Prototyping Efficient Interprocessor Communication Mechanisms

Vassilis Papaefstathiou, Dionisios Pnevmatikatos, Manolis Marazakis,
Giorgos Kalokairinos, Aggelos Ioannou, Michael Papamichael,
Stamatis Kavadias, Giorgos Mihelogiannakis, and Manolis Katevenis
Institute of Computer Science, FORTH – member of HiPEAC
Vasilika Vouton, Heraklion, GR 71110 Greece
{papaef,pnevmati,maraz,george,ioannou,papamix,kavadias,mihelog,kateveni}@ics.forth.gr

Abstract—**Parallel computing systems are becoming widespread and grow in sophistication. Besides simulation, rapid system prototyping becomes important in designing and evaluating their architecture. We present an efficient FPGA-based platform that we developed and use for research and experimentation on high speed interprocessor communication, network interfaces and interconnects. Our platform supports advanced communication capabilities such as Remote DMA, Remote Queues, zero-copy data delivery and flexible notification mechanisms, as well as link bundling for increased performance. We report on the platform architecture, its design cost, complexity and performance (latency and throughput). We also report our experiences from implementing benchmarking kernels and a user-level benchmark application, and show how software can take advantage of the provided features, but also expose the weaknesses of the system.**

I. Introduction

Chip and cluster multiprocessor systems are becoming widespread, while also growing in sophistication. To achieve efficiency, they strive for a tight coupling of computation and communication, and even propose customization of Network Interface (NI) features to meet particular application domain demands. Advanced features in the NI influence the design of, or require support from, the underlying interconnection network. Thus, our 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 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. This ability is even more important as the software-hardware interactions are only crudely (if at all) modeled in simulators.

In the context of our research and experimentation in high-speed processor-network interfaces and interconnects we have developed and FPGA-based prototyping system. Our prototyping platform consists of multiple (currently 8) workstations (PC's) linked through our custom interconnect. An FPGA development board plugs into the PCI-X bus of each PC, and is configured as its NI. A number of additional FPGA boards are configured as network switches. The key features of this platform are:

- *Remote Access Primitives:* For efficient communication we use Remote Direct Memory Access (RDMA) and Remote Queues for short messages.
- *Efficient Event Notification:* We support flexible arrival and departure notification mechanisms (selective, collective interrupts or flag setting).
- *High Throughput Network:* Each link offers 2.5 Gbits/s of net throughput per direction. Bundling 4 such physical links together (byte-by-byte or packet-by-packet) enables the creation of 10 Gb/s connections.
- *Efficient Network Operation:* Lossless communication via credit-based flow control; per-destination virtual output queues (VOQ) for flow isolation; large valency switch (up to 16×16). Bundling up to 4 switches in parallel offers up to 160 Gbits/s of full-duplex network throughput.

We are using this prototyping platform to study system-level aspects of network interface, efficient interprocessor communication primitives, and switch design, as well as evaluate their overhead and scalability for future multi-core and multi-node parallel systems. Colleagues from our Institute have used it for research in storage area networks [1]. In this paper, we report on the system architecture and performance, as well as the design cost and development experience. Our contributions are twofold: *(i)* we present the design and implementation details of an efficient, high-performance communication platform supporting advanced capabilities. *(ii)* we describe experiences and evaluation of the platform with (a) benchmarking kernels and (b) a user-level, interprocessor communication benchmark application. The evaluation gives valuable insight about the use and efficiency of the supported features and reveals bottlenecks that must be addressed in future systems.

In the rest of the paper, Section II discusses interprocessor communication primitives and Section III and IV describe in detail the NI and switch architectures. Section V presents implementation details, experimental results and discusses the efficient use of NI features. Finally, Section VI discusses related work and Section VII summarizes our conclusions.

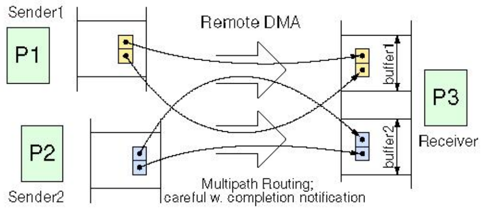

Fig. 1. Remote DMA: the receiver allocates separate buffer space per sender

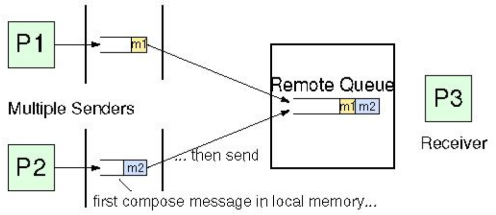

Fig. 2. Remote Enqueue: multiple senders on a single queue

II. Efficient Interprocessor Communication Primitives

To support efficient interprocessor communication, we need a set of simple, yet powerful communication primitives to be supported in hardware. This set must be as small as possible in order to reduce implementation cost, and as versatile and composable as possible, in order to maximize utility for the software. In our research we focus and base all NI functionality on just two primitives: Remote DMA and Remote Queues.

A. Remote Direct Memory Access

The Remote Direct Memory Access (RDMA) is the basic data transfer operation needed to enable *zero-copy* protocols. Zero-copy protocols deliver data *in-place*, so as to avoid the receiver having to copy them from one memory location to another. This is an important factor in overhead reduction, since data copying introduces major costs in latency, memory throughput, and energy consumption. With RDMA operation, every network packet carries the destination address where its data should be written, thus the receiving NI avoids to place the data in a temporary buffer, and then rely on protocol software to copy these data to their final location. The basic challenge in implementing RDMA is dealing with virtual-to-physical address translation and protection. Fig. 1 illustrates the RDMA operation, in the presence of multiple parallel transfers, and when packets of each transfer may be routed through different paths (“adaptive” or “multipath” routing). Multiple senders, P1 and P2, are sending to the same receiver, P3, in separate memory areas; otherwise the synchronization overhead would be excessive.

Multipath (adaptive) routing is desirable because it greatly improves network performance; however, multipath routing causes out-of-order delivery – a complex and expensive problem that many architects want to avoid. RDMA matches well with multipath routing: each packet specifies its own destination address, and it is placed in the correct place regardless of arrival order. The only problem that remains is to detect when all packets belonging to a same RDMA “session” have arrived (subsection III-E).

B. Remote Queues

Remote DMA is well suited to pair-wise (one sender, one receiver) producer-consumer type bulk communication: the transmitter controls the write pointer, while the receiver controls the read pointer. RDMA is not optimal for *small* transfers: it incurs some overhead to specify the source and destination addresses, initiate the DMA and then the transfer takes place; notice that a small transfer may be comparable in size to the RDMA descriptor. Also, if multiple senders exist, they must each be allocated a separate memory area, at the cost of additional memory usage, and also increased cost of monitoring arrival in these multiple memory buffers. Remote Queues (RQ) [2], [3] offer an effective alternative for these cases. A remote enqueue operation specifies the ID of the queue where its data will be placed. The receiving NI maintains the queue and atomically accepts messages upon arrival, Fig. 2; this property makes remote queues a valuable synchronization primitive. One important use of queues that we target in our research is to collect notifications for multiple concurrent transfers. If a receiver is expecting data from many potential sources via RDMA, and data arrival is signaled conventionally, by writing a flag at the last address of each transfer, then the receiver has to circularly poll many flag locations; this introduces latency and consumes memory bandwidth [1]. Alternatively, if arrival notifications are all placed in a single queue, the receiver can simply wait for that queue to become non-empty, and then read from that queue the information of a transfer that got recently completed.

III. NI Prototype

Our prototype NI is designed as a 64-bit PCI-X 100 MHz peripheral based on a Xilinx Virtex II Pro FPGA and uses up to 4 RocketIO multigigabit transceivers [4] for the network transport. The NI architecture is depicted in Fig. 3. We briefly describe the main modules of the system in the paragraphs below and we focus on the components supporting interprocessor communication in the next subsections.

The PCI-X module fully implements initiator, target and interrupt functions and exposes memory-mapped regions to the system. It supports 32 and 64-bit accesses in burst or non-burst mode to the target interface, while the initiator provides the DMA capabilities to read/write from/to the host’s memory, supporting 32 and 64-bit wide bursts using physical PCI addresses.

The Link interface uses the RocketIOs (each capable of 2.5 Gbps) to transmit the packets through high speed serial links. It injects control delimiters using in-band signaling, transmits the raw packet data and appends CRC checksums for error detection. Moreover, it uses a QFC-like credit-based flow-control [5] protocol to achieve lossless network transmission.

[1]If notification is done through the use of interrupts there is no need for this mechanism; however, the cost of per-transfer interrupt is excessive in high speed communications and they should be avoided whenever possible.

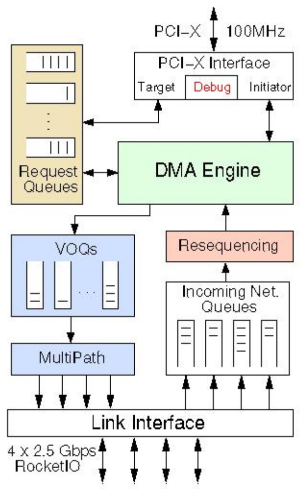

Fig. 3. NI' s architecture

The DMA Engine is the heart of both the outgoing and incoming portion of the NI which arbitrates between and serves transfers from *(i) Request Queues* that generate outgoing traffic and *(ii) Incoming Network Queues* that serve all the traffic that arrives from the network.

A. RDMA Support

Our prototyping approach allows the host processor to post *transfer descriptors* for RDMAs to memory-mapped regions which are exposed by the NI. We have chosen to support only RDMA-Write in hardware since it is the basis for RDMA communication; RDMA-Read can be implemented via system software using a rendezvous protocol. The descriptors arrive at NI's target interface, stored in *RDMA Request Queues* and served by the central DMA engine.

A transfer descriptor consists of two 64-bit words which contain all the information needed to initiate and transmit an RDMA packet. The first word specifies the PCI-X source address for the local data and the second word contains:

- a 32-bit remote host destination physical address; where the data will be transfered to,
- the size of the transfer, in 64-bit words (the maximum supported size is 512 words or 4096 bytes),
- the ID of the destination host (current support for 128 hosts) and
- an "opcode" field that controls the notification options for the transfer, as described shortly.

The *RDMA Request Queues* keep the transfer descriptors for the pending remote writes and are organized per-destination to prevent head-of-line blocking and ensure flow isolation. We currently have 8 queues, one per-destination in the network, that allow up to 128 pending transfer descriptors each.

Besides decoupling the operation of the DMA engine from the processor, these request queues support clustering of requests to the NI: 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 become available.

B. Remote Queues Support

Although RDMA mechanisms could support remote enqueues, we followed a different approach. An RDMA-write requires a transfer descriptor to be written in a request queue and then a local read DMA to be performed. This series of events entails significant latency overhead since the system bus is traversed twice. Our approach allows messages to be written directly into NIs memory – *Outgoing Message Queues* – and avoid the double traversal of the system bus.

The *Outgoing Message Queues* are organized per-destination and allow the processor to implement low-latency remote enqueue operations without posting a transfer descriptor. The processor forms the actual short messages (header and body) into these queues and the central DMA engine forwards them to the network. Moreover, processor's programmed-IO can exploit the write-combining buffers and greatly improve performance by transferring the packet data into bursts. We currently have 8 queues, one per-destination host in the network, of 2KByte each, implemented as a circular buffer in a statically partitioned 16KByte memory.

The outgoing messages contain a *QueueID*, instead of a destination address, which should be translated into a physical address at the receiver. This translation is dynamic and provides the physical addresses in a cyclic manner in order to form circular queues into the receivers host memory. The mechanism that handles these messages at the receiver uses a lookup table – *Queues Translation Table* – which keeps 128-bits per entry:

- a 64-bit base physical address which is bound with the queue ID
- queue's head pointer offset
- queue's wrap around offset

During an enqueue operation, the head pointer of the associated Queue advances and when it reaches the wrap around offset it returns to the base address. This translation table is also memory-mapped in the system's address space and can be configured by the system software. Our design allows the user to configure up to 256 circular *Remote Queues* of programmable size.

C. 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 injected into the network: when so requested by a transfer descriptor or a short message, upon departure of the transfer, the NI copies the tail pointer of the associated queue to prespecified locations in host memory, using a single-word DMA write access. Since we have per-destination queues, we also have per-destination locations in the host memory

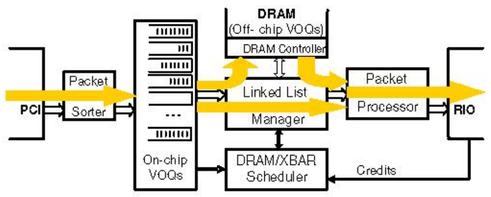

Fig. 4. VOQs block and flow diagram

for these local notifications. The addresses for these memory locations are set by software.

The processor can poll in these notification locations to determine the state of the requested transfers i.e. how many transfers have departed (transfers from a single queue depart in-order), hence recycle their slots. Processor polling in the host memory is lighter than polling the tail pointer itself – a NI control register – in I/O space.

Remote Interrupts and Remote Notifications can be used to inform the receiving node that a packet or message has arrived; usually assuming in order delivery from the network. The former are traditional PCI interrupts, while the latter are similar to their local counterparts: they write (via single-word DMA write) the last address of a completed DMA operation into a prespecified address in the receiver's host memory. Since the packets come from different hosts, we have per-source locations in the host memory for remote notifications. The addresses for these locations are set by software.

Local and remote notification options, in combination with the operation clustering option, allow for a drastic reduction in the number and overhead of interrupts [1].

D. 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 head-of-line blocking and localize the effects of congestion, multiple virtual output queues (VOQs) – one per (potential) destination – are implemented.

The initial architecture of the VOQ handling system is based on previous research [6]. Fig. 4 depicts the initial VOQs architecture where 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 in 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 VOQs through the use of linked-lists implemented in hardware.

The RDMA packets (max. 4 KBytes) that exceed the maximum network packet size, which is 512 bytes in our network, are segmented into smaller independent RDMA packets by modifying or inserting the appropriate packet headers.

The addition of multi-path support – load balancing – is highly dependent on the VOQs implementation and led to a very complex design, inappropriate for FPGA prototyping. Therefore, we simplified the VOQs block by keeping only on-chip VOQs in the current implementation and by not making use of external memories (i.e. DRAM). The current VOQs design is far more flexible and has lower latency since the linked-lists are removed and packet processing is performed in parallel with packet sorting, before packets enter the VOQs.

E. Multipath Routing and Completion Notification

Inverse multiplexing [7] is a standard technique that allows several independent links to be combined together in order to implement a ”logical link” of multiple capacity. The load on each link is switched (routed) to the destination independently and the original traffic should be distributed among the links by the transmitting NI. This technique is also suitable for internally-non-blocking switching fabrics as long as the load is *evenly balanced* among the parallel paths, on a per-destination basis. Our multipath policy balances the traffic using Deficit Round Robin (DRR) [8].

Such multipath routing may deliver packets out-of-order, at the destination. Owing to the use of RDMA semantics (each packet carries its own destination address), packet data will be delivered in-place in the host memory even if the packets arrive in scrambled order. If data were delivered in-order, RDMA completion could be signaled by the last word being written into its place, however, when packets can arrive out-of-order, the last address in the destination block can be written into before intermediate data have arrived. Thus, RDMA semantics eliminate the need for reorder buffers and data copying, but introduces the need for *Completion Notification* to guarantee transfer completion.

Currently we use resequencing to provide completion notifications. We economize on resequencing space by buffering only packet headers, while packet data are written to their destination address. After resequencing, we discard headers in-order until we encounter a notification flag; at this point we are sure that all packets before it have been received and processed, hence the notification can be safely delivered.

IV. Switch Prototype

Our switch implements an 8×8 *Buffered Crossbar* (Combined Input-Crosspoint Queuing - CICQ) architecture [9] on the Xilinx ML325 board [10]. The switch uses small buffers at each crosspoint and features *(i)* simple and efficient scheduling, *(ii)* credit-based flow control [5] for lossless communication, *(iii)* variable-size packet operation, and *(iv)* peak performance without needing any internal speedup.

Figure 5 depicts the internal structure of a 4×4 buffered crossbar switch. Incoming packets are delivered to the appropriate crosspoint buffers (2 KBytes each) 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, with a round-robin policy, a non-empty crosspoint buffer for transmission. Each OS supports cut-through operation even for minimum-size packets and hides scheduling latency by utilizing a pre-scheduling technique; schedules the next packet before the end of the previous packet transmission. As packet bytes are being transmitted to the output, the credit scheduler (CS) generates

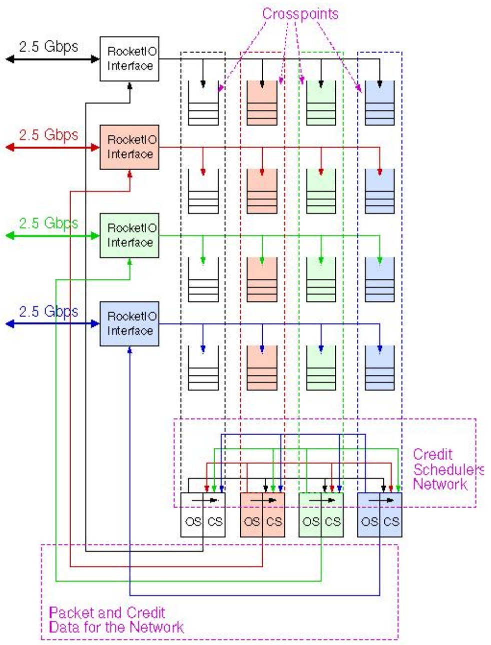

Fig. 5. The internal structure of a 4x4 Buffered Crossbar

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. The datapath of the switch is 32-bit wide, the clock frequency is 78.125 MHz as required by the RocketIO serial link interfaces, and the maximum packet size is 512 bytes.

V. Cost and Performance Evaluation

This section reports the implementation cost of our FPGA prototype and presents a performance evaluation through various benchmarks. At first we illustrate the performance of our custom network, then we present the observed DMA performance through the PCI-X bus and finally we report on the efficient use of the platform by the systems software.

A. Hardware Implementation Cost

Table I presents the hardware cost of the system blocks. The numbers refer to the implementation of the designs in a Xilinx Virtex II PRO FPGA with the back-end tools provided by Xilinx. The 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 overall design. The VOQs block is also area demanding because it involves many BRAMs to be used as packet buffers and considerable logic for their associated state.

B. Network Performance Evaluation

For the evaluation of our custom network we implemented some extra hardware functions in the NI and the Switch so as to use them for benchmarks (latency and throughput). In this special *Benchmark Mode*, the NI and Switch record cycle-accurate timestamps and append them in the payload the packets, as they pass through the stages of the system.

TABLE I
Hardware Cost Breakdown

Block	LUTs	Flip Flops	BRAMs
PCI-X - DMA Engine - Queues	2500	1400	22
Link Interface	1800	400	0
Multiple VOQs	4100	2100	37
Multipath Support	2800	1200	20
Debugging Support	2900	2100	32
Totals NI	14100	7200	111
BufXbar Switch 8×8	15800	13300	64

In *Benchmark Mode*, timestamps are recorded at the following points: (i) upon packet creation, in the request queue, when the host processor writes a transfer descriptor; (ii) upon packet departure from the NI 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 NI, whereas timestamps (iii) and (iv) measure the delay and latency in the switch. The latencies of the cables and the SERDES circuits of the RocketIO's are constant, and therefore we don't have to measure them; we simply add them to the final latency. Moreover, we 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; in this way, we factor out the software and the PCI bus latencies.

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 NI. The software, after execution of an experiment, reports the distribution of the packet latencies and the observed throughput per source. The throughput is measured using processor-cycle-accurate timestamps that start upon arrival of the first packet and finish upon arrival of the last packet, per source.

Using the special software and hardware functions, we have performed delay and throughput experiments in order to validate the simulation experiments of the switch performance that appear in [9]. For the traffic patterns, we generated packet traces with the *Traffic Generator* of [11]; at measurement time, host software loads the traces and feeds NI's request queues with descriptors at specified times. We were able to run only small scale experiments due to the limited number of hosts and memory resources, and therefore our figures correspond to experiments with a 4×4 switch: each NI had to act as either source or sink of packets, but not both, because if it were to act as both then software and the PCI-X bus would be the bottleneck, rather than being able to saturate the network. We have run each test with 50 million packets where the first few thousands of packets (warm-up) and the last few thousands were not accounted in order to have as accurate measurements as possible. The duration of these tests ranged from 1 to 14 minutes of real time traffic.

For the delay experiments we have run tests with uniformly destined traffic and uniform packet sizes. The results of the delay vs. load experiments are shown in Fig. 6 where the observed curve follows closely the simulation results of [9].

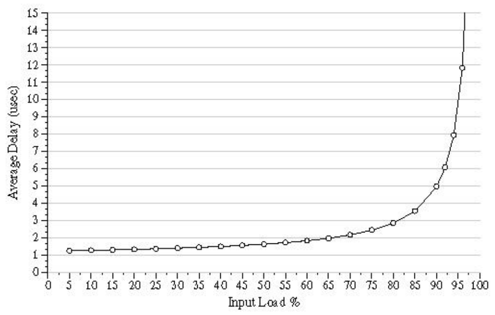

Fig. 6. Avg Delay vs. Input Load under uniform traffic. Max load is 96%.

The average end-to-end network delay of our platform under light load is *just 1.25 microseconds*, where half of this (0.64 μs) is due to propagation delays (SERDES (95%) plus cables (5%)); this figure (1.25 μs) is from the moment a packet is generated (by hardware) inside the source NI, to the moment that packet enters the destination NI. In other words this figure includes source and switch queuing, propagation, and scheduling delays, and SERDES+cable delays from source to switch and from switch to destination, but does *not* include any PCI or software delays. Our NI and switch designs proved to have modest latency even under 80% load, where the end-to-end delay is lower than 3 μs.

C. PCI-X microbechmarks

We used hardware cycle counters at the NI to examine the behavior of the host-NI interface, namely the PCI-X Target Interface (100 MHz). 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 buffer of the host processor, we can write a burst of 64 bytes of data in 24 PCI-X cycles, which translates to 4 transfer descriptors. This feature gives a 6$\times$ improvement over the simple case which would need about 160 PCI-X cycles and saves a significant number of cycles on the PCI-X bus. Notice that the use of the write combining buffer implies weak ordering and requires the user to regularly flush it (with an sfence instruction) in order to avoid undesired latency of the data in the buffer.

For a write-DMA transfer of 4 KBytes (PCI-X maximum size) to the host memory, with 64-bit data phases, we measured a delay of 570 cycles, out of which only 512 actually transfer data (90% utilization). The remaining 58 cycles are attributed to arbitration, protocol phases, and the occasional disconnects. For 4 KByte read-DMA transfers from host memory, we measured a delay of 592 PCI-X cycles, i.e. a utilization of 87%. In every DMA read request, the PCI-X bridge issues a split response and on the order of 50 cycles are needed until we receive the first data word. The remaining cycles are attributed to protocol phases and disconnects.

The theoretical maximum throughput of a 64-bit 100MHz PCI-X bus assuming zero arbitration cycles is 762,9 MBytes/sec. We managed to achieve 662 MBytes/sec in PCI-X read transfers and 685 MBytes/sec in PCI-X write transfers by using a specially designed DMA engine that performs a large series of back-to-back PCI-X accesses that employ the bus for over a minute of real time.

D. Efficient Use of the NI by System Software

For systems software to make efficient use of the capabilities offered by our NI, we need to closely match the abstractions exported by the hardware with corresponding software abstractions. Specifically, we have to be careful to use the hardware resources in a manner consistent with their design, despite the fact that this may lead to a more complicated software implementation. To illustrate this point, Fig. 7 shows the throughput achieved by two alternative implementations of a simple program that issues one-way data transfers, for a range of transfer sizes. For transfer sizes up to 4 Kbytes (single OS page), only one RDMA descriptor is posted. For larger transfers (up to 512 Kbytes), several RDMA descriptors are posted, one after the other. For each transfer size, the benchmark programs performed 100,000 transfers and PCI-X write-combining was enabled for the transmitting endpoint. The alternative versions of this benchmark differ in the details of when RDMA transfers are triggered for execution by the NI, and under which conditions to block waiting for a notification of transfer completion.

The first version (marked `v1`) treats the RDMA request queue as a linear (non-circular) command buffer where RDMA descriptors are posted in batches (up to 128 consecutive descriptors for a 512 Kbyte transfer). Only the last RDMA descriptor in the batch triggers the NI to begin transferring data from the host memory (PCI-X DMA read). This version of the benchmark waits for this last RDMA operation to complete, making use of the local notification capability offered by the NI. Since the RDMA descriptors are processed in FIFO order, this version of the benchmark waits until all pending transfers are completed before posting the next batch.

The second version (marked `v2`) treats the RDMA request queue as a circular command buffer, explicitly checking if there is space to post each of the RDMA descriptors. This is done by setting each of the RDMA descriptors to trigger the NI to begin transmission, and then checking progress toward completion by reading the head and tail values written in host memory as a result of local notifications. If no space is found, this version of the benchmark busy-waits by polling on the local notification word in host memory. Otherwise, it immediately posts the next RDMA descriptor. Thus, the usage pattern induced by this version more closely matches the way that the NI hardware actually processes RDMA descriptors. Unlike the `v1` implementation, `v2` does not have to wait until a large batch of RDMA transfers are completed and thus allows overlapping transfers with posting new descriptors.

Although `v1` achieves a throughput level of up to 632 Mbytes/sec (around 95% of the maximum achievable for this specific experimental setup, see subsection V-C), it suffers in terms of latency as transfer sizes increase. By not matching the way the NI processes pending RDMA descriptors, the `v1` implementation forces even the posting of each batch

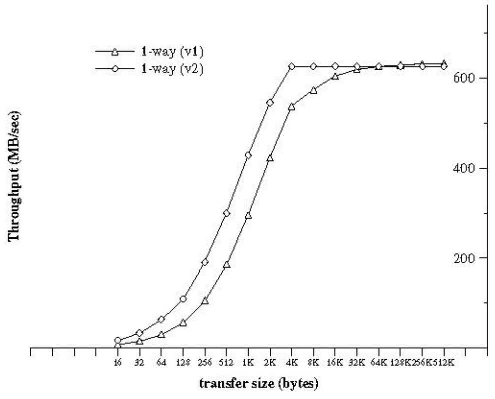

Fig. 7. Throughput comparison of two alternatives (v1, v2) in using the NI for high-speed data transfers.

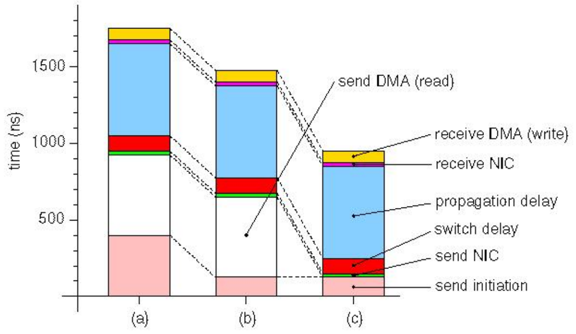

Fig. 8. Breakdown of end-to-end latency for (a) an 8-byte RDMA packet with single (uncombined) PCI-X writes, (b) an 8-byte RDMA packet with PCI-X write-combining, and (c) a single 8-byte short message with PCI-X write-combining.

of transfers to start after the whole of the previous batch is transmitted. The v2 implementation pipelines the distinct tasks of posting RDMA descriptors and processing them for transmission and achieves comparable throughput levels even for lower transfer sizes. For small transfer sizes (up to 4 Kbytes), v2 significantly outperforms v1.

The performance measurement experiments reported in this section have been taken into account in the optimization of the networked storage system described in [1]. In that system, we follow the approach exhibited by the v2 implementation of the one-way transfer benchmark to maintain a relatively constant latency for posting remote I/O operations, and their corresponding completions.

E. End-to-End Latency

Beyond the software based throughput experiments, we have measured, using NI performance counters, the end-to-end (memory-to-memory) latency in our system. Fig. 8 shows a breakdown of the one-way latency of a small (8-byte) remote DMA, with and without write-combining, and an equal-size message transmission. The overhead is divided in the following components: send-initiation, send-DMA, send-NIC, switch-delay, propagation-delay, recv-NIC, recv-DMA. The *send-initiation* component includes the PCI-X overhead during posting the transfer descriptor. The *send-DMA, recv-DMA* components include all PCI-X overhead related to the data transfer itself. Finally, *send-NIC and recv-NIC* is the time spent in the send and receive NICs. We measured these components using the corresponding cycle counters on the NIC boards. The *switch-delay* component refers to the cut-through packet delay in the switch and *propagation-delay* refers to cumulative delay of all SERDES circuits in the network paths plus the delay in the cables.

Write-combining significantly speeds-up operation initiation. In a system that uses write-combining, the two components where most of the time is spent are: propagation delay (36% of the total delay, 95% in the SERDES circuits and 5% in the cables) and the PCI-read DMA at the sending node (33% of the total delay). The majority of the PCI-read DMA cost is due to read latency, manifesting itself as PCI-X *split* duration (50 PCI-X cycles). Hence, it becomes apparent that, for short transfers, the *message* operation yields much better performance than the *remote DMA* operation, because it eliminates the read-DMA at the sender side, at a small incremental cost of 1 PCI-cycle per word, for posting each message word beyond the first two words (up to a message size of 64 Bytes), using write-combining.

F. Scalability Concerns and Challenges

In our current prototyping platform we have made several design decisions, towards simplifying the FPGA design, which raise important scalability concerns. Specifically, the RDMA Request and Message Queues are implemented in statically partitioned memory, organized per-destination. Keeping queues for every possible network destination does not scale in an environment with thousands of nodes because it would require excessive amounts of memory. Even dynamic memory management of a shared memory space cannot scale beyond a few hundreds of destinations.

Moreover, the choice of packet VOQs was required since we needed to evaluate in a real system the variable packet size CICQ architecture we proposed in previous work [9]. Additionally, we consider multipath routing, in a multiprocessor environment, to be an important feature that can boost network performance as well as allow for scalable multistage fabrics. We have experimented with multipath routing, out-of-order packet delivery and completion notification and deduced that inverse multiplexing with DRR needs $O(N^2)$ counters to be implemented and thus does not scale.

Furthermore, the cost of resequencing, even when only packet headers are stored, is excessive since the space required is proportional to the number of senders (nodes), the amount of intermediate network buffering and the number of network paths from a source to a destination.

We are currently investigating whether the NI could share memory with the processor in an environment where the NI moves closer to the CPU; from the I/O bus to the memory bus or even share processor's cache. Our ongoing work tries to address the scalability issues mentioned above. We consider replacing all per-destination queues with per-thread or per-process queues towards NI virtualization. In addition adaptive routing can provide a simple and scalable multipath solution.

Resequencing can be avoided by associating counters only to packet groups that require completion notification.

VI. Related Work

Commodity system area networks such as Infiniband [12], Myrinet [13], Quadrics QsNet2 [14], and PCI-Express Advanced Switching [15] 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. Our FPGA-based platform offers the capability to include and experiment with user-customizable functions at the NI.

In terms of the NI software interface, the Remote DMA primitives have been proposed in order to provide low-latency and high throughput communication [16], [17], [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.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].

On the switch side, buffered crossbar switches have become feasible since recent technology advances allow the integration of the memory required for crosspoint buffers. We have extensively evaluated these advantages and proved the feasibility of that support variable-size packets [9] and multipacket segments [6]. To our knowledge, there is only one FPGA-based buffered crossbar implementation done by Yoshigoe *et al.* [18], that 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.

VII. Conclusions and Future Work

We presented an FPGA-based, research 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 that 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 FPGA-based platform is open to accommodate new features and evaluate them in an actual experimental environment. Our experience so far is that the system-level operation reveals component interactions that are practically impossible to foresee and model in a simulator.

We are currently in the process of porting MPI over our NI and we plan to measure parallel applications and benchmarks. Moreover, we strive for architectures that offer tighter coupling of the NI with the processor. We consider "moving" the NI closer to the processor, as close as the cache interface.

Acknowledgments

We would like to thank all the members of the CARV laboratory at ICS-FORTH and especially Christos Sotiriou and Spyros Lyberis for their crucial support throughout this work. We are also indebted to the members of the Scalable Systems Group and Prof. Angelos Bilas for their useful discussions and for enabling this work by developing the system software for our NI. 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] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *SPAA 1995.*

[3] M. Katevenis, E. Markatos, P. Vatsolaki, and C. Xanthaki. The Remote Enqueue Operation on Networks of Workstations. *Proceedings of CANPC'98*, Jan. 1998.

[4] Xilinx Inc. Rocket I/O User Guide. http://www.xilinx.com/bvdocs/userguides/ug024.pdf.

[5] H. T. Kung, T. Blackwell, and A. Chapman. Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation and Statistical Multiplexing. In *Proceedings of the ACM SIGCOMM Conference*, 1994.

[6] M. Katevenis and G. Passas. Variable-Size Multipacket Segments in Buffered Crossbar (CICQ) Architectures. In *Proceedings, IEEE International Conference on Communications (ICC 2005)*, Seoul, Korea, May 16-20 2005.

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

[8] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM '95: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, New York, NY, USA, 1995. ACM Press.

[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] Xilinx Inc. *Xilinx ML325 Characterization Board.* http://www.xilinx.com.

[11] G. Passas. Performance Evaluation of Variable Packet Size Buffered Crossbar Switches. Technical Report FORTH-ICS/TR-328, Institute of Computer Science, FORTH, Heraklion, Greece, November 2003.

[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] 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.

[16] C. Sapuntzakis, A. Romanow, and J. Chase. The case for RDMA. http://suif.stanford.edu/ csapuntz/draft-csapuntz-caserdma-00.txt.

[17] 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.

[18] 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.