A flexible router for on-fly dynamic packet management in NetFPGA platform

Alfio Lombardo, Carla Panarello, Diego Reforgiato, Enrico Santagati and Giovanni Schembra
Department of Electrical, Electronic, Informatics Engineering (DIEEI)
University of Catania, V.Le A. Doria 6, 95125 Catania
Email: {alfio.lombardo, carla.panarello, diego.reforgiato, enrico.santagati, giovanni.schembra}@dieei.unict.it.

***Abstract*—[1] The NetFPGA platform not only is used by instructors to show how to build line rate Ethernet switches and Internet Protocol (IP) routers, but allows everyone to prototype and develop multi-Gigabit networking applications. It is an open platform and user community developed to enable researchers to build high speed, hardware-accelerated networking systems. The paper starts from the reference router implementation on the NetFPGA platform to realize an open flexible and high-performance router that allows the implementation of any strategy for on-fly packet modification. More specifically, the proposed router works at user data path level and modifies packet fields if certain conditions defined by the user through NetFPGA registers are satisfied. The project has been implemented as a fully open-source project and can also be used as an exemplar project on how to build and distribute NetFPGA applications. All the code (Verilog, system software, graphical user interface, verification scripts, makefiles, and support tools) can be freely downloaded from the NetFPGA.org website.**

I. Introduction

In the last decade the enormous innovation in computer networks and network devices has led a capillary diffusion of the Internet in the world. Unfortunately, almost the totality of network devices today used in the Internet as IP routers, bridges, hubs and switches are compacted and closed platforms which are not possible to change or enhance. Their functionalities are limited and restricted by vendors who are often hostile to allow researchers and programmers to modify and extend their products. As a consequence, this is causing a substantial decrease in the rate of innovation and improvement.

At the same time a lot of algorithms have been proposed in the past literature that require to be implemented in network devices to improve performance at both the network and application levels.

A relevant example of this matter is constituted by the need of dynamically changing the IP address of both incoming and out-coming packets crossing an access router to realize a smart proxy server working according to user-defined strategies. A possible application of this is the access router of a video server farm where many servers are used to balance the load of coding and streaming applications. In this case the same IP address can be used by all the clients, and incoming packets can be dynamically redirected to different servers according to the required service or the current load of each server, following a user-defined strategy. Although this problem can be solved in other ways (see for example [1]) with other techniques like TCP forwarding, connection splicing, IP tunneling, that best way to not alter router performance is to change IP addresses runtime. Another possible application of on-fly IP address change is for mobile IP within the same local area network: if a user needs to redirect traffic to another host without changing the local host IP addresses in order to not lose privileges associated to its IP address, it can issue a request to the access router to redirect packets to the new IP address, making a sort of call redirection, but at the IP level. Besides the above examples, an infinity of other examples can be carried relating the need of managing TCP or IP packets on fly, like for example, TCP header fields modifications to improve TCP performance in particular environments ([2],[3],[4],[5],[6],[7]), packet priority variations, service differentiations, smart firewall implementations, data encryption, and so on. Of course, all the applications cited above cannot be realized and customized in not open network devices. A first attempt to solve this problem is the use of software routers [8],[9],[10] with their implementation on standard personal computers.

However, with this approach routing and forwarding operations are all performed in the PC motherboard, and therefore both the CPU and the communication bus constitute a severe bottleneck that strongly limit performance precluding their applications in large scale environments.

With all this in mind, the target of this paper is to realize an open flexible and high-performance router that allows the implementation of any strategy for on-fly packet modification. To this purpose the router has been implemented over the NetFPGA platform [11], extending the NetFPGA reference router project [12] with capabilities never shown so far, like for example runtime computing TCP and IP header checksums in hardware.

The NetFPGA platform not only is used by instructors to show how to build line rate Ethernet switches and Internet Protocol (IP) routers, but allows everyone to prototype and develop multi-Gigabit networking applications. It is an open platform and user community developed to enable researchers to build high-speed, hardware-accelerated networking systems.

[1]The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n 258454 (Collaborative Project "ECONET"). Moreover, the work leading to this invention has benefited from a fellowship of the Seventh Framework Programme of the European Community [7 PQ/2007-2013] regarding the Grant Agreement n. PIRG03-GA-2008-231021.

The open-source NetFPGA distribution consists of gateware, hardware and software. As far as the hardware is concerned, it consists of a PCI card that has an FPGA board, memory (SRAM and DRAM), and four 1-GigE Ethernet ports (a new version of the NetFPGA platform has been recently launched with four 10-GigE Ethernet ports). Source code and scripts are provided to build reference designs, enhance a design, or create new applications using supplied libraries. Hardware description source code (gateware) and software source code are freely available online. The NetFPGA platform not only consists of the NetFPGA board, but also the development environment and scripts that allow for rapid prototyping and development of hardware projects. The development environment is available from the NetFPGA website [11]. Reference designs comprised in the system include an IPv4 router, an Ethernet switch, a four-port NIC, and SCONE (Software Component of NetFPGA). Researchers have used the platform to build advanced network flow processing systems. A single NetFPGA board can route packets over any number of subnets, and multiple NetFPGA boards can be installed in the same PC. In addition, there are several user-contributed projects available such as the Netflow probe, OpenFlow switch, the Packet Generator, and the RCP router.

The proposed flexible router allows users to define any strategy to manage packets, changing any field of both TCP or IP headers, as for example the IP source and destination addresses, the TCP source and destination ports, the TCP advertised window size, etc.. Moreover, the TCP and/or IP checksums are possibly recomputed and stored into the packets. We have called the new project the flex router. It is available at [13].

This paper is structured as follows: Section II introduces the NetFPGA platform; Section III describes the modifications we have applied to the reference router project; Section IV reports the test performed using our new module and comparisons against other solutions we have thought; Section V shows the device utilization of our system. Finally, Section VI ends the paper with the conclusions and potential future works.

II. NetFPGA Platform

The NetFPGA is an accelerated network hardware that augments the function of a standard computer. It consists of three parts: hardware, gateware, and software. The development board itself is a PCI card that can be installed in any PC with an available full-length slot. In more detail, the hardware of the board has the following core components:

- Xilinx Virtex-II Pro 50
- 4x1 Gbps Ethernet ports using a soft MAC core (recently, the new version of the NetFPGA has 4x10 Gbps Ethernet ports)
- Two parallel banks of 18 MBit Zero-bus turnaround (ZBT) SRAM
- 64 MBytes DDR DRAM

The FPGA directly handles all data-path switching, routing, and processing of Ethernet and Internet packets, leaving software to handle only control-path functions [14]. Hosted on the board are a user-programmable FPGA (with two PowerPC processors), SRAM, DRAM, and four 1Gbps Ethernet ports (10Gbps Ethernet ports in the newer version). Software and gateware (Verilog HDL source code) are available for download under an open source license from the NetFPGA website [11]. This allows jump starting prototypes and quickly building on existing designs such as an IPV4 router or a NIC. The gateware is designed to be modular and easily extensible. Designs are implemented as modular stages connected

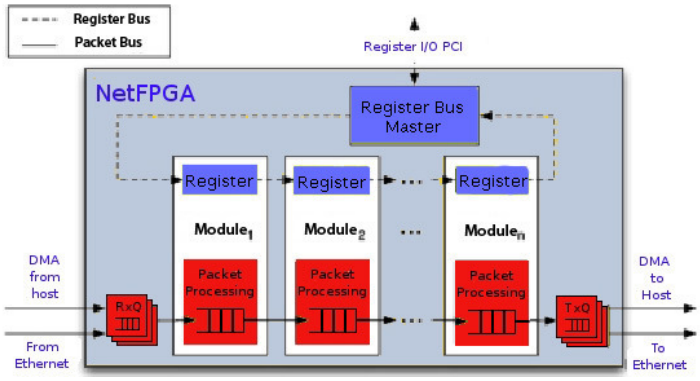

Fig. 1. Modular NetFPGA pipeline structure. The high-bandwidth packet bus (in red) is used for packet processing while the register bus (in blue) is used to carry control and status information between software modules and the hardware.

together in a pipeline, allowing the addition of new stages with relatively small effort [12]. The pipeline is depicted in Figure 1.

It is possible to develop on top of the platform Ethernet switches, Internet Prototcol (IP) routers using hardware rather than software, precise network measurement systems, and hardware-accelerated network processing systems. The platform can be used by researchers to prototype advanced services for next-generation networks.

Accent Technology [15] offers pre-assembled NetFPGA computer systems as approved by Stanford University. These pre-built and completely tested Linux-based computers are available in a compact desktop cube or standard 1U rack-mountable server configuration. In the researcher laboratories, the NetFPGA is usually installed inside a desktop PC so researchers can access the hardware [16], [17]. Several projects have already been developed in the NetFPGA (see the NetFPGA project page [18]).

III. System Architecture

Our system consists of two main components: hardware and software. The hardware component is an extended NetFPGA IPv4 reference router that, for each packet satisfying user defined constraints, changes its fields accordingly. The software component is represented by a graphical user interface which writes the constraints defined by the users into new NetFPGA registers that we have created.

A. Hardware - Changes to the reference router project

Our design modifies the User Data Path module of the reference router. Fig. 2 shows the reference router layout with the addition of the new *flex_router_preprocess* and *flex_router* modules. The *user_data_path.v* file has been altered to include the definitions of *flex_router_preprocess* and *flex_router* modules and their wire connections to the *output_port_lookup* and *output_queues* submodules. The new modules have therefore been inserted into the reference router pipeline. The

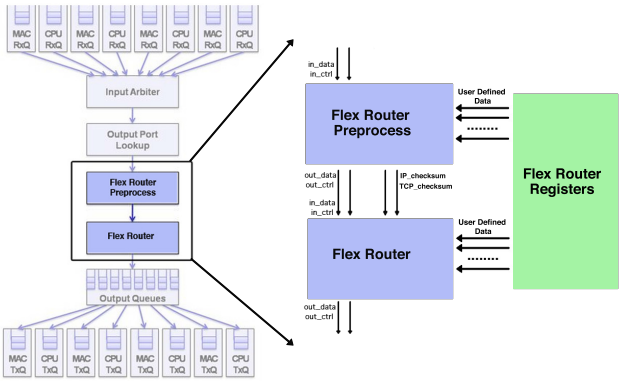

Fig. 2. Submodule layout of the modified User Data Path. The *flex_router_preprocess* and *flex_router* are new submodules and the *user_data_path* has been altered accordingly.

flex_router_preprocess takes as input data provided by the *output_port_lookup* module; hence it evaluates the users data provided using new defined registers and computes the new IP and TCP checksums for each incoming packet which satisfies these rules. Once computed, it enables the next module, the *flex_router*, allowing it to receive the new computed values. The *flex_router* module, hence, writes into the appropriate packet fields, the data specified by the users and the new computed checksums where needed.

The reason why we have used two Verilog modules and not just one is related to the word alignment for reference router TCP packets depicted in Fig. 3. Since the destination ip address is split into two different words, we need to read the following word in order to extract the complete value needed to check the user defined rules and for IP checksum computation purposes. Moreover, the two blocks structure works well even if we want to extend the flex router functionality to take into account any other field located into the payload data; in fact, as the TCP checksum lies before the payload data, we can not write it before its computation.

For such reasons, using the *flex_router_preprocess* module we are able to just read the TCP packet fields and compute the new checksum according to the fields specified by the user. The *flex_router_preprocess* module will transfer the old fields values and the new checksums to the *flex_router* module. Once the input data are ready, the *flex_router* module will compare the old field values (passed by *flex_router_preprocess* module) with those defined by the user (read by NetFPGA registers). If they match, the *flex_router* module will write into the relative

	in_data			
Words	63:48	47:32	31:16	15:0
1	eth_da			eth_sa
2	eth_sa		type	ver, IHL, TOS
3	total_length	identification	flags, frag_off	TTL, proto
4	checksum	src_ip		dst_ip
5	dst_ip	src_port	dst_port	sequence
6	sequence	ack		data_off, flags
7	win_size	checksum	urg_pointer	option
8	option			
9	payload			
.....				

Fig. 3. NetFPGA word alignment for TCP packets. Fields shaded in blue are inspected and modified according to the users rules. Fields shaded in red are the IP and TCP checksums which the module *flex_router* will update.

words (the forth, the fifth and seventh, respectively) the fields specified by the user, and the IP and TCP checksums.

Both the *flex_router_preprocess* and *flex_router* modules evolve according to the state machine shown in Fig. 4. The *flex_router* module waits for the *rd_preprocess* signal coming from the *flex_router_preprocess* module. This signal will be sent when the checksum computation ends. The *flex_router_preprocess* module uses two FIFOS to store data: the first one is used to store words already processed which have to be passed to the *flex_router* module. Words will not be passed to the *flex_router* module until the *rd_preprocess* signal is sent. The second FIFO is used to store the IP and TCP checksum of the *flex_router_preprocess* module, which are later used by *flex_router* module.

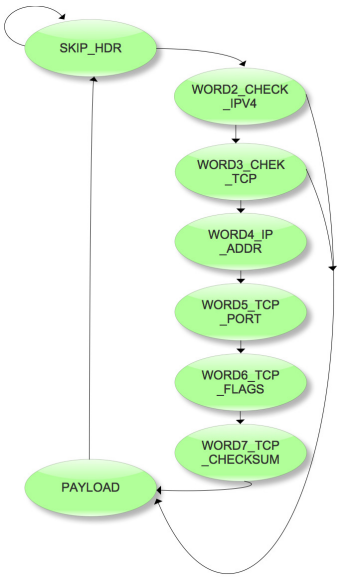

Fig. 4. Diagram of State Machine Used by the *flex_router_preprocess* and *flex_router* modules.

The state machine for *flex_router_preprocess* initially idles in the *SKIP_HDR* state waiting for the IP version field, skipping all the other words containing not relevant data for our purposes. Once the word containing the IP version arrives, the state machine moves to *WORD2_CHECK_IPV4* where

the IP version is checked to be IPv4. If it does not, then the machine moves straight to the *PAYLOAD* state where all the remaining packet words are skipped until the packet ends. If the IP version is IPv4 then the next visited state is *WORD3_CHECK_TCP* where the information about the next level protocol are read. If the protocol is not TCP then the state machine moves again into the *PAYLOAD* state; else, the next reached state is *WORD4_IP_ADDR* where the information about the IP checksum, the IP source address and part of the IP destination address are taken into account for the new IP checksum computation and stored in order to be sent later to the *flex_router* module. The next state is then *WORD5_TCP_PORT* where the remainder of the IP destination address, the source port and the destination port are fetched and used as into the previous state; moreover, in this state the computation of the new IP checksum ends as we read the whole IP destination address. *WORD6_TCP_FLAGS* is the following state where the underlying word is skipped as it does not contain relevant fields neither for checksum computation nor for field replacement. Then, the state *WORD7_TCP_-CHECKSUM* is reached. Here, we read the values of the advertised window size and the old TCP checksum field; in this state the computation of the new TCP checksum ends as well. Finally, the state machine moves to the *PAYLOAD* state, waits for the end of the current packet, and starts again with the *SKIP_HDR* state.

The state machine for *flex_router* module is similar to the one just described above for the *flex_router_preprocess*. During the state *WORD4_IP_ADDR* the new IP checksum (which is an output of the previous module) is written together with the new source IP address and part of the new destination IP address if requested by the user. During the state *WORD5_TCP_PORT*, the remaining part of the new IP destination address, the new source port and the new destination port are written as well if necessary. Finally, in state *WORD7_TCP_CHECKSUM*, the new advertised window size and the new TCP checksum are written. The reader notices that the IP and TCP checksums are written only if any of the old values have changed.

Let us consider that the Internet checksum calculation is very demanding as far as the computational resources are concerned (as specified in [19]). Whereas the IP checksum computation would affect four words (for a total of 20 bytes), the TCP checksum affects the whole packet which can be much bigger: this results in heavier computation for the design. As the operations performed are directly executed in hardware we need to decrease the computational load as much as possible: doing like that we would reduce the design complexity of the project. Luckily, the Internet checksum incremental formula defined into [20] comes to our needs. As reported in [20] the formula is defined as:

$$H' = \sim (\sim H \bigoplus \sim sum_{old} \bigoplus sum_{new})$$

where H is the old checksum, H' is the new checksum, sum_{old} and sum_{new} are the ones' complement sum computed over source IP address, destination IP address, source port, destination port, window size of the old and new values respectively (the only fields that change their value). $\sim$ indicates a bit-wise complement operation, and $\bigoplus$ indicates a ones' complement sum operation. The reader notices that, if the sum within parenthesis has carry, we need to perform a further summation of the carry before applying the most external ones' complement. As implicitly reported into the state machine description of the *flex_router_preprocess* module, we use the Internet checksum incremental formula as we do not read the entire packet but only the necessary fields.

The NetFPGA platform works like every digital electronic devices and, therefore, there is a clock signal which is the basis of each operation. As far as the reference router project is concerned, during each clock cycle, a word of a given incoming packet is processed. The reader notices that each state of the machine defined above lasts one clock cycle where fields of the current word are fetched and potential operations on them are executed. Given that the NetFPGA has a core clock that runs at 125 MHz, each clock cycle lasts therefore about 8 ns. Consequently, all the operations executed within one clock cycle have to meet this time constraint. In order to reduce the number of operations performed within each clock cycle, we have distributed those on the machine states having a lower computational load.

B. User Defined Rules

The user is allowed to decide which packet to modify by using a set of rules through a Graphical User Interface (GUI) developed in JAVA. In Fig. 5 a screenshot of the JAVA application is displayed. The column on the left represents the requirements that a packet has to satisfy; if it does, then, the values on the second column are written in the relative packet fields. As shown in Fig. 5, packets having as IP source 192.168.1.2, as IP destination 192.168.2.2 and as TCP source port 5554 will satisfy the user defined rule: the packets satisfying the rules will be hijacked: their IP source and the advertised window size fields will be overwritten, respectively, with the new values 192.168.3.2 and 52.

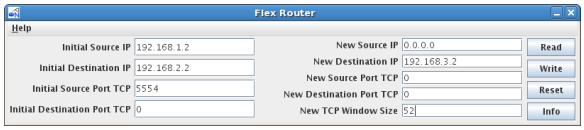

Fig. 5. Graphical User Interface.

When the user specifies the values, once he clicks on the write button, the new values will be passed from the host computer to the NetFPGA board which will write them into registers introduced in our design as remarked in [21]. If any of the parameters are not specified, then the relative field is not taken into account. The *flex_router_preprocess* module reads the registers in order to understand whether the relative field values have to be considered for the Internet checksum computations. The *flex_router* module uses the registers to check if the current packet satisfies the user defined rules. Packets which satisfy the rules on the first column will be

updated with the new values expressed in the second column of the GUI. Moreover, the TCP and IP checksums will be updated accordingly. More specifically, the GUI reads and writes the NetFPGA registers using the system calls *regread* and *regwrite* (they are two C programs which come together with the NetFPGA platform). The Verilog modules *flex_router_preprocess* and *flex_router* use the *generic_regs.v* module defined into the Verilog *utils* library in order to access them. In particular, the the registers we have used are referenced as *NetFPGA software registers* and allowed us to read their contents from hardware and write new values from software (the GUI). Conversely, *NetFPGA hardware registers* allow us to write new values from hardware and read their contents from software.

IV. Experimentation

We first verified our implementation in the simulation platform by using the Perl testing library provided by the NetFPGA. The library allowed us to create packets with arbitrary values into the IP and TCP headers. We have used Wireshark [22] to test our project for both simulation and verification. As far as the simulations are concerned, we have used it in order to capture real TCP packets and export them into simulation scripts. This allowed us to have a more realistic scenario for testing. For verification, Wireshark was useful to check that the new IP and TCP checksums together with the new header fields were correctly written into incoming packets satisfying the user defined rules. We have also used and run the regression tests of the reference router to ensure that our modifications did not break the standard functionality of the reference router. The availability of these tests greatly reduced the time required to test our design.

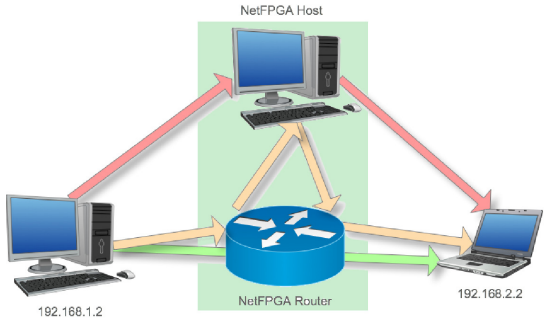

Fig. 6. Topology used for our experiments. Red line indicates the path of packets for the Click Modular Router. The orange line shows the path of packets for the software version of the flexible router; the green line represents the path of packets for the flex router project.

Having verified the correctness of our implementation, we ran different experiments in order to test our system with and without the NetFPGA platform. We have compared our system with the original reference router project. The same features have also been developed using the original reference router and writing a C program in order to capture network packets, change the TCP and IP fields, and compute the new checksums. We have called such a system the software version of the flexible router. Finally, we have installed the Click Modular Router into the same computer hosting the NetFPGA board and developed the same features into the Click. Fig. 6 shows the topology we have used for our experiments. PC_1 having IP address 192.168.1.2 is connected to the NetFPGA port $nf2c1$ whereas PC_2 with IP address 192.168.2.2 is connected to the NetFPGA port $nf2c2$.

As described above, the first two experiments have been performed using the NetFPGA platform but with different hardware designs. The first one is represented by our flex router project (the reference router augmented with *flex_router_preprocess* and *flex_router* modules). The other is represented by the reference router modified in order to forward each incoming packet to the CPU port and allow, using a software written in C, to hijack packets as the flex router project does. Finally, we have developed a new module using the Click Modular Router [8] which performs the same operations of the flex router project but via software. The results of the three experiments have been compared to the ones of the original reference router design. Fig. 6 shows three different paths that incoming packets follow for each system just described. As user may notice, for the Click Modular router, the packets following the red path are forwarded trough the software router (which resides into the host computer) to the destination without affecting the NetFPGA board. Packets into the orange path are first forwarded to the NetFPGA router which switch each of them to the host computer in order to be processed by our software implementation of the flex router. Then, they are forwarded back to the NetFPGA router which sends them to the final destination. Finally, the green line represents the path of incoming packets processed by our flex router project and they are forwarded through the NetFPGA router to the final destination. Each experiment shows the throughput of the output queue for a HTTP file transfer session of 60 seconds. Figures 7, 8, 9 show the throughput for each implemented system described above with respect the throughput of the original reference router project. The reader notices that the flex router has the highest throughput with respect to the other two systems as its throughput is the closest to the one of the reference router. It is obvious that the original reference router has higher performance than the flex router as this last has been obtained augmenting the original reference router project with other two modules which slightly increase the overall complexity.

V. Device Utilization

The device utilization of the hardware component of the flex router project is just a little bit higher than the one used by the reference router and it is displayed in Table I. Table II shows the device utilization of the reference router project.

VI. Conclusions and Future Work

An infinity of examples can be carried relating the need of managing TCP or IP packets on fly, like for example, TCP header fields modifications to improve TCP performance in particular environments, packet priority variations, service differentiations, smart firewall implementations, data encryption,

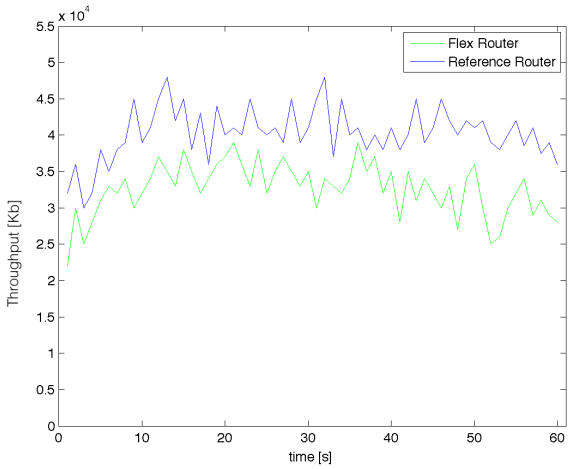

Fig. 7. Flexible Router throughput.

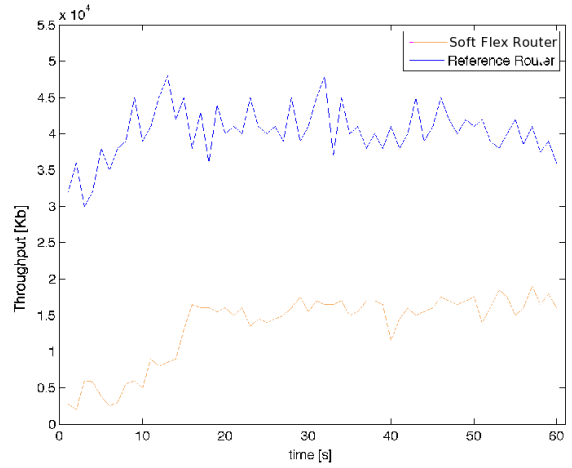

Fig. 8. Software Flexible Router throughput.

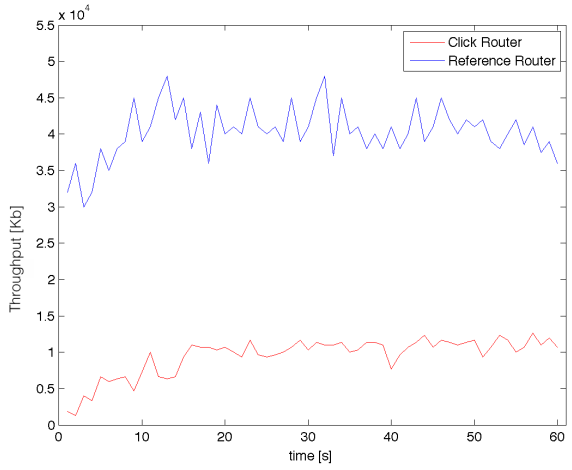

Fig. 9. Click Router throughput.

Resources	XC2VP50 Utilization	Utilization Percentage
Slices	11201 out of 23616	47%
4-input LUTS	16413 out of 47232	34%
Flip Flops	9349 out of 47232	19%
Number of BRAMs	37 out of 232	15%
Number of bonded IOBs	360 out of 692	52%
Number of GCLKs	8 out of 16	50%
Number of DCMs	6 out of 8	75%

TABLE I
DEVICE UTILIZATION FOR THE *flex_router_preprocess* AND *flex_router* MODULES.

and so on. Of course, all the applications cited above cannot be realized and customized in not open network devices.

With all this in mind, the target of this paper has been to realize an open flexible and high-performance router that allows the implementation of any strategy for on-fly packet modification. To this purpose the router has been implemented over the NetFPGA platform, extending the NetFPGA reference router project with capabilities never shown so far, like for example runtime computing TCP and IP header checksums in hardware.

Our system has been implemented on top of the NetFPGA platform. In particular, it augments the reference router project with *flex_router_preprocess* and *flex_router* modules. It can modify different TCP and IP packet fields according to rules defined by the user through a graphical user interface developed in JAVA. It also computes the IP and TCP Internet checksum in order to validate the packets with the new values. To the best of our knowledge, there is no much in literature about working with the reference router project.

Our code has been released, following the guidelines in [24], to the larger community for re-use, feedback, and enhancement. It can be downloaded from [13]. In the future, taking as starting points the works in [25] and [26], we will analyze the energy consumption of each module of the reference router included the ones we have proposed in this paper.

Resources	XC2VP50 Utilization	Utilization Percentage
Slices	9699 out of 23616	41%
4-input LUTS	14332 out of 47232	30%
Flip Flops	8221 out of 47232	17%
Number of BRAMs	27 out of 232	11%
Number of bonded IOBs	360 out of 692	52%
Number of GCLKs	8 out of 16	50%
Number of DCMs	6 out of 8	75%

TABLE II
DEVICE UTILIZATION FOR THE ORIGINAL REFERENCE ROUTER PROJECT.

REFERENCES

[1] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson, "Optimizing tcp forwarder performance," *IEEE/ACM TRANSACTIONS ON NETWORKING*, vol. 8, no. 2, 2000.

[2] P. Papadimitriou and V. Tsaoussidis, "Abstract on tcp performance over asymmetric satellite links with real-time constraints," 2007.

[3] C. Metz, "Tcp over satellite... the final frontier," *Journal IEEE Internet Computing*, vol. 3, no. 1, 1999.

[4] I. F. Akyildiz, G. Morabito, and S. Palazzo, "Tcp-peach: A new congestion control scheme for satellite ip networks," *IEEE/ACM Transactions on Networking*, vol. 9, pp. 307–321, 2001.

[5] A. Bakre and B. R. Badrinath, "I-tcp: Indirect tcp for mobile hosts," *Proc. 15th Int. Conf. Distributed Computing Syst. (ICDCS)*, 1995.

[6] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving tcp performance over wireless links," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, 1997.

[7] M. Barbera, A. Lombardo, C. Panarello, and G. Schembra, "Queue stability analysis and performance evaluation of a tcp-compliant window management mechanism," *IEEE/ACM Transactions on Networking*, vol. 18, no. 4, pp. 1275–1288, 2010.

[8] K. Eddie, M. Robert, C. Benjie, J. John, and K. M. Frans, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[9] G. Calarco, C. Raffaelli, G. Schembra, and G. Tusa, "Comparative analysis of smp click scheduling techniques," *Proc. QoS-IP*, pp. 379–389, 2005.

[10] "Zebra, http://www.zebra.org."

[11] "Netfpga team. netfpga website. http://netfpga.org."

[12] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "Netfpga: reusable router architecture for experimental research," in *PRESTO 08: Proceedings of the ACM workshop on Programmable routers for extensive services of tomorrow, pages 17, New York, NY, USA, 2008. ACM*.

[13] "http://netfpga.org/foswiki/bin/view/netfpga/onegig/ flexrouter."

[14] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "Netfpga - an open platform for gigabit-rate network switching and routing," in *International Conference on Microelectronic Systems Education*, 2007.

[15] "http://www.accenttechnologyinc.com/."

[16] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "Netfpga: An open platform for teaching how to build gigabit-rate network switches and routers," in *IEEE Transactions on Education, August*, 2008.

[17] G. Watson, N. McKeown, and M. Casado, "Netfpga - a took for network research and education." in *2nd Workshop on Architecture Research using FPGA Platforms (WARFP), February*, 2006.

[18] "Netfpga project table. http://netfpga.org/foswiki/bin/view/netfpga/onegig/projecttable."

[19] "Computing the internet checksum, http://tools.ietf.org/html/rfc1071."

[20] "Computation of the internet checksum via incremental update, http://www.apps.ietf.org/rfc/rfc1624.html."

[21] "http://www.netfpga.org/foswiki/bin/views/netfpga/onegig/registersystem."

[22] "Wireshark. http://www.wireshark.org."

[23] M. Ciesla and V. Sivaraman, "Url extraction n the netfpga reference router," *NetFPGA Developers Workshop*, 2009.

[24] G. A. Covington, G. Gibb, J. Naous, J. Lockwood, , and N. McKeown, "Methodology to contribute netfpga modules," in *International Conference on Microelectronics Systems Education*, 2009.

[25] V. Sivaraman, A. Vishwanath, Z. Zhao, and C. Russel, "Profiling per-packet and per-byte power consumption in the netfpga gigabit router," *IEEE INFOCOM Workshop on Green Communications and Networking (GCN)*, 2011.

[26] A. Vishwanath, Z. Zhao, V. Sivaraman, and C. Russell, "An empirical model of power consumption in the netfpga gigabit router," *IEEE Advanced Networks and Telecommunication Systems (ANTS)*, 2010.