_HLL" stands for the resulting HDL code obtained after the C code was synthesized with Vivado-HLS.

Regarding the resource usage, Table 1b shows the final hardware utilization of both approaches. The number of Flip-Flops (FF), Lookup-Tables (LUT), internal 18 Kbits Block RAMs (BR), as well as the percentage of FPGA resource usage (%use) and maximum frequency are reported. The rows HDL and HLL report the results for the implemented core (modules that outfit the Netflow) using the respective methodology. Meanwhile, the rows "System_HDL" and "System_HLL" describe the resources used by the whole system, which includes all the necessary modules to operate.

4.4 Discussion of results

Clearly, the use of HLL reduces the development time of the module by roughly one order of magnitude. The corresponding area is significantly larger in comparison to the HDL counterpart, but it only represents less than 2% of the total available area of the target FPGA. The reported maximum frequency of both HDL and

TABLE 1: Implementation results

(a) Number of code lines using HDL and HLL design flow.

	#Files	#Code lines	#KBytes of code	Develop. time
HDL	10	1055	83.2	3-4 months
HLL	2	324	11.0	2-3 weeks
Synth_HLL	15	4712	226.6	-

(b) FPGA resources utilization and maximum frequency.

	#FF	#LUTs	#BR	%use	Max. Freq.
HDL	1716	1137	27	1.1%	>200 MHz
HLL	2796	2632	28	1.9%	>200 MHz
System_HDL	24624	22457	46	16.5%	>200 MHz
System_HLL	25712	22965	47	17.2%	>200 MHz

HLL meet the timing specifications that allow line-rate operation.

The skills and effort necessary to design with HLL are simpler than those required in the HDL development flow, but current state-of-the-art demands hardware design knowledge to integrate the whole system. A framework and/or automatic tools are still needed to hide the complexity behind the reconfigurable hardware development and finally bridge the gap for a networking engineer.

Such a framework should also provide applications with interfaces commonly used for packet processing (*e.g.* pcap), and take into account timing issues as well. It is also necessary to manage external memory and to facilitate access to a host computer (using PCI Express) in order to directly flush packets to disk, for example. All these features are still missing from High-Level Languages in order to bring the hardware development closer to a network engineer.

5 Conclusion

The development of packet processing applications in FPGA offers undeniable advantages in terms of performance and response time predictability compared to solutions based on commodity x86 hardware. However, time and development costs make it unattractive for most network engineers. Interestingly, new tools for High-Level Synthesis are a promising way to tackle these development difficulties.

In this work we have presented how state-of-the-art HLS tools can boost the development of packet processing applications with existing FPGA-based platforms, such as the open-source NetFPGA-10G. We have also outlined the main limitations that impede the widespread adoption of High-Level Languages. Overall, the programming model of the FPGAs has changed in a way that allows capturing the designs with HLL, and the reduction of the development time of the applications from months to weeks, compared to a traditional hardware development flow based on HDLs.

By way of a practical example, *i.e.* generation of flow records at 10 Gbps line-rate, this paper shows how to develop hardware-based network applications without previous knowledge of HDLs. Additionally, using a high-level design methodology proved to be acceptable in terms of performance and hardware resource utilization. This paves the way for a packet processing application framework based on HLL (typically C/C++). Such a framework would further abstract hardware details, therefore allowing to bridge the historic gap between software and hardware development in networking applications.

References

[1] V. Moreno, P. Santiago del Rio, J. Ramos, J. Garnica, and J. Garcia-Dorado, "Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines," *Communications Letters, IEEE*, vol. 16, no. 11, pp. 1888 –1891, november 2012.

[2] M. Blott, J. Ellithorpe, N. McKeown, K. Vissers, and H. Zeng, "FPGA Research Design Platform Fuels Network Advances," *Xcell Journal*, pp. 24–29, 2012.

[3] J.-P. Deschamps, G. Sutter, and E. Cant, *Guide to FPGA Implementation of Arithmetic Functions*, ser. Lecture Notes in Electrical Engineering. Springer, 2012, vol. 149. [Online]. Available: http://dx.doi.org/10.1007/978-94-007-2987-2

[4] J. Schönwälder, A. Pras, and J.-P. Martin-Flatin, "On the future of Internet management technologies," *Communications Magazine, IEEE*, vol. 41, pp. 90–97, Oct 2003.

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

[6] Xilinx Inc., *Introduction to FPGA Design with Vivado High-Level Synthesis. UG998*, July 2013. [Online]. Available: http://www.xilinx.com/support/

[7] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, 2009.

[8] A. Cornu, S. Derrien, and D. Lavenier, "HLS tools for FPGA: Faster development with better performance," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2011, pp. 67–78.

[9] Xilinx Inc., *Vivado Design Suite User Guide. High-Level Synthesis. UG902*, July 2012. [Online]. Available: http://www.xilinx.com/support/

[10] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," in *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4. ACM, 2004, pp. 245–256.

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

[12] M. Forconesi, G. Sutter, S. Lopez-Buedo, and J. Aracil, "Accurate and flexible flow-based monitoring for high-speed networks," *Field Programmable Logic and Applications*, 2013.

Marco Forconesi (marco.forconesi@uam.es) is currently a research assistant in the Networks and Operating Systems Group in the Computer Laboratory at University of Cambridge. He received his Electronic Engineering degree from Universidad Nacional de San Juan (Argentina) in 2012 and finished his MSc degree in ICT Research and Innovation at Universidad Autónoma de Madrid (Spain) in 2013, where he held a research grant. He is currently a member of both Network as a Service Project (NaaS) and NetFPGA team. His current research topics include FPGA hardware design for high-speed PCIe and Ethernet networking.

Gustavo Sutter (gustavo.sutter@uam.es) received an MS degree in Computer Science from State University UNCPBA of Tandil (Buenos Aires) Argentina, in 1997, and a PhD degree from Universidad Autónoma de Madrid, Spain, in 2005. He has been assistant professor at the UNCPBA Argentina and is currently an associate professor at Universidad Autónoma de Madrid, Spain. His research interests includes algorithms and networking in reconfigurable computing (FPGA), digital arithmetic, development of embedded systems and High Performance Computing. He is the author of three books and more than hundred international papers and communications.

Sergio López-Buedo (sergio.lopez-buedo@uam.es) received in 2003 his Ph.D. in Computer Engineering from Universidad Autónoma de Madrid (Spain), where he currently serves as associate professor in the area of Computer Architecture. He was a visiting researcher at University of British Columbia (2005) and at The George Washington University (2006, 2007), and he has also collaborated in the doctorate program of Universita degli Studi di Trento (2007-2009). FPGA technology is his main research interest, especially high-performance reconfigurable computing and communication applications. Dr. Lopez-Buedo holds more than 50 publications, including journals, conferences and books as editor, and he is also co-founder of Naudit HPCN, a company dedicated to providing high-performance computing and networking solutions.

Jorge E. López de Vergara (jorge.lopez_vergara@uam.es) is currently an associate professor in the Electronics and Communication Technologies Department of the Universidad Autnoma de Madrid (Spain). He received his MSc degree in telecommunications from Universidad Politcnica de Madrid (Spain) in 1998 and finished his PhD in telematics engineering at the same university in 2003, where he held a 4-year research grant funded by the Spanish Ministry of Education. In 2000 he was a visiting researcher at Hewlett-Packard labs for 6 months. He has participated in several Spanish and EU research projects. In 2009, together with other university professors, he founded Naudit High Performance Computing and Networking, a spin-off company devoted to traffic monitoring and analysis. His current research topics include network, service, and distributed application management and monitoring. He has co-authored more than 100 papers in scientific conferences and journals.

Javier Aracil (javier.aracil@uam.es) Javier Aracil received the M.Sc. and Ph.D. degrees (Honors) from Technical University of Madrid in 1993 and 1995, both in Telecommunications Engineering and the Licenciatura (five-years degree) in Mathematics from UNED in 2009. In 1995 he was awarded with a Fulbright scholarship and was appointed as a Postdoctoral Researcher of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. In 1998 he was a research scholar at the Center for Advanced Telecommunications, Systems and Services of The University of Texas at Dallas. He has been an associate professor for University of Cantabria and Public University of Navarra and he is currently a full professor at Universidad Autnoma de Madrid, Madrid, Spain. He is also a co-founder of Naudit HPCN. His research interest are in traffic analysis and performance evaluation of communication networks. He has authored more than 100 papers in international conferences and journals.