A Configurable Network Protocol for Cluster Based Communications using Modular Hardware Primitives on an Intelligent NIC *

Ranjesh G. Jaganathan   Keith D. Underwood †   Ron Sass

Parallel Architecture Research Lab
Holcombe Department of Electrical
& Computer Engineering
Clemson University
105 Riggs Hall
Clemson, SC 29634-0915

Sandia National Laboratories†
P.O. Box 5800 MS-1110
Albuquerque, NM 87185-1110

{jranjes,rsass}@parl.clemson.edu, kdunder@sandia.gov

Abstract

The high overhead of generic protocols like TCP/IP provides strong motivation for the development of a better protocol architecture for cluster-based parallel computers. Reconfigurable computing has a unique opportunity to contribute hardware level protocol acceleration while retaining the flexibility to adapt to changing needs. Specifically, applications on a cluster have various quality of service needs. In addition, these applications typically run for a long time relative to the reconfiguration time of an FPGA. Thus, it is possible to provide application-specific protocol processing to improve performance and reduce space utilization. Reducing space utilization permits the use of a greater portion of the FPGA for other application-specific processing.

This paper focuses on work to create a set of parameterizable components that can be put together as needed to obtain a customized protocol for each application. To study the feasibility of such an architecture, hardware components were built that can be stitched together as needed to provide the required functionality. Feasibility is demonstrated using four different protocol configurations, namely: (1) unreliable packet transfer; (2) reliable, unordered message transfer without duplicate elimination; (3) reliable, unordered message transfer with duplicate elimination; and (4) reliable, ordered message transfer with duplicate elimination. The different configurations illustrate trade-offs between chip space and functionality.

KEYWORDS: Intelligent Network Interface Card, reconfigurable computing, networking protocols, cluster computing

1. Introduction

Reconfigurable Computing (RC), like low-cost clusters, is an alternative technology to traditional high-performance computing systems [7]. For a number of years researchers have demonstrated that hardware circuits realized in RC logic are very effective at accelerating specific applications. In an effort to exploit RC technology and broaden its general usefulness, the Adaptable Computing Cluster (ACC) project focused on an extension to the Beowulf (commodity cluster, open source software) architecture that places reconfigurable resources (FPGA) in the network data path of every node. In other words, the ACC integrates RC and a commodity NIC, forming an Intelligent Network Interface Card (INIC). This has been shown to have a significant impact on performance and is cost-effective for several classes of applications [20, 22].

The INIC offers several modes of operation. With a pass-through configuration, the INIC can operate as a standard high-speed NIC while the RC resources are free to be used as a standard RC peripheral card. Another mode, explored in several papers [19, 21], uses the INIC as a combined com-

*This project was supported in part by the National Science Foundation under NSF Grant EIA-9985986. The opinions expressed are those of the authors and not necessarily those of the foundation. Support was also received from Code 935 under NASA Grant NAG5-11329.

†Dr. Underwood was supported by a NASA GSRP Fellowship under grant number NGT5-85

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC'03, November 15-21, 2003, Phoenix, Arizona, USA
© 2003 ACM 1-58113-695-1/03/0011...$5.00

putation and communication processor where the data coming to and from the network is operated on in transit. The third mode has the INIC assisting communication by offloading protocol processing. This work explores the third mode: the INIC as a communication assist where the entire communication protocol is configured and moved to the INIC.

Ideally, one might consider putting TCP/IP or another complete communication protocol stack on the INIC. However, the dedicated microSPARC is much too slow for Gigabit and 10 Gigabit line speeds and to implement TCP in reconfigurable computing would swamp the largest FPGA chips available. Instead, we take another approach: in this paper we propose a *configurable* protocol suite. By carefully defining interfaces and building features as independent components, a programmer can simply identify the desired features of their customized protocol. The appropriate components are then stitched together, synthesized, and downloaded to the INIC's FPGAs. Since reprogramming times are very fast (milliseconds) and applications in Beowulf clusters often receive dedicated nodes, this is a viable technique for deploying application-tuned communication protocols, thus enabling a trade-off between RC resources and communication features.

Most Beowulf programmers implicitly choose TCP/IP as an underlying communication protocol. TCP/IP offers a reliable, in-order message transfer service and although most parallel algorithms are built around this assumption, it is not necessarily always required. For example, many algorithms that redistribute data do not care what order the data arrives so long as it is reliably delivered. Likewise, a common discrete event simulation code does not rely on order so long as all the messages arrive during their appropriate quantum. Certain *rendezvous* type communications implicitly force the synchronization, effectively ordering the messages. Further, many commodity clusters are deployed with network switches which guarantee in-order delivery, making the a software re-ordering component redundant. Perhaps more surprisingly, there are several cases where 100% reliable services is not necessary for correctness. For example, many iterative numerical algorithms are very robust and most network switches very reliable. In such a situation, an iterative algorithm can be programmed to simply reuse a value from the previous iteration on the rare occurrence of a lost update. Because it is robust, the algorithm can converge in nearly the same number of iterations but — because of the performance gain of using unreliable messages — there is an overall performance gain for the application.

There are several performance reasons for wanting a communication protocol in the network interface card. First, any protocol processing performed by the card is computation that the host does not have to do. In a message-passing program, a very common technique is to overlap the communication with the computation; reducing the CPU load on the communication part improves the overall performance. In addition, because the card is aware of the protocol, it can interact with the network without host intervention. This has two important consequences. First, it reduces the latency of the overall protocol because short messages such as an ACK do not have to cross the peripheral bus twice. Second, the host does not have to service an interrupt or perform a context switch — thus saving several operations that contribute factors to the latency of the response and are pure overhead. While simple unicast messages are demonstrated in this paper, the architecture is generic enough that it is easily extended to support more complex, collective communications such as barrier synchronization [20], multicast, and reductions [11].

The advantage that reconfigurable computing brings to this effort is a high-performance, yet configurable, computing fabric. Traditional processors frequently have trouble providing adequate processing power for high rate protocol processing, particularly in the power constrained embedded environment of a network interface card[18]. However, configurability is still required as protocols have a tendency to change over time[2]. More importantly, with reconfigurable hardware on the network interface, it becomes possible to customize the protocol based on the current application. This can even be extended to include application specific processing with the protocol.

The work described here concentrates on using the INIC as a communication assist for Beowulf-class architectures running message-passing parallel codes with modern, high-speed networking such as Gigabit Ethernet. The design of a component-based, configurable network protocol for the INIC is described. Several modules have been implemented to demonstrate that the architecture is feasible. An area analysis is included to illustrate the impact of customizing the protocol to meet the specific needs of an application. The goal of the work presented here is to establish the feasibility of such an architecture. Unfortunately, appropriate hardware to do complete performance testing is unavailable; however, our prototype board does firmly establish the feasibility of this approach and the performance measured is competitive.

The rest of this paper is organized as follows. In section 2 the general problem of building a communication assist for an INIC is described. A description of the protocol is then presented in section 3 followed by a description of the hardware design in section 4. In section 5, a preliminary evaluation that compares the size of each configuration is presented. In section 6 some of the related work are discussed. Finally, conclusions are presented in section 7.

2. Communications Assist

Traditional Beowulf clusters use general-purpose protocols like TCP/IP for networking. While it is easy to use, it has disadvantages that limit its usability. Generic protocols do not consider the architecture of the network or the needs of the application. Even in commodity protocols, there is a recognized need for different protocols for different applications. UDP and RTP complement TCP in an attempt to satisfy the need for a generic protocol for each general class of applications.

Since TCP/IP is often used in Beowulf-style clusters, a closer look at its drawbacks in such an environment proves useful. Most of the network switches used in Beowulf clusters provide ordered delivery of packets and have a very low drop rate. Also, unlike the Internet, the bandwidth of the network at hand is known in most clusters. Features of TCP implementations designed to accommodate these differences in the Internet limit its performance on clusters. Examples include delayed acknowledgments (requiring larger windows), slow start, and congestion avoidance.

Figure 1 (a) shows the data path between the application and the high-speed network in a computer using general-purpose protocols. Note that every packet (data and ack) has to move across the relatively slow PCI bus and every such transfer involves interrupt processing and most likely context switches. With the increasing network speeds and decreasing latency of newer commodity networks, the number of context switches will increase, causing considerable overhead. In order to reduce the number of bus transactions and to achieve maximum performance, an ideal solution is to have a hardwired protocol on the NIC. However, there are numerous competing protocols and fixing one general-purpose protocol is inconsistent with commodity general-purpose hardware. It eliminates the benefits of commodity, large-scale production.

An INIC directly addresses the desire for a on-NIC protocol without losing the advantages of commodity processors. In the communications assist mode of operation, the entire communication protocol is migrated to the INIC. As shown in Figure 1(b) the host is never interrupted when receiving ack packets and only data is transferred over the PCI bus. Transactions between the processor and the NIC now occur at the message level, and parallel programs are often written with a message-passing library. Note that while presently there is no library in our setup, there is nothing preventing the use of one. This also allows the INIC to address another limitation of commodity networks: small packets. Small packets can result from a limited network MTU or from various types of parallel communications patterns. Whatever their source, in a commodity system, small packets drastically increase the packet processing overhead by increasing the number of interrupts and bus transactions.

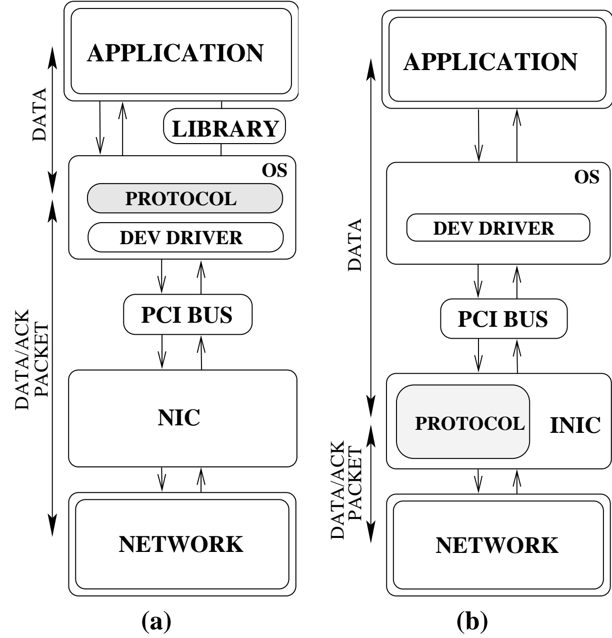


Figure 1. (a) Traditional data path between application and network (b) proposed path between an application and the network

By interfacing the INIC at the library level, the overall number of bus crossings can be drastically reduced.

The INIC, with its RC resources, offers a novel way to address the problems with generic protocols. The RC component of the INIC allows arbitrary hardware designs to be realized on the NIC. This, in turn, makes high-speed NIC-based protocols feasible. With non-RC hardware, each new protocol requires a new design, verification, implementation, and deployment cycle. To remain a commodity, these cycles sometimes take years to complete since compatibility is necessary. However, the INIC can rapidly deploy an application-specific protocol for a dedicated cluster. Since resources used in the protocol are not available to the application, it is important to minimize the resources used on a per application basis. Thus, the key trade-off in the communication assist problem is: protocol functionality versus CLB (LUT and FF) resources required.

The proposed approach uses components that can be glued together to form a new network protocol. A specific assortment of components can be integrated together based on the required functionality, or space constraints. Desired options might include (1) reliable delivery, (2) ordered delivery, (3) duplicate elimination, and (4) rate control, among others.

To study the trade-off between space and functionality, four different protocol configurations were assessed:

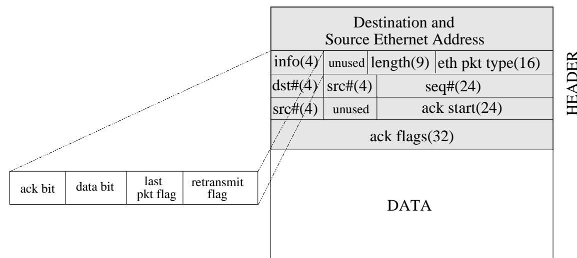


Figure 2. Packet Format

1. Unordered, unreliable delivery;
2. Reliable, unordered without duplicate elimination;
3. Reliable, unordered with duplicate elimination;
4. Reliable, ordered delivery.

These are presented in increasing order of functionality and increasing order of chip space.

3. Protocol Description

In order to implement the protocol in hardware, a very simple, yet fully functional protocol was designed. The packet format employed is shown in Figure 2. One header format is used for both data and ack packets to ease the process of assembling and disassembling the packet. As indicated by the figure, packets are built directly on top of Ethernet packets. In the current format, the length is provided in a nine bit field indicating the number of 32 bit words transfered. This could easily be extended to an eleven bit field indicating the number of bytes transfered; however, the applications utilizing this protocol, thus far, transfer data in increments of words. This is an illustration of how the protocol can be customized on a per application basis.

A four bit info field contains information about the the packet type. The packet can optionally contain an acknowledgment, data, or both. Two additional flags are reserved to indicate the last packet in a message and packets that are retransmits. The next field shown in Figure 2 is a 24 bit sequence number. The sequence number is a packet number (unlike the TCP sequence number) to simplify the ability to perform selective acks and nacks. The next two 4 bit fields include the source and destination nodes. The cluster used in this research project has only 16 nodes, so only four bit fields are needed. In larger clusters, the packet format could easily be changed. For example, the sequence number need not be 24 bits. In cluster networks, packets are never “lost in the network”, so the sequence number only needs to be large enough to handle the number of outstanding packets. An eight bit sequence number would be adequate for many systems. This would leave 12 bits each for the source and destination supporting 4096 nodes. Alternatively, the protocol format could be adapted to larger systems with a simple change to the hardware components.

The ack mechanism is also relatively simple. The first ack field indicates the last consecutively received packet. The validity of this field is indicated by the *ack bit* in the packet header. The second ack field is used to indicate which packets have been received past the last consecutively received packet. This is a 32 bit mask that uses a 0 to denote a packet that has not been received and a 1 to indicate a packet that has been received. Zeros that occur before a one indicate a packet that has been lost. Acks are included with each data packet that is transmitted (assuming packets have been received from the destination node). If packets are received from a node but no data packets are sent to that node, then a forced ack may be required. The ack building mechanism keeps track of the value of the last packet acknowledged. When a packet is received that is more than *threshold* packets away from the last packet acknowledged, an acknowledgment is forced.

On the transmitting side, only a certain number of packets are allowed to be outstanding without an acknowledgment. This value is usually at least $2\times$ *threshold*. Packet transmission stops when the outstanding packet limit is reached. A timer tracks the time between packets being sent to the host in question. When this timer reaches a threshold, the last outstanding packet is retransmitted. This last packet will force an acknowledgment from the receiver using one of two mechanisms. Either the receiver will recognize the retransmitted packet as one that has been acknowledged, and will send another acknowledgment, or the difference in the last acknowledged packet and the transmitted packet will exceed *threshold*. The packet generator recognizes messages that have been completed and disables the timer for those destinations. The simple semantics of this protocol make it easy to implement as a set of modular components in hardware. The specific components that achieve this functionality are described in the following section.

4. Hardware Design

The main design objective is to keep the protocol as modular as possible. Thus, it is important to build a collection of components that interact with each other in a very specific way. The granularity must be maintained such that, at most, one function is provided by each component. In some cases, a single function is broken into a collection of components to maximize flexibility. This enables a change in the implementation of one functionality without a change to or rewrite of the entire protocol. Thus, there is also the implication that multiple implementations of a single functionality might be available to choose from based on the appli-

cations needs. Finally, each component takes parameters to increase the flexibility in a particular configuration. Figure 3 shows the design of a reliable, ordered, rate-controlled[1] protocol using these components. The functionality and interface of each of the various components is explained in the following subsections.

4.1. Standard Interfaces

If a design is to permit component substitution, it is important to have a well defined interface between all of the components. The interface is defined in two parts. First, all of the components communicate using FIFOs. These FIFOs provide for one interface standard across the entire design and the depths of these FIFOs can be parameterized as needed. Second, the communication between components uses a very specific format. If a particular port from a component is not used, a stub is connected to that port to generate (or consume) the appropriate data.

4.2. Packet Format processing

The *xmit* and *split packet* components are the only two components that handle the format of the packet. The *xmit* component is responsible for building a complete packet from the control information it receives. These packets are injected into the network based on rate control information. There are two different types of packets, data packets and acknowledgment packets, that are distinguished by a bit in the packet header. Ack packets are used when acknowledgments are forced. Otherwise, acks are a component of the packet header. The *split packet* component is the peer of *xmit* on the receive side. This component separates the packet into header and data components. These parts abstract the packet format from the rest of the design. This ensures that only these components need to be modified to implement a change in the packet format. The interface for the *xmit* part includes

1. Packet interface to provide information about the next packet to be sent (*pinfo*)
2. Ack interface to provide acknowledgment information for the next packet (*ack_info*)
3. RAM interface to retrieve the data from memory (*ram_data*)
4. Output interface to send the packet to the network (*pkt_dout*)

The packet interface provides the starting address of data, length of the data, and the destination node for the data. The ack interface provides both the base packet number being acknowledged and a bit pattern indicating the packets beyond that base that have been received. This interface also includes a destination node and an extra bit to force the transmission of an ack packet (as opposed to placing the ack information in the outgoing packet). The RAM interface is provided for the *xmit* unit to retrieve packet data from memory. Finally, the *xmit* component can be parameterized with the packet size and the number of nodes supported.

The interface for the complementary component, *split packet*, includes

1. Input interface to receive the packet from the network
2. Header interface to pass the packet header to the design
3. Data interface to pass the packet data to the design

The *split packet* component is the demultiplexor part that splits the received packet into its components. Header information is output on one port while data is output on the other. Other components in the receive processing path require this information to determine how to handle the packet.

4.3. Packet Generation

The *genp* component is responsible for generating packet numbers when data is ready to be sent or when a particular packet needs to be retransmitted. It interfaces with the application, keeps track of the address where the application writes the packet data, monitors received acks, and provides packet information to the *xmit* component. The details of generating the next packet depend on details of the protocol. Factors that are decided by this component include the number of outstanding packets without acks and the amount of data contained in a packet. The interface exported by the *genp* component to the other components is:

1. Ack interface to receive information from incoming packets (*ack_info*)
2. Application interface to track the status of the packet buffer in RAM
3. Packet interface to provide the next packet information to the *xmit* component
4. Timer interface to receive timeout information

4.4. Building Acknowledgments

The *build_acks* component processes information from received packet headers to build acknowledgment information. It also has a port to receive information from the

[1] Rate control has not been implemented yet

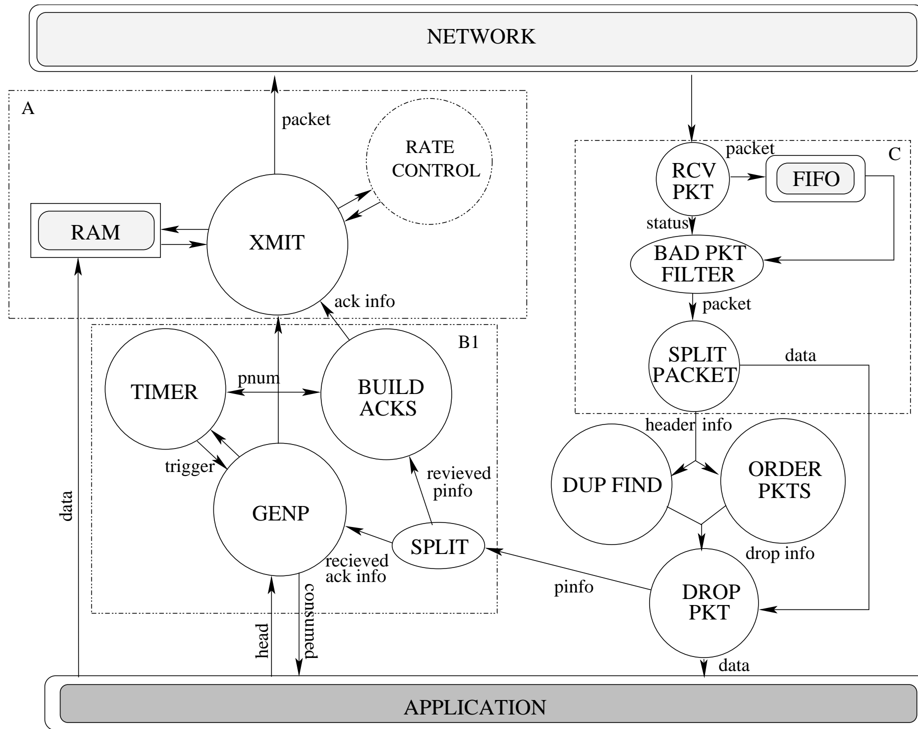


Figure 3. Design of a reliable, ordered, rate-controlled protocol

genp component to inform it about the next packet that is being sent. This allows the *build_acks* component to provide the correct ack information for the outgoing packet. It also tracks which nodes have been acknowledged recently so that forced acknowledgments can be sent if necessary. If a forced acknowledgment is needed, it injects extra acknowledgment into the stream and raises the *forced_ack* bit to notify the *xmit* component an explicit ack packet is necessary. The ports for this component include

1. Packet generation interface to obtain the next packet to be sent
2. Received packet header interface (pinfo)
3. Ack information interface to provide ack data to the *xmit* component (*ack_info, ack_data, forced_ack*)

4.5. Timer

The *timer* component is responsible for making sure that acknowledgments are received for all of the transmitted packets. The timer part monitors outgoing packet numbers and resets the timer for a given destination as soon as a packet is sent to that destination. If the ack information for that packet is received before the timer reaches a programmable threshold, the *genp* component disables the timer. If acks are not received before the timer expires, the *genp* component is notified with a signal indicating which timer expired. The *genp* component then retransmits the appropriate packet. The interface to the *timer* part includes

1. Next packet interface to indicate the next outgoing packet
2. Disable timer interface to allow the *genp* component to selectively disable timers
3. Timer expiration interface to notify the *genp* component of expired timers

In addition, parameters to the *timer* component include the number of processors in the cluster (thus, the number of timers) and the timeout threshold.

4.6. Duplicate Elimination

The *dup_find* component is used on the receiving side of the protocol to find packets with duplicate packet numbers. Packets with duplicate packet numbers occur when acknowledgments are lost. This part communicates with the *drop_pkt* part to drop the packet in such a scenario. This component makes sure that only the data is dropped in case of a duplicate packet. The packet header is passed on to the *build_acks* component to make sure that an ack is sent to the source node to inform it that the packet has been received.

4.7. Packet Ordering

The *order_pkts* component helps order the incoming packets. In the configuration shown above, the *order_pkts* part uses a go-back-N mechanism. This part makes sure that only packets with consecutive packet numbers are passed and all others are dropped. In this case, both the data and the packet info are dropped so that acks will specify that packets after the last consecutive received packet were lost. These packets will be retransmitted by the sender. A go-back-N mechanism was chosen for simplicity on the prototype INIC. A more modern INIC could easily buffer packets received with nonconsecutive numbers and pass that header information to the *build_acks* component. Thus, only the lost data would need to be retransmitted. Note that *dup_find* and *order_packets* components do not communicate with each other. The header information from the *split_packet* component is duplicated so that the *dup_elim* and *order_packets* can compute in parallel. Either of these components can cause a packet to be dropped.

4.8. Support Components

A number of components are needed for minor support functions. For example, the *rcv_pkt* component provides a clean interface to the NIC by handling the oddities of hardware interacting with a PCI NIC. In turn, the *bad_pkt_filter* drops packets that contain some form of error as indicated by the NIC status word. Finally, the *drop_pkt* component takes information from the duplicate elimination and packet ordering components to determine which good packets should be passed to the application.

4.9. Rate Control

Rate control is one of the features that is very difficult to implement in software. With the aid of the rate control component on the INIC, it will be possible to implement cycle accurate rate control and quality of service. Specifically, proper rate control implies that only one packet would be sent for each M microseconds. This is virtually impossible to achieve in software because the software timer granularity cannot (reasonably) be set at a microsecond granularity. Software solutions approximate rate control by sending a burst of P packets every $P \times M$ microseconds. This introduces significant difficulties into the implementation of proper rate controlled protocols. Hardware timers can support a much finer granularity and allow much more direct rate controls. Presently this component has not been implemented.

4.10. Receive side host interface

Transfer of data from the INIC to the host at the receive side can be configured to be either polling or interrupt driven. For performance reasons, our designs use an interrupt driven mechanism. Also the protocol can be configured at runtime with a 'rcv_chunk_size' parameter. This parameter specifies the number of bytes that should be received by the protocol before interrupting the host. The protocol takes care of pooling multiple receives until 'rcv_chunk_size' bytes of data is received and then it generates an interrupt. The ability to provide a custom protocol implementation on a per application basis provides the ability to customize interrupt mitigation to match the needs of an application. Thus, a given application might choose to have multiple interrupt mitigation techniques that could be based on such things as packet size, packet types (if multiple packet types were provided), or source of the packet.

4.11. Levels of Abstraction

At a higher level of abstraction, components can be grouped together. For example, in Figure 3 block B1 represents the set of components responsible for the flow control portion of the protocol. The entire block B1 can be replaced to get an entirely different flow control algorithm. Fine tuning of a particular flow control mechanism can be done by tweaking the parameters of the individual components. Similarly block C represents the components that receive a packet, buffer them, check for transmission errors and disassemble the packets. Finally, block A provides for the formation of packets and the provision of those packets to the network interface at a controlled rate. For simple changes to a protocol, many of the components blocks can be reused. For example, only blocks A and C need to be changed in case of a change in the packet format.

5. Evaluation

Each of the protocols evaluated were tested on a prototype Intelligent Network Interface (INIC). The experimental setup consists of the adaptable computing cluster (a Beowulf cluster with an INIC on each of the nodes). The prototype INIC used was a PCI-bus ACE2 card. As shown in Figure 4, the ACE2 card contains two Xilinx XC4085XLA FPGAs, a microSPARC chip, and a Gigabit Ethernet card. The Gigabit Ethernet card sits on the PCI Mezzanine Connector (PMC). One PLX9080 provides a bridge between the host PCI and the local I960 bus. A second PLX9080 provides a bridge to the local PCI bus.

This platform is an experimental prototype used in our ACC project. As such, it has a number of weaknesses that

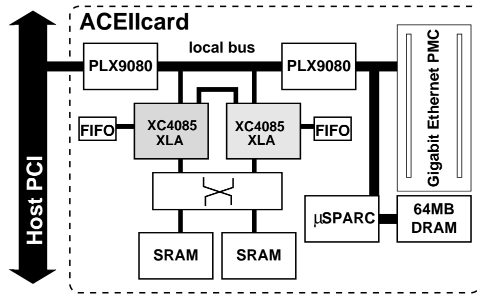


Figure 4. A prototype INIC based on the ACE2 card

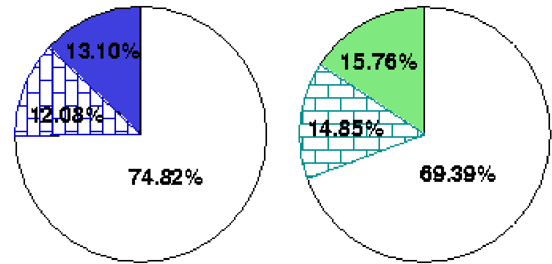
make it generally uncompetitive as a high-performance network interface. Thus, the primary concern here is to show feasibility: a configurable network protocol can be realized on an INIC. Secondary concerns include flexibility and resource trade-offs.

Each of the protocol components were implemented as VHDL entities. Four different configurations were put together as shown in Figure 5. Components in configuration 1 implement a simple unreliable packet transfer protocol. The only component that was specifically designed for this protocol is the packet generator. Unreliable packet generation is a stub that implements no flow control. Packets are generated as soon as there is enough data and sent packets are not monitored. There are no acks generated in such a protocol. In configuration 2, the components implement a reliable, unordered message transfer protocol without the elimination of duplicate packets. Such protocols are appropriate when reliability is necessary but the order of packets (and packet redundancy) do not matter. As an example, this type of protocol was used in [19] at the core of a matrix transpose for a 2D-FFT. The third configuration is a reliable, unordered message transfer protocol with the elimination of duplicate packets. Note that the entire blocks A and C are shared by each of the above protocols. In configuration 4, a complete reliable, ordered delivery protocol is implemented. Note that in this configuration blocks A, B1 and C are reused.

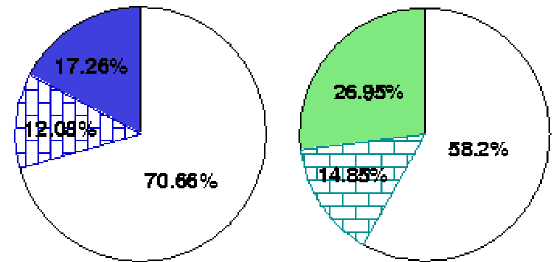
In the following subsections we discuss the feasibility and benefits of this architecture. For the sake of completeness we also discuss the performance of the INIC in latency and bandwidth tests.

5.1. Feasibility and FPGA Resource Usage

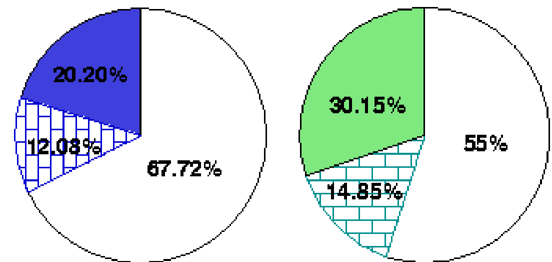
In Figure 6, the pie diagrams show the percentage of chip space occupied by each of the configurations with respect to two XC4085XLA chips. Table 1 shows the different functionalities provided and CLB space consumed by each of the protocol configurations.



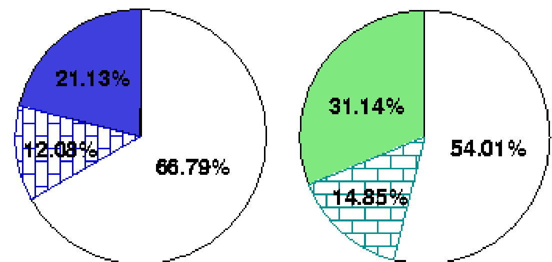
(a) Configuration CONF1



(b) Configuration CONF2



(c) Configuration CONF3



(d) Configuration CONF4

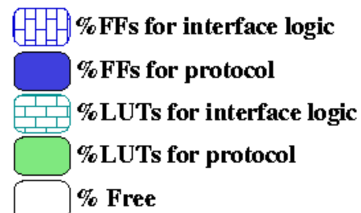


Figure 6. Flip flops (FFs) and look-up tables (LUTs) consumed by the four protocol configurations

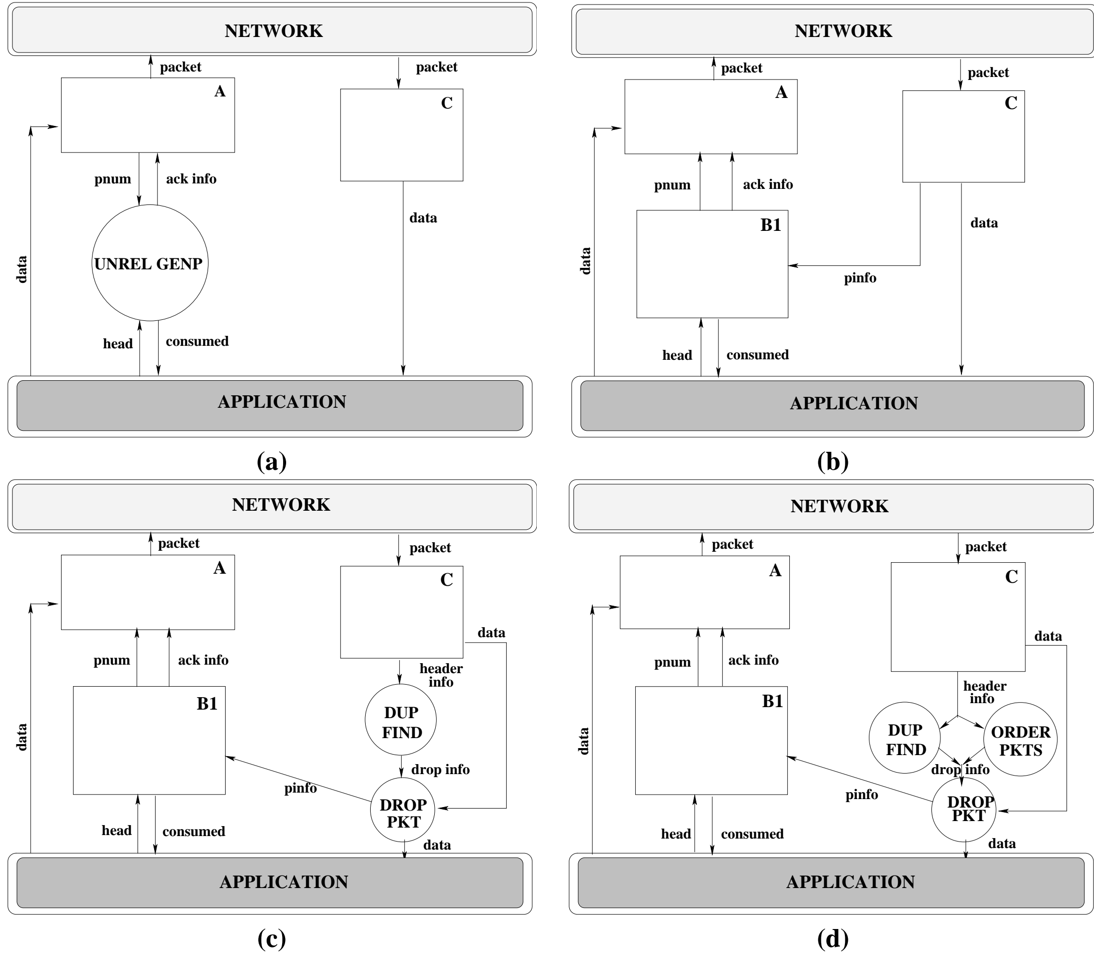


Figure 5. (a) Configuration 1: unreliable unordered transfer without duplicate elimination (**CONF1**) (b) Configuration 2: reliable unordered transfer without duplicate elimination (**CONF2**) (c) Configuration 3: reliable unordered transfer with duplicate elimination (**CONF3**) (d) Configuration 4: reliable ordered transfer with duplicate elimination (**CONF4**)

Table 1. A comparison of space and functionality for the various protocol configurations

Protocol Config	Reliable Yes/No	Ordered Yes/No	Dup elim Yes/No	Num of FlipFlops	Percentage of FlipFlops	Num of 4 LUTs	Percentage of 4 LUTs
CONF1	No	No	No	1643	13.10 %	1977	15.76 %
CONF2	Yes	No	No	2165	17.26 %	3380	26.95 %
CONF3	Yes	No	Yes	2534	20.20 %	3784	30.15 %
CONF4	Yes	Yes	Yes	2650	21.13 %	3933	31.14 %

5.2. Processor Resource Usage

One of the primary benefits of the INIC, as discussed in previous sections is that it reduces the load on the processor. Since the entire communication protocol is migrated to the INIC, the time spent by the processor in protocol processing should be minimal. Figure 7 is a plot of the average processor time usage of UDP, TCP, MPI and INIC on both the send and receive side. In the case of UDP and INIC, packets of size 1408 bytes were used. The results were calculated over a 100MB data transfer. In the case of the INIC, an unreliable ordered delivery protocol was used. Real time is the total time taken by the program to complete. User time is the time spent by the program in user mode, which would include the time taken to process user code and any library processing done (in the case of MPI). System time includes the time taken for processing system calls in kernel mode. Real time would naturally include System time and User time. It would also include time taken in context switches and interrupts and wait times (when the processor is idle). Since the interface between the host and the INIC is the same for all protocols, we can safely assume that the System and User times will be almost the same for any INIC protocol.

From Figure 7, we can see that the system and user time taken in the case of the INIC protocol is much less when compared to the other protocols. Also note that in the case of user time, the INIC takes almost no time. This is because the user program just spends time reading a bunch of registers and starting DMA transfers. The system time would be the time taken by the device driver. Also note that even though the INIC takes less system and user time, it takes more real time. This is due to the limitations of the prototype INIC. This time is spent idle by the processor while waiting for the INIC to finish transfers or receives. This extra processor time is available for other operations. From Figure 7, we can see that even though the prototype INIC does not show a win in performance[2] , the architectural benefits are clear.

In Figure 8, we can see the number of interrupts generated in the send and receive side. It can be seen that the INIC reduces the number of interrupts by a great deal. This is because host-INIC transfers are at the message level and not packet level. Each such interrupt would potentially cause a context switch. This is one of the architectural benefits of the INIC. Moreover, in the case of the receive side, the protocol generates an interrupt every time the received data has reached a threshold. A different method would check the INIC to see if more data is available once the data for the first interrupt is processed. In such a method, if a huge transfer is made, multiple interrupts will be pooled to a single interrupt, still reducing the number of interrupts.

[2]latency and bandwidth

5.3. Performance — Latency and Bandwidth

Even though the prototype INIC is not a good platform for performance study, for the sake of completeness we discuss the latency and bandwidth performance of the INIC briefly.

In Figure 9, we show the latency and bandwidth performance of the INIC compared to other protocols. The poor performance is due to some of the problems with the prototype INIC shown in Figure 4. Every packet has to cross multiple busses in prototype INIC which contributes to latency. Every packet transmitted has to cross the shared local I960 bus (32 bit 33 MHz — 132MBps) four times. This effectively limits the bandwidth that can be attained with this prototype to 264 Mbps when there is a transfer in both directions. Moreover, cycles are lost in bus arbitration further reducing performance. Finally, a bug with the media access controller forces control logic restarts often, costing cycles every now and then. Also, note that the performance results were obtained with the card running at its maximum clocking capacity of 33 MHz. In our second prototype[2], these issues have been addressed and we expect much better performance results.

6. Related Work

With the increasing popularity of Beowulf-style cluster computers, numerous efforts have been made to enhance their architecture. Both high-performance network interfaces[3, 5] and lighter weight protocols[6, 23, 24] attempt to stay within the commodity framework of Beowulf Clusters, but they fail to eliminate all of the protocol overhead on the host. Since work such as the LogP model [13] indicate that the reduction of protocol overhead is extremely important, all protocol processing should be offloaded onto the NIC.

Some modern high-performance network interfaces provide these protocol processing features as well as some other non-trivial computing capabilities [3, 15, 1, 16]. For example, Myrinet provides a LANai processor ranging up to 200MHz[3] and even commodity network cards, like those using the Alteon Tigon chipsets, have begun to add embedded processors[17]. In [4], the authors show that implementing non trivial network operations like collective communications on such hardware improves the performance significantly. Unlike other efforts where the sequential embedded processor limits protocol processing performance[17, 18, 1], the INIC distinguishes itself by providing abundant computing power and allowing the remaining reconfigurable resources to be used for computation.

Researchers have also focused on custom ASIC's in the network interface [10] to accelerate protocol processing. A similar approach to ours has been studied extensively over

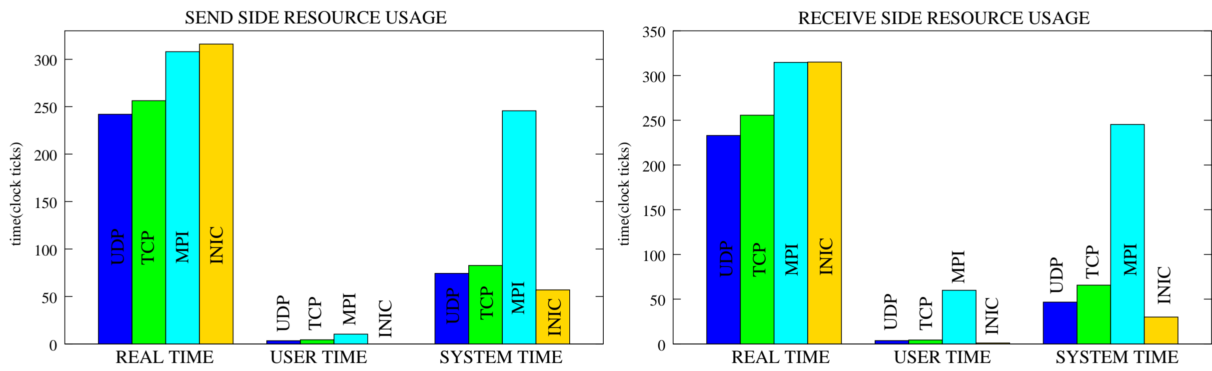


Figure 7. Processor time usage on send and receive of various protocols

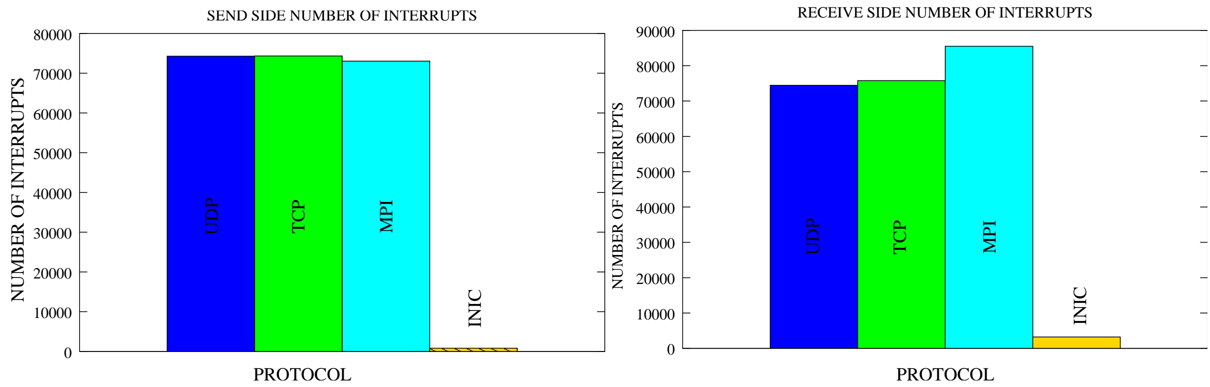


Figure 8. Processor interrupt counts on send and receive side of various protocols

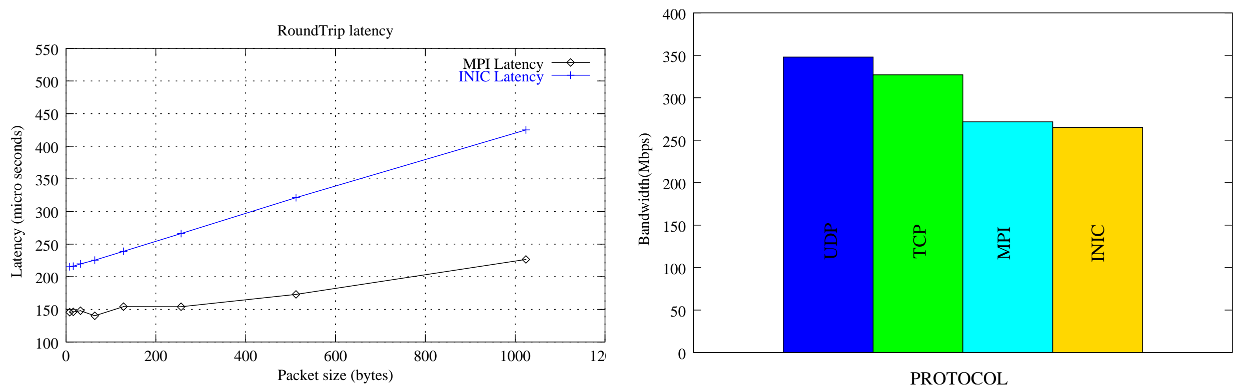


Figure 9. Latency and Bandwidth results

the last several years by researchers and Carnegie Mellon. Like our proposal, the nectar system [1] has identified the importance of moving protocol processing across the system bus and into the network interface. However, nectar is implemented with ASICs which are *fixed* hardware circuits. While ASICs have a larger capacity and could theoretically support all the usual network-level features, there is no way to introduce unanticipated, desirable features without an expensive re-engineering of the chip. Also, there is no way to introduce application-specific computations.

Other work has focused on the use of FPGAs in network interfaces. These include such things as IPSec acceleration on the network interface[2], the acceleration of network intrusion detection[9], and the acceleration of specific applications on a cluster[19]. In addition, FPGAs are commonly used in networking. A few examples include [14, 8, 12] and numerous others. This paper contributes a discussion of the capabilities of reconfigurable computing as an application specific protocol processor.

7. Conclusion

This paper presents a configurable architecture for protocol acceleration in cluster computers. Such an architecture provides excellent flexibility for cluster network interfaces. Hardware protocol options can be configured on a per application basis to maximize protocol performance and minimize the area requirements. This leaves as much hardware as possible available for application acceleration.

The goal of this paper was to demonstrate the feasibility and flexibility of a reconfigurable computing based communication architecture. The key to feasibility and flexibility is the ability to modularize a communications protocol and selectively assemble those modules to form specialized protocols. Future work will focus on building a next generation prototype and analyzing the performance advantages of the various protocol configurations.

References

[1] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste. The design of nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 24, pages 205–216, New York, NY, 1989. ACM Press.

[2] P. Bellows, V. Bhaskaran, J. Flidr, T. Lehman, B. Schott, and K. D. Underwood. GRIP: A reconfigurable architecture for host-based gigabit-rate packet processing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

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

[4] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-level Barrier over Myrinet/GM. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[5] Compaq. Compaq Servernet II SAN interconnect for scalable computing clusters, June 2000. From Whitepaper found at http://www.compaq.com/support/techpubs/whitepapers/-tc000602wp.html.

[6] I. Compaq and M. Corporations. *Virtual Interface Architecture Specification*. December 1997.

[7] K. Compton and et al. An introduction to reconfigurable computing.

[8] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim. An adaptive cryptographic engine for IPSec architectures. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 132–141, Napa Valley, CA, April 2000.

[9] R. Franklin, D. Carver, and B. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[10] C. Kosak, D. Eckhardt, T. Mummert, and P. Steenkiste. Buffer management and flow control in the credit net ATM host interface. volume 20, pages 370–378, 1995.

[11] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[12] J. W. Lockwood, J. S. Turner, and D. E. Taylor. Field programmable port extender (FPX) for distributed routing and queueing. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, pages 30–39, Napa Valley, CA, April 2000.

[13] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[14] J. T. McHenry, P. W. Dowd, F. A. Pellegrino, T. M. Carrozzi, and W. B. Cocks. An FPGA-based coprocessor for ATM firewalls. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 30–39, Napa Valley, CA, April 1997.

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

[16] C. A. F. D. Rose, R. Novaes, T. Ferreto, F. A. D. de Oliveira, M. E. Barreto, R. B. Avila, P. O. A. Navaux, and H.-U. Heiss. The scalable coherent interface (sci) as an alternative for cluster interconnection.

[17] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *Proceedings of the 2001 Conference of Supercomputing*, 2001.

[18] P. Shivam, P. Wyckoff, and D. Panda. Can user-level protocols take advantage of multi-CPU NICs? In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.

[19] K. Underwood, R. Sass, and W. Ligon. Acceleration of a 2d-fft on an adaptable computing cluster. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 2001.

[20] K. D. Underwood. *An Evaluation of the Integration of Reconfigurable Hardware with the Network Interface in Cluster Computer Systems*. PhD thesis, Clemson University, Aug. 2002.

[21] K. D. Underwood, W. B. Ligon, and R. R. Sass. Analysis of a prototype intelligent network interface. to appear in *Concurrency and Computation: Practice and Experience*, 15, 2003.

[22] K. D. Underwood, R. R. Sass, and W. B. Ligon, III. Cost effectiveness of an adaptable computing cluster. In *Proceedings of the 2001 Conference on Supercomputing*, Nov. 2001.

[23] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 40–53, Dec. 1995.

[24] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.