HERO: High-speed enhanced routing operation in Ethernet NICs for software routers

HERO: High-speed Enhanced Routing Operation in Ethernet NICs for Software Routers ⋆

Michele Petracca Robert Birke Andrea Bianco

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

Abstract

Software/open routers, PCs (Personal Computers) running open source OSs (Operating Systems) and equipped with Ethernet Network Interface Cards (NICs), are receiving increasing attention in the research community, because they can offer multi-gigabit-per-second packet forwarding speed, performance comparable to those of low-medium end commercial routers. However, commercially available NICs lack programmability. Furthermore, the use of standard NICs implies that each packet crosses the bus twice, and is processed and routed in software by the OS, thus reducing forwarding performance. In this paper we discuss the design and the implementation of an FPGA-based NIC that permits to overcome the performance bottleneck and the lack of flexibility of commercial NICs. Performance and limitations of the proposed approach are thoroughly discussed.

1 Introduction

Software/open routers are typically based on off-the-shelf hardware and open-source operating systems running on Personal Computer (PC) architectures. They are receiving increasing attention in the research community, because high-end PCs shared buses fit into the multi-gigabit-per-second routing segment with lower prices than those of commercial routers.

SRs (Software Router) based on PC architectures can be considered as a central memory packet switch. Ethernet NICs (Network Interface Cards) are connected to the PC bus, receive packets from the network and transfer them into the main

⋆ A preliminary version of this paper was presented at the conference IT-NEWS International Telecommunication NEtworking WorkShop 2008, held at Venezia (Italy) in February 2008, under the title ”HERO: High-speed Enhanced Routing Operation in Software Routers NICs”.

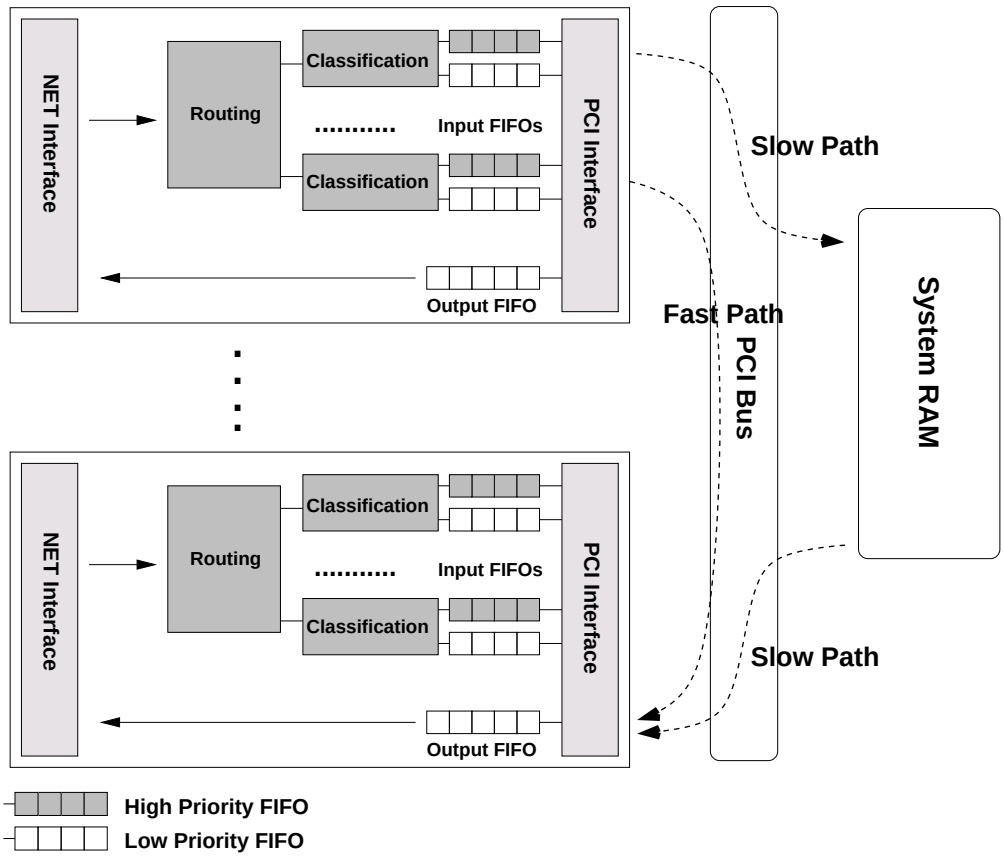

Fig. 1. Enhanced software router structure

memory. Packets are then routed by the OS (Operating System), transferred back to the NICs and re-injected into the network.

Commercially available NICs require that packets cross the shared PC bus twice. Furthermore, packets are always routed in software by the OS, thus further reducing the forwarding performance, especially when dealing with small-size packets. Fig. 1 shows how the data forwarding procedure may be optimized to increase the SR performance by exploiting the well-known fast path in a custom NIC [1]. Note that, besides performance increase, classification schemes can also be introduced, hence enabling, for example, priority enforcement to improve QoS support.

Indeed, the main performance impairment when routing packets in a software router is in the centralized nature of PC architectures. The shared memory, the CPU, and the OS involved in packet routing may easily become a bottleneck. A distributed approach, where each NIC is able to determine the output port for a significant portion of the incoming traffic, allows using the shared bus to forward packets directly between NICs. In this case, the main limitation becomes the PC bus bandwidth, since OS processing is required only for a small fraction of packets, thus reducing the occurrence of centralized processing bottlenecks. In this paper we refer to the direct exchange of packets among NICs as the *fast path*. All the packets whose destination cannot be determined locally on the input NIC, are routed by the OS

following the so-called *slow path*. Both paths coexist and permit to forward data traffic simultaneously, as shown in Fig. 1.

The *fast path* has several advantages: No memory access latency during read operations, more efficient use of the bus by reducing the bus occupancy for packet transmissions, and CPU off-loading. Furthermore, QoS oriented classification and scheduling algorithms may replace the FIFO service discipline available on commercial NICs. However, to implement direct NIC-to-NIC communication through the fast path, the NIC must perform autonomously the routing function to determine the packet destination. Furthermore, a protocol for direct NIC-to-NIC communication has to be defined and implemented.

The availability of powerful programmable logics permits to extend the open software paradigm to the hardware domain. The logic circuitry developed for FPGAs can be made public [2], reused and improved by the research community. This "open hardware" approach can open the door to low-cost hardware implementations of performance-critical functional blocks.

In this paper, we present a detailed description of a re-engineered version of a FPGA-based NIC, whose performance was partly assessed in [1]. An important motivation to develop the core is providing the research community with an open-source VHDL core implementing the fast path packet processing and capable to communicate with a PCI-X core and an Ethernet MAC core.

The paper is organized as follows. Sec. 2 gives a general overview of the main features of the custom NIC, while the next five sections present a more detailed analysis: Sec. 3 is devoted to NIC configuration, Sec. 4/Sec. 5 to incoming/outgoing packet management, Sec. 6 to the *slow path* and Sec. 7 to the *fast path* description. In Sec. 8, the outcome of the logic synthesis process is presented. Sec. 9 provides details on the hardware equipment and on the IP cores used for this project. In Sec. 10, performance results are assessed. Finally, Sec. 11 draws conclusions.

2 HERO Architecture Overview

HERO (High-speed Enhanced Routing Operation) is the name of the IP core developed within the framework of the BORA-BORA (Building Open Router Architectures - Based on Router Aggregation) project [3] . The developed IP core exploits two available IP cores managing respectively the interfaces with the Ethernet network and the PCI-X bus. HERO is organized in three main sections, respectively performing the following tasks:

- NIC configuration, through interaction with the Linux driver by means of registers and interrupts

- forwarding of incoming packets, i.e. packets received from the network. Packets are stored into either the central memory when using the *slow path* or NICs memory when exploiting the *fast path*
- forwarding of outgoing packets, i.e. packets received from the driver or from other NICs are sent to the network

Fig. 2 provides a high-level view of the HERO architecture. The NIC configuration section deals with the control path, and includes the *Register File* (RF) block, which includes 64 32-bit wide registers, and the *Interrupt Generator* block. A more detailed description is provided in Sec. 3.

The *Descriptor Queues* block controls three FIFOs, containing the RAM memory buffer addresses where packets are stored if using the *slow path*. Sec. 6 is devoted to the detailed description of this block.

Incoming packets are managed by the *Incoming Packet Management* block. This block receives packets from the Ethernet core, buffering them if possible or discarding them if the FIFOs are congested, and performs the routing and classification functions exploiting a VOQ (Virtual Output Queuing) buffering architecture, i.e., one separate FIFO queue is available for each output NIC in the router. Finally, it forwards packets either on the *slow* or on the *fast path*.

Outgoing packets are managed by the *Outgoing Packet Management* block, which forwards them to the Ethernet core.

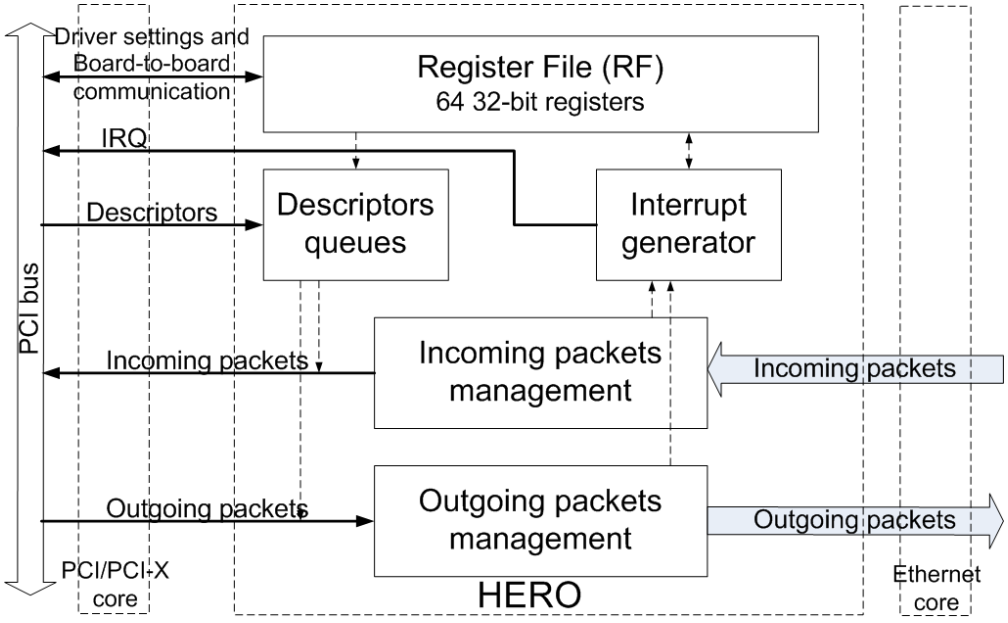

Fig. 2. HERO structure

The RF block is used by the driver to write and read configuration data. The registers can be logically grouped into 5 sets, depending on their use: command registers, IRQ registers, descriptor registers, routing and classification registers, and inter-NIC registers.

The command registers are used either to issue a set of commands to the NIC, like the reset command, or to manage the Ethernet core, e.g., enable Ethernet autonegotiation, configure the MAC address, verify the connection status.

The *Interrupt Generator* block manages three IRQ registers to signal to the OS important asynchronous events such as a packet arrival or departure. A 32-bit $R_{unmasked}$ register maps 32 possible interrupt events; currently, not all the bits are used. Whenever one of the interrupt events occurs, the corresponding bit in $R_{unmasked}$ is raised. A 32-bit R_{mask} register is set by the driver to select which events are allowed to generate an IRQ. Since a unique IRQ channel is assigned to the NIC, an IRQ is generated every time one of the enabled events occurs, if satisfying the following condition:

$$R_{unmasked} \oplus R_{mask} > 0 \tag{1}$$

The result of the $\oplus$ operation is stored in the R_{masked} register, to allow the driver to read the event that has triggered the IRQ. The *Interrupt Generator* block detects the events, masking the undesired ones and generating the IRQ signal. Every time the driver reads the R_{masked} register, it clears it to permit re-assertion of the proper bit when the next IRQ event occurs.

The use of maskable interrupts allows the driver to run in two different operating modes: IRQ and NAPI. In IRQ mode, each packet reception and transmission generates an IRQ. This operating mode is very easy to implement, but it can lead to performance degradation due to the well know IRQ trashing phenomenon [4]. IRQ trashing occurs when the CPU is flooded by IRQs and is unable to perform any other operation apart from processing IRQs. To avoid this problem, the NAPI operating mode, based on the polling idea, has been devised [4]. When adopting NAPI, the IRQ signal is used only to add the NIC to a polling list, disabling its IRQs until the NIC is active in processing packets. When no more packets are available, the NIC is removed from the polling list and NIC IRQs are re-enabled. Therefore, when the packet transfer rate is low, the NIC driver mainly works on IRQs. On the contrary, during high loads periods, polling will mostly be used, thus combining the low latency property of the IRQ scheme with the high throughput of polling systems.

The routing and classification registers are used to route and classify packets into

priority classes. These registers are managed by the driver and are used to address the incoming packets into the VOQ system. Further details about routing and classification operations are given in Sec. 4.

The inter-NIC registers are used for NIC-to-NIC signaling, needed when dealing with data transfer via the *fast path*. More details are provided in Sec. 7.

Finally, the descriptor registers are used to make packet descriptors available to the NIC. Sec. 6, describing the *slow path* operation, provides a detailed explanation on descriptor registers types and functions.

4 Incoming Packets Management Block

The *Incoming Packet Management* block is the first stage that processes a packet received from the Ethernet core. A detailed block diagram is shown in Fig. 3.

4.1 Parallelization, classification and routing

The interface with the Ethernet core comprises an 8-bit data bus and some control signals. Three control signals are defined: a *data valid* signal, asserted when a byte on the data bus is valid, a *start* (*stop*) signal, asserted for one clock cycle during the

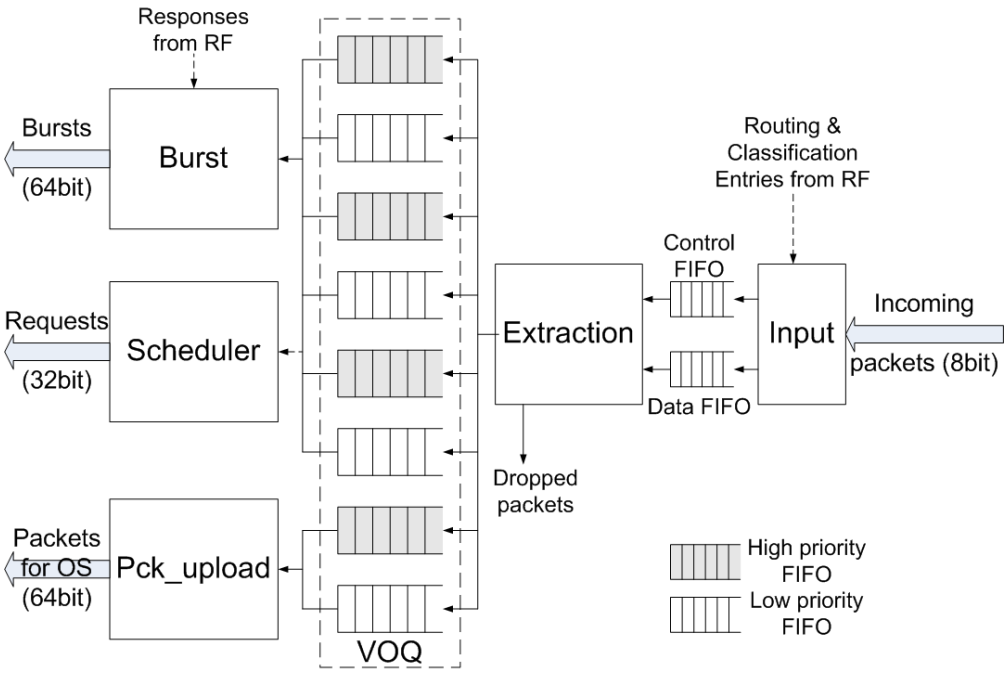

Fig. 3. Incoming packet management block structure

first (last) byte of packet transmission. The HERO core is designed to be always able to receive data from the Ethernet core, internally dropping packets if necessary.

The *Input* block (Fig. 4) transforms the serial 8-bit data into a parallel 64-bit word compatible with the PCI-X bus parallelism. During the parallelization phase, the *Open Header* block collects the initial words of the packet and transfers the Ethernet and IP headers to the routing and classification logic. Additional logic in the *Input* block decrements the TTL field and updates the IP header checksum. The routing operation is based on the destination IP address (32 bits). The classification function is based on the following fields (80 bits):

- destination IP address (32 bits)
- source IP address (32 bits)
- ToS (Type of service) (8 bits)
- protocol type (8 bits)

The system supports up to 4 custom NICs in a single PC; therefore, up to three routes and four classification rules can be provided by the driver. Every packet not matching any of the given routes is sent along the *slow path* to the OS. Each classification rule is associated with a possible destination, either one among the three NICs or the OS. If a packet matches a classification rule, it is considered as a high priority packet.

Routing and classification are based on two ternary masks. Since the FPGA uses only binary logic, the masks were implemented using two separate registers. The first register contains the pattern P to be matched. The second register defines a bit mask M, where 1 indicates a "care bit" and a 0 a "don't care" bit. A value V in the

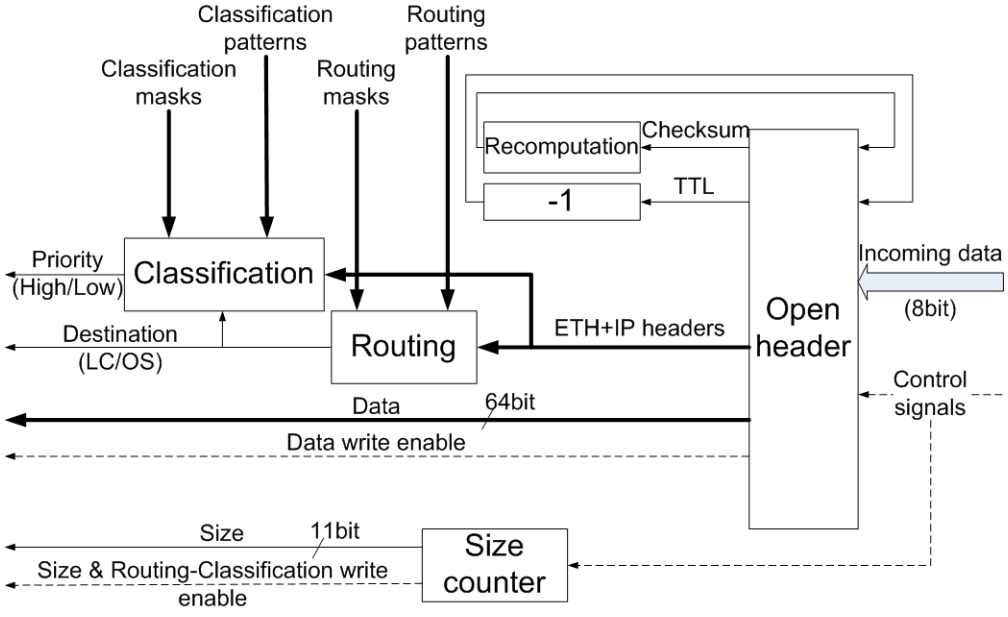

Fig. 4. Input block details: routing and classification blocks structure

packet header is a match if:

$$(V \oplus P) \cdot M = 0 \tag{2}$$

The destination IP address of each packet is compared in parallel with all three ternary masks. The packet is forwarded to the lowest ranked NIC having a matching ternary mask; otherwise, the packet is sent to the OS. The classification step is executed concurrently with the routing. The header fields are compared with all the classification ternary masks in parallel, thus obtaining one outcome for each possible destination.

By using special values for M and P, it is possible to deactivate routing and classification functions. To disable routing, a pattern of all zeros and a mask of all ones should be used. Since no packet can have an IP address set to 0.0.0.0, all the packets are sent to the OS.

To disable classification, it is necessary to set all the M bits to 0; thus, all the packets match the rule and are classified as high priority. When disabling both functions, the board becomes a standard NIC, which supports a single priority and a single FIFO queue storing packets addressed to the OS.

For each possible route, i.e., the 3 different NICs on the *fast path* and the OS on the *slow path*, a counter keeps track of the number of packets sent to the considered destination. All those values are stored in the RF and can be accessed by the driver. The driver can dynamically configure the NIC registers, removing and adding route entries. This approach allows to dynamically redirect the highest bandwidth flows on the *fast path*, forwarding the others through the *slow path*.

4.2 Dropping

After the routing and classification stages, packet destination and priority level are established, and the packet is enqueued into an intermediate FIFO. From this FIFO, packets are extracted by the *Extraction* block that enqueues the packet in the proper FIFO in the VOQ architecture, according to the result of the routing and classification process. If not enough space is available in the destination queue to store the full packet, the packet is dropped.

The intermediate FIFO is used to address some design issues. Packets could be directly dropped by the Ethernet core. Indeed, when any of the FIFOs is full, HERO could simply advertise this information to the Ethernet core by "or-ing" the control signal that detects FIFO queue overflow. Since the routing and classification have not been performed yet, there is no possibility of detecting the status of the "proper" FIFO. This solution is not optimal, because the Ethernet core would drop

also packets addressed to non-full queues. By using the intermediate FIFO, the packet is received, processed and dropped only if the proper destination FIFO is full. Moreover, a store and forward technique is needed to determine the size of the packet and to check whether it fits into the destination queue. The intermediate FIFO also simplifies the control logic allowing to perform the routing and classification process while receiving the packet, making those operation zero-latency. If no buffer stage is introduced, it would be necessary to obtain the information on the destination queue at the beginning of the packet, making the control logic more complex or delaying the packet enqueuing. With our approach, it is enough to execute routing and classification operations within a time bound equal to the duration of the shortest possible Ethernet packet. Obviously, an additional store and forward delay is paid. Finally, the intermediate FIFO is dual-clocked, and is used to decouple the 125MHz clock domain of the Ethernet core from the 133-100MHz clock domain of the PCI-X core.

4.3 VOQ enqueuing

The *VOQ* block contains four pairs of FIFOs, one pair for each supported output NIC. One FIFO collects high priority packets, the other one low priority packets. Each FIFO, as well as all the other FIFOs within the design, is physically composed by a “data” FIFO, storing packets, and by a “size” FIFO, storing the packet size needed to determine packet boundaries in the data FIFO during packet extraction operations.

The *Packet Upload* block is involved in the *slow path* forwarding, i.e., packet transmission from the two FIFO queues to the PC main memory. The *Scheduler* and the *Burst* blocks are involved in the control and data forwarding over the *fast path*.

5 Outgoing Packet Management Block

The *Outgoing Packet Management* block, shown in Fig. 5, is much simpler than the incoming packet management block. It performs the multiplexing of packets received from both the *slow* and the *fast path*. A RR (Round Robin) policy is followed, by extracting alternatively one packet from each queue. The *Transmission* block serializes the 64-bit words received from the PCI bus in words of 8-bits, the parallelism needed by the Ethernet core.

Further details about the behavior of the *Reception* and *Scheduler* blocks are provided when analyzing the NIC-to-NIC communication protocol via the *fast path* in Sec. 7.

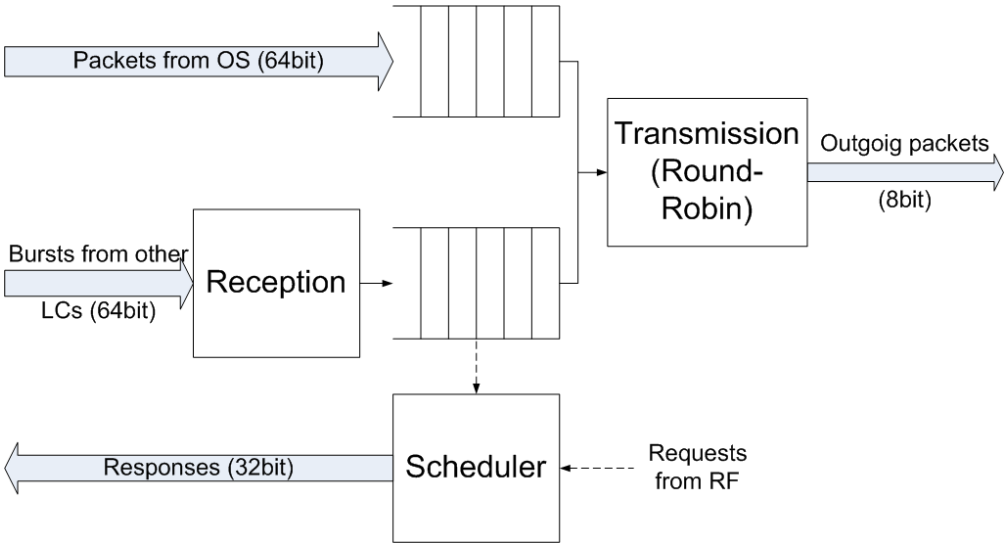

Fig. 5. *Outgoing packets management* structure

6 Slow Path

The packet passing mechanism between the driver and the NIC is based on packet descriptors. A packet descriptor is a data structure containing two basic information: packet data address in memory and packet size. These information are passed to the NIC to exploit its bus master capability to initiate a DMA transfers through the bus.

When a packet is received by a NIC from the network, it is locally processed, stored and then forwarded as soon as possible to an available central memory location. The memory location is determined by the driver by means of a descriptor. When the packet transfer ends, the event is signaled to the driver rising an IRQ signal. Thus, the driver can handle the packet to the OS, that will perform the routing operation.

Similarly, when the driver receives a packet from the OS kernel, it handles the descriptor to the NIC, which reads the packet from the central memory to send it on the Ethernet cable. When packet reception is completed, an IRQ signal is generated to free the packet memory.

To increase performance, modern NICs organize the descriptors into circular buffers called rings. These rings can be implemented at least in two different ways, with different performance. We refer to the two implementations as *scatter-gather* and *vector* mode.

The *scatter-gather* mode organizes the descriptors into a linked list; the list head is passed to the NIC. For each packet, the NIC first reads the descriptor and then starts the packet transfer. Two PCI transactions for each packet are needed. The *vector* mode tries to overcome this problem by grouping the descriptors into vectors. The

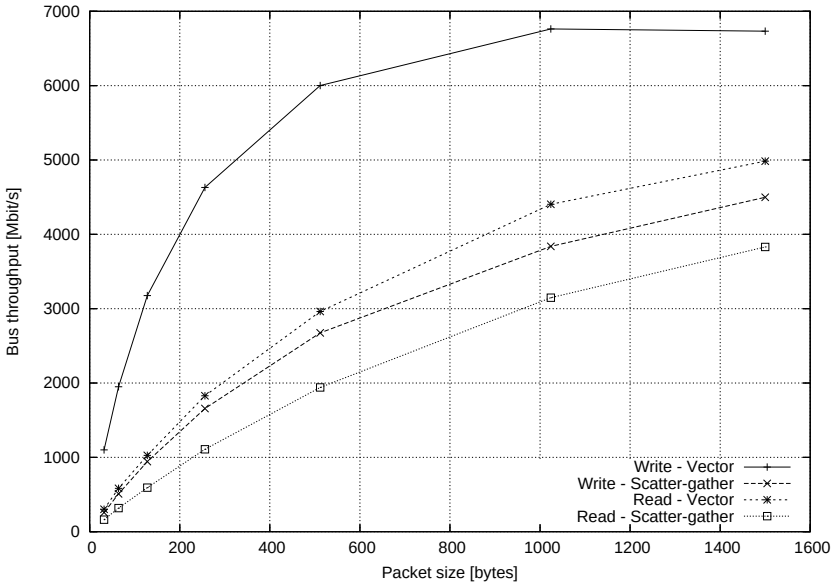

Fig. 6. Performance comparison between scatter-gather and vector mode descriptor rings (PCI-X @ 133 MHz).

NIC can read all the descriptors at once, thus optimizing the bus utilization.

Fig. 6 compares the two modes in terms of achieved throughput. The *vector* mode shows better performance both in reading and writing. Indeed, the *scatter-gather* mode needs several small bus transactions to download the descriptors. This reduces performance, due to the high initial transaction setup cost in accessing the bus. The *descriptors queues* in Fig. 2 are the FIFOs where the descriptors are stored and are organized according to the *vector* mode.

Two vectors are used to build the rings. During packet reception, the driver allocates new packet buffers into one descriptor vector. When the NIC has used all its descriptors, the driver swaps the two vectors providing new descriptors to the NIC. Two vectors are used also in the transmission side, one by the driver to group outgoing packets, while the NIC is transmitting the packets stored in the other one. When all packets are transmitted by the NIC, an IRQ is generated and the two vectors are swapped.

To better support two priorities, a second incoming buffer ring has been added.

A scheduler within the driver serves both rings. Therefore, a total of three descriptor FIFOs are needed, two for the descriptors to write incoming packets into the RAM (one for each ring) and one for reading packets from the RAM. For the sake of simplicity, the scheduler gives absolute priority to high priority packets, which implies that, in principle, low priority packets may suffer starvation. Fig. 7 summarizes the *slow path* management.

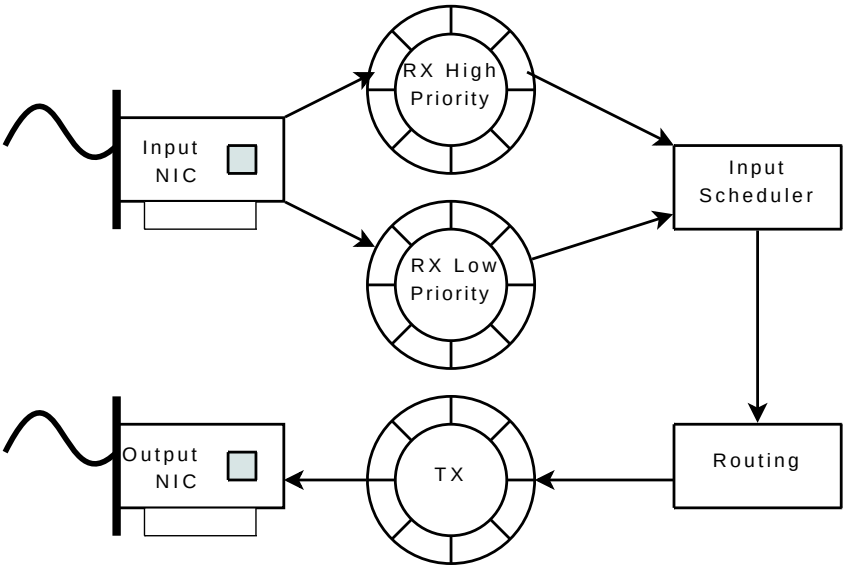

Fig. 7. Description of the *slow path*.

7 Fast Path

The *fast path* goal is to provide a high bandwidth-low latency data transfer among NICs. Indeed, the main shared resource in the PC architecture is the PCI bus. The aim of bus management is to be fair for NIC-to-NIC communications and to maximize bus throughput. For example, it is obviously not a good practice to transfer packets through the bus to a congested output NIC, since they will be discard immediately after. A better solution would be to discard these packets directed toward a congest NIC directly at the input NIC. A back-pressure mechanism can be exploited to shape the sending rate of the input NICs. Fairness must be ensured: any input NIC should be able to forward packets to any output NIC without starvation in a reasonably short amount of time.

Taking into consideration that the PCI-X protocol schedules bus access with a proprietary policy, we propose a protocol able to organize bus access requests and packet transfers through the bus to meet the above requisites. One possible approach could be to introduce the control in software: the driver first inquires the NICs state and then computes the optimal scheduling. The high latency imposed by this approach when the driver accesses the NIC registers discourages this idea. Thus, we developed a hardware-based inter-NIC communication protocol.

This protocol is inspired by the three-way schedulers proposed in synchronous slotted IQ switches, like iSLIP [5]. However, our solution is completely asynchronous. Indeed, each board can receive packets of variable size at any time; the use of the bus is granted asynchronously and independently by each output NIC. For these reasons, the protocol implements a communication protocol based on three steps:

- *request*: an input NIC sends a request containing the number of packets and the amount of data available to the proper output NIC;
- *response*: the output NIC grants to the input NIC the amount of available bytes, i.e., the available space in the output FIFO;
- *burst*: the input NIC transfers a number of packets whose total size is equal or less than the granted amount received during the *response* phase.

The structure of the protocol is partly due to some limitations related to bus access procedures. In the PC architecture, a board acting as a PCI-X bus master can perform both write and read operations toward the main memory. However, only write operations toward other peripherals are admissible. Thus, the protocol exploits write transactions only. If read operations were available, it would have been possible to use simply a *request* message; then, the output NIC would have been able to read the data directly from each input NIC.

A *request* message is a 32-bit word divided into two 16-bit sections, one for each priority. Each section contains the number of bytes stored in the corresponding FIFO. The *request* message is used to start a communication between two NICs. When both FIFO queues (one per priority) addressed to an output are empty, the communication phase ends. As soon as a new packet for that output is received, a new request message is sent, opening again the communication channel.

When a NIC receives a *request*, it sends back a *response* message containing the minimum between the value stored in the *request* itself and the size of the available memory in the *fast path* FIFO. All the free space available within the FIFO, if needed, is allocated to the current data exchange. If the memory available is not able to satisfy the request, the available buffer is first allocated to the high priority field; the remaining part, if any, is devoted to the low priority field. Note that different *response* generation policies can be easily defined and implemented. The *response* message is divided into two 16-bit sections, one section for each priority level. Each section contains the number of bytes available for the corresponding priority level.

When a NIC receives a *response* message, the data transfer occurs. Packets are grouped into a *burst*, composed by several high priority packets followed by the low priority ones. For each priority level, the amount of bytes of the packets transmitted within the burst never exceeds the value stored in the *response* message.

Each burst has a header that contains the total number of high and low priority packets in the burst. A 64-bit control word is sent prior of each packet in the *burst* message; the control word contains the packet size and it is needed to extract the packet from the burst at the receiving end. A 64-bit control word is appended at the end of the burst; it is used to piggy-back a new 32-bit *request* for the two priorities. If no other packets are available for that output NIC at the end of the burst transmission, the request bits are set to zero. The piggy-backing technique keeps the communication alive and, during high load periods, saves a bus transaction, thus

improving forwarding performance.

The generation of *request* messages is performed by monitoring the VOQs occupancy status. In Fig. 3, the block generating *request* messages is named *Scheduler*. The same physical block, as shown in Fig. 5, is also involved in the management of the requests and in monitoring the *fast path* FIFO to generate the *response* messages at the receiving side. This block is also involved in the transmission phase, as shown in Fig. 5. The *Burst* block in Fig. 3 is in charge of generating the *burst* message, taking into account the incoming *response* and the new *request* to piggy-back. When the *burst* arrives at the receiving side, it is parsed by the *reception* block in Fig. 5, and packets in the burst are stored into the proper FIFO.

Conflicts among NICs may occur in high load conditions. Indeed, two input NICs may have data available for the same output NIC. In this case, bursts are serialized and not interleaved. More precisely, the *request* message is sent by the input as soon as data arrive in an empty FIFO. *Request* messages are received and collected by the outputs asynchronously, and then processed in a Round Robin (RR) fashion. If resources are available, a *response* for the first *request* in the RR order is sent back. Then, the NIC waits to receive the *burst*. Only when the *burst* is completely received, the following *request*s are processed. This mechanism allows each NIC to wait for at most one *burst* at a time, simplifying both the resource allocation process and the communication protocol. If the input NIC, upon reception of a *response* message, is unable to send at least one packet in the *burst* due to size constraints, an empty burst is sent. The empty burst contains only the piggy-backed *request* to keep alive the communication channel.

A RR scheduler is also adopted to manage incoming *response* messages. Each NIC detects the first *response* message and creates the corresponding *burst*. Only when the *burst* has been completely sent, the NIC examines the next *response* message.

The *request* and *response* messages are written in the inter-NIC registers described in Sec. 3.

8 Digital Design Analysis

In the above sections we discussed the main features of the HERO project mapping them on the logic design of the HERO core. To better understand the physical level of our implementation, we describe the results of the logic synthesis process.

The project has been synthesized on an Altera Stratix GX FPGA equipped with 41250 *Logic Elements* (LE) and $3.3Mbits$ of embedded RAM. In a FPGA device, a LE is the basic programmable block. It is composed by a Flip-Flop (FF) and a LookUp Table (LUT) that can be configured and interconnected to compose a ba-

	LEs	% design	% FPGA
PCI	4798	20	12
MAC	4227	18	10
PHY	1541	6	4
HERO	15605	56	38
TOT	26171	100	63

Table 1
FPGA resource usage break down by core.

sic logic function. The embedded memory is composed by Static RAM (SRAM) blocks distributed on the FPGA area, that are used to implement single-clock or dual-clock FIFOs or to implement single-port or dual-port RAM blocks. The number of LEs upper-bounds the amount of logic and the number of registers the design can use, while the available memory constraints mainly the FIFO buffer size.

Tab.1 offers the break-down of the resources used for each IP core including the PCI-X core and Ethernet core (including both the MAC and the Physical layers). The entire design uses 63% of the available resources on the FPGA. This leaves a wide possibility to expand the design: i.e. more complex allocation policies in the Fast-Path scheduling protocol or more routing and classification rules. For the routing and classification entries, considering that each FF implies the use of one LE and that a routing-classification rule consists of 224 bits (see Sec. 4.1), we estimate that the available LEs permit to implement up to 100 routing-classification rules. However, the additional resource utilization will certainly affect HERO's maximum clock frequency. The relationship between clock frequency and FPGA occupancy is not linear: thus, it is not easy to predict the impact of adding new rules on the HERO core clock speed.

Fig. 8 shows the break-down of HERO resources used by each logic function. Most of the area is used by the incoming packets management, the register file and the outgoing packets management. The large percentage of area used by the packet management blocks is due to the complexity to manage packet bursts, while the register file area is mainly devoted to the registers. All other functions require a minimal resource usage.

Looking at memory utilization, most of the memory is used to implement the VOQ system, Indeed, each FIFO queue requires $32kBytes$ and a total of 24 FIFOs is required to build the VOQ architecture for three NICs. The FIFO queue used to decouple the two clock domains is set to $2kBytes$, because it never holds more than one packet at a time (the packet arrival rate is lower than the extraction rate). The queue receiving packets from the *slow path* has a $16kBytes$ size, whereas the FIFO queue receiving packets from the *fast path* is larger ($64kBytes$) to permit long packet bursts.

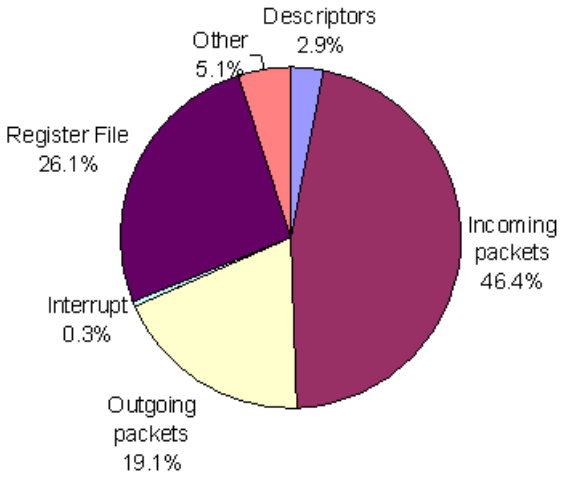

Fig. 8. HERO resource usage break down by function

9 Hardware Equipment

HERO was tested using prototyping boards from PLDApplication [6] designed for networking applications. The board hosts an Altera Stratix GX FPGA and provides two SFP (Small Form-factor Pluggable) transceiver slots able to support both optical and copper based network connectivity. Bus connectivity is provided with a 64-bit PCI-X connector.

The Altera Stratix GX family is thought for high-speed communication applications, embedding hardware support for connectivity up to $3.125GHz$. In our project, the target is a Gigabit Ethernet connectivity, which translates to a physical signal frequency of $1.25GHz$, due to the $8B/10B$ encoding of the Ethernet physical layer. The maximum clock frequency achievable on the FPGA is $325MHz$. However, the design maximum clock period is strongly affected by the size of the circuit. The clock design requirements are $125MHz$ for the network side clock and at least $100MHz$ on the bus side clock.

A commercial core developed by MTIP (More Then IP) [7] is used to manage the Ethernet interface. It fully implements both the physical layer and the hardware-related part of the MAC layer. The physical layer manages channel synchronization, auto-detection and bit transmission/reception from/to the wire. The MAC layer buffers packets into internal FIFOs allowing an easier management of data flows. Furthermore, it evaluates the Frame Check Sequence (FCS) field of incoming packets and overwrites the source MAC address of outgoing packets. It also

offers support for Jumbo frames, VPN and multicast, which are not considered in our application.

The PCI-X core is directly provided by PLDA. In master mode, it provides 4 independent DMA channels that are multiplexed in time by the core itself. The core does not buffer packets, but it simply redirects in real-time the bus control signals to the control logic that schedules data transfers, either asking for or providing the packet when the transaction is under progress. For each channel, the control logic provides to the core the base address, the data size and the transfer type: read or write.

The NIC's host is a high-end PC with a SuperMicro X5DPE-G2 main-board equipped with one $2.8GHz$ Intel Xeon processor and $2GByte$ of PC1600 DDR RAM consisting of two interleaved banks, so as to bring the memory bus transfer rate up to $3.2GByte/s$. This motherboard has three PCI-X 133 and three PCI-X 100 slots with peak transfer rates of respectively $1GByte/s$ and $0.8GByte/s$.

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

10 Performance Evaluation

The aim of this section is twofold. First, we wish to assess the NIC performance in terms of forwarding rate and latency on the one hand, and of flexibility in dealing with different priority flows on the other hand. Second, the benefits provided by the developed NIC with respect to standard NICs are examined. All tests are run with NAPI enabled on the Linux kernel.

10.1 Throughput

We first test the developed NIC as a standard commercial NIC to forward packets passing through the OS. The achievable throughputs are in line with those obtained using commercial Gigabit NICs (e.g., Intel e1000 NICs) working under the same conditions. The forwarding limit is $400Mbps$ when using minimum sized ($64byte$) Ethernet Frames. This derives from both CPU and memory latency bottlenecks, as shown in [9]. To overcome this limit, the developed NIC uses the *fast path*. On the *fast path* it is possible to route all the packets at wire speed, regardless of their size. Fig. 9 summarizes these results.

The aggregate SR throughput can be increased by adding more cards. By using 3 boards plugged into the PCI-X bus, we are able to route an aggregated traffic

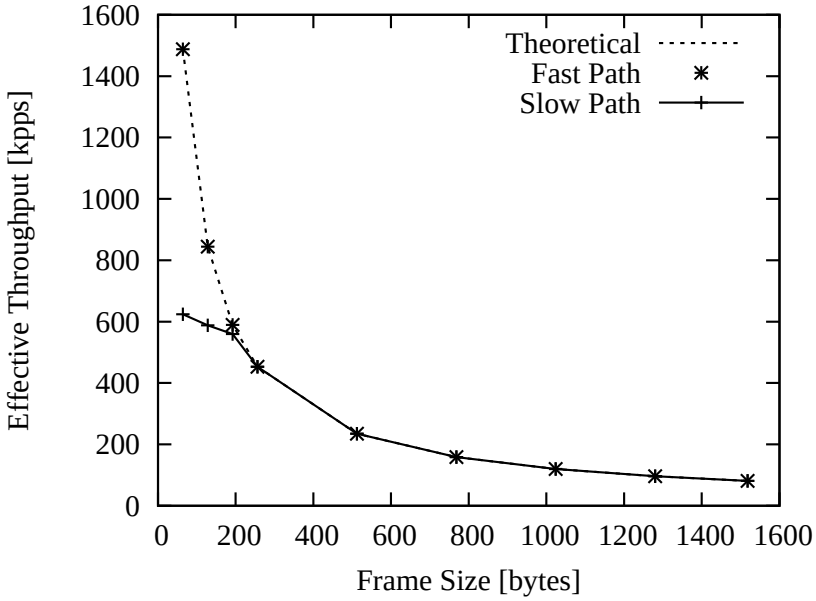

Fig. 9. Number of packets-per-seconds forwarded by each path

flow of $3Gbps$. Unfortunately, due to the lack of FPGA cards in our lab, it was not possible to try configurations with more then three NICs, even though, theoretically the PCI-X bus is able to support loads up to the peak value of $8.512Gbps$.

To improve the quality of service support with respect to the classical FIFO best-effort model provided by commercial NICs, the developed NIC is able to classify traffic in two priority classes and treat them differently. When all the traffic can be routed without losses, the classification has no impact on packets forwarding. But in overload, having different priority flows permits to allocate the limited amount of resources to packets with higher priority. Fig. 10 shows the forwarding performances along the *slow path* without packet classification, like on standard NICs. All the packets are treated the same, and losses are equally distributed between low and high priority packets. By adding packet classification, we obtain the results shown in Fig. 11. High-priority packets experience a higher forwarding rate to the detriment of low-priority packets.

Packet classification can also be exploited on the *fast path* when an output is overloaded by two or more inputs with an aggregate rate higher than the wire speed. Again, high-priority packets are privileged to the detriment of low-priority packets obtaining similar results as on the *slow path*.

10.2 Latency

The *fast path* not only permits to achieve higher forwarding rates, but also lower packet latencies. Two main reasons reduce the packet latency:

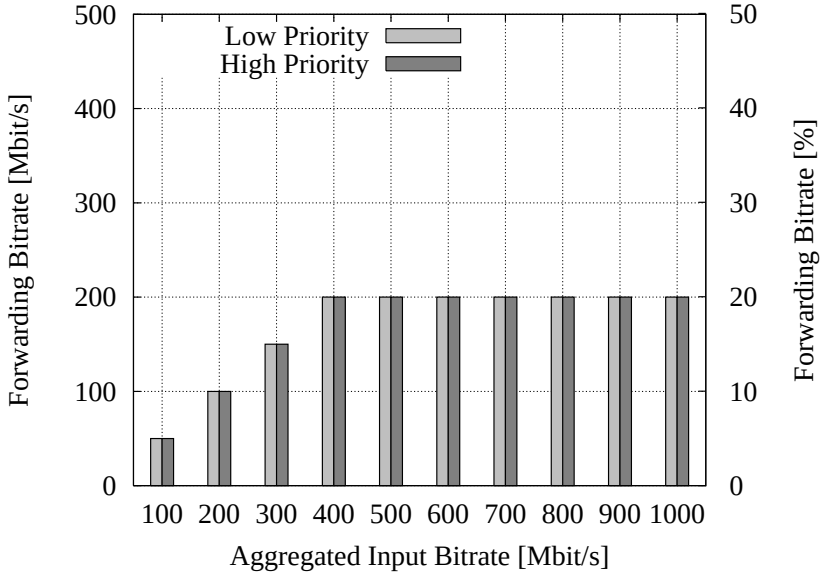

Fig. 10. Traffic forwarding with two different priority flows using standard NICs

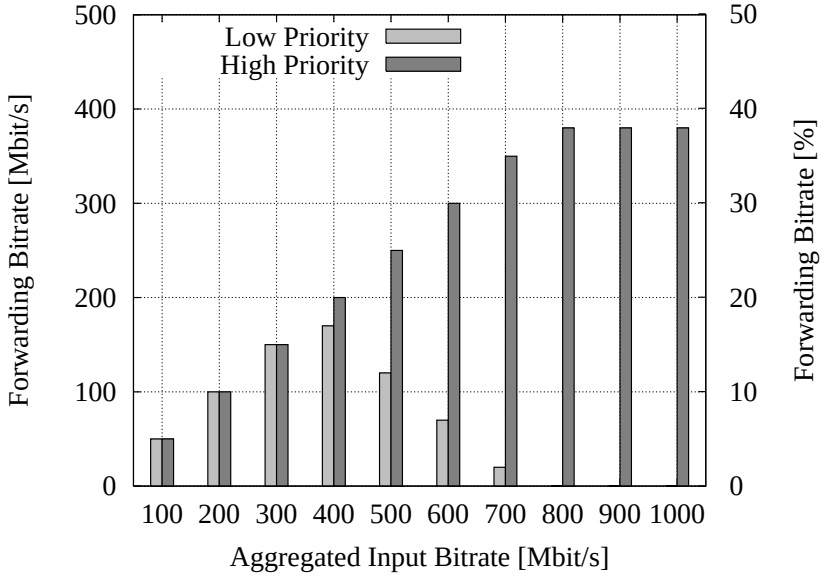

Fig. 11. Traffic forwarding with two different priority flows using HERO-based NICs

- no additional latency due to the CPU scheduling by the OS to process the packet;
- only one bus transversal per packet.

Fig. 14 compares the average latencies of the two paths close to saturation for different packet sizes, The latency is always computed at the maximum throughput with zero losses for each packet size. Fig. 12 and Fig. 13 plot the latencies versus the offered load for the slow and fast path respectively.

First of all, observe that the average latency on the *fast path* is always at least one order of magnitude lower than the one on the *slow path*. Taking a closer look at the *slow path*, the latency increases for smaller packets. This happens because the PC

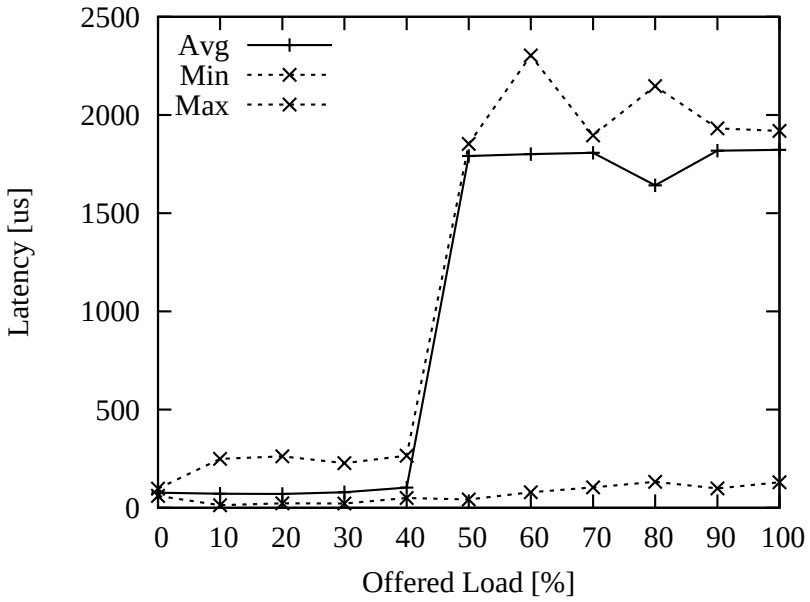

Fig. 12. Average forwarding latency over the *slow path* as function of the offered load

workload is higher for small packets. Indeed, the throughput at which the latency is computed is very close to the maximum effective throughput. With larger packet size, the load on the system decreases and the maximum throughput becomes limited by the wire speed.

When looking at the *fast path*, no increase in the latency value for small packet size is visible, because the throughput is always limited by the wire speed. The CPU processing time is mainly linear with the packet size, because the FPGA processes a fixed amount of data per clock cycle. However, around a packet size of $256bytes$, a latency drop occurs. The drop is due to a threshold phenomenon depending on the distributed message-passing scheduling algorithm adopted in the *fast path*. For small packets, due to the different transmission speeds between the PCI-X bus and the Ethernet, the probability of being able to piggy-back the next request transfer is lower than with larger packets.

Finally, Fig. 12 shows the *slow path* latency as a function of the offered load, when using $64bytes$ packets. The latency is constant until the saturation point is reached. For higher loads, the non discarded packets experience a high latency when traversing the software router. The *fast path* as shown in Fig. 13, is not affected by this performance degradation since it permits to run at wire speed. Therefore, the saturation point is never reached.

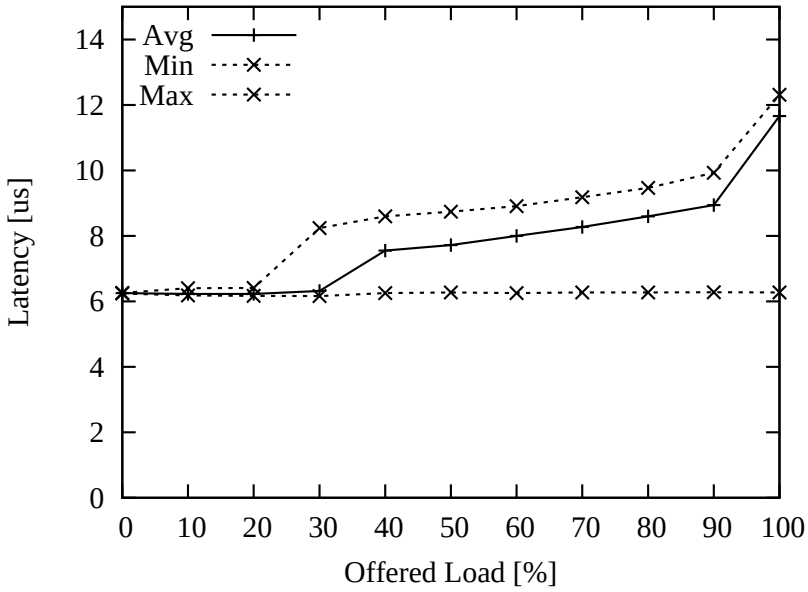

Fig. 13. Average forwarding latency over the *fast path* as function of the offered load

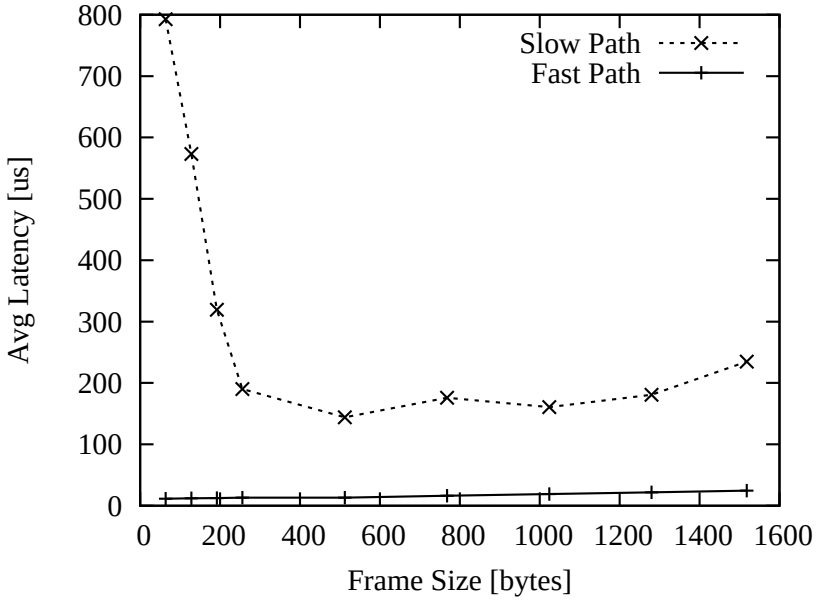

Fig. 14. Average forwarding latency over both paths for various frame sizes at the saturation point input rate

11 Conclusions

We presented the implementation choices and the design logic of a custom NIC, suitable for SR running Linux. The NIC implements extra features with respect to standard, commercial NICs reducing the CPU load and improving both QoS support and system throughput. The project is available as an open source IP core which is interfaced with a PCI-X and an Ethernet IP core to obtain a portable NIC

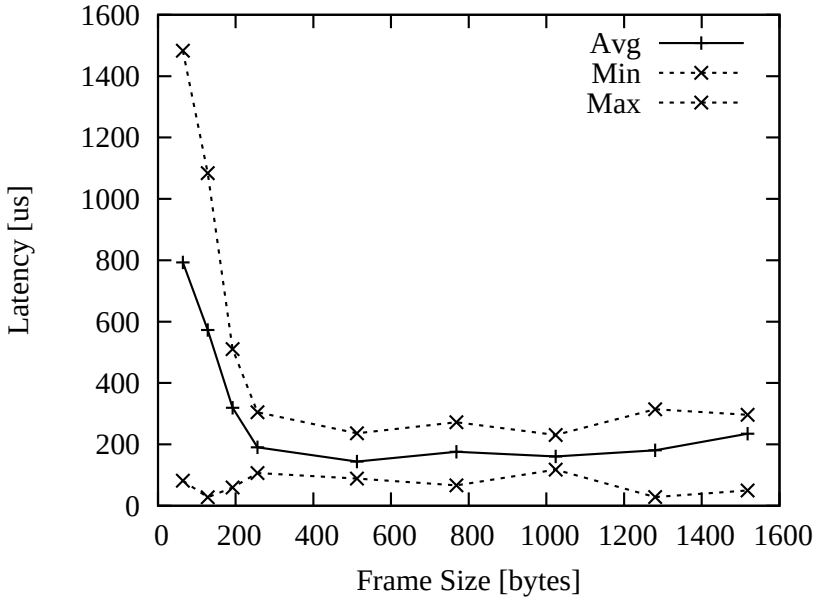

Fig. 15. Average forwarding latency over *slow path* for various frame sizes at the saturation point input rate

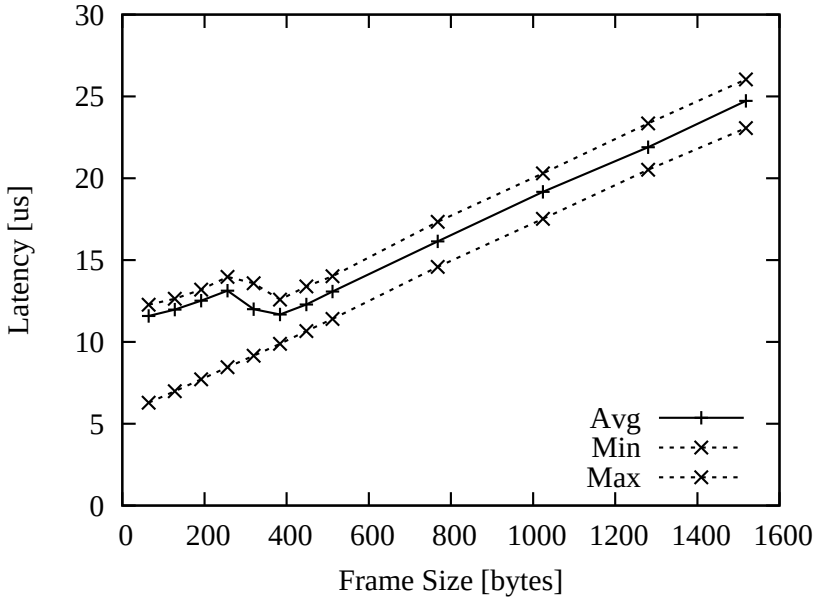

Fig. 16. Average forwarding latency over *fast path* for various frame sizes

implementation.

This core is fully compatible with the Linux OS and exploits very flexible interfaces toward both the PC bus (PCI core) and the network (Ethernet core). Therefore, it is useful not only for SRs performance analysis, but also in other applications needing non-standard network cards. e.g., security enforcement, network measurements, monitoring and analysis, development and testing of new MAC protocols for new generation optical or wireless packet networks.

The design can be modified increasing the number of possible routes and the number of possible priority classes per route. It is also possible to change the module that allocates resources in the output NIC during the response phase of the scheduling algorithm, to obtain more sophisticate fairness policies. The driver scheduler used to manage the two priorities on the *slow path* can also be modified. Finally, different mechanisms that can use the flow/packet statistics to dynamically configure the routing entries to better exploit the *fast path* are under study.

References

[1] A.Bianco, R.Birke, G.Botto, M.Chiaberge, J.Finochietto, G.Galante, M.Mellia, F.Neri, and M.Petracca, "Boosting the performance of PC-based software routers with FPGA-enhanced network interface cards," HPSR (IEEE Workshop on High Performance Switching and Routing), Poznan, Poland, June 2006

[2] "HERO: High-speed Enhanced Routing Operation for software routers." [Online]. Available: http://www.telematica.polito.it/hero/

[3] "BORA-BORA (Buildin Open Router Architectures - Based On Router Aggregation)." [Online]. Available: http://www.telematica.polito.it/projects/borabora

[4] J.C.Mogul and K.K.Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel", ACM Transactions on Computer Systems, vol.15, no.3, pp.217–252, Aug. 1997

[5] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Transaction on Networking*, vol.7, no.2, pp.188–201, 1999

[6] "PLDApplications." [Online]. Available: http://www.plda.com

[7] "MoreThanIP." [Online]. Available: http://www.morethanip.com

[8] "Agilent, N2X routertester 900." [Online]. Available: http://advanced.comms.agilent.com/n2x

[9] A.Bianco, R.Birke, D.Bolognesi, J.Finochietto, G.Galante, M.Mellia, M.Prashant, and F.Neri, "Click vs. linux: two efficient open-source IP network stacks for software routers," HPSR (IEEE High Performance Switching and Routing Workshop), Hong-Kong, May 2005