Experiences from Debugging a PCIX-based RDMA-capable NIC

Manolis Marazakis, Vassilis Papaefstathiou, Giorgos Kalokairinos, and Angelos Bilas
Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH),
Heraklion, Greece GR71110
{maraz,papaef,george,bilas}@ics.forth.gr

Abstract

Implementing and debugging high-performance newtork subsystems is a challenging task. In this paper, we present our experiences from developing and debugging a network interface card (NIC). Our NIC targets networked storage subsystems [17]. For this purpose it mainly provides support for remote direct-memory-access (RDMA) write, sender-side notification of RDMA write completion, and receiver-side interrupt generation. In our work we examine issues that arise during system implementation and debugging, both in terms of correctness as well as performance. We present an analysis of the individual problems we encounter and we discuss how we address each case. For most problems we encounter, it is not possible to rely on existing debugging tools. However, we find that most of the techniques we use in this process, rely on collecting some form of event records from software or hardware components. We believe that such capabilities can be provided for independent hardware or software components in isolation, a fairly straight-forward task, thus, significantly simplifying the debugging process in complex systems of this nature.

1. Introduction

Implementation and debugging of high-performance systems consisting of multiple hardware and software components presents a significant challenge both in terms of correctness as well as performance. Especially, when both new hardware and low-level software components are involved, debugging becomes time-consuming as most tools currently available cannot be used.

In this paper, we present our experience from implementing and debugging such a system [17]. We focus on a network interface card (NIC) that targets networked storage systems. Future storage systems will increasingly rely on a scalable interconnect, such as Infiniband [1], Myrinet [7], and PCI-Express/AS [21, 18]. Such interconnects are used extensivelly today in compute clusters. However, storage systems present new challenges due to both the different communication requirements they impose, as well as the kernel-level protocols they need.

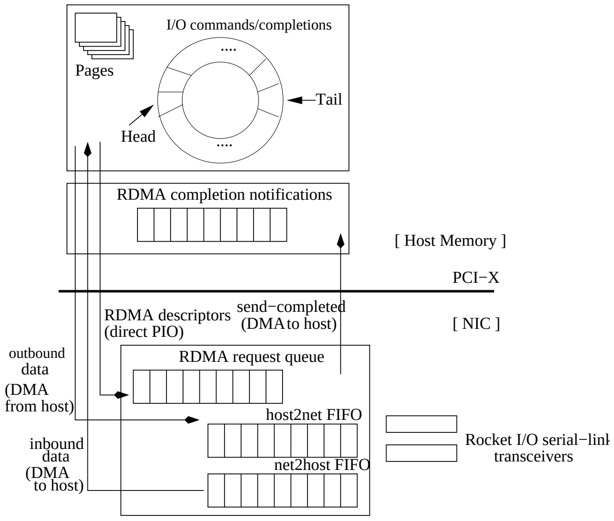
We focus on issues related to the implementation and debugging of the network interface card (NIC). Our NIC offers the following core capabilities: RDMA write, sender-side notification of RDMA write completion, and receiver-side interrupt generation. Our goal is to (1) summarize the problems we encounter during implementation, (2) describe the methodology we use to address each problem, and (3) suggest debugging mechanisms that should be developed during implementation of various system components.

We divide that problems we encounter in correctness and performance categories:

- Correctness: basic operation of the major NIC modules, interfaces between modules, interactions with the host environment.
- Performance: efficient operation of the major NIC modules, interference between modules, hardware-software interface.

The custom mechanisms we use to address the problems are mostly event counters, cycle counters, tracing memories, and various supporting software modules, both kernel-space and user-space. We find that these mechanisms constitute some form of event recording. For instance, we capture data words enqueued for transmission via the network links, to verify the interface between the DMA engine that pulls these words form the host memory and the NIC module that controls the network transceivers. Moreover, we observe that these mechanisms refer to individual system components; thus, their implementation is independent of other system components and may occur a-priori during component implementation. Therefore, we believe that providing the ability to gather certain known event records by each system component is a generic mechanism that can reduce debugging effort.

The rest of this paper is organized as follows. Section 2 outlines the design of the NIC and discusses the debugging



I/O commands/completions
Pages
Head
Tail
RDMA completion notifications
[Host Memory]
PCI−X
[NIC]
RDMA descriptors (direct PIO)
send−completed (DMA to host)
outbound data (DMA from host)
RDMA request queue
host2net FIFO
inbound data (DMA to host)
net2host FIFO
Rocket I/O serial−link transceivers

Figure 1. NIC block-level organization and main structures.

methodology and tools we use. Sections 3 and 4 present the main correctness and performance problems that occurred during implementation and how we address each of them. Section 5 presents related work. Finally, Section 6 concludes the paper.

2. Background about the NIC

We are developing a low-latency, high-throughput NIC that allows us to perform memory-to-memory transfers from initiators to targets, in a SAN environment. The NIC has been developed in-house to allow both detailed measurements of all aspects of communication in the I/O path as well as customization of communication primitives and operations. In the following, we present an overview of the design, and highlight testing and debugging challenges that we have to overcome. The NIC is implemented with a DiniGroup DK6000K10SC development board, using a Xilinx VirtexII Pro FPGA. A detailed presentation of the design is given in [13].

Figure 1 shows the block-level organization of the NIC. The NIC design initially targeted 64-bit, 66 MHz PCI. The design evolved to 64-bit, 100 MHz PCI-X [20].

The host-NIC DMA engine transfers 64-bit data words in bulk to and from the host's memory. A transceiver controller co-ordinates transmission and reception of data words over a pair of serial links. The NIC applies a credit-based scheme for network flow control [15], to prevent overflowing the receiver's NIC buffer, thus preventing loss of data in the event of heavy network load. The NIC maintains a pair of inbound and outbound word counters *per flow*. The receiving-side NIC transmits the count of words received, in a special control message (*credit*). The transmitting-side NIC has to examine the credit, in comparison with its own count of words transmitted, and determine if there is enough space at the receiving side to accept a new packet.

2.1. Network Links

At the physical layer, our NIC uses a pair of RocketI/O (RIO) serial links [3]. The serial links, each capable of 2.5 GBits/s (3.125 GBauds) full-duplex, operate at 78.125 MHz and are controlled by an FPGA-based transceiver controller. An extension of this design allows for 4 serial links, resulting in 10 GBits/s total link throughput.

The communication with each RIO transceiver is done through a 32-bit wide data-path. At the transmitter side, data is sent to the network 4 bytes per clock cycle per RIO link. Some synchronization/delimiting bytes are introduced occasionally at packet boundaries or at idle periods. The data is serialized and transmitted through differential links (one pair per RIO link). At the receiving side however, the situation is somewhat more complicated. Data deserialization can cause misalignment of the bytes, i.e. the bytes that make up a transmitted 32-bit word may be found shifted in the received word. RIO attribute settings may assist in making this problem infrequent, but it can only be overcome by using a separate circuit for each link. This circuit is a pipeline that takes as input the misaligned incoming stream, extracts serialization/synchronization information from the link, and delivers the packet in its original from.

Beyond intra-link synchronization, inter-link synchronization is required. The two RIO links are *bonded* by the NIC to appear as a single network link, as is the case in all high-end network interfaces today that use multiple physical links. Each packet is transferred over both links, multiplexing data at the byte level. Every receiving circuit incorporates a small synchronization FIFO. This is used to form 64-bit words, by combining 32-bit words from each of the links. This is necessary as the two 32-bit words are not available always at the same clock cycle from the RIO to the receiver.

Another important design issue is the difference of the link speeds at each side of the network. Each NIC has its own crystal driving the RIO transceivers, which although of the same nominal frequency, always have a slight drift. To counter this difference, we have activated a clock correction mechanism which allows the injection and removal of idle/sync characters in the stream.

2.2. RDMA Initiation

Host programs access the RDMA request queue, as well as a set of performance-related counters via a memory mapping established with the cooperation of the NIC kernel module through the `mmap()` system call. Our current design does not support *protected*, user-space access to NIC resources and is intended for use by the kernel. The main reason for this is that we are interested in exploring issues related to the I/O protocol stack, which because of transparency requirements in practice always involves the operating system kernel. However, user-space programs can still access the NIC in an unprotected manner for testing and benchmarking.

Currently, the NIC supports only *RDMA write* operations. Each remote write is specified through a transfer handle. The transfer handle specifies the local and remote physical addresses, the length of the transfer, and the desired notifications. Completion of a transfer may generate (a) a local notification in the form of a 64-bit word written by the NIC into the sender's memory and (b) an interrupt at the receiver host. Currently, the maximum message size is 4 KBytes; longer messages have to be segmented in the host library. On the receive path, data is directly transferred to the specified physical locations in memory, *without any receiver processor intervention*.

Posting a RDMA write operation requires posting the transfer descriptor to the NIC request queue, over the PCI-X target interface. In the current version of the NIC, posting a transfer descriptor requires four 32-bit PCI-X write operations in the NIC's DMA queue: Two 32-bit words to specify the source physical address, one 32-bit word to specify the destination address, and finally a 32-bit word to specify the transfer size (as a number of 64-bit words), various transfer flags, such as local and remote notification options, and the destination node, identified by a 7-bit *flow ID*. The NIC's design is oriented for mostly kernel-space use and allows for 64-bit addresses. As a provision for allowing some level of protection, the NIC can treat the 32-bit destination address word as a handle to be resolved based on pre-established bindings for remote memory regions. However, in the current version of the NIC this feature is not available; therefore, the destination word is treated as a physical address in the receiver's memory.

The DMA engine and the network transceivers operate within different clock domains. A pair of FIFO queues allows us to handle this mismatch: The DMA engine places the entire message to be transmitted in the outgoing FIFO and then signals the transceiver module to commence transmission. Regardless of clock speeds, the process of reading words from the host's memory is subject to delays that cannot be anticipated or controlled by the NIC. Having the entire message ready for transmission simplifies the operation of the transceiver module. During message reception, the transceiver module signals the DMA engine to start transferring data to the host's memory as soon as the header and a few data words have been placed in the incoming FIFO. Receiver-side cut-through offers a performance benefit, as it frees FIFO space in the receiver as fast as possible. Moreover, it allows the sender to proceed with its next RDMA transfer before the receiver has finished placing the data in host memory.

2.3. Crossbar Switch

To support multiple I/O initiators and targets, our prototype uses an in-house network switch. The switch is based on a buffered crossbar that involves small buffers in the crosspoints and allows cut-through operation. The crossbar applies distributed round-robin scheduling to its outputs. Since we expect to have hotspots in the network, i.e some outputs will be more congested than others, we rely on a credit-based back-pressure mechanism to prevent overflowing both the crossbar buffers and the receiver's incoming network buffers. The buffered crossbar operates directly on variable size packets; therefore, there is no need for segmentation and reassembly [14]. We have an 8x8 buffered crossbar switch, where each NIC is connected with one bidirectional RIO link with the switch. The NICs can then send packets to any of the destination in our system-area network by modifying the appropriate destination flow-ID field in the packet header.

2.4. Experimental Testbed

The main prototype we use in testing the NIC consists of two Dell PowerEdge 1600SC servers, each with a single Intel Xeon CPU, running at 2.4 GHz, 512 MBytes of main memory, and one 64-bit PCI-X bus running at 100 MHz. One of the nodes serves as I/O target, with 8 directly-attached SATA disks. The disks we use are Western Digital WD800JD, connected to a BroadCom RAIDcore controller, with a total capacity of 614.1 GBytes. Each node has an additional dedicated IDE system disk, and two interconnects: (a) a Gigabit Ethernet adapter for system administration and monitoring and (b) our custom-built NIC for RDMA transfers. For experiments using the switch, we can currently use from 2 up to 6 servers as end-points interconnected through the switch.

2.5. Hardware-Software Interface

Our NIC is used as part of a networked storage system prototype. Details of the remote I/O protocol, as well as a performance evaluation, are presented in [17]. We have implemented the systems software in Linux kernel version

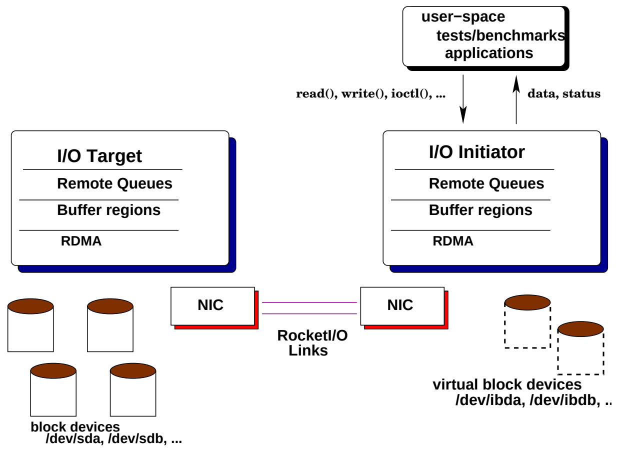


Figure 2. Overview of software architecture for remote block I/O prototype.

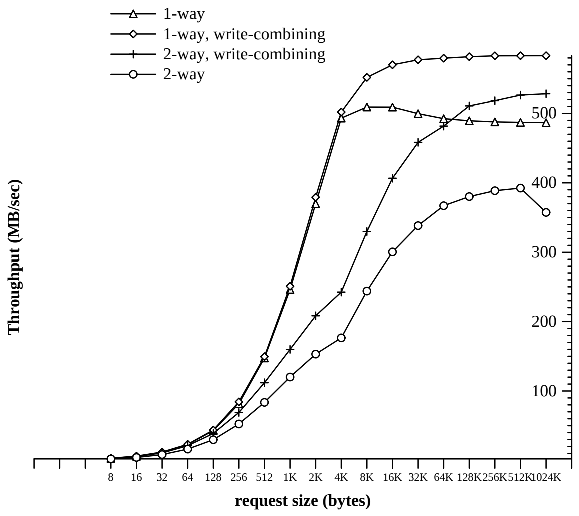


Figure 3. Base communication performance, for two versions of the NIC.

2.4.30. We divide the remote I/O path framework into two major parts, corresponding to the two sides in an I/O operation, initiator and target. Our protocol is implemented as a block-level driver module for the initiator and target. To use RDMA operations the initiator and the target each maintain a queue and a pool of data buffers. The queue at the target stores I/O commands, produced by the initiator. The queue at the initiator stores completion notifications, forwarded by the target upon completing commands that it has dequeued from its queue. The overall architecture of our I/O protocol stack is illustrated in Figure 2 [17]. The initiator and target modules that implement the remote access protocol initially establish their association over a TCP/IP socket. Then, the target informs the initiator of the actual device parameters: device size, block size, sector size, and read-ahead limits. Finally, the initiator registers the remote device with the local OS kernel. All applications at the initiator can then access the target's exported block device, as if it were a directly-attached disk. This guarantees *transparent* access to the storage available at the target. For example, we construct file systems on top of remote block devices.

2.6. Performance Evaluation

Figure 3 shows maximum achievable throughput for two simple user-space benchmarks. The *write-combining* curves correspond to a version of the NIC that supports write combining. This results in faster posting of transfer descriptors into the RDMA request queue. One of the benchmarks initiates one-way transfers without waiting for a response from the receiver, whereas the second produces a two-way, ping-pong traffic pattern. In all cases, all messages sent are of the same size. When message size exceeds 4 KBytes, the host library breaks the message to 4-KByte segments.

The maximum throughput of the host-NIC DMA engine is one 64-bit word at every 100-MHz PCI-X clock cycle, i.e. 762.9 MBytes/s (assuming zero bus arbitration and protocol overheads). The maximum throughput for the pair of RIO links is one 64-bit word at every 78.125-MHz RIO clock cycle, i.e. 596 MBytes/s. Therefore, the maximum end-to-end throughput is limited to that of the network links.

We use hardware cycle counters at the NIC level to examine the behavior of the host-NIC DMA engine. With the version of the NIC that does not support PCI-X write-combining, we measure that on the order of 10 PCI-X cycles are required for each 32-bit CPU-to-NIC write operation. Initiating a single RDMA write operation requires about 40 PCI-X cycles or about 400ns. We observe that it takes on the order of 50 PCI-X cycles (500 ns) for any DMA operation from the host's local memory (PCI-X read) to begin transferring data. With write combining, we find that writing two RDMA descriptors (i.e. 32 bytes) in the NIC request queue costs about 110 ns, whereas writing a single RDMA descriptor (i.e. 16 bytes) costs about 90 ns.

Delays for writing the RDMA descriptor and commencing transfer of data from the host memory dominate the latency for small transfer sizes. After the initial delays, the DMA engine is capable of reading one 64-bit word per PCI-X cycle and placing it in the transceiver's out-bound FIFO. If there were no further disruptions, this would result in the maximum transfer rate of 762.9 MBytes/s. For a DMA transfer of 4 KBytes (i.e. 512 64-bit words) from a host's memory, we measure a delay of 577 PCI-X cycles, of which 512 cycles actually transfer data words (89% utilization of PCI-X cycles). Of the remaining 65 cycles, 8 cycles

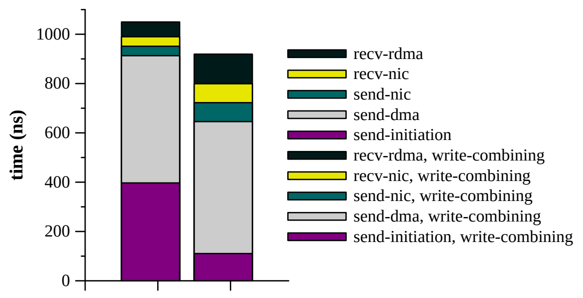


Figure 4. Breakdown of end-to-end latency for (a) a single 8-byte message with single (uncombined) PCI-X writes, and (b) two 8-byte messages with PCI-X write-combining.

are attributed to bus arbitration and PCI-X protocol phases, and 57 cycles are consumed until we receive the first data word. This last time interval is the duration for a *PCI-X split transaction* to complete, and we find it is almost constant (regardless of the DMA transfer size) and in the range 45-60 cycles.

Figure 4 shows a breakdown of the one-way latency for a small, 8-byte message (payload), with and without write combining. The overhead is divided in the following components: send-initiation, send-DMA, send-NIC, 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.

For the NIC version that only supports single PCI-X target writes, we see that the *send-path* is the most expensive part for this type of short transfer. The *send-initiation* component accounts for 40% of the overall latency, while the *send-dma* component accounts for 51.6%. We find that approximately 46 PCI-X cycles out of the *send-dma* component are related to PCI-X (PCI-X *split* duration). This initial delay before transferring any data from the sender host's memory is only amortized for larger transfer sizes. With write combining, the *send-initiation* component accounts for 12% of the overall latency, while the *send-dma* component accounts for 58.2%. Latency for a transfer of two 64-bit words is 12.5% lower than the latency for the transfer of a single 64-bit word without write combining.

2.7. Debugging Methodology

Our NIC debugging effort consists of four stages: First, the debugging of the PCI target interface, including read/write accesses from the host to the NIC, as well as DMA transfers from the host's memory to the NIC. Second, debugging DMA transfers from the NIC to the host's memory. Third, integration of the DMA engine with the network transceiver module, initially in a loop-back arrangement and subsequently in a point-to-point direct connection setup. Fourth, integration of the NIC with the buffered crossbar switch. Since the focus of this paper is debugging the NIC, we provide only a brief description of this stage. Testing the overall system (hardware plus software) reveals problems, mostly at the interfaces between modules and in the hardware-software interface, that cannot be reproduced in simulations.

Our system consists of multiple interacting hardware and system components that need to be tested as an integrated whole. A further complication rises from the fact that the systems software runs on the host processor, so its interactions with the NIC cannot be simulated with hardware simulation tools. Moreover, our networked storage system prototype is interrupt-driven, resulting in inherently asynchronous NIC-host interactions.

A fundamental issue for system-level debugging is that without a full simulation model that incorporates all system-level inputs and outputs, designers have to develop a *test-bench* that provides signals replacing the system-level signals. This can quickly become a tedious, error-prone procedure. In our work, we rely on the `Mentor Graphics ModelSim` [19] and `Cadence NCSim` [8] simulation environments for module-level functional simulation, but verify overall system functionality mostly by using signal and data capturing tools and techniques. The next sections describe in detail the most important correctness and performance debugging issues we encounter.

2.8. Debugging Aids

We have designed a tracing memory module to allow capturing, at each cycle, data words coming in or going out of the NIC. The tracing memory is 72 bits wide and can hold up to 2048 samples. We establish four points for capturing data words, corresponding to the enqueue/dequeue operations on the two FIFO queues that bind the DMA engine to the network transceiver module. We preserve tracing memory contents across host reboots, until they are explicitly reset. These contents can be recovered by a utility that accesses them through the target interface. The tracing memory module exposes an index register that allows the utility to scan through its entries and retrieve contents.

To inspect the state of the network transceiver module, we extend the NIC to include an internal bus, which conveys debugging and timing information from this module to the PCI/PCI-X target interface. We use tracing memories extensively to examine the interactions between the modules. Moreover, we add a register accessible through the target interface that allows us to see the current state in each of the finite-state machines that control the NIC's operation. Status information from the RIO transceivers is also exposed through the target interface.

3. Correctness Debugging

Table 1 summarizes the correctness issues we encounter during system implementation and characterizes the techniques we apply in each case for detecting and resolving them. The following sections discuss each of these issues.

Correctness Issues	
Issue	**Detection**
PCI-X configuration space & PCI-X target	signal capturing (logic analyzer)
PCI-X DMA transfers from/to host	kernel module + counters + signal capturing
RIO transceivers: reliable data flow from/to host, bonding of two links	signal capturing + tracing + counters
NIC flow control	signal capturing
Circuit timing issues	timing analysis tools

Table 1. Summary of correctness issues.

3.1. Lack of PCI-X Information

A main challenge in our work has been debugging issues related to PCI/PCI-X. The NIC card is plugged in a PCI-X slot owned by the south bridge chip, which arbitrates the PCI-X bus and implements protocol timings. To verify the correct functional behavior of our peripheral we need a model for this bridge in order to simulate states of the PCI-X protocol. Unfortunately, we do not have access to a simulation model from the manufacturer of the bridge. For this reason, we develop custom test-benches to simulate most of the bridge behavior according to PCI-X specifications. Such a procedure is tedious and required months to reach a satisfactory level of stability for the prototype. Moreover, there are no readily-available test patterns to simulate the software-induced events and processor intervention in a test-bench. System-level testing and verification revealed several errors that are difficult to trace. The functional simulation and verification tools, ModelSim and NCSim, helped us identify design and protocol errors. However, several corner cases were not possible to check. The fact that our test-bench was built based on an incomplete understanding of the PCI-X bridge hardware, did not allow us to run reliable functional and code coverage tests, as supported by the Cadence NCCov tool and ModelSim's coverage tool.

3.2. PCI/PCI-X Target interface

The PCI and PCI-X specifications do not directly provide a clear sequence of steps for building a DMA-capable peripheral. In our work, we find that design should start with a minimal device that initially includes only the PCI configuration space and target functionality. Having a functional PCI target allows access to event counters and tracing memories built into the NIC.

Debugging the target interface requires the use of a *logical analyzer* in order to capture the address and data lines, as well as signals that govern the interaction between the NIC and the PCI/PCI-X bridge. A main difficulty in this stage is capturing the signals of interest. As we do not have a full-scale PCI development and test environment, we use alternatively a PCI extender card [2]. This PCI extender card requires the NIC to appear to the PCI bus as a 33 MHz peripheral. This restriction does not allow for validating the NIC's timing constraints. However, the setup is sufficient for capturing and validating the sequencing of critical signal transitions, such as the assertion of a stop signal during a DMA transfer from the host's memory. Moreover, this approach is far less expensive than a full-scale PCI development and test environment.

3.3. DMA Transfers

DMA transfers are initiated by writing descriptors in the request queue of the NIC. The NIC device driver allows test programs to access the NIC via read and write accesses to a specially mapped region of memory. Moreover, this device driver reserves a number of memory pages that can be used as the source or destination of DMA transfers between the NIC and the host. This arrangement allows us to test the NIC's DMA capability within a single host, without the added complexity of the network transceiver module.

We have resolved a number of issues related to DMA transfers, by making use of event recording mechanisms embedded in the NIC. Specifically, we use counters for tracking the number of words transferred in each direction, as well as tracing memories to capture the actual words transferred. With early versions of the NIC, we have had to deal with word omission and duplication errors due to incomplete handling of PCI/PCI-X bus interactions. We take advantage of the signal capturing functionality offered by the Xilinx ChipScope tools package [4] to generate on-chip "logical analyzers" and watch signal timings via a JTAG port. ChipScope uses free logic cells in the FPGA and on-chip memory, to install on-chip "signal taps" to the data-paths of the circuit selected by the designer. An important feature of this tool is support for triggers, in a manner similar to a real logic analyzer.

Extending the functionality of the PCI-64 NIC to a PCI-X NIC requires major modifications in the structure of the DMA engine. Under the PCI-X bus protocol, a DMA read access usually requires the NIC to start the transaction in initiator mode and complete it in target mode. This happens when the PCI-X bridge signals a *split-transaction* where the data cannot be pulled immediately by the NIC; Instead, the bridge pushes them later in time, when they become available from the host memory. Meanwhile, the NIC can also be accessed by the systems software running on the host processor. Similar complications arise in handling the completion of a transaction. The validation of target interface becomes by necessity interwined with the validation of the DMA engine, a factor that increases complexity.

In many occassions, we have encountered unanticipated behavior, race conditions and corner cases that could not be modeled in functional simulations. Understanding and tracing such situations required extensive monitoring with the debugging tools embedded in the NIC. We mostly relied on using tracing memories to store time-series of events that were later decoded and inspected off-line to detect the causes of the unanticipated behaviours. We also used the Xilinx ChipScope tools package to establish triggers for signal capturing upon certain corner conditions.

3.4. Network Transceivers

We first test the network transceiver module in a loopback arrangement, and later in a point-to-point direct connection setup. We observe various forms of data scrambling. Improper RIO attribute settings lead to overflows or underflows on the data lines with varying results at the receiver side. Problems arising from clock differences between the sending and the receiving RIO transceivers are complicated to trace. Moreover, they cannot readily be revealed through simulation, as they occur after quite long periods of operation. To counter such problems, we capture data at the transceiver level. For this purpose, we use tracing memories to store data, and then software utilities to read, decode and analyze them off-line. Moreover, we use the Xilinx ChipScope tools extensively to obtain snapshots of the system state.

We find that the use of CRCs at both headers and packet payload is needed to detect data scrambling problems. We have implemented a mechanism to "freeze" the NIC after detecting such a problem. This allows us to capture state information as close in time as possible to the instant when the error occurred. The NIC includes several counters, accessible via the target interface, to track important events, such as the arrival of a packet with a CRC error.

Since the network transceiver module bonds together two RIO links, an important issue to be resolved is the synchronization of transmission and reception from the two links. This procedure relies on a pair of FIFO queues. Errors in this part of NIC manifest themselves as data corruption events. The NIC appends two CRC values to each outgoing message, for the header (CRC16) and the payload (CRC32) respectively. Loss of synchronization between the two RIO links usually results in one of the synchronization FIFOs to become empty. With the particular FIFO implementation that we use, dequeueing from an empty FIFO returns the last valid data item that was dequeued before the FIFO became empty, thus resulting in duplicate data words. We use simulation extensively to check the behaviour of the RIO transceivers, varying the parameters of the RIO transceivers to trigger the observed problems.

3.5. Credit-based Flow Control

The NIC relies on a credit-based flow control protocol, to prevent buffer overflows that would lead to data loss. In order for the credit protocol to work, correct initialization of the word counters is needed. The FIFO size at the receiving side end must be taken into account, so that at start-up this is the free space known to be available downstream. A complication that also needs to be handled is that a cumulative counter will eventually overflow. In order to compensate for this, the comparators at each side must be aware of value wrap-around; otherwise, buffer overflow is likely to occur at the receiver. To overcome this problem, we add two extra bits for each counter to implement this mechanism.

Our tests of the credit-based flow control mechanism have revealed cases where a header or its associated CRC is occasionally not calculated properly at the enqueued or dequeued data stream. In turn, this causes the word counters to slip in a slow but steady way, eventually leading to buffer overflows. Simulations failed to reveal such errors, as it would have to last for very long time periods in order to reach such states. A debugging technique that we use is to inject credits into the payload of the packets. Then, with the aid of a software utility and the tracing memories in the NIC, we track the updates of credits. Another issue is the occassional corruption of credits, as a result of interference between NIC modules. To ensure that no erroneous credit is taken into account, we extend credits to include a parity bit. A credit is discarded in case of parity mismatch.

Finally, flow control-related tests need to verify both that the protocol is neither too conservative, leading to performance degradation, nor too optimistic, leading to buffer overflows and data loss. This is a point where correctness and performance debugging become interwined. In order to check the "back-pressure" caused through the credit mechanism, during tests we disable and re-enable DMA capability in the receive-path. In configurations built around a 8x8 crossbar switch, we also carry out tests from many transmitters to a single receiver, causing network paths to become congested, thus stimulating back-pressure.

3.6. Timing Issues

Adding tracing capabilities to the NIC comes at the cost of increasing the overall footprint and complicating the place-and-route step in the synthesis process. Moreover, the selection of tracing points directly affects the timing of the overall circuit. For example, with the PCI version of the NIC it is possible to capture at each 66 MHz PCI clock cycle signals related to PCI activity. This is a tremendous help in ensuring that the NIC conforms to PCI specifications. However, this is not possible with the 100 MHz PCI-X version of the NIC, where the DMA engine's clock time has been significantly reduced (from 15.15 ns to 10 ns).

Moreover, the stricter PCI-X timing specification requires careful checks concerning the placement and routing of signals. PCI-X requires that signals follow certain timing constraints in order for the bus protocol to work reliably when the signals cross several inches of PCB on the host motherboard. Violating such constraints results in unreliable operation of the NIC as a whole, with varying manifestations, including incomplete transactions and data corruption at either the sender or the receiver-side. These constraints are challenging for FPGA-based prototypes and a significant amount of effort has been dedicated to checking conformance to the timing constraints. For timing constraint verification we use the Xilinx Timing Analyzer tool.

3.7. Integration of Crossbar Switch

We check the basic correctness of the buffered crossbar switch alone, through functional simulations on post-place-and-route models using ModelSim and NCSim. Then, we use debugging and monitoring tools inside the switch. We use ChipScope, and also a custom tool for monitoring the switch via a serial terminal. This tool consists of a PowerPC 405 [5] processor that is provided as a hard block inside the Virtex II PRO FPGAs, a custom processor bus peripheral and a serial port controller. The embedded processor runs a program that receives monitoring data from the switch and sends them to a terminal connected through a serial port. This data includes counts of the words transferred for each of the flows going through the switch, the occupancy of crosspoint buffers, and cycle counts for the time required for forwarding a packet from a source to a destination.

4. Performance Debugging

Table 2 summarizes performance issues that we have come across.

Performance Issues	
Issue	**Detection**
High interrupt frequency	counters
Sender-side notification	counters
RDMA Initiation cost	counters
Conservative flow control	rate sampling
I/O protocol inefficiencies	measurement

Table 2. Summary of performance issues.

4.1. High Interrupt Frequency

Previous work has shown that interrupt cost can be extremely high in high-performance networks [26]. Given that the I/O path in the kernel relies on interrupts for I/O request completion, it is important to reduce the number of interrupt handler activations. In our networked storage prototype, interrupts are required for messages that manipulate queue head/tail values, so that the I/O target and the I/O initiator get to consume I/O requests and completion notifications, respectively, from their respective queues.

PCI-X interrupt delivery requires asserting the level-triggered interrupt line assigned to the NIC. This line has to be cleared before the NIC can deliver a subsequent interrupt. Once the top-half interrupt handler begins its execution, it disables further interrupt delivery and schedules the execution of the bottom-half handler that handles the details of issuing a block-level I/O request (target-side) or handling an I/O operation completion (initiator-side). Once the bottom-half has finished execution, it re-enables interrupt delivery.

We modify the NIC's interrupt delivery mechanism so that the actions of enabling and disabling interrupts have the side-effect of clearing the (level-triggered) interrupt line assigned to the NIC. The bottom-half handler processes all I/O requests or completions present in the queue. In the meantime, the NIC may deliver additional requests, which however will *not* cause additional interrupts. When the bottom-half handler has finished processing all requests in the queue, it enables NIC interrupts. Since the bottom-half handler is interruptible, we have to ensure that there is no race condition between enabling interrupts and returning from the bottom-half handler by re-checking for new requests in the queue as soon as interrupts are enabled. In this manner, at high loads, the number of interrupts is reduced, since no interrupts are delivered as long as a node is processing other requests.

4.2. Sender-side Completion Notifications

On the send path, upon completion of each out-bound RDMA transfer, the issuer needs to deallocate the local handle. Traditionally, NICs notify the host with an interrupt when the send-path processing has finished. However, interrupts incur high overhead. Instead, we use NIC support for *local notifications*: When a local DMA operation completes, the NIC writes back (via DMA) a notification (in the form of a 64-bit word) to a specified location in host memory. The issuer, instead of spinning on this notification word to eagerly free the transfer handle, checks lazily for free transfer handles, when more requests are posted later. This functionality of the NIC helps to reduce the host processor's load in the send-path.

4.3. Conservative Flow Control

Flow control prevents the sending NIC from transmitting packets to a receiving host, if it is estimated that the new packet will overflow the receiver's inbound FIFO buffer. This computation relies, among other factors, on an estimate of the network round-trip time (RTT) [13]. The accuracy of this estimate affects correctness, as an optimistic estimate results in overflowing the receiver's buffer. This problem can be detected by using the status registers in the receiving NIC.

The RTT estimate also affects performance, as a conservative estimate reduces the transmission rate. In order to counter this performance issue, the NIC incorporates a tracing facility to record samples of the instantaneous throughput, both incoming and outgoing. This facility uses a tracing memory that records a number of samples, which is defined by a register exposed through the target interface. These are samples of the cumulative count of 64-bit words transmitted and received. From these samples we get throughput estimates. Moreover, the tracing facility records the occurrence of back-pressure events [15] during the time-line of the sampling process. Using this facility to study the performance of an earlier NIC version, we have been able to determine that a finer-granularity estimate of the available buffer space should be implemented. In this manner, we reduce the frequency of back-pressure events, resulting in a 10-15% increase of the achievable throughput, to the levels shown in Figure 3.

4.4. I/O Protocol Inefficiencies

The issues so far are not specific for a particular application of the NIC. We have also concerned ourselves with issues specific to the efficiency of the remote block-level I/O protocol (see Section 2.5). A limiting factor for the achievable remote I/O throughput is the presence of short-size RDMA transfers, needed by the remote I/O protocol for managing queues of I/O commands and completions. Accordingly, we have adapted the I/O protocol so that queue head and tail values are not explicitly updated, via RDMA transfers. Instead, command and completion messages incorporate a `pending` flag, so that the recipient can identify the messages that need to be processed; effectively, the recipient determines on its own how to update its queue tail. As discussed in [17], this change in the messaging pattern between the I/O initiator and the I/O target results in up to 10% improvement in the achievable block-level I/O throughput, as measured for large transfer sizes.

5. Related Work

Very little has been published about the process of debugging high-performance NICs, as the related papers mostly focus on the overall design [7, 23, 10, 24] and performance characteristics [22, 9, 16]. This is understandable, as these are the key differentiation points for high-performance NICs. However, since the debugging process is not presented in any detail, no experience or insight into the practical aspects of developing such NICs is conveyed. We believe that presenting highlights from the hands-on process that we follow to debug our NIC, is a contribution of practical interest that supplements the design and performance aspects. A related development that we find promising is the availability of test-cases to verify standards conformance for iWARP-compliant NICs [12].

A point that came up during this process is that the development of a NIC is fundamentally a problem of hadware-software co-design. Current-generation design tools do not adequately support such a co-design approach [11]. The authors in [6] present a method for performance debugging of distributed systems where the components are "black boxes". However, following this approach requires collecting traces of system activity at various interface points to infer the interdependencies between observed events. Our experience shows that such tracing facilities can be incorporated into every major system module during design. Eventually, information from these sources may be collected and processed by integrated debugging and monitoring frameworks [25].

6. Concluding Remarks

In this paper, we present our experience from developing and debugging a PCIX-based RDMA-capable NIC. We discuss the most important correctness and performance issues we encounter. We also present the methodology and tools we developed to detect and resolve each issue. Although the NIC presented in this paper is implemented as a PCI-X peripheral, we believe that the methodology and tools can also be applied in a PCI Express implementation. Since PCI Express is a packet-based interconnect [21], we could capture all packet-related information at all protocol layers, and also measure performance-related events using counters.

Overall, we believe that with high-performance system designs becoming more and more mainstream, it is essential to provide standardized mechanisms for extracting state information from each component in the system. Such mechanisms should (1) allow for selective activation, (2) include aggregate counts of events of interest, and (3) also include state information snapshots in the form of histograms and time-series. As we observe in the course of our work, these mechanisms can be very expensive to build and integrate when issues arise. Instead, they should be included from the start in the design, and be treated as important, maintainable functions.

7. Acknowledgments

We would like to thank the members of the CARV laboratory at ICS-FORTH, in particular Konstantinos Xinidis, Aggelos Ioannou, Giorgos Mihelogiannakis, and Prof. Manolis Katevenis. We thankfully acknowledge the support of the European FP6-IST program through the SIVSS (STREP-002075) and UNIsIX (MC-EXT-509595) projects, and the HiPEAC Network of Excellence (NoE-004408).

References

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

[2] Pci extender card, part no. 7564-uextm rev. b. Twin Industries, Inc, http://www.twinind.com.

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

[4] Xilinx chipscope pro. Xilinix, Inc. http://www.xilinx.com/ise/optional_prod/cspro.htm.

[5] Powerpc 405 processor block reference guide: Embedded development kit, 2005. Xilinx, Inc. http://www.xilinx.com/bvdocs/userguides/ug018.pdf.

[6] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[7] 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, 1995.

[8] I. Cadence Design Systems. Ncsim users manual. http://www.cadence.com.

[9] D. Dalessandro and P. Wyckoff. A performance analysis of the ammasso rdma enabled ethernet adapter and its iwarp api. In *Proceedings of the RAIT Workshop (in conjunction with the IEEE International Conference on Cluster Computing)*, 2005.

[10] 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.

[11] H. Hsieh, L. Lavagno, and A. Sangiovanni-Vincentelli. Embedded system codesign: Synthesis and verification, 1996.

[12] U.-I. iWARP Testing Consortium. iwarp test suites. http://www.iol.unh.edu/testsuites/iwarp/.

[13] G. Kalokairinos, V. Papaefstathiou, A. Ioannou, D. Simos, M. Papamichail, G. Mihelogiannakis, M. Marazakis, A. Bilas, D. Pnevmatikatos, and M. Katevenis. Design and implementation of a multi-gigabit nic and a scalable buffered crossbar switch. Technical Report 376, FORTH-ICS, 2006.

[14] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos. Variable packet size buffered crossbar (cicq) switches. In *Proc. IEEE Conference on Communications (ICC 2004)*, 2004.

[15] 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.

[16] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, and D. Panda. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*, 24(1):42–51, 2004.

[17] M. Marazakis, K. Xinidis, V. Papaefstathiou, and A. Bilas. Efficient block-level i/o over an rdma-capable nic. In *Proceedings of the ACM Int'l Conference on SuperComputing (ICS)*, 2006.

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

[19] I. Mentor Graphics. Modelsim se users manual. http://www.model.com.

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

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

[22] F. Petrini, F. E., and A. Hoisie. Performance evaluation of the quadrics interconnection network. *Journal of Cluster Computing*, 6(2):125–142, 2003.

[23] F. Petrini, W. Feng, A. Hoisie, S. Coll, and F. E. The quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, 2002.

[24] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2001.

[25] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. IEEE Scalable Parallel Libraries Conf.*, 1993.

[26] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *IEEE Computer*, 37(11):48–58, 2004.