An FPGA-based Prototyping Platform for Research in High-Speed Interprocessor Communication

Vassilis Papaefstathiou, Giorgos Kalokairinos, Aggelos Ioannou, Michael Papamichael,
Giorgos Mihelogiannakis, Stamatis Kavadias, Evaggelos Vlachos,
Dionisios Pnevmatikatos, and Manolis Katevenis
Foundation for Research and Technology-Hellas (FORTH) – member of HiPEAC
Institute of Computer Science (ICS), Vasilika Vouton, Heraklion, GR 71110 Greece
{papaef,george,ioannou,papamix,mihelog,kavadias,vlachos,pnevmati,kateveni}@ics.forth.gr

Abstract

Parallel and multinode computing systems are becoming widespread and grow in sophistication. Besides simulation, rapid prototyping becomes important in designing and evaluating their architecture. We present an FPGA-based system that we developed and use for prototyping and measuring high speed processor-network interfaces and interconnects; it is an experimental tool for research projects in architecture. We configure FPGA boards as network interfaces (NI) and as switches. NI's plug into the PCI-X bus of commercial PC's, and use 4 links of 2.5 Gb/s/link as network connections; we can bundle these links together, at the byte or packet level, offering 10 Gb/s of network throughput. NI's implement DMA on the PCI-X side, and remote DMA and remote notification (interrupt or flag-setting) on the network side. We configured the switch boards as buffered crossbars operating directly on variable-size packets and featuring credit-based flow control for lossless communication. Multiple, parallel switches can serve the NI links using multipath routing; NI's resequence the out-of-order packet arrivals. All boards provide extensive support for monitoring, debugging, and measurement. Colleagues adapted the Linux OS for this platform, and used it for remote disk I/O experiments [1]. We report here on the platform architecture, its design cost and complexity, latency and throughput parameters, and buffered crossbar performance. We now work on remote queues and synchronization mechanisms.

1 Introduction

Chip and cluster multiprocessor systems are becoming widespread, while also growing in sophistication. To achieve efficiency, they strive, among others, for a tight coupling of computation and communication, and even propose customization of Network Interface features to meet particular application domain demands. Advanced features in the Network Interface influence the design of, or require support from the underlying interconnection network. Thus, a goal is the integrated design of network interface and interconnect features.

Evaluating an entire system architecture before it is built is very complex and requires approximations. Simulation and rapid prototyping are the available tools, each with its own pros and cons. Rapid prototyping is becoming increasingly important, owing to the availability of large field-programmable gate arrays (FPGA), which enable the design and operation of systems that approximate the actual ASIC designs with very high accuracy compared to simulators.

We present an FPGA-based system that we developed and are currently using for prototyping and measuring high speed processor-network interfaces and interconnects; it constitutes a research tool for European architecture projects. We report on the system architecture and performance, and on our design cost and experience. The prototyping platform, in its current form, is shown in figure 1. It consists of (currently 8) commercial personal computers (PC's) linked through our custom interconnect. An FPGA develop-

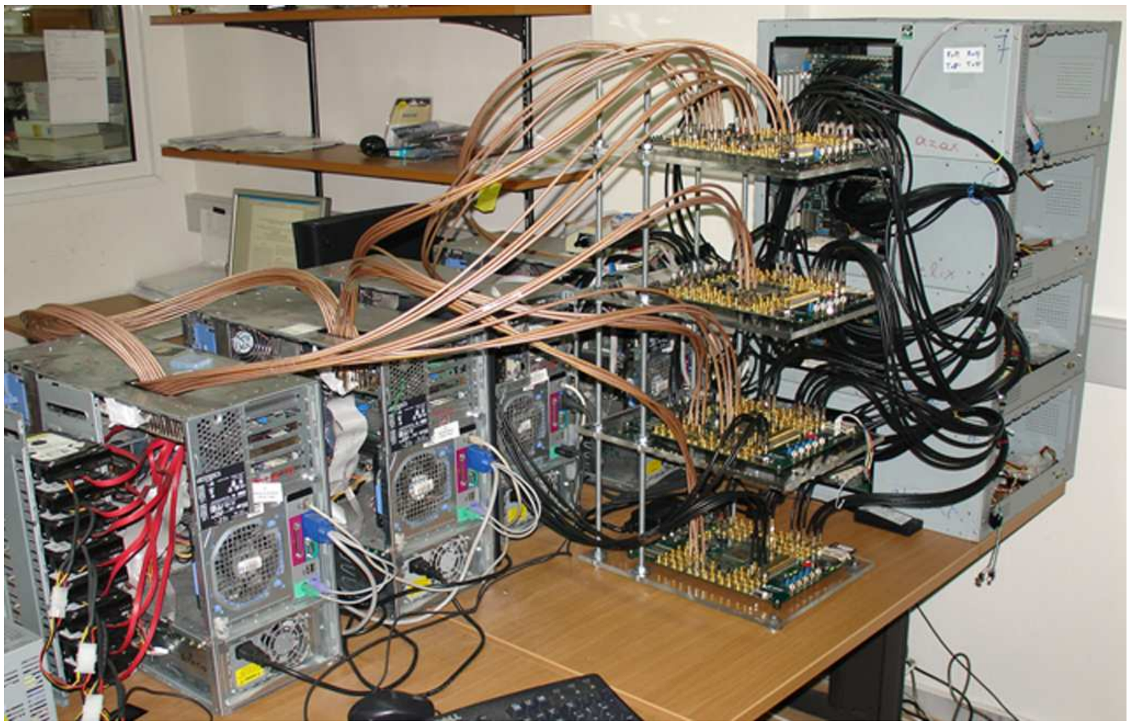

Figure 1. Photograph of the current system

ment board plugs into the PCI-X bus of each PC, and is configured as its network interface (NI). A number of additional FPGA boards (4 in the current system, arranged vertically in the middle) are configured as network switches. We currently use Xilinx Virtex II-Pro FPGA's, which offer a number of high-speed serial ("RocketIO") links each. The key features of this platform are:

- *High throughput network:* each (bidirectional) link operates at 3.125 GBaud, offering 2.5 Gbits/s of net throughput per direction [2]. Each NI connects to the network via 4 links. Bundling these physical links together enables the creation of 10 Gb/s connections. Two types of bundling are supported: byte-by-byte, where the bytes of each packet are evenly distributed to all links, and packet-by-packet, where each packet is transferred via a single link, but successive packets are distributed to all links.

- *PCI-X interface:* the NI communicates with the host processor via a 64 bit wide PCI-X bus, currently running at 100 MHz; higher speed grade FPGA's would allow 133 MHz operation. The corresponding peak throughput is 6.4 or 8.5 Gbits/s [3].

- *Remote Direct Memory Access (RDMA)* based operation for efficient communication; flexible arrival and departure notification mechanisms (selective, collective interrupts or flag setting); lossless communication via credit-based flow control; per-destination virtual output queues (VOQ) for flow isolation and quality of service (QoS) guarantees even under congestion.

- Extensive performance, debugging, and event logging counters for effective prototyping and evaluation.

- *Large valency switch:* the switch board features 20 bidirectional links; depending on the complexity of the internal logic, switches up to 20$\times$20 can be configured. In our "XC2VP70K" FPGA, we have been able to fit up to 12 switch ports so far (12$\times$12 switch). The 4 switches in parallel can offer up to 200 Gbits/s of network throughput (120 Gb/s in the current maximal FPGA configuration).

We are using this platform to experiment with system-level aspects of network interface and switch design. Colleagues in our Institute have used it for research in storage area networks [1]. The contributions of this paper are: *(i)* variable-size packet operation of a buffered crossbar switch demonstrated in actual operating hardware for the first time; *(ii)* correct event notifications in the presence of out-of-order packet deliveries caused by multipath routing (RDMA + header resequencing); *(iii)* reporting on 5 person-year hardware project, and its design, implementation, and test process and cost; *(iv)* describing a platform that is useful for European architecture projects.

In the rest of the paper, section 2 presents the NI architecture, section 3 describes the switch, and section 4 talks about supporting cache coherent NI's. Implementation details and metrics are given in section 5, with performance evaluations presented in section **??**. Section 6 discusses related work.

2 Network Interface (NI) Card

The Network Interface Card (NIC) is designed as a 64-bit, 100MHz PCI-X peripheral. It is based on a Xilinx Virtex II Pro FPGA, and it features 4 physical multigigabit transceivers that run at 2.5 Gbps (3.125 Gbaud). Its main components are shown in Figure 2.

2.1 PCI-X interface

The PCI-X interface implements the full PCI-X initiator and target functionality and interfaces to the rest of the NI by means of the DMA request queues. The PCI-X target interface provides the host processor with a memory mapped interface to: the configuration space of the card; the control, debugging, and performance counters of the NI; and to the DMA request queues. It supports 32 and 64-bit accesses in burst or non-burst mode and also implements the interrupt functionality of the card.

The PCI-X initiator interface provides the functionality of a DMA engine from/to the host's memory. It supports 32 or 64-bit wide bursts using physical PCI addresses, and its operation is controlled by the DMA engine described next.

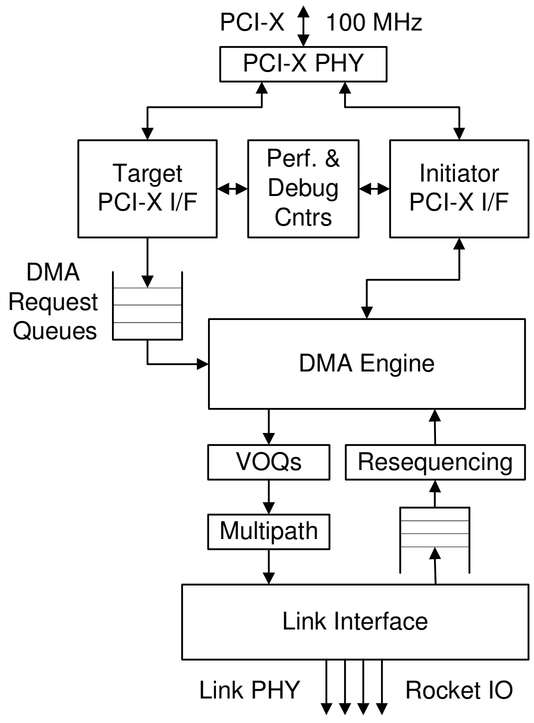

Figure 2. NIC block diagram

2.2 DMA Engine and DMA Request Queue

The DMA engine is the heart of both the outgoing and incoming portion of the NIC. Outgoing (remote write) transfer commands are queued in the memory-mapped *DMA Request Queue* by means of writing a **two 64-bit word** transfer descriptor via the PCI-X target interface. The first word specifies the local data PCI source address. The second word contains *(i)* a 32-bit remote host destination address (physical address) where the data will be transfered to; *(ii)* the size of the transfer, in 64-bit words (the maximum supported size is 512 words or 4096 bytes); *(iii)* the flow ID of the destination host (current support for 128 flows); and *(iv)* an "opcode" field that controls the completion notification options for the transfer, as described shortly.

We currently have 8 DMA request queues, one per destination host in the network. When congestion exists, multilane (per-flow) backpressure (credit-based flow control) stops the flow of packets (hence RDMA's) to some destinations, while those targeted to other hosts proceed unobstructed; separate per-destination request queues allow the latter to proceed while the former are blocked, thus preventing head-of-line blocking. Our current request queue size is 128 transfer descriptors per destination, implemented as a circular buffer in a statically partitioned 1024-descriptor memory. Besides decoupling the operation of the DMA engine from the processor, the request queue supports clustering of requests to the NIC: the host processor can write multiple transfer requests to the queue (and even write them in non-sequential order), while holding their processing back until a special "Start Flag" bit is set in the last one of the clustered requests; at that time, all clustered requests are released to the DMA engine for processing. One example for such use would be to prepare a scatter operation before the actual data are computed, then release the entire scatter when the data are available.

For packet transmission, the DMA engine reads the packet data from the specified address of the host memory, prepends the appropriate header and hands them to the VOQs block via a synchronization FIFO; for DMA operations larger than the maximum packet size, segmentation is handled by the VOQ block (section 2.5). The reverse path is symmetrical, except for using a single "lane"; upon reception, packet data are passed from the link interface to the DMA Engine via a reception queue and are then written to the host memory. Each outgoing VOQ (8 in our case) is 8 Kbytes of on-chip memory; there is a single incoming queue, of size 8 KBytes. The DMA engine also arbitrates between the incoming and outgoing paths, as to which one will be served next, on a per-packet basis.

2.3 Notification Mechanisms

The NI provides three notification options: *(i)* local notification; *(ii)* remote interrupt; *(iii)* remote notification. Local notification is used to inform the sending node that the packet was sent to the network: when so requested by a DMA descriptor, upon departure of this DMA transfer, the NI copies the tail pointer of the request queue to a prespecified location in host memory, using a single-word DMA write access. The processor can poll that location to determine how many transfers have departed (transfers from a single queue depart in-order), hence recycle their descriptor slots; for the processor, this is lighter than polling the tail pointer itself –a NIC control register– in I/O space.

Remote interrupts and remote notifications can be used to inform the receiving node that a packet has arrived. The former are traditional PCI interrupts, while the latter are similar to their local counterparts: they write (via single-word write DMA) into a prespecified address in the receiver's host memory. Local and remote notification options, in combination with the operation clustering option, allow for a drastic reduction of number and overhead of interrupts. For example, if the network provides in-order delivery, a large multi-packet transfer can use interrupt or notification only for the last packet. In a network storage application, this allowed a 3 to 4 times improvement in bandwidth [1].

2.4 Link Interface

On the link wires, packets are delimited by a *start-of-packet (sop)* control symbol (byte); this can be followed by zero or more credits, which are followed by the 64-bit packet header, a CRC-16 covering the header, the packet payload, and a CRC-32 covering the payload; one or more *comma* control symbols (clock synchronization idle characters) appears before the next packet's *sop*. The header consists of a 7-bit packet flow-ID, 5-bit operation bits, 10-bit are reserved for resequencing (see 2.6), 10 bit size and the 32-bit destination address at the target.

Up to four links can be bundled together to provide an aggregate bandwidth of $4 \times 2.5 = 10$ Gb/s. We support byte-by-byte and packet-by-packet bundling modes. The byte-by-byte mode distributes the bytes of a single packet evenly across all links, approximating a single 10 Gbps link very closely, but it can only be used in direct NIC-to-NIC connections. The packet-by-packet bundling mode sends each packet to a single link, balances the load across all links, and is suitable for multipath routing. This is a very flexible approach, however, depending on the network topology and guarantees, it may lead to the out-of-order arrival of packets sent via different paths (section 2.6).

Our credits are 16-bit wide, and consist of: *(i)* 1 bit to differentiate a credit from a packet header; *(ii)* 7 bits of flow identifier; *(iii)* 7 bits of credit value; and *(iv)* 1 bit of parity. The credit value is the LS part of a "cumulative number of words" count (QFC-like protocol). This QFC-like credit protocol relies on two counters on each communicating NIC: one counts the cumulative number of words transmitted to the network since power up and the other counts the cumulative number of words that have been forwarded from the network, again since power up. This second counter is transmitted and a simple comparison allows the link interfaces to determine the free space downstream. In order to shorten the credit encoding, we do not send the entire counters - 12 bits in our system - but only the 7 most significant bits. This approximation coarsens the granularity of credits to 32 words. The *Round-Trip Time (RTT)* in our system is calculated and measured to be:

$$RTT = (85cc + Cut\_Through\_Latency)$$

45 to 50 out of the 85 clock cycles (cc) are due to the RocketIO serializer-deserializer and transceiver; the rest account for link-to-DMA latency (about 5 cc), and credit granularity (32 cc). The cut-through latency depends on whether the receiving link resides on the switch or on the NIC. The cut-through latency of the switch is at most 8 clock cycles, while for the NIC it is PacketSize/4 cycles, due to the relative frequency of the communicating blocks.

2.5 Multiple VOQ support

The use of a single output queue for all outgoing traffic regardless of destination leads to head-of-line blocking resulting in significant performance loss. In order to avoid the head-of-line blocking effect multiple virtual output queues (VOQs) - at least one per potential destination - are implemented. The use of VOQs instead of a single queue greatly improves the NICs performance and localizes the effects of congestion.

The architecture of the VOQ handling system is based on previous research [4, 5]. However implementation-specific requirements led to a slightly differentiated design, which includes a novel packet processing mechanism. Figure 3 depicts the systems architecture. The thick arrows show the packet flow through the various modules.

Traffic is segmented in variable-size multi-packet segments and only the first segments of each VOQ reside on on-chip memory. When a VOQ becomes excessively large its body migrates to external memory (SRAM and/or DRAM) which is partitioned in blocks of configurable size and dynamically shared among the available VOQs through the use of linked-list structures implemented in hardware.

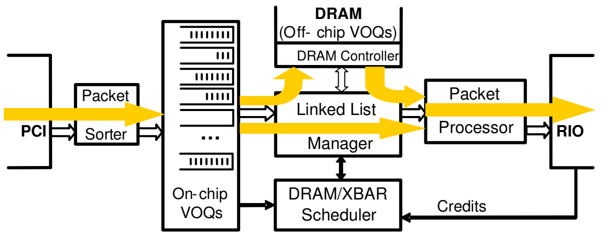

Figure 3. VOQs block diagram and flow

In order to avoid the delay/buffer cost of packet reassembly on the receiving end, a novel packet processing mechanism was introduced. Upon departure towards the link interface each segment - that may contain a part of a stream of packets chopped at arbitrary points - is handled by a packet processing module. Each segment is transformed into separate independent packets which can be immediately handled upon reception at the receiving end, instead of waiting in a buffer for reassembly purposes.

The system is fully configurable and can easily be adapted to specific implementation requirements. The number and width of VOQs, maximum segment size, size of on-chip memory, as well as other parameters can be manually set at synthesis time.

2.6 Multipath support

Inverse multiplexing [6] is a standard technique that allows k links of capacity C each to be combined together in order to implement a high speed link of capacity $k \cdot C$. In the general case, the load on each link can be switched (routed) to the destination independently. This implies that the granularity in which the original traffic is partitioned is at least per packet (or maybe per "flow"), since routing can be applied to whole packets only.

Inverse multiplexing is needed for the lowest-cost ($O(N \cdot logN)$) internally-non-blocking switching fabrics (Benes, Clos) to operate. For internally-non-blocking operation, the load must be *evenly balanced* among the parallel links, on a per-destination basis.

Such multipath routing may deliver packets out-of-order, at the destination. Owing to the use of DMA semantics (each packet carries its own destination address), packet payload can be delivered in-place in the host memory, avoiding the need for reorder buffers, the associated storage cost, and the cost of data copying. This solves the problem of delivering the correct data to the correct place, even if they arrived in scrambled order, but it does not solve the problem of completion notification. If data were delivered in-order, completion can be signaled by the last word being written into its place (e.g. the last word of the block is a 1-flag written into an originally-0 word). However, when packets can arrive out-of-order, the last address in the destination block can be written into before intermediate data have arrived.

Our current method to provide completion notification is to resequence packet headers. Notice that we economize on resequence buffer space by writing packet data into its destination address, and only keeping packet headers in the resequence buffer. After resequencing, we discard headers in-order until seeing a header that contains a notification flag; when this header is seen *after* resequencing, we are sure that all packet before it have been received and processed, hence the notification can be safely delivered. We use the resequencing protocol proposed by Khotimsky et al. [7]; it allows for detection of and recovery from single-packet losses. The sender side implementation (insertion of resequencing information in packet headers) conforms to per-destination queuing and per-path traffic partitioning used for inverse multiplexing. On the receiver side, per-sender and per-path queues are needed. Per-path queues can share space, and, because only headers are stored, their total size can be kept small. Scaling this scheme to thousands of communicating nodes may be difficult, because of the need to keep per-sender and per-path head and tail pointers at the receive side. We are examining alternative methods, based on packet counting, for scalability; one must allow for interleaved transmissions of groups of packets, while separately counting each group. The challenge is to come up with a proper programming interface that eases the assignment of each transfer to one of the (few) existing groups, and to later recycle group identifiers.

3 Switch

We implemented the switch on a Xilinx ML325 board [8], which provides twenty RocketIO SMA interfaces. The current FPGA configuration implements a *Buffered Crossbar* (Combined Input-Crosspoint Queuing - CICQ) switch, in order to demonstrate, on actual working hardware, the architecture that we have been working on in recent years [9]. Buffered crossbars use small buffers at each crosspoint, and feature *(i)* simple and efficient scheduling, *(ii)* variable-size packet operation, and *(iii)* peak performance without needing any internal speedup.

Figure 4 depicts the internal structure of a 4$\times$4 buffered crossbar switch. Incoming packets are delivered to the appropriate crosspoint buffers according to their headers and the output scheduler (OS) is notified. If sufficient credits exist and the outgoing link is available, the output scheduler for that link selects a crosspoint buffer for transmission. As packet bytes are being transmitted to the output, the credit scheduler (CS) generates the corresponding credits that will be transmitted back to the source of the packet. These credits are multiplexed with the other packets destined to the initial source (section 2.4). Next, we describe these in detail.

Crosspoint Buffers: each of them consists of a 2 Kbyte dual ported show-ahead FIFO implemented in one BRAM. The crosspoint control logic is very simple: a head and tail pointer, a small FSM triggered by start-of-packet and end-of-packet indicators, and a synchronizer to notify the output scheduler of packet arrivals (the FPGA places the top half and the bottom half RocketIO's in two separate clock domains, hence many crosspoints have different input and output clock domains). Datapaths are 32 bit wide; clock frequency is 78.125 MHz; enqueue and dequeue throughput are one word per cycle, each. The minimum-size packet payload is 24 Bytes (6 clock cycles), and the maximum-size packet payload is 512 Bytes (128 cycles). The packet header, CRC, and framing overhead on links wires is 16 Bytes, as discussed in section 2.4.

Output Schedulers (OS): they keep track of the number of packets enqueued in each crosspoint of their column, and they select in a round-robin fashion the next crosspoint to be served among the crosspoints with non-zero occupancy counts. The round robin policy is implemented using a priority enforcer, which operates in a single clock cycle. The latency from crosspoint logic to OS decision is 5 clock cycles, most of which (about 3 clock cycles) is the synchronization de-

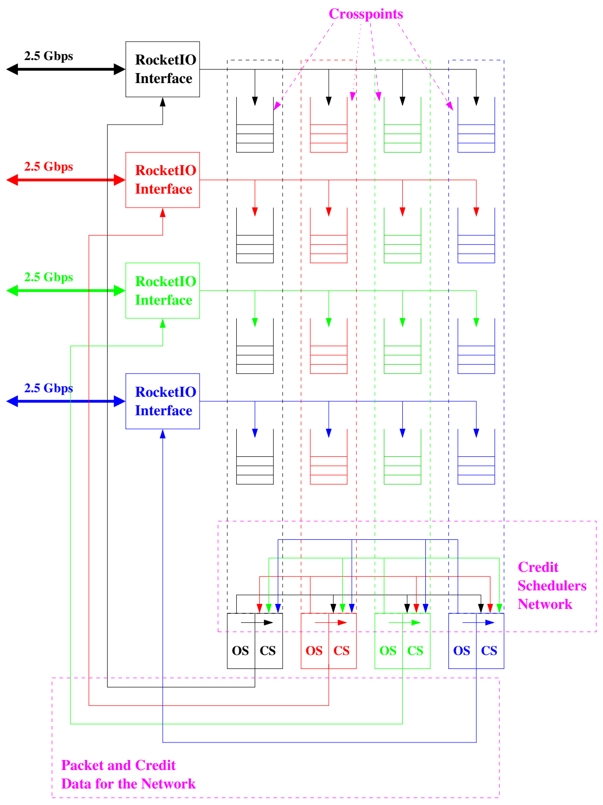

Figure 4. The internal structure of a 4x4 Buffered Crossbar

lay. Since this latency is less than the minimum-packet time (40 Bytes on the wire = 10 clock cycles), cut-through operation is supported even for minimum-size packets. The OS hides scheduling latency and supports back-to-back packet transmissions by initiating scheduling 3 clock cycles before the end of a previous packet transmission. The first word of the selected packet appears on the column multiplexor and the OS reads the packet size in it. For the selected packet to be dequeued and transmitted, the OS must have sufficient credits for the outgoing path (credits are single-lane at this stage –switch outputs do not currently have per-flow queuing or backpressure). When the transmission starts, the OS sends the packet size to the credit scheduler (CS) of the input where the packet came from.

Credit Schedulers (CS): they are associated with a specific input port each; each of them maintains per-output counts of the number of bytes that originated from this input and have departed through that output. We use a QFC-like protocol [10]: counter wrap-around and infrequent credit losses do not affect correct protocol operation. The CS is informed about departing bytes from the output schedulers, which, however, may reside in a different clock domain; synchronization costs about 3 clock cycles in latency. The CS also interacts with the link interface, and, when requested (on packet boundaries), provides the credit data to be transmitted, one credit at a time. Multiple credits, corresponding to multiple flows, share the same link: the CS transmits them in round robin order. Round robin transmissions are applied first to the active flows (those whose credits have changed since their last transmission); on a much longer time scale, all credit counter values are transmitted, to guard against potential credit losses on the wire (corrupted bit transmissions).

4 In preparation for future Coherent NI

Future network interfaces (NI), especially in Chip Multiprocessors (CMP), will be tightly coupled to the processor in order to provide low latency communication, as opposed to the currently predominant long-latency coupling through the I/O bus. When the network interfaces to the processor at the same level as caches do, we need a *cache-coherent NI* [11]: incoming data may have to be delivered to the cache, rather than to main memory, to avoid long fetch latencies; alternatively, when data are delivered to memory but stale versions of the same words reside in the cache, the latter must be invalidated. In other words, the NI must participate to the cache coherence protocol.

We plan to prototype systems with coherent NI's in the future, using the PowerPC processors that are embedded in the Xilinx Virtex-II Pro FPGA's as the host processors, i.e. *not* using Pentium's as the host processors and talking to them through a PCI interface. In preparation for such prototyping, we need processors with data caches that can run a coherence protocol. Unfortunately, the data caches of the PowerPC's inside Virtex-II Pro FPGA's do *not* run a coherence protocol.

For this reason, we have implemented data caches equipped with a MESI-like coherency protocol, inside Virtex-II Pro FPGA's but outside their embedded PowerPC's. Our two caches inside an FPGA connect directly to the Data Side PLB interface of each of the two PowerPC processors, and not to the PLB as peripherals, in order to save some cycles. We also implemented a switch-based snooping-bus-like interconnect between the two caches. Each cache uses this connection in order to broadcast coherence requests and receive data. It also snoops the interconnect activity, in order to respond to coherence requests whenever is needed. Accesses that miss in both coherent caches are forwarded to the DDR controller that manages the 256 MB external DDR memory. Finally, each processor has its own PLB bus, in order to have access to instructions and private data. Connection to PowerPCs DSPLB interface is shared between the PLB bus and the coherent memory system. A request is served by either the coherent memory system or the PLB bus depending on the address being accessed. Our coherent caches support update requests, too; coherent NI's will use them to directly deliver data in the cache.

5 Implementation

The prototypes of the system have been implemented in the Verilog and VHDL hardware description languages using Xilinx tools Xilinx for FPGA development. Our designs have been synthesized, mapped, placed and routed in the Xilinx ISE environment. Before mapping each design in the FPGA we have made numerous exhaustive functional simulations using the Cadence's LDV Suite and Mentor's ModelSim.

5.1 Design and Hardware Cost

The functional simulations helped us eliminate the most obvious errors of our designs, although debugging in the system platform was inevitable. The tests in the system platform revealed several hidden bugs that could not be found during simulations, since we cannot have accurate models of the real system components. For instance the PCI-X bridge behavior and the host processor intervention are events that cannot be modeled. Therefore, we have long debugging cycles in the system and spent several months on this.

Table 1 shows the person-months spent in the blocks of the design along with their complexity. The most complex block of the system is the PCI-X interface & DMA engine, that required in depth study of the PCI-X protocol and conformance with the standards. The PCI-X and DMA blocks were the most difficult and time consuming to debug since we had to deal with the interfaces with the host computer, in a system with high volatility and uncertainties. Furthermore, we have spent several months for the RocketIO link interface to reach a stable version, since we had to deal with problems ranging from the physical interfaces (connectors and cables) to high level block interfaces. It is important to note that the numbers that we present for the buffer crossbar switch concern only the porting of the design that was provided by [10] to the specific FPGA environment and the specific RocketIO link interfaces.

Table 2 presents the hardware cost of the system blocks. The numbers refer to the implementation of the designs in the Xilinx Virtex II PRO FPGA with the back-end tools provided by Xilinx. The Monitoring and Debugging block is one of the biggest blocks in terms of area because it contains a suite of benchmark, performance and monitoring sub-blocks that occupy many LUTs and BRAMs and represent approximately 33% of the complete design. The VOQs block is also area demanding block because it involves many BRAMs to be used as packet buffers and considerable logic for their associated state. Notice that the *Equivalent Gates* count includes the memory (BRAM) bits counted in terms of gates.

5.2 PCI-X microbechmarks

We use hardware cycle counters at the NIC level to examine the behavior of the Host-NIC DMA engine, namely the PCI-X Target Interface. For single word PCI-X write transactions, on the order of 10 PCI-X cycles are required. Therefore, initiating a single RDMA write operation (writing a transfer descriptor) requires about 40 PCI-X cycles, or about 400 ns. Leveraging the write combining processor feature, we can write a burst of 64 bytes of data in 24 PCI-X cycles which translates into 4 transfer descriptors. This feature gives a 6x improvement over the simple case which would need 160 PCI-X cycles. Naturally, the cost of the write combining feature is the latency of the data in the write combining buffer but it saves significant cycles in the PCI-X bus.

For a DMA write transfer of 4 KBytes (PCI-X maximum size) to the host memory with 64-bit data

Block	Lines of code	Development	Debugging	Person-Months
Previous PCI Version	8000	5 months	8 months	20
PCI-X Interface - DMA Engine	6000	3 months	6 months	12
RocketIO Network Interface	1200	2 months	5 months	10
Multiple VOQs	1500	2 months	3 months	6
Multipath Support	1000	3 months	3 months	6
Buffered Crossbar switch	3300	3 months	3 months	6
Totals	8000 + 13000	$\sim$ 1 year	$\sim$ 1 year	20 + 40 = 60

Table 1. Development and Debugging time of the basic blocks

Block	LUTs	Flip Flops	BRAMs	Equivalent Gates (K)
PCI-X Interface - DMA Engine	2500	1400	22	1200
RocketIO Network Interface	1800	400	0	20
Multiple VOQs	4100	2100	37	2500
Multipath Support	2800	1200	20	600
Monitoring - Debugging Support	2900	2100	32	2200
Totals NI	14100	7200	111	6520
Buffered Crossbar Switch 8x8	15800	13300	64	4400

Table 2. The FPGA Hardware cost of the basic blocks

phases, we measured a delay of 570 cycles out of which only 512 actually transfer data (90% utilization of PCI-X cycles). The remaining 58 cycles are attributed to arbitration, PCI-X protocol phases and the occasional disconnects. For 4 KByte DMA read transfers from the host memory, we measured a delay of 592 cycles yielding a utilization of 87% PCI-X cycles. In every DMA read transfer, 50 cycles are consumed until we receive the first data word from the corresponding split completion and we found that a split completion sequence completes on average in 3 transactions. For each of these 3 transactions we have an average latency of 6 cycles between them and we also need 4 cycles for the PCI-X protocol phases in every transaction. Finally, the PCI-X bridge issues a split response to all host memory read requests. The above measurements have been recorded in a PCI-X bus where the NIC is the only peripheral and we expect the above latencies to increase with the addition of other PCI-X cards on the same PCI-X bus.

The theoretical maximum throughput of a 64-bit 100MHz PCI-X bus assuming zero arbitration cycles is 762,9 MBytes/sec. We manage to achieve 662 MBytes/sec in PCI-X read transfers and 685 MBytes/sec in PCI-X write transfers.

5.3 Network Performance Evaluation

For the evaluation of the system we implemented some extra hardware functions in the NIC and the Switch so as to use them for benchmarks. In order to measure the latency and the throughput of the components in the system, we can set the NIC and the Switch in a *Benchmark Mode*, where they record cycle accurate timestamps inside every packet, as it passes through the stages of the system. The timestamps are kept in the payload of the packet and are processed by software at the receiver side, where each packet has passed through all the stages.

In *Benchmark Mode* the timestamps are put at the following points: (i) upon the packet creation in the request queue when the host processor writes a transfer descriptor, (ii) upon the packet departure from the NIC to the network, (iii) upon packet arrival at the switch port and (iv) upon departure of the packet from the switch. Timestamps (i) and (ii) measure the queuing delay and the pipeline latency in the NIC where timestamps (iii) and (iv) measure the delay and latency in the Switch. The latency in the cable and the SERDES circuits of the RocketIOs are constant and therefore we don't have to measure them but we can simply add them to the final latency. Moreover, we can bypass the process of reading the payload of the packet from the host memory (through a PCI-X DMA read) and simply generate a packet payload with zero values. Note that the flow ID , the size and the destination address still come from the transfer descriptor that is written by the host processor. This setup gives us the flexibility to measure the net NIC latency without the PCI-X overhead.

All packets are written in the destination host memory through DMAs in the appropriate addresses and are then collected by a linux kernel module which is developed inside the device drivers of the NIC. The software after the execution of an experiment reports a graph with the distribution of the packet latencies and reports the observed throughput per source. The throughput is measured by using processor cycle accurate timestamps that start upon the arrival of the first packet and finish upon the arrival of the last packet per source.

We have tested the system under many different traffic patterns with random packet sizes and we measured throughput and latency in the receiver side for every source.

The light-load (baseline) latencies of the system are as follows: For the transmission of a minimum size packet from the NIC, namely 40 bytes, we spend 16 PCI-X clock cycles. These cycles are attributed as follows: (i) the DMA engine block requires 2 memory accesses to read the transfer descriptor plus 1 cycle to start the transmission, so we spend 3 cycles in the DMA Engine. (ii) we spend 3 cycles in the synchronization FIFO between the DMA engine and the VOQs block plus 5 clock cycles (minimum packet size is 40bytes = 5 * 8 bytes words) for the packet to enter the VOQs and then handed to the link interface. (iii) we spend 3 cycles in the synchronization FIFO between the VOQ block and the Link interface plus 2 cycles for the link to start transmitting to the network.

As far as the throughput is concerned we have measured the maximum of 2.4 Gbps out of 2.5 Gbps in a setup where 1 node sends to 1 destination. In the case where 3 nodes send to 1 destination we have measured 0.79 Gbps for every flow which gives a sum of 2.37 Gbps.

We are currently measuring the NIC and Switch under more sophisticated traffic patterns that include packet size and destination distributions similar to those in [9] in order to practically verify the performance metrics of the proposed architecture. *These figures will appear in the final version of the paper!*

6 Related Work

Commodity system area networks such as Infiniband [12], Myrinet [13], Quadrics QsNet2 [14], and PCI-Express Advanced Switching [15, 16, 17] have been proposed to offer scalability and high performance switching. Many of these systems may also offer Network Interface Cards that are programmable at the (usually system-) software level but do not provide any hardware customization capability. Being FPGA-based our platform offers this capability and opens the opportunity for the inclusion and experimentation with new custom functionalities that a user might want to add to the NIC.

In terms of the NIC software interface, the Remote DMA primitives have been proposed to provide low-latency and high throughput communication [18, 19, 12]. These primitives are already available in high-performance networks [13, 14] and show up even in relatively low-cost Gigabit Ethernet controllers that support RDMA functionality over TCP, e.g. Broadcom BCM5706 [20]. We also believe that the RDMA primitives are attractive and we have added the flexible notification mechanisms that has been shown to be very effective in improving the interrupt processing cost [1]. Furthermore, our platform features two PowerPC processors in the FPGA; while we have not utilized these processors, they can be used to off-load processing from the system CPU or to implement protocol extensions.

On the switch side, buffer crossbar or CICQ switches have become realistic with the recent technology advances that allow the integration of enough memory for the crosspoint buffers. We have extensively evaluated these advantages and proved the feasibility of that support variable-size packets [10, 9] and multipacket segments [5]. To our knowledge, there is only one FPGA-based buffered crossbar implementation done by Yoshigoe *et al.* [21]. That work used older, low-end FPGA devices. Another important difference is that our switch can operate directly with variable-sized packets, and that we offer a complete reconfigurable system that includes the network interface card and the necessary (linux-based) system software.

7 Conclusions

We presented an FPGA platform for prototyping high-speed processor-network interfaces and interconnects. This platform includes both the network interface card and the switch card and offers built-in efficient primitives and can be adapted to new paradigms and protocols.

We believe that an experimental evaluation of new ideas is important and yields better accuracy and confidence as compared to simulation. Our platform, being FPGA-based, is open to accommodate new features and evaluate them in an actual experimental environment.

Acknowledgments

We would like to thank the members of the CARV laboratory at ICS-FORTH and especially the Scalable Systems Group. We are indebted to Manolis Marazakis, Kostas Xinidis and Prof. Angelos Bilas for their useful discussions and for enabling this work by providing us with Linux device drivers for our PCI-X peripheral. We thankfully acknowledge the support of the European FP6-IST program through the SIVSS (STREP 002075), UNIsIX (MC EXT 509595), SARC(FET 027648) projects, and the HiPEAC Network of Excellence (NoE 004408).

References

[1] M. Marazakis, K. Xinidis, V. Papaefstathiou, and A. Bilas. Efficient remote block-level i/o over an rdma-capable nic. In *Proceedings, International Conference on Supercomputing (ICS 2006)*, Queensland, Australia, June 28-30 2006.

[2] Xilinx Inc. Rocket i/o user guide. http://www.xilinx.com/bvdocs/userguides/ug024.pdf.

[3] I. Mindshare and T. Stanley. *PCI-X System Architecture*. Addison-Wesley Professional, 2001.

[4] G. Kornaros, C. Kozyrakis, P. Vatsolaki, and M. Katevenis. Pipelined multi-queue management in a vlsi atm switch chip with credit-based flow control. In *Proceedings of ARVLSI'97 (17th Conference on Advanced Research in VLSI)*, pages 127–144, Univ. of Michigan at Ann Arbor, MI USA, Sept 1997. IEEE Computer Soc. Press.

[5] G. Passas M. Katevenis. Variable-size multipacket segments in buffered crossbar (cicq) architectures. In *Proceedings, IEEE International Conference on Communications (ICC 2005)*, Seoul, Korea, May 16-20 2005.

[6] J. Duncanson. Inverse multiplexing. *IEEE Communications Magazine*, 32(4):34–41, April 1994.

[7] Fabio M. Chiussi, Denis A. Khotimsky, and Santosh Krishnan. Generalized inverse multiplexing for switched atm connections. In *Proceedings of IEEE GLOBECOM'98: The Bridge to Global Integration*, volume 5, pages 3134–3140, Sydney, Australia, Nov 1998.

[8] Xilinx Inc. *Xilinx ML325 Characterization Board.*

[9] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos. Variable packet size buffered crossbar (cicq) switches. In *Proceedings, IEEE International Conference on Communications (ICC 2004)*, Paris, France, June 20-24 2004.

[10] D. Simos. Design of a 32x32 variable-packet-size buffered crossbar switch chip. Technical Report Technical Report FORTH-ICS/TR-339, Institute of Computer Science, FORTH, Heraklion, Greece, May 16-20 2005.

[11] S. Mukherjee, B. Falsafi, M. Hill, and D. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of 23rd ACM Int. Symposium on Computer Architecture (ISCA 1996)*, pages 247–258, Philadelphia, PA USA, May 1996.

[12] Infiniband Trade Association. An infiniband technology overview. http://www.infinibandta.org/ibta.

[13] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovicm, and W. Su. Myrinet: A gigabit-per-second local-area network. *IEEE-Micro*, 15(1):29–36, July-August 1995.

[14] J. Beecroft, D. Addison, D. Hewson, M. McLaren, D. Roweth, F. Petrini, and J. Nieplocha. Qsnet2: Defining high-performance network design. *IEEE-Micro*, 25(4):34–47, July-August 2005.

[15] PCI-SIG. Pci express. http://www.pcisig.com.

[16] A.S.I. SIG. ASI technical overview. http://www.asi-sig.org.

[17] D. Mayhew and V. Krishnan. Pci express and advanced switching: Evolutionary path to building next-generation interconnects. In *Proceedings, 11th IEEE International Symposium on High Performance Interconnects*, 2003.

[18] RDMA Consortium J. Pickerton. The case for rdma. http://www.rdmaconsortium.org/home/The_Case_for_RDMA-02053.prf.

[19] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B.Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE-Micro*, 18(2):66–76, 1998.

[20] Broadcom Corporation. Bcm5706 ethernet controller. http://www.broadcom.com/collateral/pb/5706-PB04-R.pdf.

[21] Kenji Yoshigoe, Ken Christensen, and Aju Jacob. The rr/rr cicq switch: Hardware design for 10-gbps link speed. In *Proceedings, IEEE International Performance, Computing, and Communications Conference*, pages 481–485, April 2003.