

Scalable Intelligent Video Server System

<i>Title</i>	FORTH Switch Feasibility and Research Contribution Report
<i>Revision</i>	B
<i>Deliverable #</i>	5.4
<i>Author</i>	M. Katevenis, with contributions by many others
<i>Company</i>	FORTH
<i>Date</i>	2006-02 (February 2006)
<i>Filename</i>	D54_forthSwitch_revB.doc
<i>Dissemination²</i>	CO

REVISION	DATE	DESCRIPTION
A	2005-12	Created by M. Katevenis (1 st draft)
B	2006-02	First normal Release

² **CO** = Confidential (only for members of the consortium + EC); **RE** = Restricted to a stated circulation list (+ EC)
 [replace this footnote with the list]; **PP** = Restricted to other FP6 participants (+ EC); **PU** = Public

TABLE OF CONTENTS

0. EXECUTIVE SUMMARY	3
1. DOCUMENT OVERVIEW.....	3
2. PERFORMANCE CONTRIBUTION: QOS GUARANTEES	3
3. NEW RESEARCH CONTRIBUTION: REDUCING THE SRAM COST OF BUFFERED CROSSBARS OPERATING ON VARIABLE SIZE PACKETS.....	4
4. HARDWARE PROTOTYPES.....	6
4.1 SYSTEM OVERVIEW	6
4.2 BUFFERED CROSSBAR DESIGN AND IMPLEMENTATION.....	9
4.2.1 Buffered Crossbar Organization.....	9
4.2.2 Crosspoint Buffers.....	11
4.2.3 Output Scheduler (OS).....	12
4.2.4 Credit Scheduler (CS).....	12
4.2.5 RocketIO Network Interface.....	12
4.2.6 Tests and Verification in the System Platform.....	13
4.3 NIC DESIGN AND IMPLEMENTATION	15
4.3.1 PCI-X DMA Engine Module.....	15
4.3.2 RocketIO Network Interface Module.....	32
5. CONCLUSIONS	39
6. REFERENCES	39
7. APPENDIX: RESEARCH CONTRIBUTION	40

LIST OF FIGURES

Fig. 1 System Overview	7
Fig. 2 Photograph of the system	8
Fig. 3 Buffered Crossbar Internal Architecture	11
Fig. 4 Photograph of the switch.....	13
Fig. 5 PCI-X DMA Engine Block Diagram.....	15
Fig. 6 The DMA Request Queue.....	23
Fig. 7 Network Interface Module Block Diagram	32
Fig. 8 Host to Network FSM.....	34
Fig. 9 Network to Host FSM.....	35
Fig. 10 The packet format at various interfaces.....	37
Fig. 11 The Packet Header Format.....	37
Fig. 12 The Credit Format.....	38

0. EXECUTIVE SUMMARY

In this Workpackage 5, “Deliver High Performance IOSS for Integration with SIVSS Platform”, FORTH contributed as follows:

1. Prior to the switch change of plans, in the first part of year 2 of the project, FORTH completed the careful study of the TeraChannel Switch scheduling mechanisms and how these are to be used to provide bandwidth and delay guarantees in the presence of other, (low priority) congested traffic, including the use of confidence intervals (as requested by the reviewers). The results of these studies are summarized in section 2 below, and were incorporated into D4.4, which provided the appropriate context for them, yielding revision D of that document (March 2005) (including confidence intervals plus many new simulations), and then revision E which included all further studies beyond March 2005. Please refer to D4.4 revision E for that topic.
2. FORTH developed and successfully tested extensive hardware prototypes on its recently built FPGA-based platform (see section 4.1):
 - We implemented and successfully tested, thus validating, the *buffered crossbar* architecture proposed in WP4 as a possible modification for the next-generation switch: see section 4.2. Buffered crossbars offer very significant advantages relative to the current unbuffered ones, like the TeraChannel, thus being appropriate for the next generation switch. Buffered crossbar switches, communicating with each other via the RECN protocol will provide efficient, scalable, and low-cost switching fabrics.
 - We implemented the network end-point (network interface card – NIC) functionality needed for the software prototyping and performance analysis work of WP9. Besides supporting remote DMA (RDMA) and remote notifications (interrupt-based or flag-based), this NIC also includes a large set of performance and debug monitors and counters which were essential for WP9: see section 4.3.
3. On the research front, regarding the roadmap beyond the initial SIVSS platform, FORTH's research was targeted to the optimization of buffered crossbars, which operate on variable size packets, with respect to minimizing the on-chip SRAM buffer size. The novel scheduling methods proposed are briefly reviewed in section 3, and are described in a paper that is appended to this deliverable.

1. DOCUMENT OVERVIEW

Section 2 reviews FORTH's contribution towards the performance of the TeraChannel (TC) switch. Section 3 describes our new research effort on optimizing the buffered crossbar architecture by reducing its SRAM buffer size. Section 4 presents details on the hardware prototype platform and its use to (a) validate the buffered crossbar architecture (section 4.2), and (b) provide the network endpoint functionality needed for the software prototyping and measurements of WP9 (section 4.3).

2. PERFORMANCE CONTRIBUTION: QOS GUARANTEES

The TeraChannel (TC) Switch provides high throughput and includes powerful mechanisms that, when appropriately used, will provide quality of service (QoS) guarantees to the applications running on the system. Such QoS guarantees are essential for the desired operation of the system, and

especially for ensuring the real-time properties of the video streams.

The contributions made by FORTH were presented in D4.4 and are summarized here for the purpose of completeness:

1. in deliverable *D4.4 - revision C* (Feb. 2005), we described our first set of experiments demonstrating that the bandwidth allocated to the competing flows, which all try to get as much throughput as possible, is within 2% accuracy of the ideal value predicted by the ratio of their (programmable) weight factors. We also figured-out that a high-priority flow sees a *mean queueing delay of less than 1 or 8 μs*, dependent on the traffic pattern, when its throughput is 5% or more below its “fair share” of bandwidth (as determined by its programmable weight factor), even though the link through which it passes is saturated with interfering traffic from several other flows to the point where the aggregate throughput of that saturated link is 99% of its peak capacity.
2. in deliverable *D4.4-revision D* (March 2005), we repeated the same experiments in order to be confident that our measurements are bounded within a known interval: that interval was less than 5% with probability greater than 95%. We also described a new set of experiments showing (a) successful bandwidth allocation to a greater number of competing flows and (b) distribution of excess bandwidth to flows in proportion to their weights.
3. in deliverable *D4.4-revision E*, we reported experiments under realistic, video-server-oriented traffic, showing again successful bandwidth allocation.

3. NEW RESEARCH CONTRIBUTION: REDUCING THE SRAM COST OF BUFFERED CROSSBARS OPERATING ON VARIABLE SIZE PACKETS

The TC employees a traditional, *unbuffered* crossbar architecture, like the majority of current high-end commercial switches, i.e. all payload buffer space is contained in the port (TP) devices and the crossbar (TX) devices do not contain any such buffer space.

An alternative crossbar architecture, known as “*buffered* crossbar” or “Combined Input-Crosspoint Queueing (CICQ)”, is to include (relatively) small buffers at each crosspoint within the crossbar. This leads to important gains, provided the amount of buffer space is sufficient. The reason why buffered crossbars were not popular in industry, in the past, is that the required amount of buffer memory inside the crossbar, for large switches (large port counts), was infeasible in past technologies (0.25-micron CMOS or wider), while they were believed to be barely feasible in current technologies (0.18- or 0.13-micron CMOS). With FORTH’s present research contribution, this required amount of buffer memory is *further reduced*, so that now it is clearly *feasible and economical*.

Within the research community, there has been some work on buffered crossbars in the 90’s, most notably [1]. In recent years, there have been more research publications on this topic, plus one notable industrial study by IBM Zurich Research Lab [2]; this latter publication concerned a design that was not transferred to production (possibly because of its high cost), however it showed the advantages of buffered crossbars. FORTH has contributed in the past on buffered crossbar research, being one of the two initial (1987) proposers of the architecture; in recent years, FORTH has studied the implementation of weighted round robin (WRR) in buffered crossbars [3], showing that its implementation is easier than in unbuffered crossbars.

All these background studies had concluded that buffered crossbars have significant advantages over the previous, traditional bufferless configuration, mostly because of the simplicity and efficiency of

the scheduling operation (point 1 below), and support for multicast (point 2). However, it recently became apparent that buffered crossbars have additional advantages, related to variable-size packets—points 3, 4, and 5 below:

1. The scheduling task is dramatically simplified; WFQ-type QoS is easier to implement; there are no scheduler inefficiencies to be compensated by speedup.
2. Multicast can be supported and scheduled in a natural, straightforward way.
3. The crossbar can operate directly on variable-size packets, hence there is no need for segmentation and reassembly circuits.
4. Internal speedup is not needed, because there is no packet segmentation and no scheduler inefficiencies; hence, the external line rate can be as high as the crossbar line rate.
5. The egress path of the switch needs no buffer memory—at least no large, off-chip memory—because packet reassembly is not needed, and because, in the lack of internal speedup, there is no output queue build up; this eliminates a major cost component.

Following these observations, it became apparent that buffered crossbars are a major candidate for the roadmap beyond the initial SIVSS platform, hence research was needed within SIVSS to determine several of their aspects that were yet unclear. Our research contributions to this end have been described in three publications – [5][6][7] - presented at International IEEE Conferences, and they were appended to deliverable D4.4, revision C (year-1 deliverable).

Our new research contribution, within SIVSS year-2, in WP5, concerns the optimization of buffered crossbars which operate on variable size packets with respect to the crosspoint buffer size requirements. Buffered crossbars directly operating on variable size packets require at least one maximum-packet worth buffer per crosspoint. This results to an on-chip SRAM memory requirement which under some circumstances, for example when very large packets are to be switched, may not be possible to satisfy. For this reason, in [6] we described a method to segment packets in order to reduce the size of the crosspoint buffer needed, while inducing no padding byte overheads; we also connected our scheme to the ingress line card memory management operations. Although the segmentation method described in [6] was shown to be very effective, it has two undesirable features: (a) a reassembly buffer is needed at the egress path of the switch and (b) a packet reassembly delay is induced.

In our new research effort, in WP5, we apply packet mode scheduling to buffered crossbars in order to remedy the packet segmentation shortcomings mentioned above. Packet mode scheduling had only been studied for bufferless input-queued switches: when the central switch scheduler establishes an input-output port pairing, the pairing is maintained until all cells of the packet are forwarded. The extension to buffered crossbars is not trivial because the scheduling process does not necessarily determine such pairings. We introduce two schemes for buffered crossbars: probabilistic and deterministic packet mode scheduling. The probabilistic case assumes independent crossbar output schedulers, but requires reassembly buffers. Using simulation we show that it reduces packet delay compared to the system with pure segmentation and reassembly; for a representative traffic pattern the average packet delay is improved by more than 80% for loads up to 50%. Deterministic packet mode scheduling sacrifices some scheduler independence in order to eliminate reassembly buffers; based on our simulations, it performs very close to buffered crossbars without segmentation (which use very large crosspoint buffers), and it also performs always better than packet mode scheduling in bufferless crossbars. Deterministic packet mode scheduling yields very good performance even though the *crosspoint buffer size is determined by its scheduler round-trip time*, and not by the size of the

packets that it handles, thus holding the promise quite small buffer sizes will suffice for good performance!

FORTH's new research contribution has been submitted for publication; the version of January 2006 is appended to the present deliverable.

4. HARDWARE PROTOTYPES

In this section we present the hardware prototypes and the system that we built in order to (a) prove the feasibility of the Buffered Crossbar switch and (b) provide network interface cards (NIC) for the WP9 performance analysis work. The prototypes are developed on high performance FPGAs and offer numerous high speed interfaces that are used to create a rich and complex environment to host network services.

4.1 System Overview

Our system consists of NIC cards which are attached to host machines and the Buffered Crossbar Switch which provides a fast and efficient interconnect between the NICs of different machines. The switch allows the machines of the system to communicate with each other and benefit from the services offered by other machines. The NICs use the remote DMA (RDMA) technique to achieve the fastest possible transfer of large volumes of data and communicate with each other across the switch. The NICs also have extensive monitoring and debugging capabilities in order to measure the performance and export the state of the system. Fig. 1 illustrates an overview of the system and a real photo of the system is shown in Fig. 2.

The switch is a Buffered Crossbar switching fabric developed on a Xilinx Virtex II Pro FPGA with tens of RocketIOs and offers the connectivity between the NICs. The switch regulates the traffic and handles congestion with a backpressure flow control protocol between the switch and the NICs.

Each NIC card is developed on a Xilinx Virtex II Pro FPGA and is attached to the PCI-X bus of a host machine (Intel Xeon). The NIC card is also attached to the switch via the RocketIO network interface which provides the connectivity with the rest of the system. The PCI-X is the interface of the NIC with the host processor that runs the actual applications. This interface is used to initiate outgoing packets to the network and deliver incoming packets to the host memory. The RocketIO is a fast network interface and allows large data volumes to be exchanged with the other nodes of the network.

The system is used and tested with user initiated traffic but beyond this we have used a remote storage framework to allow more realistic traffic patterns. In this storage environment one of the machines hosts storage services and transparently shares a large volume of disk space among the other machines that belong to the network.

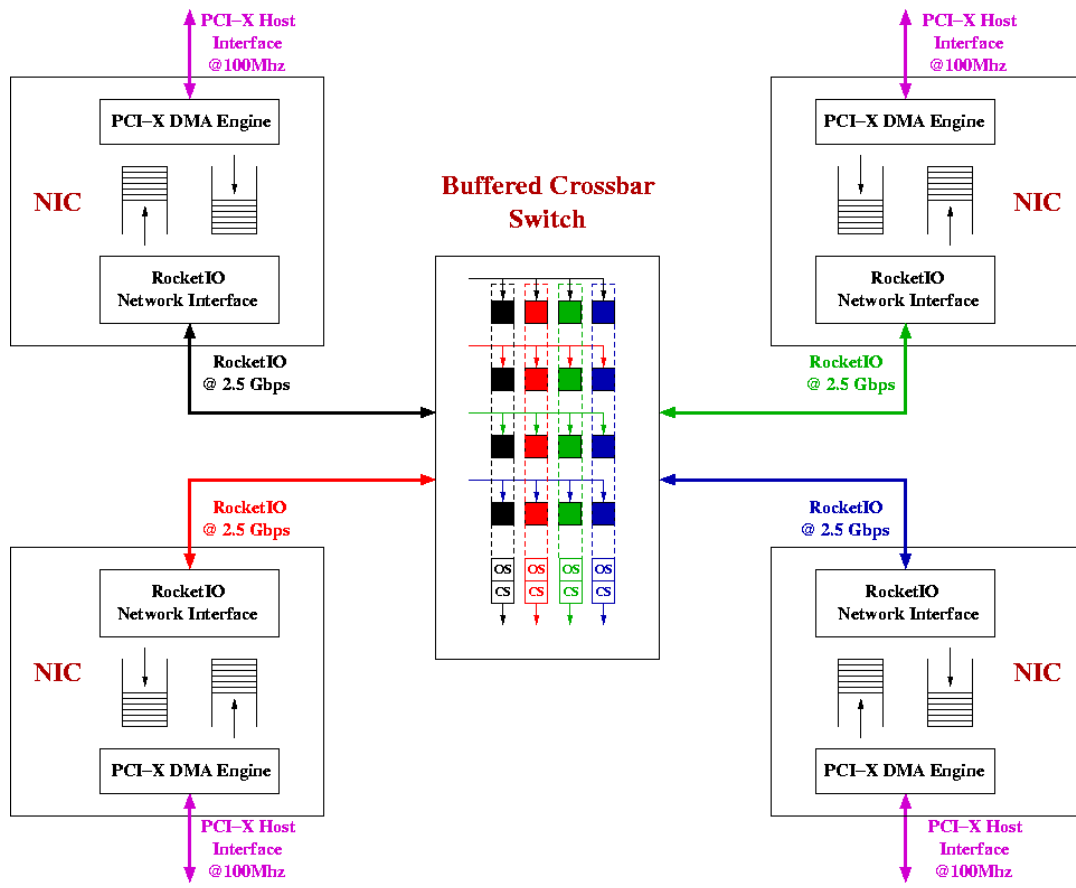


Fig. 1 System Overview

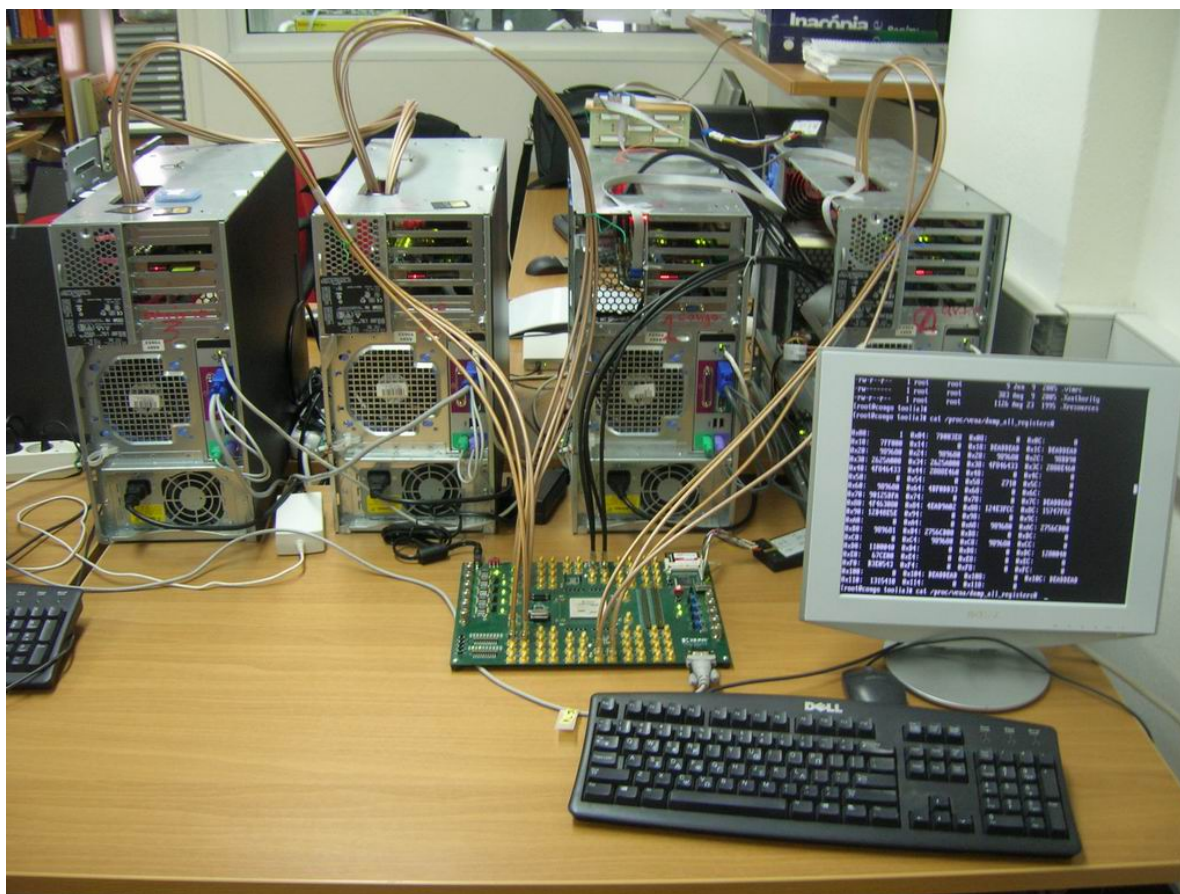


Fig. 2 Photograph of the system

Buffered Crossbar Switch

The prototype of the switch is developed in a board made by Xilinx, namely the ML325 board [9], and features a Virtex II Pro FPGA with 2 embedded PowerPCs, an SDRAM and has 20 Rocket IO SMA interfaces and therefore is an ideal solution for a switch.

The switch that provides the connectivity in the system is a Buffered Crossbar and therefore the CICQ Architecture is implemented and RocketIOs serve as network interfaces. It is an 8x8 switch where only 4 ports are used until now. The switch prototype has 64 crosspoint buffers where each of them is 2Kbytes. It has a single priority and has of course inherent ability to switch variable size packets while no segmentation and reassembly is needed. The operating frequency is 78.125MHz required by the RocketIOs and combined with the width of the internal paths is sufficient to fully utilize the network bandwidth. The switch implements a round robin scheduling policy with an output scheduler (OS) per output port and supports cut-through operation. The flow control is credit based (QFC-style) and is assigned to the credit schedulers (CS) that exist per input; these credits are interleaved between the packets.

A more detailed description of the Buffered Crossbar Switch is provided in section 4.2.

NIC Cards

The prototypes of the NIC cards are developed on boards made by DiniGroup, namely the DN6000K10SC board [10], and feature a Xilinx Virtex II Pro FPGAs. Each development board has a rich collection of peripheral components such as SDRAMs, SRAMS and Flash memory while the

Virtex II Pro device has 2 embedded PowerPC processors and 4Mbits of on-chip memory. Moreover the board has 4 RocketIO SMB interfaces and a 64-bit PCI-X interface.

The NIC design implements a 64-bit PCI-X interface which operates at 100MHz and can provide a maximum theoretical throughput of 760 Mbytes/sec to the host memory. The calculation of this throughput assumes a datum transferred in every cycle but actually there is the PCI-X protocol overhead. The DMA engine of the NIC has a fully functional 64-bit PCI-X Initiator with DMA capabilities and thus it can directly read or write to the host memory. The PCI-X Target interface of the NIC is also 64-bits and is used to accept commands from the host processor or deliver status and performance information.

Commands to the NIC from the host processor reach the PCI-X target interface and are placed in a memory mapped region which is called DMA Request Queue (section 4.3.1.2). The request queue allows a maximum of 1024 pending operations with a maximum of 4Kbytes per operation (i.e. one operating systems page). The request queue also allows the user of the NIC to cluster operations and offers on-demand processing of every single operation. Another important feature of the request queue is the local notification which informs the processor about the completion of an operation. When the operation is completed, the NIC writes a value to a pre-specified memory address. The processor can then poll that memory address for the completion of a specific operation. Moreover the NIC offers the remote notification feature which can inform the processor for an incoming packet; this notification comes in the form of a level-triggered interrupt.

The NIC also implements a RocketIO network interface which is a high speed network interconnect offering 2.5Gbps of full-duplex network traffic. The RocketIO MultiGigabit Transceivers are offered by the Virtex II Pro devices and are used for the network transport. A custom network protocol and a credit based flow control is used to take advantage of the 300 Mbytes/sec provided by the network.

The NIC has two 8Kbyte network FIFOs associated with the incoming and outgoing network traffic. The outgoing FIFO is used as an elastic buffer but also as a classic input queue regardless of the destination – a virtual output queue implementation for every network destination is being implemented and we are ‘almost’ ready to use it and replace single outgoing FIFO. The incoming FIFO is also used as an elastic buffer but also covers the temporal uncertainties of the PCI-X DMA accesses. The credit based flow control mechanism is QFC-style and keeps state about every network destination (crosspoint buffers in the switch). Credits and data share the same links and the credits are interleaved between the packets.

A more detailed description of the NIC is provided in section 4.3.

4.2 Buffered Crossbar Design and Implementation

In this section we present the organization and the building blocks of the buffered crossbar switch regarding the prototype implementation in a high performance FPGA. This implementation is based on the research results presented in [5] and the initial design [8] that proved the feasibility of a 32x32 buffered crossbar in modern ASIC technology. The current report focuses on porting the latter design in a high end FPGA environment which can support millions of transistor logic, several Mbits of on-chip memory and many multi-gigabit network links.

4.2.1 Buffered Crossbar Organization

The buffered crossbar switch is implemented in a development board made by Xilinx. We use the ML325 Characterization Board [9] with features a Virtex II PRO XC2VP70K device and has 20

RocketIO SMA interfaces on the board. The FPGA device has 74K logic cells and 6Mbits of on-chip memory in 328 embedded Block RAMs (BRAMs) of 18Kbits each. Moreover, its 20 RocketIO hard-blocks operate at 156.25 Mhz and each one provides 2.5Gbps of full duplex network throughput. The internal interfaces of the RocketIOs operate at 78.125Mhz on a 32-bit datapath, hence this fact sets the frequency limit of the crossbar. Since the RocketIOs are placed in both the top and bottom side of the FPGA, the manufacturer demands two clocks deriving from different crystals and therefore we consider two clock domains for the switch. Those features of the board enable us to develop a prototype of the buffered crossbar with decent size, performance and real life characteristics.

The architecture of the crossbar is based on the work of [5] and [8] where all the related details are discussed. We have implemented an 8x8 buffered crossbar, with 32-bit datapath, on the FPGA whereas we managed to achieve a 10x10 configuration utilizing 65% of the FPGA logic. This configuration seems to be the practical limit for the current FPGA device since there are too many wires to be routed.

For the 8x8 configuration we have 64 crosspoint buffers with 2Kbytes and 8 RocketIOs for the network interfaces. In our custom network protocol the packet headers/trailers are 16 bytes as described in section 4.3.2.4 and the minimum payload size is 8 bytes (one 64-bit word of NIC's PCI-X). Consequently the minimum size of a packet is 24bytes which is translated in 6 clock cycles in our 32-bit datapath. We also defined the maximum payload size to 496 bytes hence the maximum packet size is 512 bytes (128 cycles). This maximum size is an effect of the size of each crosspoint buffer and the round trip time (RTT) of the network (section 4.3.2.2), as explained in [5].

The building blocks of the switch are:

- the crosspoint buffers and associated logic
- the output schedulers (OS) per output
- the credit schedulers (CS) per input
- the RocketIO interfaces per link which are described in section 4.3.2

The block diagram of a 4x4 buffered crossbar is shown in Fig. 3 and a real photo in Fig. 4 .

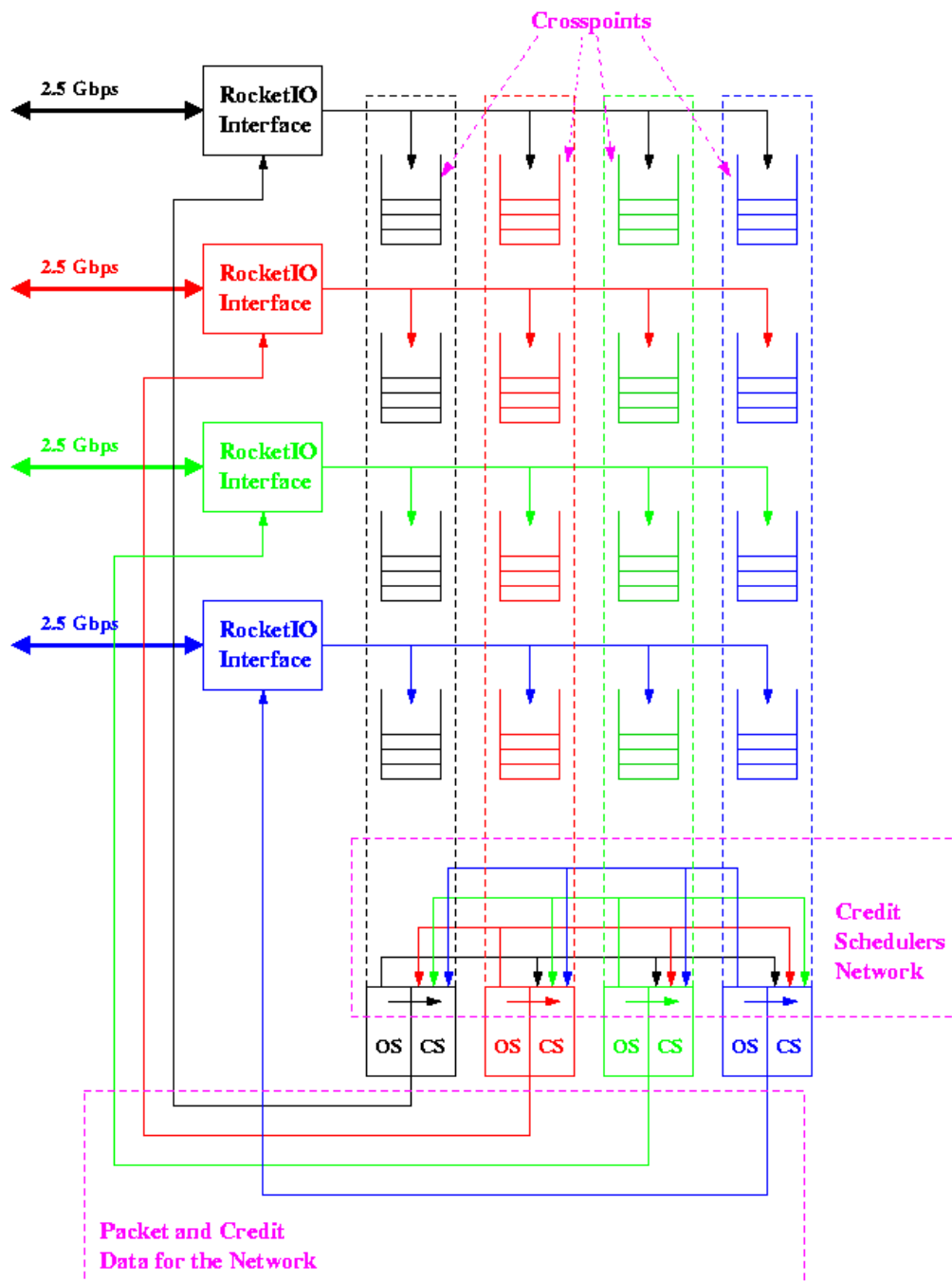


Fig. 3 Buffered Crossbar Internal Architecture

4.2.2 Crosspoint Buffers

Each crosspoint buffer is a 2Kbyte dual ported show-ahead FIFO in a 512x32 configuration and utilizes 1 BRAM for each buffer. The logic of the crosspoint receives the packets and the associated flags (start of packet and end of packet) from the network interface and enqueues them to the memory. When a new packet starts entering the crosspoint buffer then the logic informs the output scheduler (OS) of the crossbar column through a synchronizer circuit; remember that there are two clock domains. Additionally, the crosspoint buffer is able to dequeue a 32-bit word in every cycle when the OS requests it.

4.2.3 Output Scheduler (OS)

The output scheduler (OS) is responsible to select an eligible flow (crosspoint buffer) from the column and forward the existing packet(s) to the network interface. It also generates the corresponding credit (the size of the outgoing packet) and sends that to the associated credit scheduler (CS) of the input.

For every crosspoint of the column the OS keeps the number of enqueued packets for each buffer in counters and based on these it generates an eligibility mask. When there are eligible flows in the column then the OS selects one of them by applying plain round robin policy and informs the network interface. The round robin policy is implemented with a priority enforcer which keeps state of the last served flow and it requires a single clock cycle for the decision. The crosspoint logic needs 3 clock cycles due to synchronization to inform the OS for the packet and then the OS applies the scheduling policy and informs the network interface. This sequence of events requires only 5 clock cycles and therefore the OS can support cut-through operation even for minimum sized packets (24 bytes = 6 clock cycles). In order for OS to support back to back transmission to the network and to hide the scheduling latency it performs pre-scheduling. While a packet is transmitted and there are eligible flows, then the OS selects the next eligible flow according to the policy and informs the network interface 3 cycles before the previous packet finishes.

The packets are dequeued from the network interface when it has credits for the outgoing path while the crosspoint selection of the OS is transparent to the interface. When the transmission starts then the OS informs the associated CS with the size of the packet.

4.2.4 Credit Scheduler (CS)

The credit schedulers (CS) are dedicated per input, follow the QFC-like approach [8] and hold the number of bytes that departed for a specific pair of input and output. Each CS receives from every OS the number of bytes that departed and originated from the associated input. The CS keeps state, in counters, of the accumulated bytes that departed whereas the wrap-around of the counters does not affect the protocol and the correctness of the system as described in [5],[8]. Note also that the OS and CS might work in different clock domains so the credit data are synchronized in 3 clock cycles.

The CS also interacts with the network interface and provides the credit data to be transmitted when requested; one credit at a time. However, CS is responsible to provide credits for every possible destination/output the input sends the packets. Therefore, the CS provides the credits to the network interface in a round robin fashion. The round robin policy is applied initially at the modified credits values and later at the unmodified to compensate for possible credit corruptions or losses.

4.2.5 RocketIO Network Interface

Each RocketIO network interface is responsible to decode the packet headers and enqueue the incoming packets in the appropriate crosspoints of the associated crossbar row. It also keeps state of the incoming credits that concern the incoming buffer of the connected NIC. Besides, the network interface should transmit the outgoing packets and the credits it receives from the associated OS and CS whenever it has credits. Detailed description is provided in Section 4.3.2.

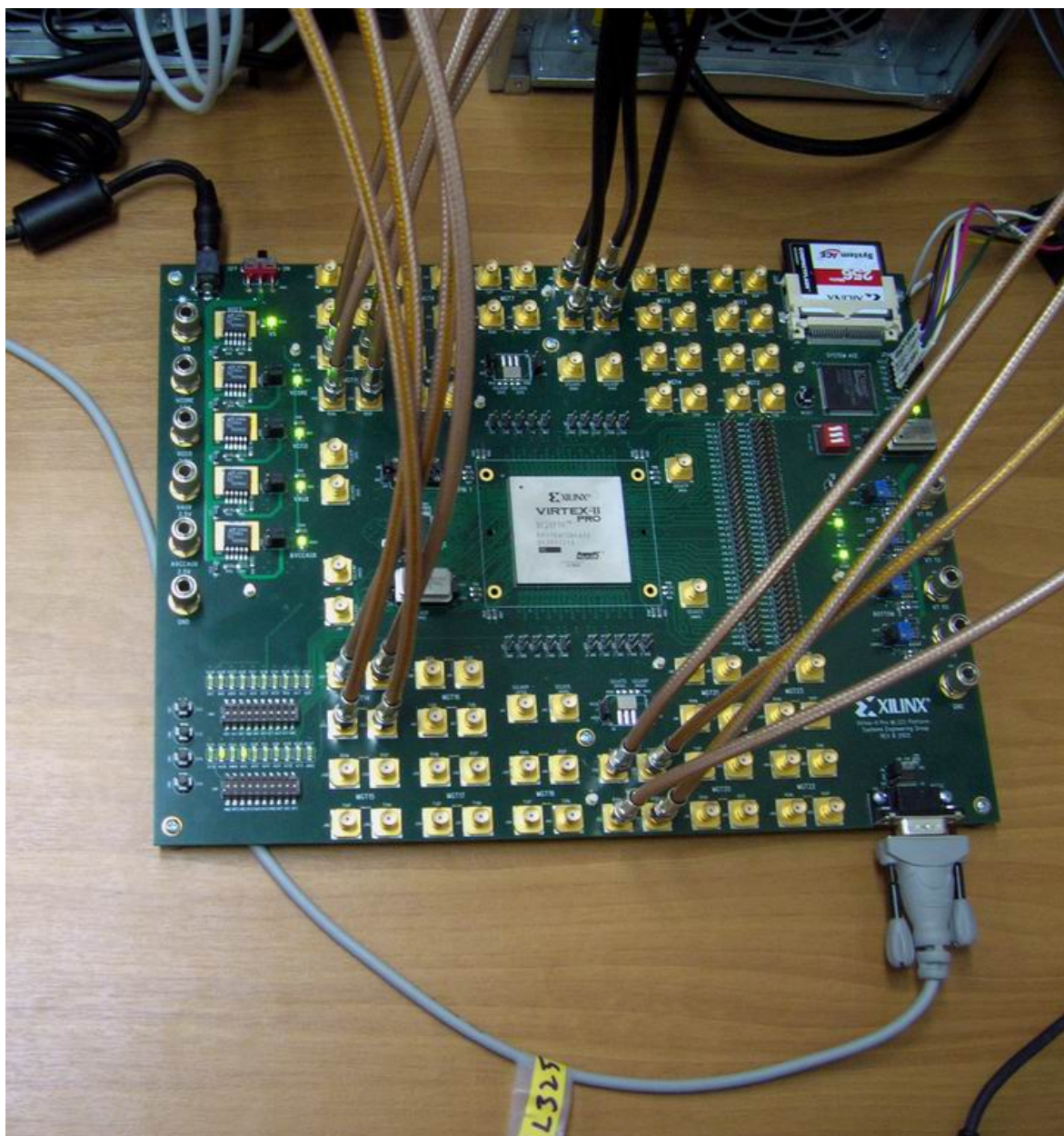


Fig. 4 Photograph of the switch

4.2.6 Tests and Verification in the System Platform

The buffered crossbar switch as shown in section 4.1 is the interconnect between the NICs of different machines and transports the network traffic. The system platform is a rich and complex environment which includes machines with Intel Xeon processors, PCI-X host interfaces with the NICs and RocketIO interconnects with the Buffered Crossbar switch. The test and verification of the switch involves software initiated traffic from the NICs to the crossbar.

For the verification of the switch we developed software modules that initiate network traffic from the NICs to the switch with programmable packet sizes and programmable network destinations. These modules also examine the received and transmitted packets and report CRC error, packet losses or

other system errors. We have run extensive tests on the system to verify the correct operation of the switch. The tests we have run are the following:

1. *Ping-pong test between two NICs through the switch:*

One of the machines is the initiator of the test which transmits a packet and waits the other machine, the target, to respond with the same packet. We have run this test with many different packet sizes in the range of 24bytes – 512bytes. We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests.

2. *1-way test through the switch:*

One of the machines is the initiator of the test which transmits back to back packets to a target machine through the switch without waiting any response. We have run this test with many different packet sizes in the range of 24bytes – 512bytes. We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests. The measured throughput was 287 Mbytes/sec (theoretical max. 300 Mbytes/sec).

3. *1-way test to itself through the switch:*

The initiator of the test transmits back to back packets to itself through the switch without waiting any response. We have run this test with many different packet sizes in the range of 24bytes – 512bytes. We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests. The measured throughput was 270 Mbytes/sec since there is both incoming and outgoing traffic to the NIC (theoretical max. 300 Mbytes/sec).

4. *1-way test with 3 senders and 1 receiver:*

There are 3 initiators that transmit back to back packets to 1 single receiver through the switch without waiting any response. This test stresses out the switch because it generates output contention and activates the flow control mechanisms of the NICs and the switch. We have run this test with many different packet sizes in the range of 24bytes – 512bytes. We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests. The measured throughput was 93 Mbytes/sec on every sender, hence 279 Mbytes/sec on the receiver (theoretical max. 300 Mbytes/sec).

5. *Round robin 1-way test to any network destination*

There are 4 initiators that transmit in round robin fashion to all the network destinations without waiting any response. The round robin policy is applied in a per packet basis. This and all the other tests have fair temporal randomness since the network nodes obtain their packet data through PCI-X DMA accesses which come under the arbitration policy of each host's PCI-X bridge. This test also stresses out the switch because it generates output contention and activates the flow control mechanisms of the NICs and the switch. We have run this test with many different packet sizes in the range of 24bytes – 512bytes. We have made billions of iterations and overnight tests involving billions of packets to ensure the correct operation. Our system has not reported errors; hence the system passed successfully these tests.

The next step is to accurately measure the metrics that indicate the buffered crossbar performance under balanced or unbalanced traffic, with more random network destination and packet sizes.

4.3 NIC Design and Implementation

4.3.1 PCI-X DMA Engine Module

The block diagram of the PCI-X module is shown in Fig. 5.

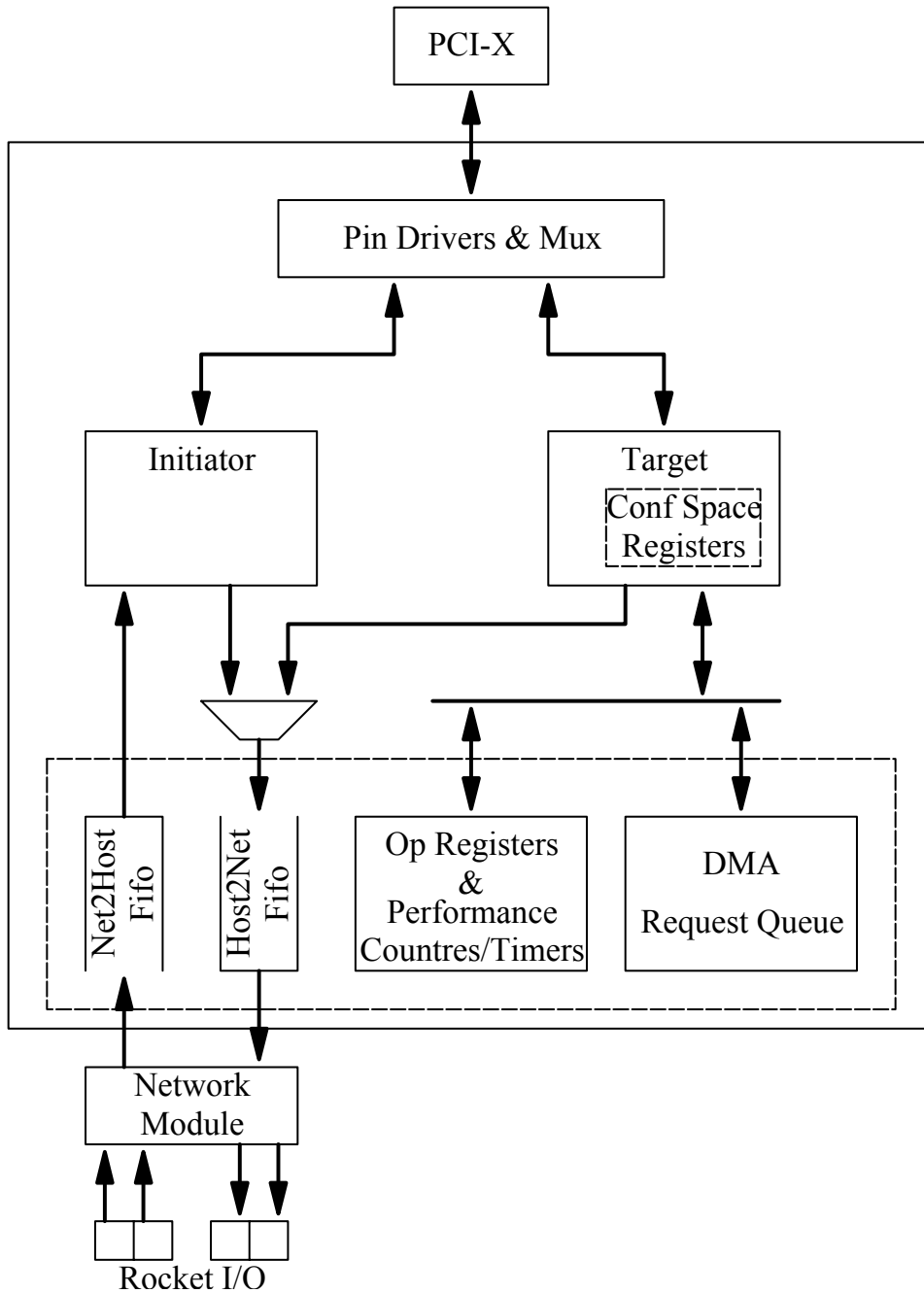


Fig. 5 PCI-X DMA Engine Block Diagram

The **Target** block implements the PCI-X target interface. The same block is used for the configuration space and therefore contains all the necessary configuration registers defined by the PCI-X standard. The DMA request queue, DMA operation registers and the performance counters/timers are updated and read through this target interface. These transactions can be 32 or 64-bits wide in burst or non-burst mode. The target block also supports dual address cycle. Moreover, one outstanding split completion is supported. Each such split completion can be 32 or 64 data bits wide. Finally, the target asserts the interrupt line upon receipt of the appropriate signal from the DMA Fifo block.

The **DMA Initiator** consists of a PCI-X master interface while it also provides the full functionality of a DMA engine from/to the host's memory. It uses physical PCI addresses. It starts the DMA transfers according to their order in the DMA request queue. These PCI-X transactions are multiples of 64-bits (ACK64_ is ignored) and can issue a dual address cycle access if the 32 most significant address bits are non-zero.

The **Operation Registers & Performance Counters/Timers** block contains a control register, registers that store the PCI addresses for the remote and local notifications and several other performance counters. These registers are further explained in section "DMA Operation Registers".

The **Host to Net Fifo** is written by data acquired from the PCI bus, either from a block read transaction performed by the initiator, or by split completion transactions received by the target module. It also provides clock domain synchronization between the PCI and network clock. In the current configuration, the size of this fifo is 1023 words of 64 bits each. The FIFO size is not 1024 because of the design limitation relative to head and tail pointers.

The **Net to Host Fifo** is written by the network module. It also provides clock domain synchronization between the PCI and network clocks. In the current configuration, the size of this fifo is 1023 words of 64 bits each.

The **DMA Request Queue** holds up to 1024 outstanding requests, set by the host (later on they will be set by the on-chip CPU as well).

Finally, the **Network Interface** block enables the communication of the PCI to the network.

4.3.1.1 PCI Configuration Space

All PCI functional devices must employ a block of 64 configuration 32-bit words for the implementation of their configuration registers. The next table illustrates the format of the configuration header region implemented in the NIC.

Address	Name			
0x00	Device ID		Vendor ID	
0x04	Status		Command	
0x08	Class Code			Revision ID
0x0C	BIST	Header Type	Latency Timer	Cache Line Size
0x10	Base Address Register 0			

0x14	Base Address Register 1		
0x18 to 0x30	Not implemented		
0x34	Capabilities List		
0x38	Not implemented		
0x3C	Not Implemented	Interrupt Pin	Interrupt Line
0x40	Application Interrupt Register		
0x44	Not implemented		
0x48	PCI-X Command	Next Capability	PCI-X Capability ID
0x4c	PCI-X Status		

- **Device ID**

This 16-bit value identifies the type of the device. Current value is: 0x17DF

- **Vendor ID**

This 16-bit value identifies the manufacturer of the device. Current value is: 0x1600

- **Status Register**

The status register tracks the status of PCI bus-related events. The next table explains the fields implemented in the NIC:

Status Register		
Bit Location	Description	Action
4:0 (LS)	Reserved	Hardwired to zero.
5	66MHz-Capable	Current value is: 1. The NIC is capable of operating at 66 MHZ.
6	UDF Supported	Current value is: 0. Device does not support UDFs.
7	Fast Back-to-Back Capable	Current value is: 0. Device does not support fast back-to-back transfers.
8	Data Parity Reported	Set by hardware if the master asserts the SERR_ (PCI line) and the parity error response bit in the

		Command Register is set.
10 : 9	Device Select Timing	Current value is: 0x02. The target device's decode speed is slow (decode time C).
11	Signaled Target Abort	Set by target whenever it terminates a transaction with target-abort.
12	Received Target Abort	Set by the master whenever its transaction is terminated by a target-abort from the currently addressed target.
13	Received Master Abort	Set by the master whenever its transaction is terminated due to a master-abort.
14	Signaled System Error	This bit should be set whenever a device generates a system error on the SERR_ PCI line.
15 (MS)	Detected Parity Error	A device should set this bit whenever it detects a parity error.

- **Command Register**

This 16-bit register provides basic control over the device's ability to respond to and/or perform PCI accesses. The next table explains the fields implemented in the NIC:

Command Register		
Bit Location	Description	Action
(MS) 15:10	Reserved	No action.
9	Fast Back-to-Back Enable	Currently hardwired to 0. Thus disabling fast back-to-back transfers (PCI-X specification requirement).
8	System Error Enable	When set the NIC can drive the SERR_ line.
7	Wait Cycle Enable	This bit is hardwired to zero (No stepping supported).
6	Parity Error Response	When set the device can report parity errors by asserting the PERR_ PCI line.
5	VGA Palette Snoop Enable	Currently hardwired to 0. Disable VGA Palette Snoop.

4	Memory Write and Invalidate Enable	The NIC DMA initiator does not support memory write and invalidate operations. This bit is ignored.
3	Special Cycle Recognition	The NIC do not respond to special cycles. This bit is ignored.
2	Master Enable	Enables the DMA initiator when set.
1	Memory Access Enable	When set, the device responds to PCI memory accesses.
0 (LS)	I/O Access Enable	Currently hardwired to 0. Disable I/O access.

- **Class Code**

This is a 24-bit read only register that defines the revision ID. Current value is: 0x020000

- **Revision ID**

This 8-bit value is assigned by the manufacturer to identify the revision number of the device. Current value is: 0x17

- **BIST**

The NIC does not implement built-in self-test so the value is currently hardwired to 0x00

- **Header Type**

Currently hardwired to 0

- **Latency Timer**

The Latency Timer defines the minimum amount of time, in PCI clock cycles that the master can retain ownership of the bus. The default value is 0x20 and the recommended value is 0xFE.

- **Cache Line Size**

This read/write configuration register specifies the system cache line size in 32-bit words. The recommended value is 0x80.

- **Base Address Register 0**

This register is written by the BIOS at boot time and contains the base address used to access the NIC RAM (SRAM and DRAM). Currently 27-bit address space is supported by the NIC.

- **Base Address Register 1**

This register is written by the BIOS at boot time and contains the base address used to access the DMA initiator registers (addresses 0x000 to 0x1FF) and the DMA request queue BRAM (addresses 0x200 to 0x3FF).

- **Interrupt Pin**

This register demonstrates which interrupt pin the device uses. The NIC uses the INTA_ PCI line so the current value is: 0x01

- **Interrupt Line**

The Interrupt Line register is read/write and is used to communicate interrupt line routing information's.

- **Application Interrupt Register**

This register is read/write and contains information about the interrupts generated from the NIC. The next table explains the fields of the register:

Application Interrupt Register Bits		
Bit Location	Description	Action
0 (LS)	Remote Interrupt Enable	If set, an interrupt is generated when a data packet from the network has arrived, provided that the corresponding operation code for this packet had the Interrupt bit set.
3:1	Reserved	Reserved for future expansion
4	Remote Interrupt Flag	Set by hardware when a data packet from the network has arrived, and its operation code had the Interrupt bit set.
7:5	3'b0	These bits are hardwired to zero.
15:8	8'b0	These bits are hardwired to zero.
23:16	Tail Pointer	Value of the request queue tail pointer.
23	1'b0	This bit is hardwired to zero.
30:24	Head Pointer	Value of the request queue head pointer.
31(MS)	1'b0	This bit is hardwired to zero.

- **Capabilities List Register**

The Capabilities Register points to the first item in the list of capabilities. This item is the PCI-X Command Register set therefore the Capabilities Register is hardwired to 0x48

- **PCI-X Capability IC**

This field is hardwired to 0x07 and identifies the Capabilities List as a PCI-X register set

- **PCI-X Next Capability**

This field is hardwired to 0x00 and identifies that the PCI-X register set is the last entry of the Capabilities List

- **PCI-X Command**

This register is read/write. The next table explains the fields of the register

PCI-X Command		
Bit Location	Description	Action
0 (LS)	Data Parity Error Recovery Enable	If set the device will attempt recovery from data parity errors.
1	Enable Relaxed Ordering	If set the device is permitted to set the relaxed ordering bit in the requester attribute phase.
3:2	Maximum Memory Read Byte Count	This register sets the maximum byte count the device is permitted to use when the initiating a sequence with one of the burst memory read commands.
6:4	Maximum Outstanding Split Transactions	This register sets the maximum number of split transactions the device is permitted to have outstanding at any time.
(MS) 15:7	Reserved	These bits are hardwired to zero.

- **PCI-X Status**

This register is read only. The next table explains the fields of the register

PCI-X Status		
Bit Location	Description	Action
2:0 (LS)	Function Number	Read only. Contains the Function Number.
7:3	Device Number	Read only. Contains the Device Number.
15:8	Bus Number	Read only. Contains the Bus Number.
26	64-bit Device	Hardwired to 1. The device supports 64 bits transfer.
17	133 MHz Capable	Hardwired to 1. The device is 133 MHz capable.
18	Split Completion Discarded	This bit is set if the device discards a split completion because the requester would not accept it.
19	Unexpected Split Completion	This bit is set if an unexpected split completion with this device's requester ID is received.
20	Device Complexity	Hardwired to 0 representing a simple device.
22:21	Designed Maximum Memory Read Byte Count	Hardwired to 2'b11. The maximum memory read byte count is 4096 bytes.

25:23	Designed Maximum Outstanding Split Transactions	Hardwired to 3'b000. The device supports only one outstanding split transaction.
28:26	Designed Maximum Cumulative Read Size	Hardwired to 3'b010 The maximum cumulative read size is 4K bytes
29	Received Split Completion Error Message	This bit is set if the device receives a split completion message with the split completion error attribute bit set.
(MS) 31:30	Reserved	These bits are hardwired to zero.

4.3.1.2 DMA Request Queue

The DMA Request Queue is a cyclic queue and uses two 2-port memories (implemented with BRAMs) in parallel with 2048 word x 32-bits configuration each and two pointers. We use two memories to allow 32-bit or 64-bits burst accesses to the queue via the PCI target interface. Each request is described with 2 entries in the queue. The first defines the PCI source address and the second the Operation, word count and the remote host destination address as shown in Fig. 6.

The Head Pointer points to the DMA request to be served and the Tail Pointer to the next free Entry of the DMA Queue (the head pointer follows the tail pointer even they wrap-around). The Tail Pointer is 10 bits (we can have up to 1024 pending Requests). The Head Pointer is 11-bits. The 10 most significant bits point to the currently served DMA Request and the least significant bit is used to copy each of the 2 request entries to the DMA engine. The DMA Request Queue is memory mapped and is accessible by the Host using the PCI target interface in the address range (Base Address Register 1 + 0x4000) up to (Base Address Register1 + 0x4FFF).

The structure of an entry in the DMA Queue is:

- The PCI Source address is used to define the physical host address where the data to be transmitted exist. These data will be read from the host memory with a DMA operation.
- The Remote Host Destination address is the physical address of the receiver where the data will be transferred. Bits [31 : 0].
- The Word count field indicates the size of the transfer in 64-bits words. The maximum size of each transfer is 512 words therefore 4096 bytes. Bits [41 : 32].
- The Operation field indicates the features of the transfer. Bits [63:59]. These bits include a Remote Notification Flag (Bit [62]) , Local Notification Flag (Bit [61]), Remote Interrupt Flag (Bit [60]) and a Start Flag (Bit [59]) which initiates the requests from the head pointer until the current request (clustering). *See also the "NIC Cards" paragraph in section 4.1.*

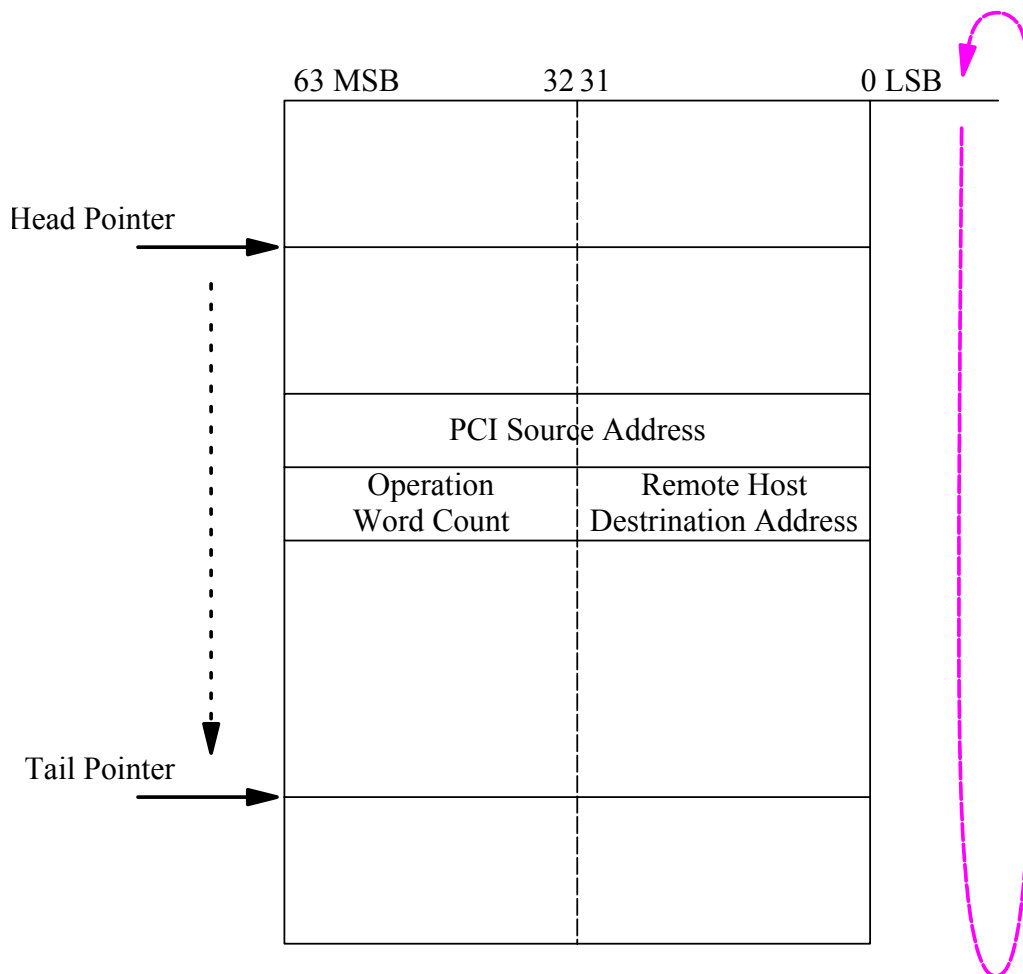


Fig. 6 The DMA Request Queue

4.3.1.3 Operation Registers & Performance Counters/Timers

This block contains several registers. These registers are accessible via the PCI target interface using the content of the *Base Address Register 1* and the address offsets given by the next table:

Address Offset	Register Name
DMA Operation Registers	
0x00	Control Register
0x04	Head Tail Pointers
0x08	Remote -Notification PCI Address Low
0x0C	Remote-Notification PCI Address High
0x10	Local -Notification PCI Address Low

0x14	Local -Notification PCI Address High
Performance Counters/Timers	
0x20	Host2NicDMAops Counter
0x24	Nic2HostDMAops Counter
0x28	Host2NicReq Counter
0x2C	Nic2HostReq Counter
0x30	Host2NicQWord Counter
0x34	Nic2HostQWord Counter
0x38	Host2NicDMAactive Timer
0x3C	Nic2HostDMAactiveTimer
0x40	Host2NicBusGranted Timer
0x44	Nic2HostBusGranted Timer
0x48	Host2NicDisc Counter
0x4C	Nic2HostDisc Counter
0x50	IRQ Active Timer
0x54	IRQ Assertions Counter
0x58	Local Notification Counter
0x5C	Remote Notification Counter
0x60	Split Operations Counter
0x64	Split Duration Timer
0x68	Packet Body CRC Error Counter
0x6C	Packet Header Alignment Error Counter
0x70	Cumulative Timer
0x74	Host2NetfiFullError Counter
0x78	Nic2HostfiFullError Counter
0x7C	Interval Timer Max Value Register

0x80	Sample Counter
0x84	Sampling Memory Data
Network Module Counters	
0x80	Outgoing Net Cycles Counter
0x84	Incoming Net Cycles Counter
0x88	Rocket I/O Unknown Character Counter
0x8C	Rocket I/O Loss of Sync Counter
0x90	Rocket I/O Parity Error Counter
0x94	Rocket I/O Tx Buffer Full Counter
0xA8	Outgoing Net Packets Counter
0xAC	Dequeued Words Counter
0xB0	Incoming Net Packets Counter
0xB4	Enqueued Words Counter
0xB8	Header CRC Error Counter
0xBC	Net Fifo Alignment Error Counter
0xC0	Net Backpressure Cycles Counter
0xC4	Host2Net Start Counter
0xC8	Net2Host Start Counter

4.3.1.3.1 DMA Operation Registers

- **DMA Control Register**

This register is used for debugging purposes. The next table explains the fields of the register:

DMA Control Register [31:0]		
Bit Location	Description	Action
31:30	Net Loop Back	When set the net loop back is activated.

29:25		Not currently used.
24	Head and Tail Pointers Reset	When set it resets the head and tail Pointers
23:5		Not currently used.
4	Software Reset	If set by software, the fifos are cleared and the FSMs are set to their initial states.
3:1		Not currently used.
0	Initiator Enable	DMA Initiator enable bit.

- **Remote-Notification PCI Address Low**

The Remote-Notification PCI Address Low 32-bit R/W register contains the low order bits of the PCI address to which the remote notification message will be written.

- **Remote -Notification PCI Address High**

The Remote-Notification PCI address high 32-bit R/W register contains the high order bits of the PCI address to which the remote notification message will be written.

- **Local-Notification PCI Address Low**

The Local-Notification PCI address low 32-bit R/W register contains the low order bits of the PCI address used for the DMA finish notification.

- **Local -Notification PCI Address High**

The Local-Notification PCI address high 32-bit R/W register contains the high order bits of the PCI address used for the DMA finish notification.

4.3.1.3.2 Performance Counters / Timers

- **Host2NicDMAops Counter**

The Host to NIC DMA Operations counter is a 32-bit R/W counter incremented every time a DMA transfer from Host to NIC is completed (request queue FSM transition to state NoPndOp).

- **Nic2HostDMAops Counter**

The NIC to Host DMA Operations counter is a 32-bit R/W counter incremented every time a DMA transfer from NIC to host is completed (request queue FSM transition to state NoPndOp).

- **Host2NicReq Counter**

The Host to NIC Request counter is a 32-bit R/W counter incremented every time a PCI bus request from Host to NIC is completed (request queue FSM transition to state NoPndOp).

- **Nic2HostReq Counter**

The NIC to Host Request counter is a 32-bit R/W counter incremented every time a PCI bus request from NIC to host is completed (request queue FSM transition to state NoPndOp).

- **Host2NicQWord Counter**

The Host to NIC Quad-Word counter is a 32-bit R/W counter incremented every time a 64-bit

word is transferred from the host to the NIC during a DMA operation.

- **Nic2HostQWord Counter**

The NIC to Host Quad-Word counter is a 32-bit R/W counter incremented every time a 64-bit word is transferred from the NIC to the host during a DMA operation.

- **Host2NicDMAactive Timer**

The Host to NIC DMA Active timer is a 32-bit R/W timer measuring the duration in clock cycles of the DMA transfers from host to NIC. The time interval starts when the REQ_ PCI signal is asserted and ends when the DMA is completed (request queue FSM transition to state NoPndOp).

- **Nic2HostDMAactive Timer**

The NIC to Host DMA Active timer is a 32-bit R/W timer measuring the duration in clock cycles of the DMA transfers from NIC to Host. The time interval starts when the REQ_ PCI signal is asserted and ends when the DMA is completed (request queue FSM transition to state NoPndOp).

- **Host2NicBusGranted Timer**

The Host to NIC Bus Granted timer is a 32-bit R/W timer measuring the sum of the time intervals during which DMA transfers from host to NIC take place. The time interval starts when the FRAME_ PCI signal is asserted and ends when the last data phase is completed (request queue FSM transition to state NoPndOp).

- **Nic2HostBusGranted Timer**

The NIC to Host Bus Granted timer is a 32-bit R/W timer measuring the sum of the time intervals during which DMA transfers from NIC to host take place. The time interval starts when the FRAME_ PCI signal is asserted and ends when the last data phase is completed.

- **Host2NicDisc Counter**

The Host to NIC Disconnect counter is a 32-bits R/W counter. The value of the timer is incremented every time a DMA transfer (Host to NIC) is disconnected due to a latency timer timeout.

- **Nic2HostDisc Counter**

The NIC to Host Disconnect counter is a 32-bits R/W counter. The value of the timer is incremented every time a DMA transfer (NIC to host) is disconnected due to a latency timer timeout.

- **IRQActivet Timer**

The Interrupt Request Active timer is a 32-bits R/W timer. This timer counts the PCI clock cycles that the interrupt request line INTA_ remains Active.

- **IRQ Assertions Counter**

The IRQ Assertion counter is a 32-bits R/W Counter. This counter is incremented at the interrupt line's negative edge.

- **Local Notification Counter**

The Local Notification counter is a 32-bits R/W counter. This counter is incremented with the completion of a local notification.

- **Remote Notification Counter**

The Remote Notification counter is a 32-bits R/W counter. This counter is incremented with the completion of a remote notification.

- **Split Operations Counter**

The Split Operations counter is a 32-bits R/W counter. This counter is incremented upon receipt of a split response.

- **Split Duration Timer**

The Split Duration timer is a 32-bits R/W counter. This counter is incremented at every clock edge between the receipt of the split response and the receipt of the first datum of the first split completion is supplied.

- **Packet Body CRC Error Counter**

The Packet Body CRC Error counter is a 16-bits R/W counter. The value of this counter is incremented every time a packet body CRC error is detected.

- **Packet Header Alignment Error Counter**

The Packet Header Alignment Error counter is a 16-bits R/W counter. The value of this counter is incremented every time an alignment error is detected

- **Cumulative Timer**

The Cumulative timer is a 32-bits R/W timer. The timer is activated by writing an initial value (eg: 0). Upon the assertion of the REQ_PCI signal of the first DMA transfer the timer starts counting. Each subsequent read of this timer returns the time interval from the 1st request until the completion of the last DMA transfer that has finished. In other words, this timer is updated with the clock cycle count on the UpdtCnt state of the request queue FSM.

- **Host2NetFifoFullError Counter**

The Host to Net Fifo Full Error counter is a 32-bit R/W counter incremented every time a 64-bit word is transferred from the host to the Net during a DMA operation and is lost due to full Fifo enqueue.

- **Net2HostFifoFullError Counter**

The Net to Host Fifo Full Error counter is a 32-bit R/W counter incremented every time a 64-bit word is transferred from the Net to the host during a DMA operation and is lost due to full Fifo enqueue.

- **Configurable Sampling Facility**

The board offers a configurable sampling facility to estimate the inbound and outbound bandwidth. After specifying a sampling interval duration (expressed in PCI-X cycles) and a sample count, the board records in internal memory the number of 64-bit words transmitted and received during each of the sampling intervals. The internal sampling memory can hold up to 2048 samples. The collection of samples begins with the first DMA transaction after setting up the sample count parameter. The counts of inbound and outbound 64-bit words are accumulative. A utility program can read the sampling memory and compute the aggregate bandwidth (in MBytes/sec).

- **Interval Timer Max Value Register**

This 32-bit read write register holds the sampling duration expressed in PCI-X cycles.

- **Outgoing Net Packets Outgoing Net Packets Sample Counter**

This 32-bit read write counter holds the number of samples to be collected.

- **Sampling Memory Data**

For each sampling interval, this memory module holds two 32-bit counts, corresponding to the number of 64-bit words transmitted and received during each of the sampling intervals. This memory module can hold up to 2048 samples.

4.3.1.3.3 Network Counters

The next 19 counters are used for debugging purposes and are read clear counters. The software can read their value performing a load operation to the corresponding offset or clear all the counters simultaneously performing a store operation to the offset 0x80.

- **Outgoing Net Cycles Counter**
The Outgoing Net Cycles Counter is incremented in every network clock cycle the network block transmits data or credits.
- **Incoming Net Cycles Counter**
The Incoming Net Cycles Counter is incremented in every network clock cycle the network block receives data or credits
- **Rocket I/O Unknown Character Counter**
The Rocket I/O 0 Unknown Character Counter is incremented every time the Rocket I/O 0 detects an Unknown Character.
- **Rocket I/O Loss of Sync Counter**
Rocket I/O 0 Loss of Sync Counter is incremented every time the Rocket I/O 0 detects a Loss of Sync.
- **Rocket I/O Parity Error Counter**
The Rocket I/O 0 Parity Error Counter is incremented every time the Rocket I/O 0 detects a Parity Error.
- **Rocket I/O Tx Buffer Full Counter**
The Rocket I/O 0 Tx Buffer Full Counter is incremented every time the Rocket I/O 0 is forced to discard a byte due to the Tx buffer being full.
- **Outgoing Net Packets Counter**
The Outgoing Net Packets Counter is incremented every time a packet is send.
- **Dequeued Words Counter**
The Dequeued Words Counter counts the numbers of words send by the network block.
- **Incoming Net Packets Counter**
The Incoming Net Packets Counter is incremented every time a packet is received.
- **Enqueued Words Counter**
The Enqueued Words Counter counts the number of words received from the network block.
- **Header CRC Error Counter**
The Header CRC Error Counter is incremented every time the network module detects a header CRC error.
- **Net Fifo Alignment Error Counter**
The Net Fifo Alignment Error Counter is incremented each time incoming network data are misaligned.
- **Net Backpressure Cycles Counter**
The Net Backpressure Cycles Counter represents the network clock cycles the network module was forced to wait due to lack of credits (backpressure).
- **Host2Net Start Counter**

The Host2Net Start Counter counts the assertions of the Host2Net Start signal.

- **Net2Host Start Counter**

The Net2Host Start Counter counts the assertions of the Net2Host Start signal

4.3.1.4 Network Interface

The network interface module utilizes mainly two FIFOs that are used so as to store the outgoing (Host2NicFIFO) and incoming (Net2HostFIFO) data to/from the network modules of the NIC. The size of the FIFOs is 1023 68-bit words and they also provide four control signals, namely empty, almost empty, full and almost full, along with the actual utilization of the FIFO (i.e. the number of entries currently occupied). The words are 68-bits since the 64 LS bits are used for the data and the other 4-bit carry control information such as start of packet (bit 65), end of packet (bit 66) and the next 2 bits are reserved for word enables to show which 32-bit data words are valid. Those FIFOs are connected to the network modules using the interface specified in the following table.

“PCI-block to Network” Module Interface			
Pin Name	Type	Size (bits)	Description
Host2NetFiDt	Output	68	Host to network FIFO data out.
Host2NetFiDeq	Input	1	Network to host FIFO dequeue.
Host2NetPReady	Output	1	Host to network packet ready.
Net2HostfiDt	Input	68	Network to host FIFO data in.
Net2HostfiEnq	Input	1	Network to host FIFO enqueue.
Net2HostfiSt (Synchronized with the Network Module)	Output	9	Network to host FIFO status: Number of 64-bit words contained in the net to host FIFO.
Net2HPReady	Input	1	Net to host packet ready. This signal indicates that the incoming packet is in the net to host FIFO.
Net2HHReady	Input	1	Net to host header ready This signal indicates that the header of the incoming packet is in the net to host FIFO.
NetPcktError	Input	1	Net packet error. This signal indicates the network detects a header error.
Loop	Output	2	Network Loop Back. 00: Normal operation. 01: Not supported. 10: Not supported.

On the outgoing path, Host2NetPReady is asserted every time a full packet is enqueued into the Host2NetFIFO and thus it is ready to be transmitted over the network. The Host2NetPReady signal is

synchronous to the net clock and is asserted for one clock cycle. Then the network module, when it decides that it can process the packet, is responsible for raising the Host2NetFiDeq signal so as to start the dequeue from the Host2NetFIFO. The Host2NetFiDtout is the 68-bit data bus that is connected to the output of the Host2NetFIFO. There is also the Host2NetFiSt status bus that is synchronized with the network module’s clock and is used so as to allow the network module to read the status of the Host2NetFIFO.

A network packet consists of the header part, the data part (body) and the CRC part. The header part is the first 68-bit words of the packet and is explained in the table below. Their four most significant bits identify all such parts. A value of zero represents a data word, a value of one a packet header and a value of two a body CRC word. The CRC we use is the popular CRC32-Ethernet / AAL5 which is 32-bits and its polynomial is:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1.$$

Header Part Fields		
Bit Location	Description	Action
(MS) 67:64	Word Identifier	The value of these bits is 4'b0001 for the header part.
63	Packet Type Identifier	This bit is set to 0
62:56	Node ID	128 nodes are supported
55	Operation Code	This bit is set to 0
54		Remote Completion Notification Flag
53		Local Completion Notification Flag
52		Remote Interrupt Flag
51		This bit is set to 1
50:42	Not implemented	Hardwired to zero
41:32	Packet Size	This field contains the payload size in 64 bit words
31:0 (LS)	Designation Address	This field contains the destination PCI Address

4.3.2 RocketIO Network Interface Module

The Network Interface (NetIF) Module is the part of the design that enables the communication of the rest of the fpga modules with the network. This interface is implemented with the Xilinx RocketIO transceivers. The communication with the rest of the fpga is accomplished through a very simple interface, which mainly uses asynchronous fifos (as different clock domains are crossed), and just a few handshaking signals.

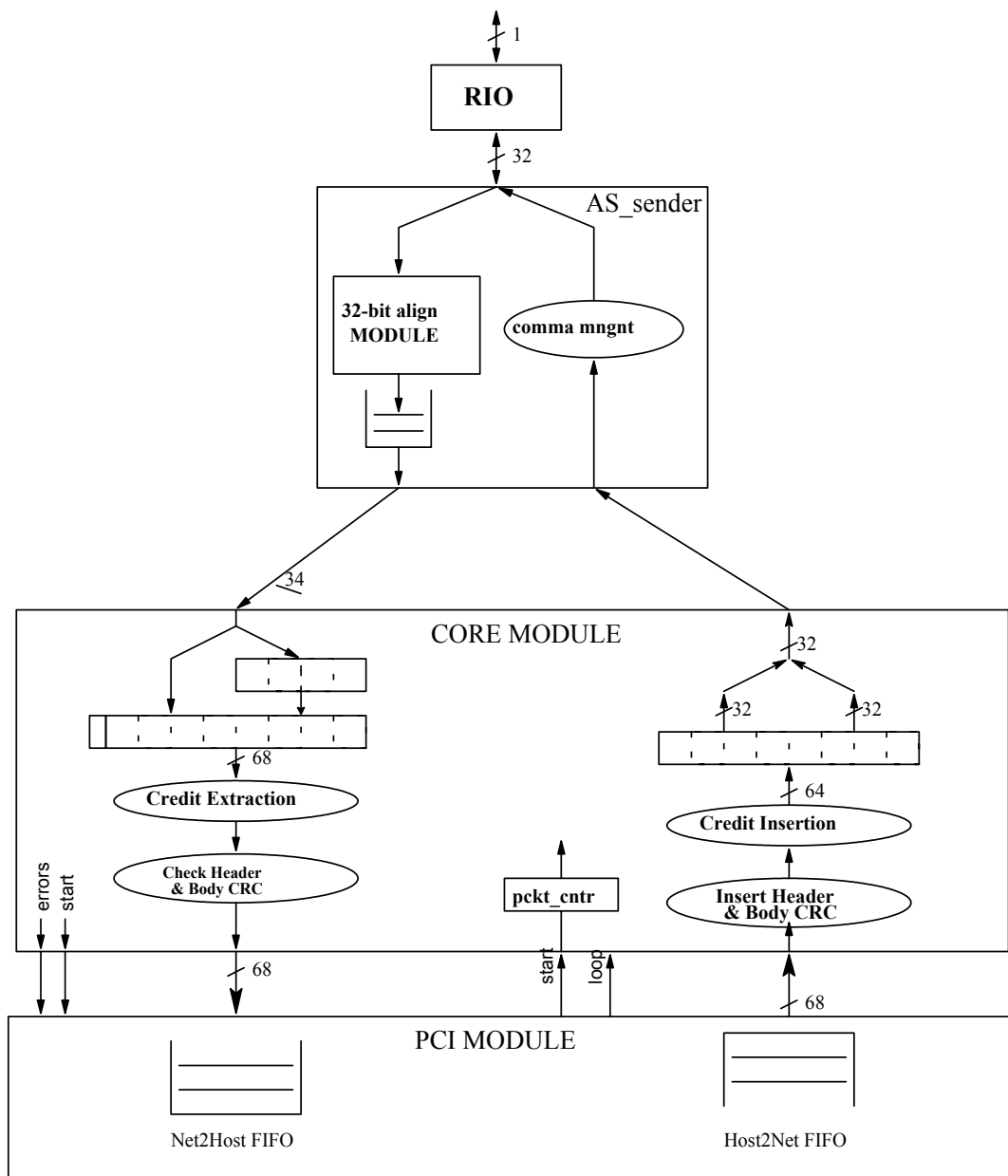


Fig. 7 Network Interface Module Block Diagram

The basic structure of the NetIF is outlined in the block diagram of Fig. 7. We will carry on explaining the functionality of the NetIF as we look at this block diagram. At the bottom of the diagram we see a

small part of the PCI module, which is the one directly linked to the NetIF. For each direction, incoming and outgoing, the PCI module deploys a separate asynchronous fifo. The width of the fifo is 68 bits, from which 64 are used for data, and the remaining 4 bits for tagging (flags for several purposes). This width of each of the two fifos is directly linked to the data format exchanged among the two modules. The data to be exchanged is formed into packets, and is then send to the NetIF. The NetIF on the other hand, takes that packet, sends it to the network after applying the needed modifications and additions. At the receiving side, it takes a packet, and after striping any network overhead, gives it to the PCI module (or any other receiving module). Explanations on the packet format can be found in section 4.3.2.4.

Let us first describe **the transmitting side** of the diagram. After the PCI module has inserted a complete packet into the Host2Net fifo, it raises the start signal for a single clock cycle. The NetIF knows that now at least a single full packet resides in the fifo. As many packets can simultaneously reside in the fifo, it is the NetIF's responsibility to keep track of the number of packets still residing therein. This of course must be done correctly even if a packet transmission is active at the time a "start" is signaled. This functionality is carried by the module tagged `pckt_cntr` (packet counter) in the diagram. Any time a full packet resides in the fifo, the NetIF has the responsibility to read it, and safely transmit it to the network. To have additional error checking capabilities, the NetIF adds a header CRC while transmitting, right after the header is send. This is a CRC calculated on the header bits, which are the most critical data of a packet. Details on the CRCs used can be found in the table below. As seen in the diagram the data are sent to the network through the RocketIO transceiver. The RocketIO can reach up to 3.125 GBaud (2.5 Gbps). We use them at this maximum rate. The AS_sender module, which is directly linked to the RocketIO, adds synchronization information to the link, at the time periods that no useful network traffic exists.

	# bits	Function
Header CRC	16	Polynomial : (0 5 12 16), data width : 64, initial CRC value : 16'hFFFF first serial data bit : D[63]
Body CRC	32	polynomial:(0 1 2 4 5 7 8 10 11 12 16 22 23 26 32), data width : 64, initial CRC value : 32'hFFFF_FFFF first serial data bit : D[63]

At **the receiving side**, each AS sender has a small synchronization fifo. This is used in order to form 64-bit words, by combining the two 32-bit words of each link. This is needed, as the exact arriving time of the bits at each link may and will differ, and so the 32-bit words are not presented always at the same clock cycle from the RocketIO to the AS_sender. The fifo then serves for removing this uncertainty. Before this, the problem of the alignment must be solved.

As the RocketIO sends data through a serial link, the **deserialization process** at the receiving side can cause misalignment of the bytes. That is the bytes may be found shifted in the received 32-bit word. This problem can be overcome by using a separate circuit for each link, which uses two consecutive words to form a valid one. This module is the `rcv_align` module. This is responsible for taking the misaligned incoming stream, extract the SOP and synchronization information from the link, and deliver the packet in its original form. The data are also accompanied by a start flag, in order to pass delimiting information to the core module.

The whole information (data and flags) are enqueued in the small fifo mentioned earlier, and through

it are delivered to the core module. After the data are restored to the state they were sent, the **incoming data processing** can be made easier. In the beginning we check for **credits**, and after zero or more credits, zero or one data packets can follow. The **data packet** starts with the header, followed by the header CRC. The header CRC is checked and if it is found to be wrong the packet received is not passed to the PCI module. If it is correct, it is send, but only after it is correctly added to a 68-bit word, and the flag bits (the 4 MS bits) of each packet word are accordingly set. At the end of the packet the body CRC is checked. Correct or not, the packet has already been enqueued, so the result of the body CRC is just passed across to the other module. On a NIC, both the header and the body CRC are striped from the packet before it gets delivered. However on the slightly differentiated version for the crossbar switch, the body CRC is delivered. Finally let as mention that the output labeled “errors” represent a number of signals that inform the other module of some common errors (header and body CRC error, alignment error). The last data word is accompanied by an eop flag (bit 65).

4.3.2.1 Describing the Logic Through the Related FSMs

Outgoing path (Host to Network)

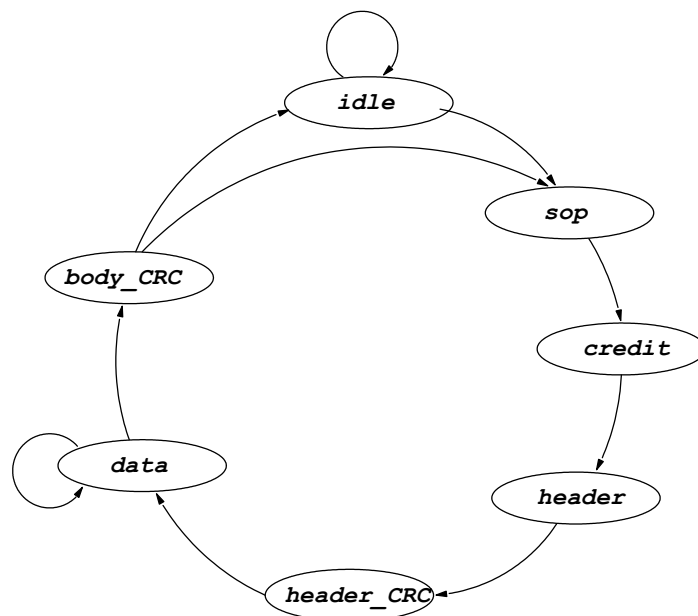


Fig. 8 Host to Network FSM

Fig. 8 depicts the FSM that serves the outgoing path (PCI module to network). This diagram just includes the states and the transitions, but bare in mind that quite excessive checking is included in most of the states, in order to accommodate for proper traffic manipulation. This also stands for the outgoing path FSM, which we will describe shortly. As we see in Fig. 8 the FSM starts from the idle state, and stays there as long as no traffic exists. When a full valid packet is found in the Host2Net fifo, the circuit starts sending the module. As a complete packet is enqueued before this process starts, the FSM can handle the whole packet non-stop, as the difference between the frequencies of the clocks for the two different sides does not matter. First the SOP is send to the network. This is not included in the packet, and is added as a network delimiter to a new packet. Then the header is read and sent. At the next cycle, a header CRC is calculated and is sent over the network. Again, this is not part of the original packet, and will be used by the receiving network interface, and will then be striped off. Then the data is sent. The circuit has two ways to double check when the data ends. One is the size that was

found in the header, and the other is the flag at the last 68-bit word that signals the last word. At the end the body CRC, which was calculated all the way through the payload data transmission, is given. Notice that in order to enhance performance, the next state of “body_CRC” is not only the “idle”, but it can also be “sop”. This is done when a new packet is ready at the time the current packet is being sent. At this circumstance, we can immediately move to the “sop” state, and avoid the many unneeded commas.

Incoming path (Network to Host)

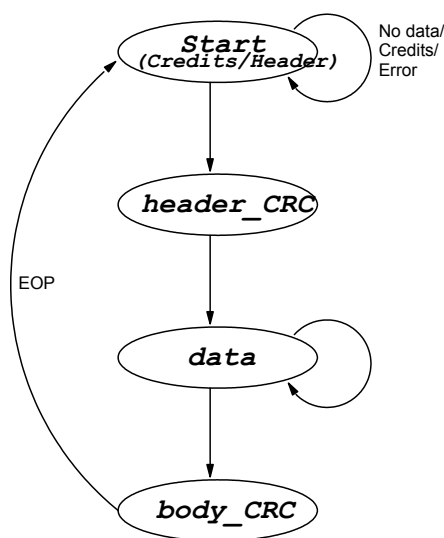


Fig. 9 Network to Host FSM

Fig. 9 depicts the FSM that serves the incoming path (network to PCI module). It starts with the start state. This state has the responsibility to find a start of packet from both of the links, which have been already properly aligned. This state also manages the incoming credit management. Transmissions can consist only of credits, or credits and a data packet, in which case one or more credits precede the packet. If errors occur, the erroneous packet is discarded, and a new packet arrival is awaited. If the process succeeds, we move on to the “headerCRC” state along with the enqueueing of the header to the Net2Host fifo. However this is not done before this state, the “header_CRC”, receives the Header CRC, and checks its correctness. If it is valid the packet will be enqueued to the fifo, as the FSM advances to the next state. However, if it is found to be wrong, the enqueueing is deactivated, and the rest of the states are used to just discard the packet. At this time, the NetIF signals to the PCI module that a packet has arrived, and so the later can start dequeuing it from the Net2Host fifo. Although this is not absolutely true, it will not result to any errors, as the network runs with a faster clock compared to the PCI module (78.125MHz vs. 66.67MHz). This way cut-through is implemented in the network-to-host stream. In later versions that use PCI-X with faster clocks (100MHZ-133MHZ) this is not implemented. Instead a mechanism which is aware of the two frequencies (through user definitions) calculates a suited time in order to signal start and thus begin cut-through as soon as possible. When all the data are enqueued to the Net2Host fifo, the EOP flag will be received along with the last data word. Then the body_CRC will be checked and the FSM will get to the “start” state.

4.3.2.2 Round Trip Time (RTT) Considerations

The time that is needed for information to travel from the transmitter to the receiver and again back at

the transmitter (the RTT) is a quite important parameter of a system. For this system we have measured the RTT with two different manners: theoretically and through lab testing. In this way we have been able to compare the two results and see if they agree, in order to have a more reliable RTT value. For the lab testing we have added some logic in our original design that performs the following measurement. It counts cycles spent from the beginning of a packet transmission up to the point that an updated credit gets received. In this way we are sure that the packet has started being delivered at the receiving host, and after some time, a credit arising from that packet delivery is formed and sent back to the transmitter. Different experiments have been carried, with different packet sizes to make sure of the correctness for the lab measurement. The resulting RTT is about 90 network side clock cycles (78,125MHz clock) plus a number of cycles equal to packet-size/4. This last term is due to the cut-through latency, explained at section 4.3.2.1, which delays a packet from being delivered from the NIC to the receiving host. The “/4” is for the latest system that uses PCI-X at 100MHz, and thus is capable of starting delivering the packet to the host, after the first quarter has arrived from the network.

$$RTT = (90 + \frac{packet_size}{4})cc$$

This number conforms to the theoretical approach and thus the lab measurement seems quite robust. Briefly referring to the theoretical approach, we mention that the 90 clock cycles mostly involve RocketIO-to-network and vice-versa (about 45cc), credit granularity (32 cc - details on section 4.3.2.5) and NIC-to-PCI latency (about 5cc).

4.3.2.3 The RocketIO Transceiver Instantiation

In this paragraph we will take a look at some detail concerning the way the RocketIO is used by our design. The reader should be familiar with Xilinx’s RocketIO transceiver. A reference for this is “RocketIO Transceiver User Guide” by Xilinx. The GT_CUSTOM primitive was used, as it is the most flexible, allowing the most user modifications. In the current fpga, xc2vp40, the X1Y1 and X2Y1 MGT locations at the top of the fpga were used. The COMMA character was chosen to be the default one, K28.5 (‘hBC), and K27.7 (‘hFB) is used as the SOP character. BREFCLK2 was used as the clock input, as the clock we provide is of a frequency greater than 2.5GHz. Actually the crystal mounted on the board gives the higher accepted frequency, which is 3.125GHz. BREFCLK2 was used instead of BREFCLK, as the crystal output on the board enters the fpga through pins H17 and J17, and so the internal clock routing of the MGTs obligates us to use BREFCLK2. The USERCLKs are produced from the clock given to the MGTs, with the help of a DCM. The USERCLK is given the MGT clock divided by two, as we use a user datapath width of 32 bits. The USERCLK2 is the USERCLK shifted by 180 degrees. As far as clock recovery is concerned, CLK_CORRECT_USE is set to true, to allow correction. CLK_COR_REPEAT_WAIT is set to 0 and CLK_COR_KEEP_IDLE is set to false, to allow the maximum capability for clock correction. The RX_BUFFER is set to true to also aid this cause. CLK_COR_SEQ_1_1 is set to the COMMA character, and CLK_COR_SEQ_LEN is set to 1, to allow single COMMA to be the sequence that is allowed to be removed. The CRC insertion is set to inactive, as the header CRC insertion and checking is done through user designed circuits. One last thing concerning the RocketIO attributes is that the TX_DIFF_CTRL is set to 800 instead of the default value 500, and TX_PREEMPHASIS is set to 3 instead of the default value 0. These two values are based on lab testing, as the default values lead to a very high error rate. Note however that even more moderate setting (e.g. 500 and 2 respectively) could probably lead to a working set. Finally, as far as loopback is concerned, the core_AS module has a 2-bit input that accepts the loopback mode. The coding of the modes is the one given by the RocketIO specifications, and the user must be cautious to drive this input at the correct value at all times.

4.3.2.4 Data Packet Format

The data to be exchanged with the NetIF module are formed into packets. Fig. 10 will guide us through this description. Each packet is formed from 68-bit words at the interface. These words exchanged to and from the NetIF module include 64 MS bits for the actual data, and the remaining 4 MS bits of each word add tagging (flags) and are only used to enhance the communication of the two modules (SOP, EOP, RFU...). The data includes header and body (payload).

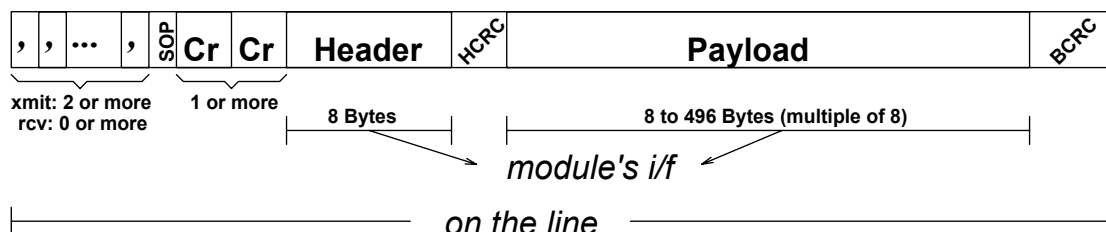


Fig. 10 The packet format at various interfaces

The packet starts with a 68-bit (one word) header. The format of the actual header (the 64 bits) is shown in Fig. 11. The 32 LS bits include the PCI address. The next 10 bits correspond to the packet size. The size is measured in words (64 bit words), and only the payload (the useful data -or body-) are measured. The header is thus excluded. The next 10 bits are Reserved for Future Use (RFU). Then, bits 51 through 55 form the opcode, something of no relevance to the NetIF. This will just be transmitted and will be used by the receiving side's PCI module. The next 7 bits carry the flow ID, which must be taken into account by the NetIF. The bit 63 is a flag that tells if the word is a header or a credit. 0 is used for header, 1 for a credit. Finally the 4 MS bits (only for the interface - not shown) carry the flags. For the header bit 64 bit must be set to 1 to signal SOP and the rest three to 0. For the other words all four must be zero, excluding the last word which must have bit 65 set to 1 to signal EOP.

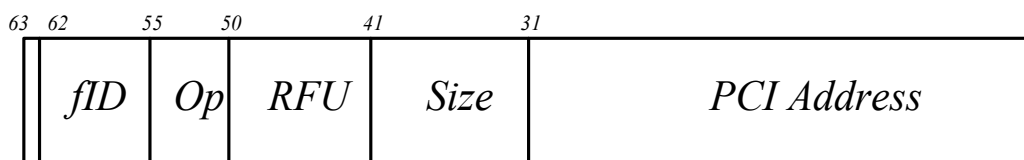


Fig. 11 The Packet Header Format

4.3.2.5 Credit Format and Credit Protocol

Fig. 12 depicts the **credit format**. Each credit is 16 bits wide. The first 7 bits concern the credit value. How this value is calculated will be explained shortly. The next 7 bits describe the flow ID, which means the flow that this credit is meant for. With 7 bits we can support as much as 128 different flows. Bit 14 is the parity, which is the xor of the previous 14 bits (bits [13:0]). This is created at the transmitter and checked at receiver. If it is found to be wrong, the credit is discarded, i.e. it is not taken into account. Finally the MS bit, bit 15, is the flag that differentiates between a credit and a header, and should be 1 to designate a credit.

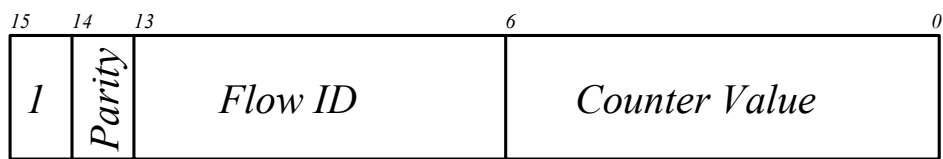


Fig. 12 The Credit Format

Now let us see the **credit protocol** in some detail. This way we will also understand the meaning of the “Counter Value” field, seen at Fig. 12. The protocol is based on two cumulative counters at each NIC. One of these is counting the complete number of words that have been transmitted to the network since power up. The other counter keeps track of the number of words that have been forwarded from the network to the host, again in a cumulative manner since power up. This second counter is transmitted through the “Counter Value” of the credit to the other side. When the other side receives this credit, it has to compare it with its first counter (the one counting transmitted packets to the network) and see if there is enough space to send a new packet with a known size (the packet size is known through its header). In order for this protocol to work, correct initialization of the counters is needed. That is the fifo size at the other end must be taken into account, so that at start up this is the free space downstream. One other thing to take into account, is that a cumulative counter will sometime overflow. In order to compensate for this, the comparators at each side must be aware of value wrap around. Two extra bits for each counter are enough to implement this mechanism. A last issue concerns the width of the counters. Based on the RTT of the system, and the need for wrap around support, our counters end up being 12 bits wide. So the “Counter Value” of the credit is a subfield of the corresponding counter. Only the 7 most significant bits (bits [11:5]) are transmitted through the credit. The remaining 5 LS bits are not transmitted, and the receiver treats them as being all zero. This gives as a more coarse grained granularity for the credits, as an updated credit at the receiver will only be seen for every 32 (2⁵) words transmitted. This however does not create any trouble.

5. CONCLUSIONS

During year 2 of the project, within WP5, FORTH has:

1. Concluded the verification of the excellent behavior of the teraChannel scheduling mechanisms for bandwidth allocation and delay guarantees –see D4.4 revision E.
2. Developed innovative methods for reducing the size of the crosspoint buffers of buffered crossbars to a size determined by its scheduler RTT rather than by the packet sizes, while maintaining the other excellent properties of buffered crossbars. This yields a realistic architecture with excellent performance properties, appropriate to become the switching node in the next generation switching fabrics that will be flow-controlled using the RECN protocol.
3. Implemented and successfully tested on its FPGA-based prototyping platform the buffered crossbar architecture operating on variable-size packets, thus verifying its correctness.
4. Implemented network interface cards (NIC's) on its FPGA-based prototyping platform that provide advanced remote DMA and remote notification services, as well as extensive debug and performance monitors and counters, for use by the WP9 performance analysis.

6. REFERENCES

- [1] D. Stephens, Hui Zhang: "Implementing Distributed Packet Fair Queueing in a Scalable Switch Architecture", IEEE INFOCOM'98 Conference.
- [2] F. Abel, C. Minkenber, R. Luijten, M. Gusat, I. Iliadis: "A Four-Terabit Packet Switch Supporting Long Round-Trip Times", IEEE Micro Magazine, Jan./Feb. 2003, pp. 10-24.
- [3] N. Chrysos, M. Katevenis: "Weighted Fairness in Buffered Crossbar Scheduling", Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR 2003), Torino, Italy, June 2003, pp. 17-22; http://archvlsi.ics.forth.gr/bufxbar/bxb_scheduling.html
- [4] G. Sapountzis, M. Katevenis: "Benes Switching Fabrics with O(N)-Complexity Internal Backpressure", Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR 2003), Torino, Italy, June 2003, pp. 11-16; <http://archvlsi.ics.forth.gr/bpbenes/>
- [5] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, N. Chrysos: "Variable Packet Size Buffered Crossbar (CICQ) Switches", *Proc. IEEE International Conference on Communications (ICC 2004)*, Paris, France, 20-24 June 2004, vol. 2, pp. 1090-1096.
- [6] M. Katevenis, G. Passas: "Variable-Size Multipacket Segments in Buffered Crossbar (CICQ) Architectures", *Proc. IEEE International Conference on Communications (ICC 2005)*, Seoul, Korea, 16-20 May 2005, CR-ROM paper ID "09GC08-4", 6 pages.
- [7] N. Chrysos, M. Katevenis: "Multiple Priorities in a Two-Lane Buffered Crossbar", *Proc. IEEE Globecom 2004 Conference*, Dallas, TX, USA, 29 Nov. - 4 Dec. 2004, CR-ROM paper ID "GE15-3", 7 pages;

[8] D. Simos: "Design of a 32x32 Variable-Packet-Size Buffered Crossbar Switch Chip", *Technical Report FORTH-ICS/TR-339*, Inst. of Computer Science, FORTH, Heraklion, Crete, Greece; M.Sc. Thesis, Univ. of Crete; July 2004, 102 pages.

[9] Xilinx ML325 Characterization Board.

URL: http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=HW-V2P-ML325&sGlobalNavPick=PRODUCTS&sSecondaryNavPick=BOARDS

[10] DiniGroup DN6000K10SC Development Board

URL: <http://www.dinigroup.com/DN6000k10SC.php>

7. APPENDIX: RESEARCH CONTRIBUTION

FORTH's new research contribution in WP5 is described in the following paper, which is appended here:

- G. Passas, M. Katevenis: "Packet Mode Scheduling in Buffered Crossbar (CICQ) Switches", FORTH-ICS, submitted for publication, version of Jan. 2006.

Packet Mode Scheduling in Buffered Crossbar (CICQ) Switches

Georgios Passas and Manolis Katevenis*

Inst. of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH) - member of HiPEAC
ICS-FORTH, P.O. Box 1385, Vassilika Vouton, Heraklion, Crete, GR-711-10 Greece
<http://archvlsci.ics.forth.gr/bufxbar/> - {katevenis,passas}@ics.forth.gr

Abstract—Buffered crossbars have emerged as an advantageous architecture mainly because of their scheduling efficiency and capacity for variable size packets. Buffered crossbars directly operating on variable size packets require at least one maximum packet worth buffer per crosspoint. When we cannot afford such large buffer sizes we are forced to segment packets. Although variable size segments can be used to avoid padding overheads, we are still left with the cost of reassembly buffers and reassembly delays. In this paper, we apply packet mode scheduling to buffered crossbars in order to remedy these packet segmentation shortcomings. Packet mode scheduling had only been studied for bufferless input-queued switches: when the central switch scheduler establishes an input-output port pairing, the pairing is maintained until all cells of the packet are forwarded. The extension to buffered crossbars is not trivial because the scheduling process does not necessarily determine such pairings. We introduce two schemes for buffered crossbars: *probabilistic* and *deterministic* packet mode scheduling. The probabilistic case assumes independent crossbar output schedulers, but requires reassembly buffers. Using simulation we show that it reduces packet delay compared to the system with pure segmentation and reassembly; for a representative traffic pattern the average packet delay is improved by more than 80% for loads up to 50%. Deterministic packet mode scheduling sacrifices some scheduler independence in order to eliminate reassembly buffers; based on our simulations, it performs very close to buffered crossbars without segmentation (which use very large crosspoint buffers), and it also performs always better than packet mode scheduling in bufferless crossbars. The probabilistic scheme performs similar to the deterministic scheme for some traffic patterns, and better than it for other traffic patterns.

I. INTRODUCTION

Crossbars are the building blocks for modern switching fabric and router systems. Traditional crossbars are bufferless – with buffering provided only on the ingress and egress line cards – hence transmissions through the switch have to be synchronized with each other, leading to operation with fixed-size segments, called *cells*. As illustrated in figure 1, variable-size packets are segmented at the inputs (where virtual-output queues (VOQ) reside), the cells are switched through the crossbar, and the packets are reassembled at the outputs (where real-output queues (ROQ) / reassembly buffers reside), before they are transmitted on the line.

* The authors are also with the Dept. of Computer Science, University of Crete, Heraklion, Crete, Greece.

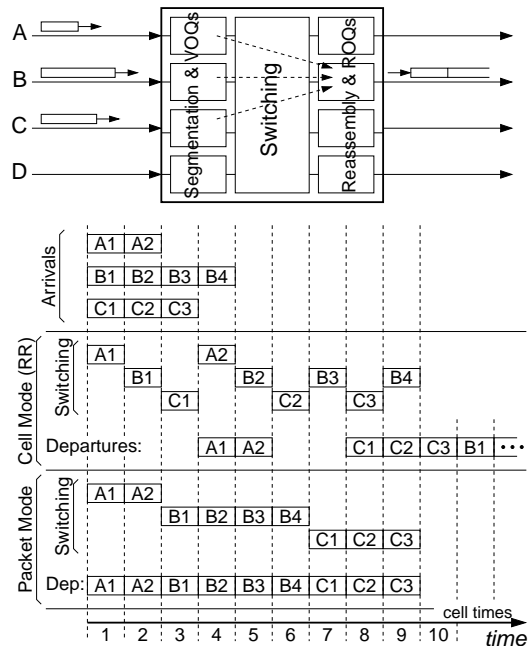


Fig. 1. Packet mode versus cell mode switch scheduling

If the switch is scheduled ignoring which cell belongs to which packet, *Cell Mode* operation results, as illustrated in the first part of the figure: the cells of a packet (say *B*) arrive at the egress line card interleaved in time with cells of other packets (*A*, *C*) arriving from other inputs. Packet transmission on the output line cannot start until the egress line card is certain of receiving the last cell of the packet. Given that scheduler decisions cannot be predicted, store-and-forward operation is enforced, and cut-through cannot be used.

Packet Mode Scheduling [1] - [5] is the alternative illustrated in the second part of the figure: when the switch makes a “connection” from an ingress to an egress line card for the first cell of a packet, that connection is maintained until all cells of the packet are switched. Since the egress line card knows that all cells of a packet will arrive consecutively in time, (a) it needs no reassembly buffer, and (b) it may start transmitting the packet right away, i.e. *cut-through* is allowed. Figure 1 illustrates that, under some circumstances,

packet-mode scheduling reduces packet delay¹; the advantage is particularly important in systems requiring low latency (e.g. multi-processor cluster interconnects), and becomes especially noticed when cut-through is supported and when traffic includes large packets (e.g. jumbo frames).

Packet mode scheduling has only been studied for bufferless crossbars. Recently, *buffered crossbars* (combined input-crosspoint queueing - CICQ), have emerged as an advantageous architecture; they contain small buffers at their crosspoints, and use backpressure to the ingress line cards to prevent these crosspoint buffers from overflowing. The first observation about buffered crossbars concerned the simplicity and high efficiency of their scheduling [6] - [8]; no internal speedup is needed to compensate for scheduler inefficiencies, thus allowing the increase of port speed. A subsequent observation was that buffered crossbars can directly switch *variable-size* packets [6] [9]. Doing so, without any segmentation, eliminates the need for speedup to cope with cell-padding overhead; in turn, the lack of speedup eliminates egress queueing, and the lack of segmentation eliminates reassembly buffers, thus reducing cost [10].

Buffered crossbars that use no segmentation at all have a limitation: the required buffer size per crosspoint is at least one maximum-size packet plus one round-trip-time (RTT) worth of data [10]. That buffer space becomes expensive in high valency switches (the number of crosspoint buffers equals the square of the port count) and in switches supporting large packets (e.g. jumbo frames).

To overcome this limitation, segmentation and reassembly (SAR) has to be re-introduced in buffered crossbars. However, SAR can be introduced in such a way that no padding overhead is incurred, hence no internal speedup is needed: in a previous paper [11], we proposed SAR using *variable-size multi-packet segments* to achieve this goal. The maximum segment size can be much smaller than the maximum packet size, thus drastically reducing crosspoint buffer size; the segment size is variable, thus eliminating padding overhead; and multiple (small) packets can be placed in a same segment, thus avoiding small segments so as to reduce relative header overhead. By contrast, bufferless crossbars need to synchronously schedule all ports, hence can only operate with fixed-size segments, and thus require padding. It is a good thing that SAR can be introduced into buffered crossbars without incurring a speedup penalty, but how about the reassembly delay (and preclusion of cut-through) and the reassembly buffer cost that SAR implies?

This paper applies packet-mode scheduling to CICQ switches, so as to reduce the reassembly delay, provide the opportunity for cut-through transmission and/or eliminate the egress reassembly buffers when these switches are forced to use SAR. Packet-mode scheduling in buffered crossbars is not a trivial extension of the bufferless case. Figure 2 illustrates the two kinds of crossbar. In bufferless crossbars (left), there is a

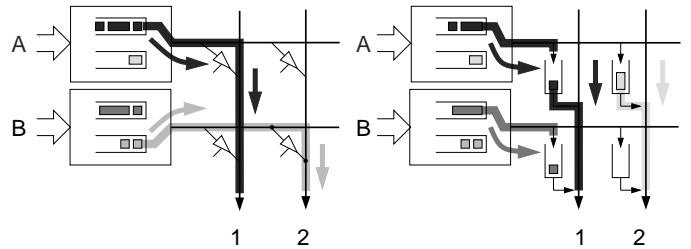


Fig. 2. Bufferless (left) versus buffered crossbar (right)

central scheduler which determines input-output port pairings (connections). Packet-mode scheduling is simple: once a connection is made, keep it until the last cell of the packet. In buffered crossbars (right), scheduling is distributed and independent: each input selects a non-full crosspoint and sends to it; each output selects a non-empty crosspoint and reads from it. Two or more inputs (A and B in the figure) may send to the same column; two or more outputs (1 and 2 in the figure) may read from the same row. Thus, the scheduling process does *not* necessarily determine input-output pairings (it does not need to, and that is why it is more efficient). Packet mode, however, requires such pairings.

We describe two methods for packet mode scheduling in buffered crossbars. The first assumes distributed and independent scheduling at switch input and output ports, as in the classical buffered crossbar architecture. We synchronize the input and output schedulers so that, whenever their independent decisions result to an input-output pairing, that pairing is maintained for the lifetime of the corresponding packet transmission. We call this first scheme *probabilistic packet mode scheduling* and we describe it in section III. Section IV describes our second method, *deterministic packet mode scheduling*, which obtains determinism (hence eliminates reassembly buffers) by reducing scheduler independence. Section V compares qualitatively our schemes to the original bufferless crossbar packet mode scheduling. Section VI presents our simulation results, evaluating the performance of our schemes and comparing them to previous systems. We show that probabilistic packet mode scheduling reduces total delay compared to the system with pure SAR; for a traffic pattern with average packet size 550 Bytes, the average packet delay (weighted with packet size) improves by more than 80% for loads up to 50%. Deterministic scheduling achieves performance very close to buffered crossbars without SAR (those that require very large crosspoint buffers). Furthermore, deterministic scheduling in buffered crossbars performs always better than packet mode scheduling in bufferless crossbars. Probabilistic may be superior to deterministic scheduling depending on traffic patterns. In the next section II, we start by defining the queuing and scheduling architecture under study.

II. SYSTEM ARCHITECTURE

The baseline architecture considered in this paper is the one described in our previous work [11]. This section lists its key features for the purpose of completeness. We assume a

¹the timing diagrams of fig. 1 were drawn ignoring the delay (assuming zero delay) of the line card and scheduler logic: a cell can be switched in the same time slot when it arrives; a packet departure may start in the same time slot when its last cell is known to have been scheduled.

classical buffered crossbar switch with VOQ's in the ingress path of the linecards, and one, small, FIFO buffer at each crosspoint in the switch core. External packets are converted into segments in the ingress path and they are switched through the crossbar. The probabilistic scheduling scheme needs small, per-flow buffers in the egress path of the linecards, where the packets are reassembled before they are transmitted on the line. With the deterministic method the packets need no reassembly: they can be directly transmitted from the crossbar output port on the line.

The size of the internal segments is determined by the occupancy of each VOQ and by the maximum internal transfer unit, s – a system parameter. For queue occupancies above s , the transfer unit consists of the first s bytes; for smaller queue occupancies, the entire queue contents form a single transfer unit – a variable-size segment, without any padding overhead; see fig. 3. This segmentation policy results from the memory management operations described in [11]: congested flows have VOQs in DRAM, and they are served at DRAM block granularity; DRAM blocks coincide with maximum-size segments. Uncongested flows bypass the DRAM, and are held in SRAM. A maximum segment size of 256 to 512 Bytes is small enough for crosspoint buffers to have a very reasonable cost², and at the same time is large enough for modern SDRAM to easily sustain its peak throughput when transferring exclusively such maximum-size segments [11].

Schedulers are placed (i) on the ingress path of each linecard (IS_i), to resolve input contention; (ii) at each output port of the crossbar (OS_j), to resolve output contention; and (iii) on the egress path of each linecard, serve contending reassembled packets, when packet reassembly is required. The egress schedulers are simple round-robin schedulers. The operation of IS_i and OS_j is the subject of this paper, discussed in sections III and IV.

We *never* use internal speedup (core overspeed), since we do not need it to achieve top performance; we consider this to be a key feature of buffered crossbars. Internal speedup increases power consumption, and thus limits port and line-rate scalability. Hence, we always assume that the external and the crossbar links run at the same rate. We assume control lines, separate from data lines, running from the crossbar to the linecards and carrying the credits of the flow control protocol, which is used to prevent crosspoint buffers from overflowing. The bandwidth of each control line suffices for the transmission of one credit during each minimum-segment time. The crossbar has credit queues, used to buffer credits resulting from segments that depart at about the same time from the same crossbar row; contenting credits are served round robin. This paper considers a single priority (class of service) level, hence a single queue per crosspoint (each input i - output j pair defines a single flow); see [12] on how a system with just twice as much buffer space per crosspoint can effectively support multiple priority levels.

² $\lceil \frac{RTT \times R}{s} \rceil \times s$, where R is the line rate; for usual RTT values, this reduces to one maximum-size segment per crosspoint buffer.

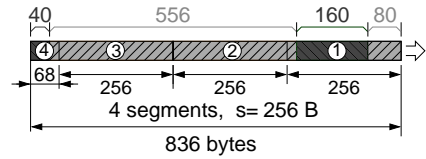


Fig. 3. Segmentation to variable size, multipacket segments

III. PROBABILISTIC PACKET MODE SCHEDULING

A. Detecting pairings and entering packet mode

We say that the classical buffered crossbar scheduling process determines an (i, j) pairing when a segment of a packet P is being written to crosspoint buffer (i, j) by input i while the same or a previous segment of P is concurrently being read by output j from the same crosspoint buffer. When such a pairing occurs, under some appropriate timing constraints, the input scheduler IS_i and the output scheduler OS_j will keep serving the same flow f_{ij} until the whole packet P has been completely forwarded to the crossbar fabric and to the egress card respectively. We will say that the schedulers operate in *packet mode* for packet P of flow f_{ij} .

For example, in fig. 3, a packet mode transmission may start for the 556 Byte packet at the time the input is writing segment 2 or segment 3 to the relative crosspoint buffer, while the output is reading segment 2 or segment 3 from the same crosspoint buffer. We assume that the input and output schedulers serve flows at segment granularity, so if a packet mode transmission is initiated for the 556 Byte packet, the 40 Byte packet will get a free ride to the egress. Similarly, the first bytes of the 556 Byte packet got a free ride to the egress during the transmission of segment 1.

In order to ensure correct operation, both IS_i and OS_j must observe a pairing *in time*. The decision of OS_j to operate in packet mode for a packet of f_{ij} presumes that IS_i keeps writing the rest of the packet to the crosspoint buffer (i, j) , so that no buffer underflow occurs. Symmetrically, the decision of IS_i to operate in packet mode for a packet of flow f_{ij} presumes that OS_j keeps reading from the same crosspoint buffer (i, j) , so that no buffer overflow occurs. Of course, in case a packet is completely stored in a crosspoint buffer, or in case its last fragment is currently being written, OS_j can enter packet mode scheduling independent of IS_i . Similarly, IS_i can enter packet mode scheduling independent of OS_j when there is enough room in the crosspoint buffer for the entire part of the packet which is pending at the corresponding VOQ. Actually, the problem of having two schedulers entering packet mode scheduling at the correct time is a *synchronization* problem, resembling the traditional “producer-consumer problem” of operating systems. Below we describe the proposed solution.

OS_j can infer a pairing at the time of its occurrence: it just needs to observe both the read and write enable signals of the crosspoint buffer, while also checking that the same packet is being read and written. An input scheduler cannot observe the pairing sooner than half RTT from the moment it occurs.

Since the input does not normally see the output decisions, it needs a *special notification* from the crossbar in order to know the occurrence of the pairing.

We claim that if an (i, j) pairing occurs at time t_r , while the crosspoint reads a segment s_k of a packet P and writes a segment s_{k+n} of P , it is safe for OS_j to enter packet mode scheduling for P if and only if:

$$t_r - t_w \leq \Delta t_s - RTT \quad (1)$$

where t_w stands for the time s_{k+n} starts being written to buffer (i, j) and Δt_s for the maximum segment transmission time. When the segment size is smaller than the maximum, it contains the entire tail of the packet and thus OS_j can enter packet mode scheduling independent of IS_i . If (1) holds, IS_i is guaranteed to observe the pairing before the transmission of segment s_{k+n} is completed. Note that RTT includes the decision making latency of IS_i , so this scheduler will see the notification in time for it to enter packet mode.

On the other hand, if $t_p - t_a > \Delta t_s - RTT$, the notification will arrive at the input after the segment transmission is completed. In the meanwhile, IS_i may have started serving other flows and thus it is possible that it will not respond to OS_j in time. We name the time interval $t_p - t_a$ *synchronization distance* - SD - and its maximum allowed value for a pairing to lead to packet mode transmission is $SD_{max} = \Delta t_s - RTT$. SD_{max} is graphically shown in the “space-time diagram” [13] of fig. 4. Our method works only as long as the round-trip time is smaller than the maximum segment transmission time. We consider this to be achievable in usual, realistic systems, as e.g. with 512B maximum segment size at 10Gb/s and with RTT below 400ns [10].

Following the above discussion, in the proposed method, the input and output schedulers toggle between two modes. In the *segment mode*, they serve flows in a (weighted) round robin fashion, as in the classical buffered crossbar scheduling. In the *packet mode*, they keep serving the same flow until the whole packet has been forwarded. An output scheduler transits from segment to packet mode when a pairing occurs and the synchronization distance is smaller than the maximum allowed. At the time it transits to packet mode, a pairing notification is sent to the input counterpart scheduler. An input scheduler transits from segment to packet mode when a pairing notification from the crossbar core arrives. Note that at most one pairing notification arrives at an input while the relative scheduler is in segment mode and no pairing notification arrives while the scheduler is in packet mode. Both input and output schedulers transit from packet to segment mode scheduling when the packet has been totally forwarded. We assume that both input and output schedulers serve flows at segment granularity.

Once the schedulers have entered packet mode scheduling, their decisions can be predicted and thus the packet can start being transmitted from the egress to the switch output port before it is completely stored at the reassembly buffers. What is needed is that the output scheduler communicates its mode of scheduling to the egress path.

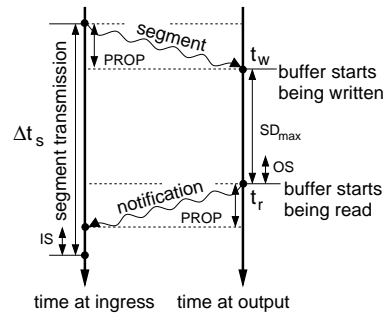


Fig. 4. Maximum allowed synchronization distance.

B. Piggybacking Control Signals

A traditional buffered crossbar exchanges one control signal per data segment: a credit is sent from the crossbar to an ingress linecard for each data segment that departs from the crossbar to an egress linecard. By contrast, our system may need up to three control signals per data segment: (a) pairing notification to the ingress linecard, (b) packet mode notification to the egress for cut-through purposes, and (c) credit to ingress. Fortunately, we can piggyback the notifications (a) in the credit signals, and notifications (b) in the data segment headers. Piggyback (a) has a penalty: we lose the opportunity to notify some pairings because when an input starts writing to an output after that output has started reading, the credit has already departed from the crossbar core. To partly overcome this inefficiency, the output scheduler could “look behind” it after each packet segment transmission is completed: whenever it fails to enter packet mode scheduling, due to the synchronization distance constraint, it “revisits” the same crosspoint during its subsequent decision slot and dequeues from it once more if a pairing occurs.

RTT should be increased to include the delay the credit may suffer in credit queues. If the credit scheduler always gives priority to credits carrying a pairing notification, then this delay is at most one credit time. Thus, the maximum nominated synchronization distance should be reduced by a credit time. Alternatively, the output scheduler should be notified that the credit was delayed and cancel its packet mode decision and the pairing notification to the input should be dropped.

C. Discussion

The proposed scheme is a combination of packet and segment mode scheduling. Under light loads, when input and output contention is rare, the buffered crossbar almost always pairs an input to an output port, packet mode transmissions are very likely to succeed and our method is very efficient. As input load increases and flows become congested, the schedulers are more likely not to synchronize themselves, the percentage of packet mode transmissions decreases and our method’s effectiveness degrades. Simulation results in section VI-B confirmed these hypotheses. When the schedulers are not synchronized, some packets - or a part of them - are transmitted in segment mode possibly interleaved with segments of other

packets from/to other links. Thus, egress reassembly buffers are needed to collect these segments. Since no guarantee for packet mode transmission is provided, the reassembly buffers must be as large as the respective ones in segment mode scheduling: one maximum packet memory space per flow, per egress line card.

The importance of the proposed scheme lies in that it provides the opportunity for cut-through transmission. Cut-through is beneficial under light loads since it greatly reduces packet latency.

Even though the egress buffer size requirement of our method could be considered a drawback compared to packet scheduling in bufferless crossbars, its independent and distributed nature are appealing.

IV. DETERMINISTIC PACKET MODE SCHEDULING

A. Scheduling Operation

The protocol described in section III spends egress reassembly buffers to gain scheduling independence: an output scheduler is free to start the transmission of a packet even if the packet is partly stored in the corresponding crosspoint buffer and there is no guarantee that the input counterpart scheduler will respond in time with the continuation of the packet. If we wish to eliminate the egress buffers, we can modify the scheduling of buffered crossbar: an output scheduler should start serving a flow f_{ij} whose head packet is partly stored in the corresponding crosspoint memory, when it is guaranteed that its input counterpart scheduler IS_i will start writing to buffer (i, j) no later than the time OS_j finishes transmitting the part ΔP of the packet already stored there; Δt refers to this time interval. Thus, when Δt is guaranteed to satisfy $\Delta t \times R \leq \Delta P$, where R is the line rate, we can deterministically synchronize the input and output schedulers. As a result, each packet may be transmitted in packet mode and thus the reassembly buffers can be eliminated.

To guarantee that $\Delta t \times R \leq \Delta P$, the buffered crossbar is scheduled as follows.

1. A flow f_{ij} is eligible by the output scheduler OS_j if and only if a packet is completely stored in the crosspoint buffer (i, j) or $\Delta P \geq (RTT + \Delta t_s) \times R$. In this inequality, Δt_s represents the max-size segment time: when input i receives the pairing request, it may delay honoring it for a time interval up to Δt_s , because it is busy transmitting a segment to another crosspoint. Each output scheduler serves the eligible flows round robin at packet granularity. An output scheduler asserts a flag in the flow control credit (as in the probabilistic method) each time it starts the transmission of a partly stored packet, requesting synchronization with the corresponding input.
2. Before an output scheduler OS_j starts the transmission of a partly stored packet at crosspoint (i, j) it must acquire a *lock* associated with input i . If that fails, because that input is currently connected to another output, the output scheduler proceeds serving the next eligible flow in the round robin schedule. Notice that the lock acquisition

is an operation entirely *internal* to the buffered crossbar chip, and does not involve any transaction with the ingress linecards. As long as the lock for an input i has been acquired by an output scheduler OS_j all flows $f_{ik}, k \neq j$ having a partly stored packet at the crosspoint (i, k) are ineligible until the lock is released.

3. An ingress linecard receiving a synchronization request - inferred by the relative bit at the credit packet - gets synchronized to the requesting output right after its current segment transmission, if there is one, or right away if it is idle. If no synchronization request is received, the input scheduler schedules flows in a round robin fashion. Whether synchronized or not, the input schedulers schedule at segment granularity.

With constraint (1) we impose that even if IS_i has initiated a transfer to output k at the time a request for synchronization with OS_j has arrived, it will be able to respond to OS_j in time. With constraint (2) we guarantee that there are no more than one requests arriving simultaneously at an input and that no requests arrive while an input is synchronized with an output.

Note that the deterministic method does not impose a constraint on the *RTT* relative to the maximum segment size, like the probabilistic method does. However, each crosspoint buffer should be large enough in order for at least $(\Delta t_s + RTT) \times R$ bytes of the packet to fit there. In case the *RTT* is large, it is possible that an input has finished the transmission of the last fragment of the packet while an output requests synchronization for that packet. To avoid synchronization for different packet transmissions we associate an *id* with each credit, which is the same with the one of the corresponding packet. If an input scheduler receives a synchronization credit corresponding to a packet already departed from the VOQs, the scheduler simply discards the synchronization request.

As mentioned above, in this method the input scheduler serves flows at segment granularity. On the other hand, output schedulers operate at packet granularity: each output scheduler keeps serving the fragments of a packet³, until the end of the packet. The flow control credits are released each time such a fragment is transmitted and corresponds to the size of that fragment.

B. Discussion

At first glance, deterministic packet mode scheduling resembles the classical request-grant-accept scheduling algorithm for bufferless crossbars. Resemblance concerns “large packet” transmissions: the transmission of packet segments to the crossbar core can be considered as the request phase for packet mode transmissions, the output scheduling as the grant phase while the lock acquisition as the accept phase. The crucial observation that discriminates our scheme from the classical bufferless crossbar scheduling is that a *sufficient* first portion of a packet should be stored at a crosspoint buffer in order for the corresponding flow to become eligible for the output scheduler.

³for example the four fragments of the 556 Byte packet in fig 3, which are spread in four segments

If k is the fraction of the maximum packet P that can be stored at a crosspoint buffer (i, j) , input scheduler IS_i is allowed to be “unfaithful” to output j , transmitting to one or more other outputs, for a time interval up to $\frac{k \times P}{R} - RTT$, while j has already started a transmission for packet whose tail resides at input i . Furthermore, it can be easily shown that at most $\frac{1}{1-k}$ output schedulers are allowed to be synchronized with the same input at the same time. Thus, the greatest the part of the packet that can be stored at a crosspoint buffer, the greatest the decoupling of the schedulers. In the extreme case that the whole packet can be stored, the schedulers are completely independent. In this paper, we assume that $k = \frac{(\Delta t_s + RTT) \times R}{P}$ and thus k is a little bit larger than zero, assuming $RTT < \Delta t_s$ and $\Delta t_s \times R \ll P$. So, at most one output may be synchronized with an input. However, inputs are decoupled from outputs in the very short run, since they can still transmit to an output other than the one synchronized to them, and for an interval equal to the maximum segment time. Thus input-output pairings for packet mode transmissions are still configured in an approximate manner.

C. Relation to Memory Management Scheduling

There is an interesting analogy between packet mode scheduling for buffered crossbars and the scheduling proposed in [14] for the memory system of ingress line cards. In [14], the bulk of each VOQ is held in long-latency DRAM, while the head (and tail) of the VOQ is held in low-latency SRAM; a memory management controller (scheduler) tries to keep all VOQ head SRAM buffers non-empty, so that these can serve the (unpredictable) requests by the crossbar scheduler. The analogy is as follows: VOQ’s in long-latency DRAM in [14] are analogous to our VOQ’s in the ingress linecards; VOQ head SRAM buffers in [14] are analogous to our crosspoint buffers; the memory management controller of each ingress linecard in [14] is analogous to the input scheduler in each of our ingress linecards; the (central) crossbar scheduler in [14] is analogous to our output schedulers.

In spite of the analogies between the resources to be scheduled, our scheduling algorithms differ from those of [14]. If we were to apply the memory management algorithm of [14] (*Earliest Critical Queue First - ECQF*) to our input schedulers, we would have to delay the decisions of the output schedulers by approximately $N \times \Delta t_s$ byte times, where Δt_s is the maximum segment time; the dequeue requests of each output scheduler would suffer a worst case latency of around $N \times \Delta t_s$ byte times which would impose the requirement for egress buffering.

The amount of SRAM for VOQ heads, per linecard, in [14], is approximately the same as the crosspoint buffer space, per crossbar row, in our scheme. The difference is that we assume complete memory partitioning among the flows, while [14] assumes complete sharing. Sharing is needed by *ECQF* in order to allocate more memory space to the queues that need urgent replenishment than the less “critical” queues. In an extended version of [14], the authors also study the requirements for the VOQ head SRAM size when buffer space is com-

pletely partitioned between flows [15]. Then, the buffer size requirements are larger than the requirements of our method⁴. A performance comparison between our scheduling algorithm and the memory management algorithms in [14][15] when they are used for packet mode scheduling in buffered crossbars gives rise to future work.

V. PACKET MODE SCHEDULING IN BUFFERED VS. BUFFERLESS CROSSBARS

In this section, we compare qualitatively our scheduling schemes for buffered crossbars to the original packet mode policy for bufferless, input queued crossbar switches. A quantitative comparison appears in section VI.

The first difference concerns packet size granularity. Crosspoint buffers allow the input and output schedulers in the crossbar to operate independent of each other, thus eliminating the requirement for synchronized decisions and fixed size cells: buffered crossbars can directly switch variable size packets with fine size granularity⁵ [10]. In this paper, we maintain scheduling independence, although to a reduced extent. As a result, our methods yield asynchronous operation and can switch variable size segments, with fine-grained segment size; padding overheads are eliminated.

By contrast, packet mode scheduling in bufferless crossbars [3] assumes cell granularity for packet size. The cell size is bounded below by the time needed for the crossbar scheduler to find a bipartite matching; this results in cell sizes usually in the range of 64 to 128 Bytes [14]. Fixed-size cells incur padding overhead, which requires internal crossbar *speedup*, by a factor of around 2 in the worst case⁶.

Second, we point-out a shortcoming of packet mode scheduling in bufferless crossbars that does not appear in buffered ones. Bufferless crossbars require *exact* pairings *at all times*. Consider a heavily loaded switch where all input ports are currently connected to one output each, and conversely all output ports are being fed by an input each. Connections are held for an entire packet duration. Consider a case where, when one of the connections is terminated –because the corresponding packet has been delivered in its entirety– *no other* connection happens to have terminated at the same time. Then, there is only a *single input* and a *single output* port that have become available for new pairing(s), hence the scheduler is forced to again pair the same input to the same output. Figure 5 shows an example of such a traffic pattern. Under such traffic, the scheduler is forced to maintain the crossbar configuration “locked” into a fixed set of connections (flows $f_{0,0}, f_{1,1}, f_{2,2}, f_{3,3}$ in the figure), thus starving all other flows.

By contrast, buffered crossbars allow *temporary* situations of “inexact” pairings, i.e. times when multiple inputs forward traffic to a same output or multiple outputs read packets from crosspoint buffers that had been fed by a same input at different times in the past. These periods of “inexact” input-output

⁴at least twice memory space per flow.

⁵size granularity equals to the crossbar datapath width (4 Bytes in [10]).

⁶e.g. with 65-Byte packets in a 64-Byte-cell switch.

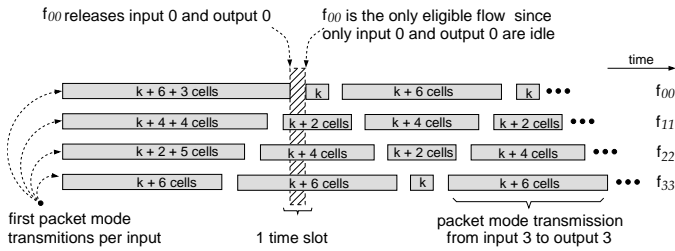


Fig. 5. A staircase-like traffic pattern that leads a packet-mode scheduler for a bufferless crossbar to “lock” in a fixed configuration. Assume a 4×4 switch; the figure shows the 4 flows that are constantly served; other flows with non-empty queues exist, but are never served.

matchings, allow buffered crossbars to escape from the above “locked” configurations.

Ongoing work: Although the above traffic scenario is very unlikely to occur, we are worried that the crossbar “locking” problem may appear in the short run even in usual, realistic traffic patterns. This would result in an increased standard deviation of packet delay; we are currently investigating this issue in our simulation study. We are also investigating whether buffered crossbars could also get “locked” in fixed configurations under some, even more rare, traffic patterns.

VI. PERFORMANCE STUDY

A. Method

We developed a byte-time-accurate simulator to model the buffered crossbar switch. It handles variable size packets at the switch interfaces and variable size transfer units in the core. It keeps track of time using the discrete event simulation approach [16]. We modelled the probabilistic scheme (section III-B), labeled *PPM* below, and the deterministic scheme (section IV), labeled *DPM*. In *DPM*, we assume that the time to acquire a lock is equal to the output scheduling time; the locks are granted to the requesting outputs in a round robin manner. We compare *PPM* and *DPM* to buffered crossbars with full, variable-size packets (no SAR, large crosspoint buffers) [10], labeled *VPS*, and to buffered crossbars with SAR and plain “segment mode” (not packet-mode) scheduling [11], labeled *SM*; we used our models from [10] and [11] for these comparisons. We also compare to bufferless crossbars, labeled *IQ*, that use input queueing (VOQ’s) and operate in cell-time granularity using packet mode scheduling; we simulated the packet mode modification of *iSLIP* from [1]; cell size is 64 Bytes and one iteration is assumed. Table I shows the default parameters for all the simulated systems.

For the buffered crossbar we assumed minimum packet size 40 Bytes and maximum 8 KB with 1 byte packet size granularity. For the bufferless crossbar we consider minimum packet size 64B (1 cell) and maximum 8KB (128 cells), with one-cell packet size granularity. Although this integer-cell size granularity favors *IQ* (padding overhead is a serious disadvantage of *IQ* - section V), we made this assumption in order to compare pure scheduling efficiency, factoring out padding overhead. Three packet size distributions were used:

- *bimodal* - 95% of the packets are minimum sized and 5% are maximum sized.
- *uniform* - packet size is uniformly distributed in the range between minimum and maximum size.
- *constant* - all packets have the maximum size.

For the buffered crossbar we assumed Poisson packet arrivals, while for the bufferless crossbar we modelled packets as bursts of cells, following the same approach as [3]. We chose the traffic models such that they offer insight on the scheduler performance under some “clear and extreme” traffic circumstances, rather than using just a single, “real life” traffic model. The maximum packets were 8KB large in order to show that our methods efficiently switch very large packets. In sections VI-B, VI-C we assume uniform destinations. In section VI-D we present results for unbalanced traffic.

The delay of a packet in a switch queue was defined as the time interval between the first byte of the packet arriving to the queue and its first byte departing from the queue. The total queueing delay of a packet in the switch was computed as the time interval between the first byte of the packet arriving to a VOQ and its first byte departing from the reassembly buffer, when the system requires egress reassembly buffers, or departing from the crossbar output port, when it does not. *Constant delays*, such as propagation and scheduling times, were subtracted. The delay was averaged over the number of packets with each packet delay contributing the same portion to the average (*average delay*), or over the number of bytes with each packet delay contributing proportionally to the packet size (*average weighted delay*). The latter measure emphasizes the delay of the large packets. The reported delay values are in units of 512 byte times (the transmission time of a 512B segment).

B. Probabilistic packet mode scheduling

We first report the results for bimodal packet size. In the buffered crossbar with pure SAR, for loads up to 0.2, the large packets were delayed in the reassembly buffers for around one packet store time (16 segment times). In the rest of the load range, this delay increased significantly, due to the packet interleaving in the switch core. With *PPM*, the delay at the egress buffers was almost zero for loads up to 0.5, because cut-through transmissions were possible at the egress path and the packet interleaving was limited. *PPM* becomes less efficient for higher loads: egress delay reduction, compared to *SM*, goes down from almost 100% for light loads to 70% for loads around 0.7, and to only 30% for a load of 0.95. The delay of small packets was almost the same in *PPM* and *SM*. Fig. 6 shows the total average *weighted* delay in the two systems. Total weighted delay was reduced by more than 80% for all loads up to 50%. When delay is not weighted with packet size, the improvement offered by *PPM* appears very small, since the delay of small packets - the greatest portion of traffic - is the same in both systems. Under uniform and constant packet size, similar results were seen. However, performance improvement was visible in terms of *average delay* as well because the average packet size is much longer

System	Xpoint Buffer Size (1)	Segment Size (1)	RTT (2)	Credit Time (2)	Scheduling Time (2)	$\frac{XbarRate}{Ext.LinkRate}$
PPM	1	[0.078-1]	0.95	0.033	0.031	1
DPM	3	[0.078-1]	0.95	0.033	0.031	1
SM	1	[0.078-1]	0.95	0.033	0.031	1
VPS	20	N/A	0.95	0.033	0.031	1
IQ	N/A	0.125	0	N/A	0.125	1

TABLE I
DEFAULT SIMULATION PARAMETERS

(1) in units of 512 Bytes, (2) in units of 512 Byte times. All switches are 16×16 .

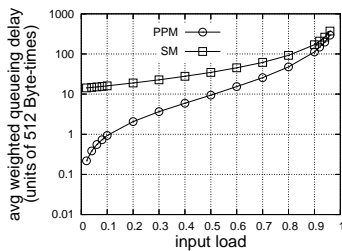


Fig. 6. Performance of probabilistic packet mode scheduling (PPM) compared to segment mode scheduling (SM). The traffic is uniformly destined and the packet size distribution is bimodal.

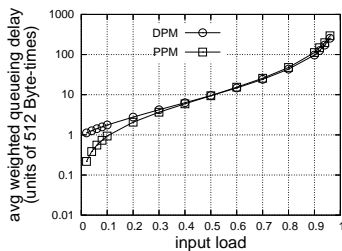


Fig. 7. Performance of deterministic packet mode scheduling (DPM) compared to probabilistic packet mode scheduling (PPM). The traffic is uniformly destined and the packet size distribution is bimodal.

than the segment size. Plots are not shown because space is not enough.

C. Deterministic packet mode scheduling

(i) Comparison to probabilistic packet mode scheduling:

Fig. 7 compares *DPM* and *PPM*, assuming bimodal packet size and using again the *weighted* delay metric. *DPM* imposes a crosspoint buffer delay and that is why it is slightly worse than *PPM* at light loads. If desired, one can patch-up this inefficiency by combining *DPM* with *PPM*. For loads between 0.4 and 0.8, the *weighted* delay was almost the same for both systems, while for loads between 0.8 and 0.95 *DPM* is slightly better; *PPM*'s extra delay is due to the delay in reassembly buffers. On the other hand, *DPM* slightly delays the small packets for loads greater than 90%. Using the average delay metric, the performance of *PPM* and *DPM* was almost the same. Under constant (maximum) packet size, for loads above 90%, *PPM* total delay is slightly less than *DPM*, even though the *PPM* delay includes the delay at egress. This shows the cost of lock acquisition in *DPM*: *PPM* exploits the full matching opportunities of the

buffered crossbar scheduling, while *DPM* forbids input contention when “large” packets are involved. Note that under bimodal packet size lock acquisition was almost transparent. Under uniform packet size, the lock acquisition cost of *DPM* almost equaled the delay of *PPM* in the egress buffers, and thus the performance was almost the same.

(ii) Comparison to buffered crossbar with full size packets:

Fig. 8 (upper row) shows a comparison between *DPM* and *VPS* in terms of average delay, and for all of the considered packet size distributions. For bimodal packet size, the delay curves are almost indistinguishable, but for loads above 90% *DPM* is slightly better. The reason is that *DPM* delay of small packets is shorter. In *VPS*, small packets face the transmission of the large packets which are ahead of them both at the input and at the crossbar output. The same happens in *DPM*, but to a lesser extent, since the input schedulers must be first synchronized with the output schedulers in order for a packet (mode) transmission to begin. Thus, small packets have greater chances to be transferred to the crossbar core before a packet transmission begins. For uniform or constant packet size, *VPS* was superior because of the increased buffer size, which is always greater than the average packet (burst) size, and the cost of the lock phase in *DPM*. Note that, for the simulated configuration, *VPS* uses 85% more memory in the crossbar chip.

(iii) Comparison to input queueing:

Figure 8 (bottom row) compares packet mode scheduling in buffered and bufferless crossbars, in terms of average delay, and for all of the considered packet size distributions. The superiority of *DPM* is more pronounced when the average packet size is small because the efficiency of the buffered crossbar scheduling is better exploited then.

D. Unbalanced Traffic

We used the destination distribution model described in [17] that is based on the Zipf law. Preliminary results suggest that *PPM* and *DPM* present worst case throughput slightly greater than 85% for all of the simulated packet size distributions and when the exponent k is close to 2. Final results will be included in the final version of the paper.

CONCLUSION

We proposed and evaluated a *probabilistic* and a *deterministic* packet mode scheduling scheme for buffered crossbar switches. The deterministic scheme performs virtually as well

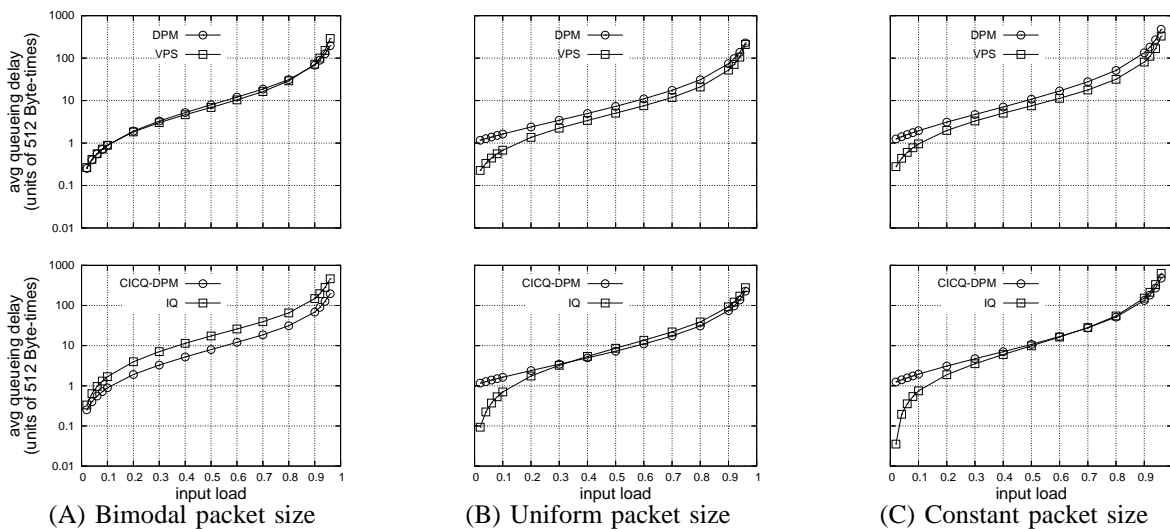


Fig. 8. Upper Row: Deterministic Packet Mode (DPM) Scheduling vs (Pure) Variable Packet Size Scheduling (VPS) in Buffered Crossbar. Bottom Row: Deterministic Packet Mode (DPM) Scheduling in Buffered Crossbar (CICQ) vs. Packet Mode Scheduling in Input Queuing (IQ). The traffic is uniformly destined.

as buffered crossbars that use no segmentation, and eliminates egress reassembly buffers like the latter systems do, while using crosspoint buffers whose size is only linked to the crossbar-ingress round-trip time –and not to the maximum packet size– hence can be much smaller than in buffered crossbars without segmentation. Performance is always better than bufferless crossbars with packet mode scheduling. Probabilistic packet mode scheduling performs even better than the deterministic scheme for some traffic patterns, while it performs similar for the rest, and allows independent output schedulers in the crossbar, but it does need the extra cost of egress reassembly buffers. Simulation results showed that for a traffic pattern with average packet size 550 Bytes the average packet delay (weighted with packet size) improves by more than 80% for loads up to 50%, compared to the system with blind segmentation and reassembly.

ACKNOWLEDGEMENTS

The authors would like to thank Nikos Chrysos for his early observations on the capacities of central schedulers in buffered crossbar switches, Enrico Schiattarella for his suggestions on simulation traffic patterns, and Alejandro Martinez for valuable discussions during the last few months. This work was performed within the project “Scaleable Intelligent Video Server System (SIVSS)”, supported by the European Union FP6 IST programme (contract 002075).

REFERENCES

- [1] Sung-Ho Moon, Dan Keun Sung: “High-performance variable-length packet scheduling algorithm for IP traffic”, *GLOBECOM 2001*, no. 1, Nov 2001 pp. 2666-2670
- [2] M. Ajmone Marsan, A. Bianco, P. Giaccone, E. Leonardi, F. Neri, “Packet scheduling in input-queued cell-based switches”, *IEEE INFOCOM 2001*, no. 1, April 2001 pp. 1085-1094
- [3] M. Ajmone Marsan, A. Bianco, P. Giaccone, E. Leonardi, F. Neri: “Packet-Mode Scheduling in Input-Queued Cell-Based Switches”, *IEEE/ACM Tr. on Networking*, vol. 10, no. 5, October 2002, pp. 666-678.

- [4] S. Mukherjee, F. Silla, P. Bannon, J. Emer, S. Lang, D. Webb: “A Comparative Study of Arbitration Algorithms for the Alpha 21364 Pipelined Router”, *Proc. of the ACM ASPLOS-X Conf.*, San Jose, CA USA, Oct. 2002, pp. 223-234.
- [5] X. Zhang, L. Bhuyan: “Deficit Round Robin Scheduling for Input-queued Switches”, *IEEE Journal of Selected Areas in Communications*, May 2003, pp. 584-594.
- [6] D. Stephens, H. Zhang: “Implementing Distributed Packet Fair Queuing in a scalable switch architecture”, *Proc. INFOCOM’98 Conf.*, San Francisco, CA, March 1998, pp. 282-290.
- [7] F. Abel, C. Minkenbergh, R. Luijten, M. Gusat, I. Iliadis: “A Four-Terabit Packet Switch Supporting Long Round-Trip Times”, *Proc. IEEE Micro Magazine*, vol. 23, no. 1, Jan./Feb. 2003, pp. 10-24.
- [8] N. Chrysos, M. Katevenis: “Weighted Fairness in Buffered Crossbar Scheduling”, *Proc. IEEE Workshop on High Performance Switching and Routing (HPSR 2003)*, Torino, Italy, June 2003, pp. 17-22.
- [9] K. Yoshigoe, K. Christensen: “A Parallel-Polled Virtual Output Queued Switch with a Buffered Crossbar”, *Proc. IEEE Workshop High Perf. Switching & Routing (HPSR 2001)*, Dallas, TX, USA, May 2001, pp. 271-275; <http://www.csee.usf.edu/~christen/hpsr01.pdf>
- [10] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, N. Chrysos: “Variable Packet Size Buffered Crossbar (CICQ) Switches”, *Proc. IEEE International Conference on Communications (ICC 2004)*, Paris, France, 20-24 June 2004, vol. 2, pp. 1090-1096.
- [11] M. Katevenis, G. Passas: “Variable-Size Multipacket Segments in Buffered Crossbar (CICQ) Architectures”, *Proc. IEEE International Conference on Communications (ICC 2005)*, Seoul, Korea, 16-20 May 2005, CR-ROM paper ID “09GC08-4”, 6 pages.
- [12] N. Chrysos, M. Katevenis: “Multiple Priorities in a Two-Lane Buffered Crossbar”, *Proc. IEEE Globecom 2004 Conference*, Dallas, TX, USA, 29 Nov. - 4 Dec. 2004, CR-ROM paper ID “GE15-3”, 7 pages; <http://archvlsi.ics.forth.gr/bufxbar/>
- [13] L. Lamport: “Time, Clocks and the Ordering of Events in a Distributed System”, *Communications of the ACM*, vol 21, no 7, 1978
- [14] S. Iyer, R. Kompella, N. McKeown “Analysis of a Memory Architecture for Fast Packet Buffers”, *IEEE - High Performance Switching and Routing*, Dallas, Texas, May 2001, pp. 368-373.
- [15] S. Iyer, R. Kompella, N. McKeown “Designing Buffers for Router Line Cards”, Stanford University HPNG Technical Report - TR02-HPNG-031001, Stanford, CA, Mar. 2002
- [16] Sheldon M. Ross: “Simulation”, Academic Press, 3rd Edition, 2001, ISBN 0125980531
- [17] Network Processing Forum (NPF): “Fabric Benchmarking Traffic Models”, available from http://www.npforum.org/benchmarking/fabric_bm.shtml