Boosting the Performance of PC-based Software Routers with FPGA-enhanced Network Interface Cards

Boosting the Performance of PC-based Software Routers with FPGA-enhanced Network Interface Cards

Andrea Bianco*, Robert Birke*, Gianluca Botto*, Marcello Chiaberge*, Jorge M. Finochietto*, Giulio Galante†, Marco Mellia*, Fabio Neri*, Michele Petracca*

* Dipartimento di Elettronica, Politecnico di Torino, 10129 Torino, Italy, Email: {firstname.lastname}@polito.it

† Networking Lab, Istituto Superiore Mario Boella, 10138 Torino, Italy, Email: galante@ismb.it

***Abstract*— The research community is devoting increasing attention to software routers based on off-the-shelf hardware and open-source operating systems running on the personal-computer (PC) architecture. Today's high-end PCs are equipped with peripheral component interconnect (PCI) shared buses enabling them to easily fit into the multi-gigabit-per-second routing segment, for a price much lower than that of commercial routers. However, commercially-available PC network interface cards (NICs) lack programmability, and require not only packets to cross the PCI bus twice, but also to be processed in software by the operating system, strongly reducing the achievable forwarding rate. It is therefore interesting to explore the performance of customizable NICs based on field-programmable gate array (FPGA) logic devices we developed and assess how well they can overcome the limitations of today's commercially-available NICs.**

I. Introduction

Software routers based on off-the-shelf personal-computer (PC) hardware and open-source software are becoming appealing alternatives to proprietary network devices because of the wide availability of multi-vendor hardware, the low cost and the continuous evolution driven by the PC-market economy of scale. Indeed, the PC world benefits from the de-facto standards defined for hardware components, which enable the development of an open multi-vendor market, and the large availability of open-source software for networking applications, such as Linux [1], Click [2] and the BSD derivatives [3] for the data plane, as well as Xorp [4] and Zebra/Quagga [5] for the control plane.

Several criticisms can be raised against software routers, e.g., software limitation, lack of hardware support, scalability problems, lack of advanced functionalities; even though, performance limitations are compensated by the natural PC-architecture evolution. Current PC-based routers and switches have a traffic-switching capability in the range of a few gigabits per second, which is more than enough for a large number of applications. However, when looking for high-end performance, commercially-available network interface cards (NICs) are affected by many limitations. In [6], where commercial NICs were used to build a router running both the standard Linux and the Click Internet Protocol (IP) stack, we showed that it is not possible to route a single 1-Gbit/s traffic flow consisting of only minimum-size Ethernet frames, even if the PCI bus bandwidth is 8 Gbit/s. The main limitations stem from central-processing unit (CPU) overloading and from large host-memory-read latency, which represents up to 90% of the minimum-size-packet transfer time.

One possible solution is to build custom NICs implementing the well-known direct NIC-to-NIC packet transfer technique, which is not possible with today's commercial NICs, because they lack programmability. This approach has several advantages: *(i)* the read-latency is reduced, because packets must not be transfered any longer to the host memory; *(ii)* the PCI bus is used more efficiently, since packets traverse it only once; *(iii)* CPU resources are freed up; *(iv)* more-sophisticated quality-of-service (QoS)-oriented classification and scheduling algorithms can substitute the classical first-in first-out (FIFO) service discipline available on commercial NICs. However, to implement direct board communication, the NICs must be able to autonomously route IP packets and a scheduling algorithm as well as a NIC-to-NIC communication protocol must be defined and implemented.

In this paper we describe the implementation of a custom NIC on a PCI Extended (PCI-X) development board equipped with a field-programmable gate array (FPGA) logic device and a Gigabit-Ethernet transceiver. We focus only on data plane performance, ignoring all the issues related to management functions and to the control plane. We aim at assessing the packet-forwarding rate of high-end PCs equipped with custom FPGA-enhanced Gigabit-Ethernet NICs, under the Linux operating system.

Note that the availability of powerful programmable logic devices permits to extend the open-software paradigm into the hardware domain. The logic circuitry developed for the FPGAs could be made public, reused, and improved by the research community. This *open-hardware* approach would enable the low-cost implementation of performance-critical functional blocks in hardware. See, for instance, [7].

The paper is organized as follows. Section II gives a quick introduction to the PC architecture, describes the operations and the bandwidth limitations of its key components, and explains how a PC can be used as an IP router. Section III describes the main features of the FPGA-enhanced NIC.

0-7803-9569-7/06/$20.00

Section IV introduces the experimental setup, describes the tests performed, and comments on the results obtained. Finally, Section V concludes the paper.

II. Software Router Architecture

A PC comprises three main building blocks: the central processing unit (CPU), random access memory (RAM), and peripherals, glued together by the *chipset*, which provides complex interconnection and control functions.

As sketched in Fig. 1, the CPU communicates with the chipset through the front-side bus (FSB). The RAM provides temporary data storage for the CPU, and can be accessed by the memory controller integrated on the chipset through the memory bus (MB). The NICs are connected to the chipset by the PCI shared bus.

Today's state-of-the-art CPUs run at frequencies up to 3.8 GHz. High-end PCs are equipped with chipsets supporting multiple CPUs connected in a symmetric multiprocessing (SMP) architecture. Typical configurations comprise 2, 4, 8 or even 16 identical CPUs.

The front-side bus is 64-bit wide and is driven by a 100- to 266-MHz *quad-pumped* clock, allowing for a peak transfer rate ranging from 3.2 Gbyte/s to 8.4 Gbyte/s.

The memory bus is usually 64-bit wide and runs at 100, 133, 166, or 200 MHz with *double-pumped* transfers, providing a peak transfer rate of 1.6, 2.1, 2.7, or 3.2 Gbyte/s. The corresponding double-data-rate (DDR) synchronous dynamic RAM (SDRAM) chips are soldered on dual-in-line memory modules (DIMM) marketed with the names *PC1600, PC2100, PC2700*, and *PC3200* respectively. In high-end PCs, the memory bandwidth is further doubled, bringing the bus width to 128 bits, by installing memory banks in pairs. Note that this allows to match the memory-bus peak bandwidth to that of the front-side bus.

The PCI protocol is designed to efficiently transfer the contents of large blocks of contiguous memory locations between the peripherals and the RAM, without requiring any CPU intervention. As the bus is shared, no more than one device can act as a *bus-master* at any given time; therefore, an *arbiter* is included in the chipset to regulate the access and fairly share the bandwidth among the peripherals. Depending on the PCI protocol version implemented on the chipset and the number of electrical paths connecting the components, the bandwidth available on the bus ranges from about 125 Mbyte/s for PCI 1.0, which operates at 33 MHz with 32-bit parallelism,

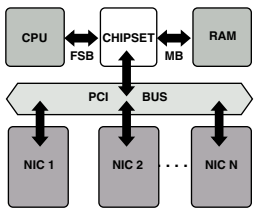

Fig. 1. Key components in a PC-based software router

to 2 Gbyte/s for PCI-X 266, when transferring 64 bits on a double-pumped 133-MHz clock.

Typically, Ethernet NICs operate as bus-masters to offload the CPU from performing bulk data transfers between their internal memory and the RAM. Incoming packets are stored directly on ring buffers available in RAM. Each NIC is connected to one interrupt-request (IRQ) line, that is used to notify the CPU of events that need service from the operating system. On the other hand, it is usually possible to switch IRQ generation off altogether, leaving to the operating system the burden of periodically *polling* the NIC hardware and reacting accordingly.

Summarizing, common PC hardware enables to easily implement a shared-bus, shared-memory router, where NICs receive and transfer packets directly to the RAM, the CPU routes them to the correct output ring buffer in RAM, and NICs fetch packets from the RAM and transmit them on the wire. In such configuration, each packet travels twice through the PCI and the memory bus, effectively halving the bandwidth available for routing traffic. Therefore, a high-end PC equipped with a 1 Gbyte/s 64-bit-wide PCI-X bus running at 133 MHz should in principle be able to deal with 3-4 Gigabit-Ethernet NICs.

III. FPGA-enhanced-NIC Operation

The PLDA [8] development board used for the custom NIC includes an optical Gigabit-Ethernet physical layer (PHY) chip, an Altera Stratix GX FPGA [9], and can be plugged in a PCI-X bus connector.

A. FPGA Architecture

As depicted in Fig. 2, the FPGA comprises three functional macro-blocks implementing *(i)* the Ethernet medium access control (MAC) protocol interfacing to the Ethernet PHY on the PLDA board, *(ii)* IP packet processing, and *(iii)* the PCI protocol.

The Ethernet-MAC and the PCI interface are implemented in two intellectual-property cores available from the market. Instead, we designed the blocks sitting in between that implement an IP-routing module, a packet classifier, and a strict-priority distributed scheduler.

Input Processing: The Ethernet frames received from the MAC core are parsed to extract the Ethernet protocol type and the IP header before undergoing IP routing — to determine their next-hop destination — as well as classification and scheduling — to enforce QoS.

Routing: The routing module is extremely simplified, does not implement a full-blown longest-prefix-matching algorithm, and can store only a limited number of IP routes (in this implementation, one toward each of the other NICs plugged in the router). The routing-process outcome is essentially binary. If the packet at hand matches one of the FPGA-stored routes, it is processed by the IP module, classified, and enqueued to be directly forwarded to the selected destination NIC along a *fast path*, avoiding any CPU intervention. Otherwise, it is left untouched, classified, enqueued to be transferred to the RAM

along a *slow path*, routed by the Linux IP stack, and finally handed to the intended destination NIC for delivery.
IP routing is implemented comparing the packet IP destination with the 32-bit ternary masks describing the IP subnetworks reachable through each of the NICs plugged in the router, and selecting the one that matches. Obviously, this only works if all subnets are disjoint. Moreover, since it would be difficult to implement the address resolution protocol (ARP) in hardware on the FPGA, each routing entry contains, besides a pointer to the destination NIC, the next-hop router's Ethernet-MAC address, rather than its IP address.
Finally, the routing module includes a simple header-rewriting function for the packets undergoing fast-path forwarding that updates the Ethernet-frame MAC destination address, decrements the IP time-to-live (TTL) header field, dropping the packets for which it reaches zero, and recomputes the IP-header checksum.

Classification: Each packet is classified comparing an 80-bit string extracted from its IP header comprising the type of service (ToS), the source address, the destination address, and the transport protocol with a per-destination 80-bit ternary mask stored on the FPGA. The packet is assigned either to the high-priority class, in case of match, or to the low-priority class, otherwise, and is stored in the per-class per-destination *input* FIFO architecture feeding the QoS scheduler.

Slow-path: The FPGA-enhanced-NIC operation in slow-path routing mode is similar to commercial-NIC operation under NAPI [10], basically a polling scheme. The only significant difference is that the FPGA-enhanced NIC provides the driver with two different reception rings, one for each traffic class. This way, the driver, which in the current version implements a simple strict-priority software scheduler, can enforce QoS by handling the packets belonging to the two traffic classes differently.

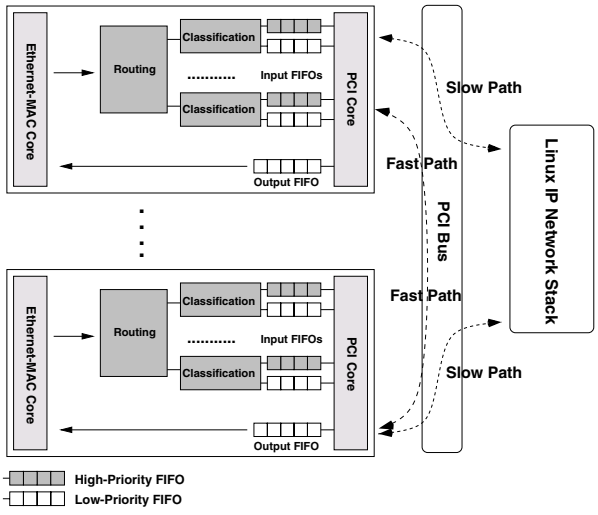

Fig. 2. Architecture of the FPGA-enhanced NICs and their interactions with the PCI bus and the Linux operating system

Fast Path: Packet transfers on the PCI-bus are regulated by a distributed (asynchronous) scheduling algorithm based upon in-band messages exchanged among the FPGA-enhanced NICs, described in Section III-B. The aim is to let each output NIC select the amount of traffic to be transferred from each input NIC, so as to prevent any packet drops at the output-NIC FIFO by confining them only to input FIFOs, and avoid any PCI-bus-bandwidth waste.

Output Processing: Packets received from the PCI bus are stored in two different *output* FIFOs (even though only one of them is shown in the Fig. 2) depending on whether they followed the fast or the slow path. A simple round-robin scheduler extracts packets from such FIFOs, giving strict priority to those coming from the slow-path FIFO, and transfers them to the Ethernet-MAC core.

B. Distributed Scheduling Algorithm

The scheduling algorithm is based on a two-step negotiation, involving the transmission of a *request* (REQ) message, waiting for the reception of the corresponding *response* (RES) message from the output NIC, so as to agree on the amount of traffic to be transmitted in the subsequent *burst* packet transfer. The burst is then immediately followed by a new REQ message to keep the algorithm going.

1) REQ Message Generation: An output NIC becomes *active* at a given input NIC when there is at least one packet arrival in any of the two associated input FIFOs when both of them are empty. Note that such NIC remains active until it is sent a REQ message, returning *inactive* immediately thereafter.

The input-NIC *request scheduler (i)* continuously monitors the input-FIFO occupancy, *(ii)* selects one out of the active output NICs, *(iii)* sends it a REQ message containing the amount of bytes stored in each of the two associated input FIFOs, *(iv)* waits for the corresponding RES message before going to *(i)*, and generating any other REQ.

Since in the general general case several output NICs may be active at the same time, a mechanism is needed to break ties and enforce fairness. The simplest solution is to scan the input FIFOs for active output NICs in a round-robin fashion, define a *request pointer* that keeps track of the last output NIC to which a REQ message has been sent, and resume the round-robin scan from it, after receiving the wanted RES message.

2) RES Message Generation: The output-NIC *response scheduler* operates similarly to the request scheduler by *(i)* continuously monitoring the REQ messages received, *(ii)* selecting one input NICs, *(iii)* sending it a RES message containing the number of bytes that can be transferred to the output FIFO from each of the two input FIFOs, *(iv)* waiting for the REQ message following the packet burst before going to *(i)* and generating any other RES.

Also here, since in the general general case several REQ messages may have been received from the different input NICs, ties are broken in a round-robin fashion, defining a *response pointer* that keeps track of the last input NIC to which a RES message has been sent.

The RES message contains two fields specifying the number of bytes the output FIFO can accept for each priority. Their values are assigned according to a strict-priority criterion, privileging first high-priority traffic, leaving what is left, if any, to low-priority traffic.

3) Burst Generation: The input-NIC *burst scheduler* operates similarly to the request scheduler by *(i)* continuously monitoring the RES messages received, *(ii)* selecting one output NICs, *(iii)* sending it a packet burst, *(iv)* generating a new REQ message before going to *(i)* and restarting the loop.

Again, since in the general general case several RES messages may have been received from the different output NICs, ties are broken in a round-robin fashion, defining a *burst pointer* that keeps track of the last output NIC to which a packet burst has been sent.

The scheduler generates and sends a packet burst that satisfies the output-FIFO size constraints specified in the selected RES message. The burst header contains the number of high- and low-priority packets it conveys, if any. High-priority packets are sent first, followed by low-priority ones; each packet being preceded by a header indicating its length. Finally, a trailing REQ is appended to the end of the burst to notify the destination NIC of the current FIFO state. If both FIFOs become empty, a null REQ is sent, preventing any further protocol exchange. If both the head-of-line packets in the selected input FIFOs are larger than the space available in the output FIFO indicated by the RES message, an empty burst message consisting only of the header and a trailing REQ message is generated and sent anyway, so as to keep the protocol exchange active.

C. Implementation Details

Driver Interface: At boot time, the NIC register file is mapped in the PC memory address space, so that the driver can easily access configuration data. Some registers are reserved to store the ternary routing and packet classification masks. In the current implementation, such masks are created by hand for each test configuration and uploaded to the board with command line scripts.

DMA-channel Assignment: The board performs all data transfers to and from the PC memory as a PCI bus master, using the four-channel direct-memory-access (DMA) engine provided by the PCI-X core. Each channel can be completely programmed by the FPGA application, specifying the transfer direction, the transfer size, and the transfer starting address. The scheduling of the four channels is proprietary and unknown, since the core is encrypted. In our prototype, the four channels are used as follows: *(i)* to download outgoing packets from the RAM to the output FIFO (slow path output), *(ii)* to upload incoming packets from the input FIFO to the RAM (slow path input), *(iii)* to send REQ and RES messages (distributed scheduling algorithm), and *(iv)* to send packet bursts (fast path).

FPGA Occupation: In the current implementation, each programmable NIC supports two service classes and fast-path routing toward up to three FPGA-enhanced NICs, requiring about 33% out of the 41 250 FPGA logic elements. The eight input FIFOs (three pairs for fast-path forwarding plus one pair for slow-path forwarding) and the two output FIFOs, fitting up to 15 kbyte each (roughly corresponding to 10 maximum-size Ethernet frames), are instead implemented on the 3-Mbit FPGA DRAM, which can also temporarily buffer debug information.

Clock Speed: The circuit complexity and size affect the signal propagation delay and skew, so that sequential-logic set-up times constrain the maximum FPGA-clock frequency for the PCI core and the IP-handling macro-block to 66 MHz, even though the PCI core could also run at 133 MHz. Nevertheless, the 64-bit parallelism (theoretically) permits to reach a 4-Gbit/s peak transfer rate. The Ethernet-MAC core is instead clocked at 125 MHz, to be able to sustain a 1-Gbit/s throughput on its 8-bit input-output interfaces. Finally, a multiplexing-demultiplexing block is need to interconnect the 64-bit PCI core and the 64-bit IP-handling block running at 66 MHz with the 8-bit Ethernet MAC running at 125 MHz.

IV. Performance Evaluation

The aim of this section is twofold. First, to assess the performance of the FPGA-enhanced NICs in terms of forwarding rate and of flexibility in dealing with different-priority flows. Second, to evaluate the benefits of FPGA-enhanced NICs with respect to standard commercial NICs. Section IV-A introduces the testbed setup, whereas Section IV-B and Section IV-C describe the routing and QoS experiments performed, and comment on the results.

A. Testbed Setup

The FPGA-enhanced NICs were plugged in a high-end PC based on the SuperMicro X5DPE-G2 mainboard, equipped with one 2.8-GHz Intel Xeon processor and 1 Gbyte of PC1600 DDR RAM. The RAM installed is split into two interleaved banks, so as to bring the maximum memory-bus transfer rate to 3.2 Gbyte/s. The motherboard features three 133-MHz and three 100-MHz PCI-X slots, supporting a peak transfer rate of 1 Gbyte/s and 0.8 Gbyte/s respectively.

An Agilent N2X RouterTester 900 [11], equipped with 8 Gigabit-Ethernet ports, which can transmit and receive Ethernet frames of any size at full rate, was used for sourcing and sinking traffic in the tests.

All experiments were run on NAPI-enabled Linux kernels, using 64-byte minimum-size Ethernet frames, because they produce the highest CPU-usage and the largest bus-transfer overhead.

We considered both *unidirectional* flows, where each router port either transmits or receives packets for all the experiment duration, as well as *bidirectional* flows, where all router ports send and receive packets at the same time. All the tests lasted 30 s and no routing-configuration changes were uploaded to the board during any experiment run.

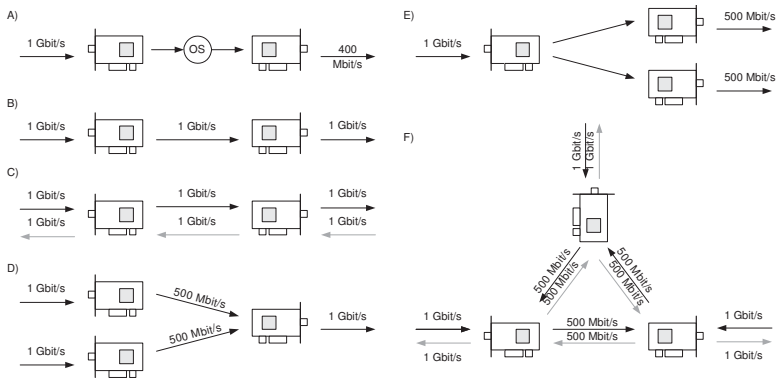

Fig. 3. Test configurations

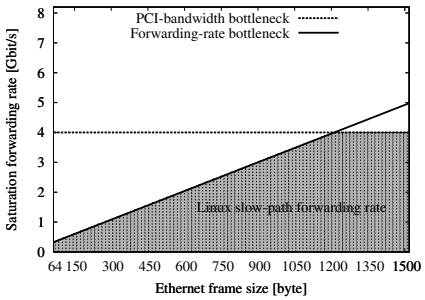

Fig. 4. Slow-path saturation forwarding rate for a Linux router using commercial NICs vs. Ethernet-frame size

B. Routing Tests

Several tests were run to verify the FPGA implementation's correctness and assess the FPGA-enhanced NICs' performance. Fig. 3 shows the scenarios explored.

The aim of the first experiment is to test the FPGA-enhanced NIC implementation's correctness in configuration *A*, by running it as a bare-bones commercial NIC to forward packets through the operating-system IP stack, via the standard NAPI interface. The forwarding throughput (not reported) for minimum size packets of the FPGA-enhanced NIC closely matches the performance of the commercial NIC. Note that the achieved saturation throughput strongly depends on the operating-system configuration; when running a standard Linux system straight out-of-the-box, the saturation throughput is roughly equal to 400 Mbit/s.

It may be surprising to notice that the forwarding rate is limited to 400 Mbit/s, even for the custom NIC. However, as shown in [6], this stems from both CPU and memory-read-latency limitations, not from NIC-operation flaws. Nevertheless, the FPGA-enhanced-NIC forwarding performance can be greatly improved by fast-path packet transfers regulated by the direct board-communication protocol. This way, the FPGA-enhanced NICs can route without any losses one unidirectional 1-Gbit/s flow in configuration *B*, or two bidirectional 1-Gbit/s flows in configuration *C*, for a net throughput of 2 Gbit/s.

Indeed, the performance of PC-based routers relying only on commercial NICs is mainly limited by two factors, as shown in Fig. 4, which depicts the theoretical saturation forwarding rate for a PC with an 8-Gbit/s PCI bus versus the Ethernet-frame size. First, for small-size packets, the *packet-rate bottleneck*, stemming from the 600-kpkt/s maximum packet rate the architecture can forward because of CPU availability and memory-read-latency constraints. Second, for large-size packets, the PCI-bus maximum bandwidth. The 4-Gbit/s figure is due to the slow-path routing architecture, which forces packets to go twice through the 8-Gbit/s PCI bus.

A third card was added to test the round-robin scheduling in configuration *D*, where each of two input NICs sends one unidirectional 1-Gbit/s flow to the same output NIC. Note that the output NIC cannot transmit the 2-Gbit/s flow resulting from the merge. In a conventional slow-path scenario, the output NIC would end up receiving the 2-Gbit/s aggregate flow and drop from each flow the half of the packets that cannot fit the 1-Gbit/s output link, wasting 2 Gbit/s of the PCI-bus capacity. On the other hand, fast-path forwarding and round-robin packet scheduling allow to draw from each input NIC just half of the packets in the incoming flow, dropping all overspill traffic at the input NIC and preserving precious PCI-bus bandwidth.

In configuration *E*, a single unidirectional 1-Gbit/s flow consisting of packets to be forwarded to each of the two output NICs in a 1:1 ratio was correctly fast-path routed and delivered to the right output ports by the input FPGA-enhanced NIC.

In configuration *F*, fast-path forwarding allows to route three bidirectional 1-Gbit/s flows, yielding a 3-Gbit/s aggregate throughput. Unfortunately, due to the limited availability of FPGA-enhanced NICs in our lab, it was not possible to evaluate routers with more than three ports.

C. QoS Tests

To improve the quality of service with respect to the classical FIFO best-effort model, the FPGA-enhanced NICs can classify traffic into two service classes handled with different priority. The tests in this section are performed with traffic

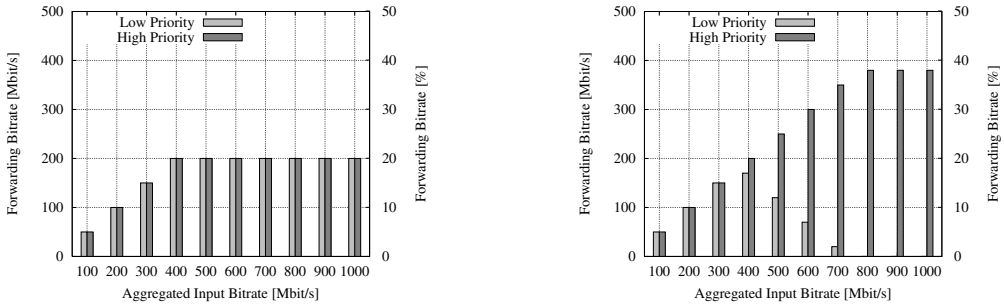

Fig. 5. Forwarding rate of the FPGA-enhanced NIC vs. the input packet rate in the *A* slow-path configuration, when packet classification is disabled (left) or enabled (right)

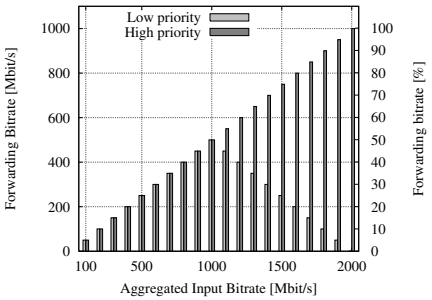

Fig. 6. Performance of the FPGA-enhanced NIC with two-priority traffic. Test configuration D. Fast path exploited

flows consisting of a mix of low- and high-priority packets identified by different values in the ToS IP-header field.

The left-hand side of Fig. 5 shows the router forwarding rate as a function of the input traffic rate, in the *A* slow-path configuration when the input FPGA-enhanced NIC ignores the existence of the two traffic classes, eventually working as a bare-bones commercial NIC. Obviously, all the packets are handled in the same way regardless of the class they belong to, and both low- and high-priority packets experience the same loss rate. The maximum forwarding rate is still upper bounded by the 400-Mbit/s packet-forwarding bottleneck.

Enabling packet classification, we obtained the results presented in the right-hand side of Fig. 5. In this scenario, high-priority packets are recognized and privileged by the FPGA-enhanced input NIC and, as long as the input packet rate is less than the forwarding bottleneck, only low-priority packets are dropped on its input FIFO.

Packet classification can be used jointly with fast-path routing in configuration *D*. The results are shown in Fig. 6; similarly to the previous scenario, high-priority packets receive a better service whereas only low-priority packets experience losses.

V. Conclusions

In this paper we assessed the feasibility of building a high-performance FPGA-enhanced NIC for a software-based IP router. We designed, programmed, and tested three FPGA-enhanced NICs that offload the CPU from performing IP routing and directly transfer packets across the PCI bus, completely bypassing the standard Linux IP stack.

We ran a number of experiments to test the implementation correctness and evaluate the maximum data-plane throughput, completely ignoring all control-plane-related issues. The results are promising, since a software router based on a high-end off-the-shelf PC and commercial NICs can forward up to 400 Mbit/s, when handling 64-byte packets, whereas the use of up to three FPGA-enhanced NICs allows to increase the forwarding throughput up to 3 Gbit/s. Unfortunately, we have not been able to assess how close we can get to the 4-Gbit/s saturation limit of a PCI-X bus running at 66 MHz, because we had only three PLDA boards.

Acknowledgments

This work was performed in the framework of two projects, named EURO [12] and BORA-BORA, partly funded by the Italian Ministry of University, Education, and Research (MIUR), and developed in the high-quality Internet Protocols and Architectures Laboratory (LIPAR) at Politecnico di Torino.

References

[1] L. Torvalds, “Linux OS.” [Online]. Available: http://www.linux.org
[2] E. Kohler, R. Morris, B. Chen, and J. Jannotti, “The Click modular router,” *ACM Trans. on Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.
[3] “BSD Unix.” [Online]. Available: http://www.bsd.org
[4] M. Handley, O. Hodson, and E. Kohler, “Xorp: An open platform for network research,” in *Proc. of the 1st Workshop on Hot Topics in Networks*, Princeton, NJ, USA, Oct. 28–29, 2002.
[5] GNU, “Quagga.” [Online]. Available: http://www.quagga.net
[6] A. Bianco, R. Birke, J. M. Finochietto, G. Galante, M. Mellia, P. M.L.N.P.P., and F. Neri, “Click vs. Linux: Two efficient open-source IP network stacks for software routers,” in *Proc. of the IEEE Workshop on High Performance Switching and Routing (HPSR 2005)*, Hong Kong, P.R. China, May 12–14, 2005, pp. 18–23.
[7] “Opencores Project.” [Online]. Available: http://www.opencores.org
[8] PLDA, “PLD Applications.” [Online]. Available: http://www.plda.com
[9] “ALTERA.” [Online]. Available: http://www.altera.com
[10] J. H. Salim, R. Olsson, and A. Kuznetsov, “Beyond Softnet,” in *Proc. of the 5th Annual Linux Showcase & Conference (ALS 2001)*, Oakland, CA, USA, Nov. 5–10, 2001.
[11] Agilent, “N2X RouterTester 900.” [Online]. Available: http://advanced.comms.agilent.com/n2x
[12] “EURO: University Experiment on Open-Source Routers.” [Online]. Available: http://www.diit.unict.it/euro