Appearing in: *Proceedings of the 2002 IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Cambridge, Massachusetts, November, 2002.

An Evaluation Architecture for a Network Coprocessor*

Jason Hatashita and James Harris
Department of Electrical Engineering
California Polytechnic State University
San Luis Obispo, CA 93407
email: jhatashi@alumni.calpoly.edu
jharris@calpoly.edu

Hugh Smith and Phillip L. Nico
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
email: husmith@calpoly.edu
pnico@acm.org

Abstract

This paper outlines research currently being conducted by the Cal Poly Intelligent Network Interface Card (CiNIC) project to develop a network coprocessor. The purpose of this coprocessor is to free the host machine from its network processing duties as well as to allow for additional functionality such as hardware-based firewalling or quality of service (QoS) support.

We provide an overview of the current CiNIC architecture as well as an introduction to and evaluation of the next generation CiNIC architecture. Our evaluation consisted of analyzing the performance and capabilities of an FPGA processor in order to determine whether it will meet our future development requirements. The FPGA's performance was tested by timing the execution of the uClinux TCP/IP stack during send operations. The processor's capabilities were tested by adding custom logic to the system and interfacing it with the uClinux operating system. We determined that both the performance and flexibility of the FPGA make it an ideal next generation CiNIC architecture.

Keywords: Network Interfaces, Network Performance, Reconfigurable Architecture, Network Coprocessor

1 Introduction

In recent years, computer systems have become more and more reliant on networks in the performance of their most common tasks. The increase in network communications has brought with it a corresponding increase in the computational burden on the processor to support them. At the same time, microprocessors have become less expensive and more powerful leading many to consider the potential benefits of adding a dedicated network coprocessor to free the host machine's resources for more important work. This goal applies equally to true parallel architectures[8] and networked workstations[7].

The goal of the Cal Poly Intelligent Network Interface Card (CiNIC) project is to offload the network functions from a host machine onto a dedicated network coprocessor. In this case, we define network functions to be those processes that manage the movement of data to and from the network, including TCP/IP stack processing, firewalls, routing, and Quality of Service (QoS) features. Our current development architecture consists of a x86 Linux host machine and a STRONG-ARM Linux coprocessor which sits in the host's PCI bus and appears to the host to be a simple PCI device. The system architecture is shown schematically in Figure 1.

In spite of its simple appearance, the CiNIC coprocessor is a complete linux system in itself. Running a full operating system on our coprocessing board provides developers with a familiar, easy-to-use, environment and affords access to industry-proven software for the coprocessor's network functions. The flexibility of a microprocessor and full operating system on the coprocessor system allows for experimentation with any of the host's software routines. Rapid software prototyping and easy access for performance evaluation make this architecture a powerful and flexible platform for exploring the capabilities of a network coprocessor.

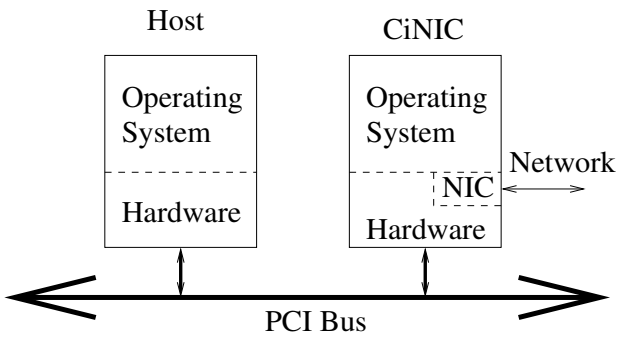

Figure 1. Logical Diagram of CiNIC Architecture

The only major limitation of the current architecture is that its hardware configuration is fixed. Because the demands placed on a network processor may differ significantly from those on a general purpose processor, design decisions based on the expected task-set of one may not be applicable to the other. That is, there might be significant opportunities to improve performance or functionality of the CiNIC if it were possible to tune the hardware architecture.

The next generation architecture for CiNIC intends to incorporate programmable logic into the design in or-

*The work described in this paper was partially supported by the Department of the Navy, Office of Naval Research.

der to explore the benefits of architectural flexibility. One such technology that is well suited to the task is Altera's NIOS processor[1, 2]. This is a 16 or 32 bit microprocessor and peripherals that can be programmed onto a Field Programmable Gate Array (FPGA). This technology provides a processor, peripherals, and custom logic all on one chip. Another reason we chose to evaluate the NIOS processor was the availability of the Microtronix Linux Development Kit[6]. The Microtronix LDK contains hardware expansions for an existing NIOS development board and a port of the uClinux operating system. This provided us an FPGA development board running uClinux that we used to test the performance and features of the NIOS technology.

This paper is an overview of the current CiNIC architecture and discusses our experience evaluating a new programmable logic-based hardware architecture. The first section is a brief overview of the hardware and software architecture of the current CiNIC system. The second section is a discussion of the goals for the next generation CiNIC architecture. The third is brief overview of the NIOS technology and Microtronix uClinux. The final section presents the results of some experiments used to evaluate the capabilities and performance of the NIOS system.

2 Overview of CiNIC Architecture

The goal of the CiNIC architecture is to build a network coprocessing engine that is completely transparent to the user. The current CiNIC architecture consists of a coprocessing system mounted in a host computer. The coprocessing system is running a full Linux 2.4 operating system which provides a software development environment, a TCP/IP stack, and services such as a firewall and QoS support. It is this complete Linux implementation that makes our coprocessing scheme different from others. Instead of the host operating system processing a network function, the parameters for that function call are packaged up and shipped to a dedicated coprocessor for execution. The results are propagated back to the host system after the network function is completed[4].

The intent of offloading the entire TCP/IP stack is to save valuable host processing cycles by allowing network functions to be executed in a coprocessor. In addition to processing savings, one can add network functionality such as firewalling, QoS, and traffic shaping without adding a processing load to the host. Similar to the way that video cards began to offload video processing, the CiNIC architecture takes steps to providing offloaded network functions.

The original CiNIC architecture began with just a simple coprocessor board, but evolved when it was realized that running a second operating system within the coprocessing system provided many benefits. The key to running two operating systems within one system is the use of a non-transparent PCI-to-PCI bridge to separate the devices of the host system and the devices of the coprocessor system. Communication between the two operating systems is accomplished via shared memory scheme in which the host maps some of the coprocessor memory into its own address space.

2.1 Hardware Overview

The current CiNIC coprocessor is an Intel EBSA-285 board comprising the following components:

- SA-110 microprocessor running at 233 MHz,
- 21285 core logic chip set,
- 148 MB of system memory for development purposes,
- a set of flash memory for BIOS images,
- and a serial port to use as a console.

Although a real commercial network coprocessor would not need all of these components, they are included to provide for our development environment with the greatest possible flexibility. This coprocessor card is plugged into an Intel 21554 EB backplane[5]. The bridge is a non-transparent PCI-to-PCI bridge that separates the host PCI devices from the co-host PCI devices, effectively creating primary and a secondary PCI bus. The host processor controls devices on the primary bus and the coprocessor controls devices on the secondary bus. Figure 2 is a detailed diagram of this configuration.

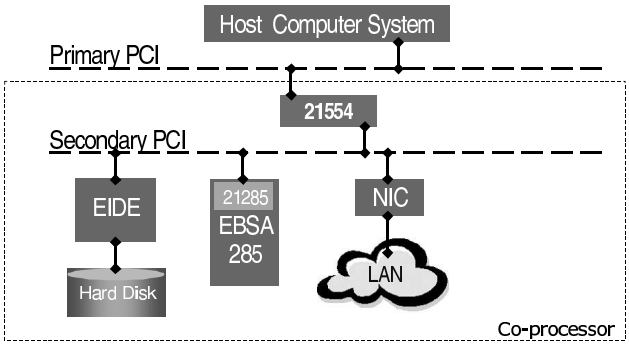

Figure 2. Detailed Diagram of CiNIC Architecture

The addition of a secondary PCI bus, allows any devices having Linux drivers to be easily added to the coprocessing system. In the case of Figure 2, an IDE disk has been added to allow for web caching. What are not shown in Figure 2 are potential devices on the host computer system.

The piece which allows the two computer systems to co-exist is the non-transparent PCI-to-PCI bridge. From both the host computer system and the coprocessor viewpoint, the bridge appears to be a single PCI device, but is responsible for moving data between the two buses. To do this the bridge performs address mappings and address translations. An address range in the primary bus can be

mapped to an address range in the secondary and vice-versa. This allows the bridge to respond to PCI transactions destined for the other side by performing address translation as shown in Figure 3.

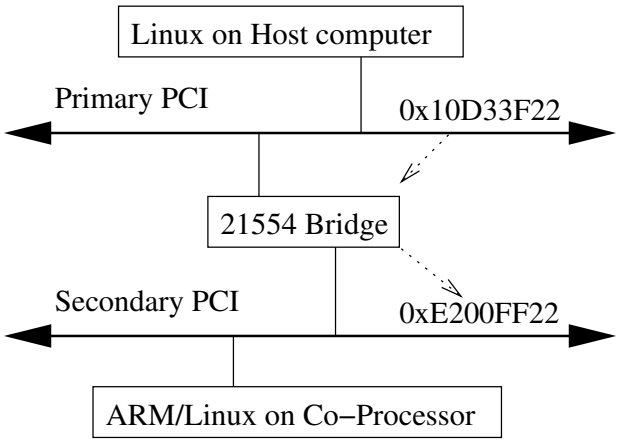

Figure 3. Address mapping through the 21554 bridge

In the example shown in Figure 3, a transaction is started on the primary bus at address 0x10D33F22. The bridge sees this address is in the range of addresses it is translating. Using an internal translation base register, it maps the host address of 0x10D33F22 into the coprocessor's address space on the secondary PCI bus at 0xE200FF22. It then carries out the transaction on the secondary PCI bus using this translated address. If this were a read, then the bridge would return the read data on the primary PCI bus. In the case of a write, the bridge would execute the write on the secondary PCI bus using the data passed to it by the primary bus [9].

2.2 Software Overview

The goal of the system software is to make the coprocessor invisible to applications while providing all TCP/IP state processing. The entire data transfer process between the two computing systems is based on a shared memory scheme. This shared memory is located in the coprocessor's memory and is mapped to the PCI space of the secondary bus via registers in the 21285 core logic. This secondary PCI space is in turn, mapped to the primary PCI space via the bridge. It is in this way that the host system can access the SDRAM of the coprocessor operating system.

To execute the host's network functions, the coprocessor must intercept and redirect the TCP/IP stack calls to the network coprocessor. In the CiNIC architecture this is done at the socket layer. When an application makes a socket system call, a kernel module takes over the processing of the system call. It marshals the parameters for the call, places them in shared memory and invokes the coprocessor to complete the function call. When the coprocessor has completed the function call, it passes the parameters back to the host. The host saves process cycles by scheduling another process while waiting for the socket call to return.

The example above is the ideal operation of each system call. Many situations are much more complex. Functions such as select() which can execute on local file descriptors and sockets, needed to be coded with care. Once the functions were coded, the end result is a library of socket system calls that to the applications and user behave exactly like the standard system calls. Standard applications such as FTP were used to test the functionality of this coprocessing architecture[4].

3 Goals for the Next Generation CiNIC Architecture

The current CiNIC architecture is based on a commercially available microprocessor. Our goal is to move to an architecture using programmable logic which will make more parts of the system customizable. While there is nothing fundamentally wrong with the old CiNIC architecture, a completely customizable system will allow us to experiment with the hardware aspects of offloaded network functions and give us the ability to adapt the hardware architecture to match the needs of the software. This new architecture will no longer be static, but can be changed to enhance the functionality and performance of the coprocessing system. The new architecture that was tested is based on Altera's Excalibur technology: RISC processors synthesized and running on programmable logic. The processor provides the power to run an operating system and the flexibility of programmable logic to implement custom hardware on the chip. It is a classic example of hardware/software co-design: designing hardware and software concurrently to meet system-level objectives[3].

This architecture allows us to conveniently decide what to implement in hardware and what to implement in software. With the added capability of programmable logic, much more of the system is under our control. It would allow us to use a physical shared memory, rather than just mapping the SDRAM of the coprocessor to the host. Since we utilize very few of the PCI bridge's capabilities, this is another component that could be replaced with a more elegant, custom solution. The original CiNIC architecture was developed to aid in network performance analysis and experimentation. Moving the entire platform to programmable logic will provide extra features and allow further analysis and experimentation.

4 NIOS and Microtronix uClinux

The core piece of this project is the Altera NIOS development board. The Altera board is part of a technology they call Excalibur. It is based on running RISC microprocessors on a FPGA and providing room for user-defined logic. The NIOS technology is a soft-core processor, mean-

ing that the hardware for the processor is implemented in a Hardware Description Language. This code is compiled and programmed into the FPGA, making part of the FPGA into a microprocessor. One advantage of this architecture is the ability to use leftover logic on the FPGA for processor peripherals. This allows peripherals like a UART, Ethernet controller, PCI Logic, or IDE controller to be implemented on the same chip as the microprocessor. In addition to peripherals, the architecture lends itself to easy implementation of user-defined logic on the chip. This allows the functionality of a microprocessor, peripherals, and ASIC all on the same chip (see Figure 4). The result is an architecture with extremely flexible software and hardware, perfect for embedded systems work[1].

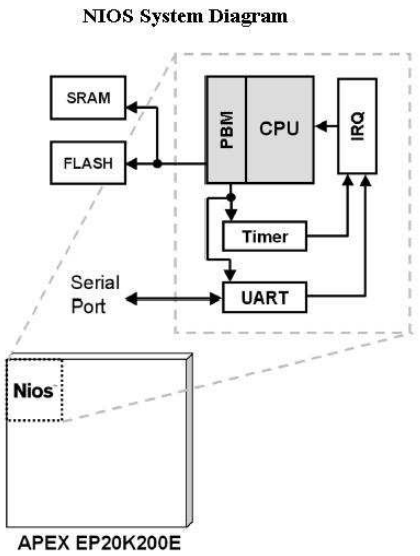

Figure 4. NIOS System Diagram[2]

The Microtronix Linux Development Kit (LDK) is a hardware and software expansion to the NIOS board. It contains extra memory, in the form of 16 Mbytes of SDRAM, a real time CMOS clock, and an 10BaseT Ethernet controller chip plus transceiver. The software expansion is a port of uClinux, a real-time, embedded version of Linux. The embedded aspect refers to the small memory footprint of the operating system kernel. Also ported were the standard Linux tools and commands, cp, ls, etc. With the addition of this kit, the NIOS processor had a fully functional operating system. For more information on the contents of the Microtronix LDK see [6].

5 NIOS System Evaluation

The evaluation of this technology to use as a next generation architecture comprises two aspects: performance and capabilities. First, we needed to determine whether the FPGA processing system could handle the Linux network processing code. Next experimentation needed to be done in order to determine how easy it is to add custom logic to the system and how that custom logic integrates with the Linux operating system. Once these two aspects were analyzed, it was shown that the NIOS system was a good target for moving the next generation CiNIC architecture.

5.1 Performance

Since the purpose of the CiNIC architecture is to improve performance related to network processing, we conducted a series of timing measurements to determine network processing latencies on the new coprocessor architecture. The experiment consisted of running a test application under uClinux on the NIOS development board and measuring the time required to traverse the stack. The test application sent marked packets using the `send()` function. A timer was added to the hardware of the FPGA specifically for counting how long a `send()` function took to execute on the NIOS system. The server that was receiving the packets simply discarded them, since the timing was all done while the data traversed the TCP/IP stack of the NIOS system.

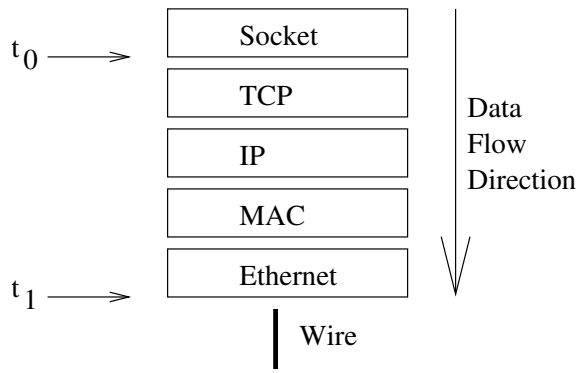

Figure 5. Model of uClinux TCP/IP stack

The timing model is relatively straight-forward. The timings for each measurement were based on the buffer size of the `send()` function. These buffer sizes range from 64 bytes to 64,000 bytes. The timer for the system is started when the `send()` function enters kernel space. The timer is stopped after all the data for the send has been written to Ethernet controller. Figure 5 shows the approximate placement of the timing hooks in the stack.

The hooks themselves are snippets placed into the kernel stack that start, read, and stop the timer. In order to make the ends of the buffers recognizable, each buffer was marked with 6 0xAA characters at the front and 6 0xEE characters at the end. On each buffer size, 2 timestamps were recorded: entry and exit times. *Entry time* is defined to be the time when the first packet of the buffer (the one with the 0xAA's) hits the t_0 marker (see Figure 5). The *exit time* is defined to be the time when the last packet of the buffer (the one with the 0xEE's) hits the t_1 marker.

This test was run 25 times for each buffer size. From there we took the median value of the 25 runs and calcu-

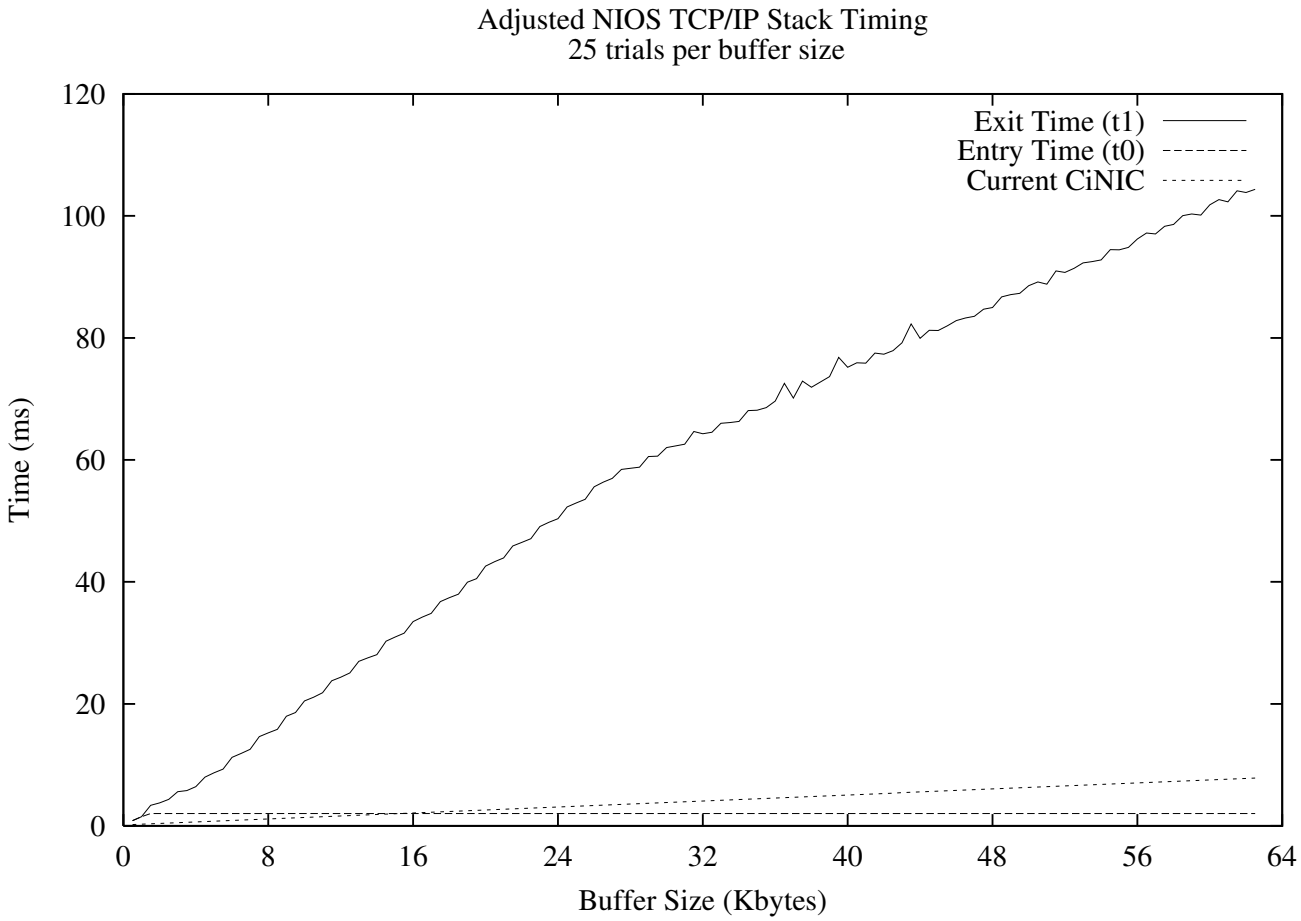

Figure 6. Adjusted NIOS TCP/IP Stack Timing

lated the standard deviation of each of the measurements. The timings showed that 99.5% (3 standard deviations) of the measurements were within 3% of the median value. This means that the uClinux stack on the NIOS system has a consistent processing time. Figure 6 shows the results—adjusted to eliminate the cost of the timestamping itself—and compares them to the results of the current CiNIC TCP/IP stack timings.

The entry time stayed constant as was expected. Whether sending a buffer of 64,000 bytes, or 1500 bytes, the time it takes for the first packet to hit the Ethernet controller is the same. The exit time is proportional to the buffer size that was sent. That is, the processing time is linearly dependent on how much data needs to be sent.

The TCP/IP stack measurements for the NIOS system give us an idea of the network performance of uClinux running on a NIOS processor. When comparing this performance measurement to the measurements from the CiNIC system[4], the NIOS processing appears to be approximately an order of magnitude slower. The reasoning for this, however, can be attributed to three major differences in the hardware and software between the two systems.

First, the architectures and processor speeds of the two systems are different. The current CiNIC using a StrongARM processor runs at 233MHz while the NIOS processor runs at 33.333 MHz. That is a difference of 7x. Second, there is a difference in the Linux kernel versions. The CiNIC's kernel version was 2.4 and the NIOS has version 2.0. There are large differences in the TCP/IP stack portion of these two kernels. Many of the modifications between the 2.2 and 2.4 kernels were improvements to network processing. Further analysis in this area is needed, but when comparing these results it is important to remember that the NIOS and CiNIC are running similar (but different) stacks.

The third difference is the fact that the NIOS has a 10 BaseT Ethernet interface, while the tests for the CiNIC were done with a 100 BaseT Ethernet interface. This issue suggests a difference of 10x if the network interface is the limiting factor. This hypothesis was easily tested since the driver for the controller chip contained a "chip busy" state. There is a finite amount of memory for packets in the chip, so when this is full the driver will wait and try again. A single debug statement proved that the driver was entering the "chip busy" state during multiple packet sends. This shows the NIOS processor and the uClinux stack are capable of

generating packets faster than 10 Mbps. The addition of 100 Mbps Ethernet would allow us to determine if the processor is capable of 100Mbps speeds.

5.2 Capabilities

This section discusses the use of programmable logic within the NIOS FPGA. The idea of adding custom hardware is one of the main advantages of this programmable system. The process of modifying or adding hardware can be done quickly, giving us flexibility not present in the current CiNIC architecture. The process of examining the capabilities of custom hardware, was implementation of a very simple custom hardware device and integrating it with uClinux.

The simple hardware device was a two register device which used control values in register A to decide when to perform a one's complement on the value in register B. This was implemented in Verilog code, along with the interfacing logic necessary for tapping into the NIOS system bus. From the NIOS system side, a user-defined interface was setup on the bus and assigned a region in the memory-mapped space. This allowed all software to see the registers of the custom hardware at a particular physical address. Since NIOS does not contain an MMU, it only has to dereference a pointer to that physical address to access the registers.

The process of adding custom hardware to the system was found to be very painless. Once the initial busing structure was understood. Integration with uClinux was also simple due to the lack of a virtual addressing for the NIOS processor. Our custom hardware wrote data to register B, flipped a bit in register A, waited for the hardware to flip the A bit back, and read out the "processed" value. This allowed us to implement a simple scenario, where the processor waits for the custom hardware to finish[1].

6 Conclusion

This paper presented a network coprocessing architecture based on an FPGA running a modified version of the Linux operating system. The architecture will allow us to implement changes and analyze the hardware/software trade-off very quickly. We studied both the performance and capabilities of this architecture. In the performance aspect, the NIOS under-performed the Strong Arm based CiNIC. But, it was shown that this may have been due to the NIOS's 10 Mbps Ethernet versus the CiNIC's 100 Mbps Ethernet. We feel confident that the NIOS, FPGA based architecture can adequately perform network functions. Also, if Moore's law holds true, the performance of the FPGA based architecture will continue to improve rapidly, providing more performance in the near future. In the capabilities arena, NIOS was found to be very flexible and fit our needs from an ease of use and development standpoint. The uClinux operating system provided us with the needed software support and the ability to add custom hardware to the system.

[1] NIOS 2.0 has the capability to provide up to five custom instructions. I.e., there now is a direct implementation for custom hardware instructions.

References

[1] ALTERA CORP. *Excalibur Backgrounder, Version 1.0*, 1.0 ed., June 2000.

[2] ALTERA CORP. *NIOS Embedded Processor—Getting Started, Version 1.1*, Mar. 2001.

[3] DE MICHELI, G., AND GUPTA, R. Hardware/software co-design. *Proc. IEEE 85*, 3 (Mar. 1997), 349–365.

[4] FISCHER, J. CiNIC—CalPoly intelligent NIC. Master's thesis, California Polytechnic State University, June 2001.

[5] INTEL CORP. *21554 PCI-to-PCI Non-Transparent Bridge Evaluation Board User's Guide, Version 1.0*, Jan. 2001.

[6] MICROTRONIX CORP. *LDK Getting Started Guide, Version 1.5*, Sept. 2001.

[7] ROSU, M., SCHWAN, K., AND FUJIMOTO, R. Supporting parallel applications on clusters of workstations: the intelligent network interface approach. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing* (Portland, OR, Aug. 1997), pp. 159–168.

[8] STEENKISTE, P. Analysis of the Nectar commuinication processor. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems* (Feb. 1992), pp. 1–3.