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

Michele Petracca, Robert Birke, Andrea Bianco

Dip. di Elettronica, Politecnico di Torino
Corso Duca degli Abruzzi 24, 10129 Torino, Italy
{lastname}@polito.it

***Abstract*— Increasing attention has been recently devoted to software routers based on off-the-shelf hardware and open-source operating systems running on Personal Computer (PC) architectures. Today's high-end PCs PCI shared buses fit into the multi-gigabit-per-second routing segment, for a price much lower than that of commercial routers. However, commercially available Network Interface Cards (NICs) lack programmability, and require not only packets to cross the PCI bus twice, but also to process them in software by the Operating System (OS), reducing routing performance. In this paper we discuss the design of an FPGA-based NIC that permits to overcome the limitations of commercial NICs and provide a detailed description of its implementation.**

I. Introduction

A PC-based software router can be considered as a central memory packet switch. Severals standard Ethernet NICs (Network Interface Cards) are connected to the PCI bus and transfer the packets into/from the main memory. These packets are routed by the OS, transferred back to the NICs and re-injected into the network.

Several criticisms can be raised against software routers, i.e. limitation of software, lack of system support, scalability problems, lack of advanced functionalities. In [1], we focused on the limited performance of the data plane and assessed the feasibility of building a high-performance FPGA based NIC for software routers, showing some scenarios where the implementation of basic functions in hardware can significantly improve the throughput of the entire system.

Fig.1 shows how it is possible to introduce a classification scheme and to optimize the data forwarding procedure in a shared bus architecture. Classification enables enforcing priority policies, improving QoS support. The data forwarding optimization offers a significant performance increase.

It is well known that the main performance impairment when routing packets in a software router is in the centralized nature of PC architectures. The memory, the CPU and the Operating System, involved in packet routing, may become the system bottleneck. A distributed approach, where each NIC is able to determine the output port for a significant portion of the incoming traffic, allows to use the shared bus to forward packets directly between NICs. In this case the only limitation is the bus bandwidth, but there is no central processing that can represent a bottleneck. We refer to the exchange of packets directly among NICs as *fast path*. All

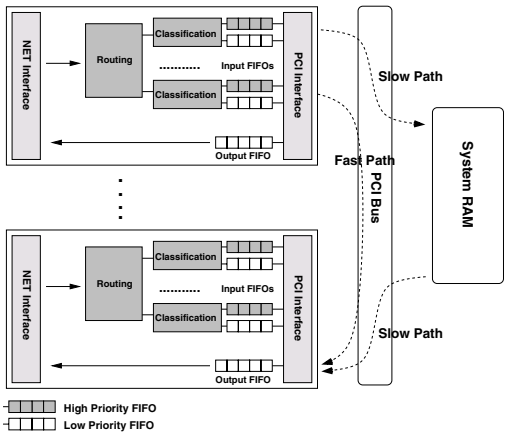

Fig. 1. Enhanced software router structure

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 forward data traffic simultaneously, as shown in Fig.1.

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

The availability of powerful programmable logics allows 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 with the purpose to provide to the community an open-source VHDL core implementing the packet processing and able to communicate with a PCI-X core and an Ethernet MAC core.

The paper is organized as follows. Sec.II describes the hardware and the IP cores used for this project. Sec.III gives a general overview of the main features of the custom NIC, while the next five sections present a more detailed analysis: Sec.IV is devoted to NIC configuration, Sec.V/Sec.VI to incoming/outgoing packet management, Sec.VII to the *slow path* and Sec.VIII to the *fast path* description. Finally, Sec.IX concludes the paper.

II. Hardware Equipment

The project used prototyping boards from PLDApplication [3] designed for networking applications. The boards host an Altera Stratix GX FPGA and provide two slots for SFP (Small Form-factor Pluggable) Transceivers able to support both optical and copper based network connectivity. The bus connectivity is provided by 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 connectivity at 1Gbps, which translates to a physical signal frequency of 1.25GHz, due to the 8B/10B encoding of the Ethernet physical layer. Further characteristics of the FPGA are: 41,250 logic elements, 3.25 Mbit of embedded RAM and 325 Mhz maximum clock frequency. The clock frequency is however strongly affected by the size of the circuit which can cause long signal propagation delays. The clock design requirements are 125MHz for the network side clock and at least 100MHz on the bus side clock.

Two commercial cores have been used: one to manage the Ethernet interfaces and one to manage the PCI-X interface. The Ethernet core is produced by MTIP [4]. It fully implements both the physical layer and the hardware-related part of the MAC layer. The physical layer manages the 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, the core provides 4 independent DMA channels that are multiplexed in time by the core itself. Thereby, it is possible to activate up to 4 simultaneous master operations, which will be scheduled by the core according to an internal scheduling policy. The core does not buffer packets, but it simply redirects in real-time the bus control signals to the control logic that schedules data transfer, 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.

III. HERO Architecture Overview

HERO (High-speed Enhanced Routing Operation) is the name of the IP core we developed within the framework of the BORA-BORA (Building Open Router Architectures - Based on Router Aggregation) project [5] . The core is functionally positioned between the two IP cores managing the network and the PCI interfaces. HERO is basically composed by three sections, respectively performing the following tasks:

- NIC configuration, through interaction with the driver by means of registers and interrupts
- forwarding of incoming packets, i.e. packets received by the driver from the network, by storing them either into the central memory when using the *slow path* or into other NICs memory over 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 an overview of the HERO architecture. The configuration section deals with the control path and includes the *Register File* (RF) block and the *Interrupt Generator* block. In the RF block, 64 independent registers, each of 32-bit, are available. Details about both blocks will be given in Sec.IV.

The *Descriptor Queues* block controls three FIFOs, containing the memory buffer addresses in RAM where packets are stored when using the *slow path*, described in detail in Sec.VII.

Incoming packets are managed by the *Incoming Packet Management* block. This block receives the packets from the Ethernet core, buffers them, eventually discarding them if the FIFOs are congested, and performs routing and classification exploiting a VOQ (Virtual Output Queuing) buffering architecture. 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 the packets to the Ethernet core.

IV. HERO Configuration Section

The HERO RF (Register File) block is used by the driver to write and read configuration data. The registers can be 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, i.e., enable Ethernet auto-negotiation, configure the MAC address or verify the connection status.

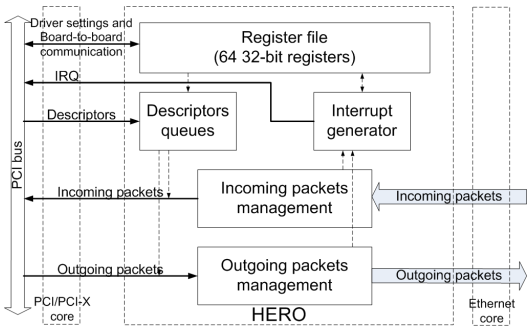

Fig. 2. HERO structure

The *Interrupt Generator* block manages three IRQ registers to signal to the operating system important asynchronous events such as a packet arrival or departure. A 32-bit register ($R_{unmasked}$) maps 32 possible interrupt events, although currently not all bits are used. Whenever one of the events occurs, the corresponding bit in $R_{unmasked}$ is raised. Another 32-bit register (R_{mask}) is set by the driver to select which events are allowed to generate an IRQ. Recall that a unique IRQ channel is assigned to the NIC. Thus, an IRQ is generated every time one of the enabled events occurs, i.e. $R_{unmasked} \oplus R_{mask} > 0$. The result of this 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 at the next IRQ event.

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 IRQ trashing phenomenon. 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, NAPI operating mode, based on the polling idea, has been devised. When adopting NAPI, the IRQ signal is used only to add the device to a polling list disabling its IRQs, until the device is active in providing data. When no more data are available, the device is removed from the polling list and IRQs are re-enabled. Therefore, when the data transfer rate is low, the device driver will mainly work on IRQs, while during high loads periods polling will mostly be used.

The routing and classification registers are used to route and classify packets into two priority classes. These registers are managed by the driver and used to address the incoming packets into the VOQ system. Further details about routing and classification operations are given in Sec.V. In the current implementation, these registers are statically configured by the driver, not taking into account the dynamic evolution of the system.

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.VIII.

Finally, the descriptor registers are used to make packet descriptors available to the NIC. Sect.VII, describing the *slow path* operation, provides a detailed explanation on the type and functions of descriptor registers.

V. 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.

A. Parallelization, classification and routing

The interface with the Ethernet core is composed by an 8-bit data bus plus some control signals. Three control signals are defined: a *data valid* signal, high when a byte on the data bus is valid, a *start* and a *stop* signal, asserted for one clock cycle respectively during the first and the last byte of the packet. HERO is designed to always be 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, 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. Routing is based on the destination IP address (32 bits), classification 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)

We sized the system to support up to 4 custom NICs in a single PC; therefore, up to three routes and four classification rules can be provided by the driver. Everything 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, including the OS. If a packet matches a classification rule, the packet is considered as high priority.

Routing and classification are based on two ternary masks. The masks were implemented using two separate registers. The first one contains the pattern P to be matched. The second one defines a bit mask M, where 1 indicates a "care bit" and a 0 a "don't care" bit. A value V in the packet header is a match if $(V \oplus P) \cdot M = 0$.

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. Simultaneously, the classification compares the header fields with the classification ternary masks, obtaining one result for each possible destination. The outcome of the routing phase selects then the final result among the 4 classification results.

By using special values for M and P, it is possible to deactivate routing and classification functions. When disabling

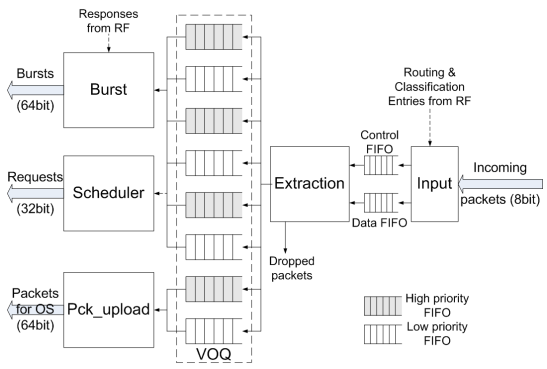

Fig. 3. Incoming packet management block structure

both functions, the board becomes a standard NIC, supporting a single priority and a single FIFO queue storing packets addressed to the OS.

B. 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 of the VOQ array according to the result of the routing-classification process. If the destination queue does not have enough space for the entire 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 "anding" 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 is not optimal, because the Ethernet core would drop also packets addressed to non-empty queues. By using the intermediate FIFO, the packet is received, processed and then potentially dropped, but 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-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 payed. 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.

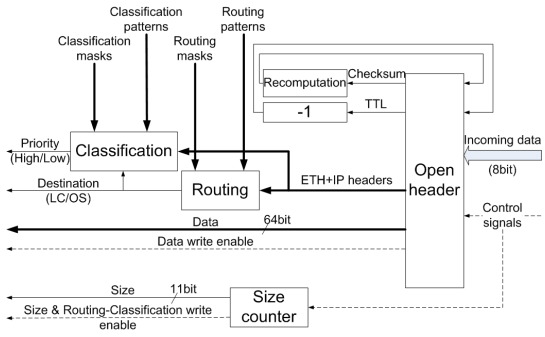

Fig. 4. Input block details: routing and classification blocks structure

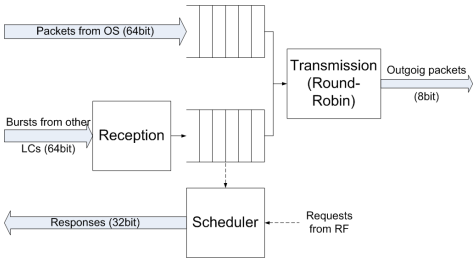

Fig. 5. *Outgoing packets management* structure

C. VOQ enqueuing

The *VOQ* block is an array of 4 pairs of FIFOs, one pair for each possible destination. 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. The data FIFOs size is 8Kbyte.

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*. Further details are provided in the relative sections.

VI. 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 Round-Robin policy is followed, extracting alternatively one packet from each queue. The size of the FIFO that collects packets from the *fast path* is set to 2Kbyte. The *Transmission* block serializes the 64-bit words coming from the PCI bus in groups 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 via the fast-path in Sec.VIII.

VII. 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 and packet size. These information are passed to the NIC to exploit its bus master capability to initiate DMA transfers. When a packet is received, it is immediately written in a free buffer by the NIC. When the packet transmission ends, the event is signaled rising an IRQ signal; the driver passes the newly arrived packet to the kernel. Similarly, when the driver receives a packet from the kernel, it passes the descriptor to the NIC, which will read the packet from the central memory to send it on the wire. When packet reception is completed, an IRQ signal is generated to free the packet memory.

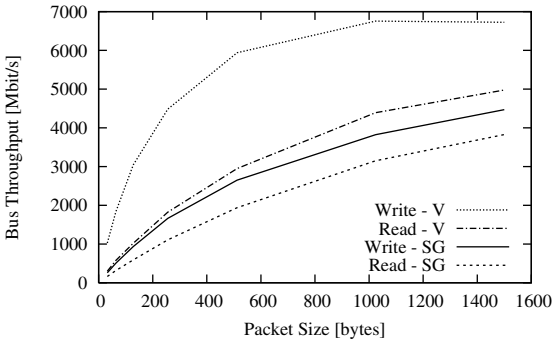

Fig. 6. Performance comparison between scatter-gather (SG) and vector (V) mode (PCI-X @ 133 MHz)

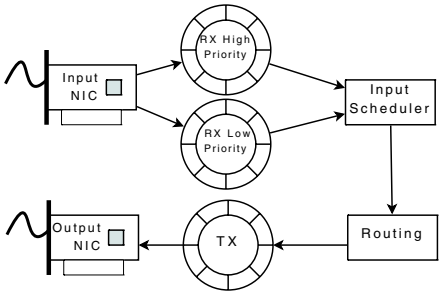

Fig. 7. Description of the slow path.

To increase performance, modern NICs batch these two operations and organize the descriptors into circular buffers called rings. These rings can be implemented in two different ways, which give different performance. We refer to these two possibilities as *scatter-gather* mode 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 NIC can read them all at once, optimizing bus usage.

Fig.6 compares the two modes in terms of achieved throughput. The *vector* mode shows better performance both when reading and writing. This is due to the fact that *scatter-gather* mode needs several small bus transactions to download the descriptors, and each transaction has a high bus-management overhead. The *descriptors queues* in Fig.2 are the FIFOs where the descriptors are stored and are organized according to the *vector* mode.

To build the circular buffer, two vectors are needed. During packet reception, the driver allocates new packet buffers into a second descriptors vector. When the NIC has used all its descriptors, the driver swaps the two vectors providing new descriptors to the NIC. Also in the transmission side two vectors are used, the first one by the driver to group outgoing packets, while the NIC is transmitting the packets stored in the second one.

To support two priorities, a second incoming buffer ring has been added. A scheduler within the driver serves both rings. Therefore, 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. Low priority packets may suffer starvation. Fig.7 summarizes the slow path management.

VIII. Fast Path

The *fast path* provides a high bandwidth-low latency data transfer among NICs.

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 idea to transfer data to a congested output NIC only to discard the packets afterwards. It would be better to discard these packets directly at the input NIC. A congestion can occur if two or more inputs send traffic to the same output with an aggregate throughput higher than the network line rate. In this case, a back-pressure mechanism has to shape the sending rate of the inputs. Also, fairness must be ensured: any input NIC should be able to forward packets to any output NIC without any starvation.

Taking into consideration that the PCI-X protocol schedules bus access with a proprietary policy, we propose a protocol able to organize both access requests and data transfers to meet the above requisites. One possible approach could be to execute the control at the software level, where the driver inquires the NICs state and computes the optimal scheduling. The high latency and low bandwidth imposed by this approach when the driver accesses the NIC registers discourages this approach. Instead, we developed a hardware-based inter-NIC communication protocol.

This protocol is inspired by the three-way schedulers proposed in slotted IQ switches, like iSLIP [6], but it is completely asynchronous. 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 control communication protocol based on three steps:

- *Request*: an input NIC communicates to the output NIC how many data are available for that output;
- *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 an amount of data equal or less than the granted amount received during the *response* phase.

The structure of the protocol is due, in part, to some limitations related to bus access procedures. In our architecture, a board acting as a PCI-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.

A *request* message is a 32-bit word divided into two 16-bit section, 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 queues addressed to an output are empty, any communications toward this output ends. As soon as a 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. The *response* is organized as a two 16-bit section in a 32-bit message, one for each priority level. In our policy, all free space within the FIFO, if needed, is allocated to the current data exchange. If the memory available is not able to satisfy the request, the free space is first allocated to the high priority field; the remaining space, if any, is devoted to the low priority field.

When a NIC receives a *response* message, the data transfer can take place. The packets are grouped into a *burst*, composed by several high priority packets followed by the low priority ones. For each priority level, the number of packets 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*; 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*. If no more packets are available for that output NIC at the end of the burst transmission, the request bits are set to zero. The piggy-back technique keeps the communication alive and, during high load periods, saves a bus transaction. Similarly to the vector mode described in Sec.VII, saving a transaction and grouping packet transfers together improves bus performance. This also motivates the introduction of packet transmissions in *burst*s.

The generation of *request* messages is performed by monitoring the VOQs occupancy status. In Fig.3, the block generating *request* messages is the *Scheduler*. The same physical block, as shown in Fig.5, is also involved in the management of the requests and in the monitoring of the fast path FIFO to generate the *response* messages at the receiving side.

The *Burst* block in Fig.3 is in charge of generating the *burst*, 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 stored into the FIFO.

Conflicts may occur in high load conditions: two input NICs may have data available for the same output NIC. Since it is not possible to receive simultaneously two *burst*s, they are serialized. In particular, the first *request*s are sent by the inputs as soon as there are data. These 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* is sent back. Then, the NIC waits for the *burst*. Only when the *burst* is completely received the next *request*s are processed. This mechanism allows each NIC to wait at most for one *burst* at a time, simplifying both resource allocation and communication protocol. If the input NIC, upon receiving a *response* message is not able to fit at least one packet into the *burst*, it sends an empty burst containing only the *request* to keep alive the communication channel.

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

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

IX. Conclusions and Future Work

In this paper we presented the implementation choices and the design logic of a custom NIC, able to implement extra features with the aim of reducing the CPU load and improving both QoS and bus throughput. This work is made available as an open source IP core, with two interfaces toward a PCI-X and an Ethernet IP cores, to obtain a portable NIC implementation.

This core is fully compatible with the Linux OS and exploits very flexible interfaces toward both the bus (i.e. PCI core) and the network (i.e. Ethernet core). It is therefore useful not only in software routers performance studies, but also in other applications needing a non-standard network card. Such applications could be, but are not limited to, security, network measurements, monitoring and analysis, development and testing of new MAC protocols for new generation optical or wireless packet networks.

The design is far from being complete. Among the possible improvements, we emphasize:

- increase the supported number of routes and priority classes
- improve the driver internal scheduler
- management of dynamic route updates to maximize the number of packets going through the *fast path*.

X. Acknowledgment

This work was performed in the framework of the MIUR project BORA-BORA [5], and developed in the high-quality lab LIPAR at the Politecnico di Torino.

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," in *HPSR 2006 (IEEE Workshop on High Performance Switching and Routing)*, Poznan, Poland, June 7-9 2006.

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

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

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

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

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