LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

StarT-Jr: A Parallel System from Commodity Technology

Computation Structures Group Memo 384
October 1, 1996

James C. Hoe, Mike Ehrlich

This paper describes reserach done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150. Paper to appear in the Proceedings of the 7th Transputer/Occam International Conference, Tokyo, Japan, November 1996.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

StarT-Jr: A Parallel System from Commodity Technology

James C. Hoe, Mike Ehrlich

October 1, 1996

Abstract

StarT-jr is an experimental parallel system composed of a network of personal computers (PCs). The system leverages the momentum of the microprocessor and PC industries to achieve excellent single node performance at a low cost. For parallel processing, StarT-jr uses the Flexible User-level Network Interface (FUNi) to provide low-overhead, user-level interprocessor communication over two IEEE 1394 High Performance Serial Busses. This efficient message-passing mechanism enables StarT-jr to exploit fine-grained parallelism for good parallel performance.

FUNi is based on an embedded processing system on a PCI card. Custom network hardware assembled from a commercial IEEE 1394 chip set provides FUNi with access to the IEEE 1394 network. In message passing, FUNi's embedded processor serves as a network coprocessor and manages an user-accessible message-passing interface in the host memory. User-level applications directly manipulate the interface location in host memory using cached reads and writes. Costly physical I/O accesses to device registers on the PCI bus are avoided. Currently, FUNi can efficiently support both fine-grain message passing and direct memory-to-memory transfers of large data blocks. FUNi can also support globally coherent shared memory by capturing and responding to memory accesses within a designated global address range. FUNi maintains a globally coherent shared memory cache to minimize global memory access latency. The necessary coherence protocol processing and communication is performed by the FUNi coprocessor.

We have demonstrated a two-node prototype of StarT-jr and are awaiting fabrication of additional interface cards in order to assemble an eight-node system. StarT-jr currently supports an active message-based light-weight communication library for the C programming language. Preliminary measurements of the communication library demonstrated overheads of $1 \sim 4$ μsec for sending or receiving small (≤ 40 bytes) messages, and an user-to-user latency of 85 μsec. Direct memory-to-memory transfers can sustain 3.4 MByte/sec on an unloaded network. With regard to the shared memory operation, reading a shared-memory location cached in FUNi takes approximately 2 μsec.

Keywords: StarT-jr, FUNi, network of workstations, parallel processing, network interface, user-level, interprocessor communication

1 Introduction

StarT-jr is an experimental parallel system based on a network of workstations (NOW). The goal is to demonstrate that an efficient and powerful parallel system can be inexpensively constructed from commodity technology. Thus, the first-order design directive of the system is to make maximal use of existing and proven commercial technology. Commodity-like Intel Pentium-based personal computers (PCs), in stock configuration, currently serve as the processing nodes to offer state-of-the-art single-node performance at a consumer-level price. In addition to a standard local area network (LAN) providing normal network services, two IEEE 1394 High Performance Serial Busses[9] provide the necessary interconnect for scalable parallel performance. Stock Linux operating system, a PC freeware version of Unix, controls the operation of the PCs.

As a corollary from the first directive, the detail design of StarT-jr must not only use the most suitable technology available, but it must also adopt to more suitable technologies as they emerge. Given this objective, the Flexible User-level Network Interface (FUNi) for StarT-jr is designed for the industry-standard PCI[17] bus, as opposed to a specific processor-memory bus. Although some aspects of performance is sacrificed, by maintaining this generality, StarT-jr systems can also be "scalable" through time by continually adopting more effective PC or workstation platforms with minimal loss in non-transferable software and hardware investment. On the other hand, FUNi itself also incorporates a modular design to facilitate upgrade in network performance. The on-board embedded processor, which controls the functions of FUNi, is packaged as an upgradable module. The lowest-level network adaptor module is also contained within an interchangeable daughter card to allow for different interconnection technologies.

To support a highly scalable parallel system, FUNi is targeted for fine-grain parallel processing where interprocessor communication is short but frequent. Any inefficiency in communication overhead (computation cycles lost to communication) will be magnified under such usage. This seems to counter FUNi's PCI-based design since physical I/O accesses to a PCI device – typically many tens of cycles per access – can add up to a very large overhead during communication. To overcome this obstacle, FUNi uses a decoupled interface paradigm to avoid direct interaction between the host processor and the FUNi hardware. FUNi itself is based on a commercial Intel i960 embedded processing system with direct read and write accesses to the host memory. Thus, FUNi can act as a network coprocessor to implement a message-passing interface (e.g. send and receive message queues) in the host memory. User processes only have to manipulate these message interface locations using normal memory reads and writes. Figure 1 illustrates this idea.

The FUNi coprocessor firmware can be reprogrammed to support many flavors of message passing interfaces and functions. In general, the FUNi interfaces are designed such that communication overhead is transferred from the host processor to the FUNi coprocessor. Currently, FUNi efficiently supports both fine-grain message passing (≤ 80 bytes of payload per message) and direct memory-to-memory transfers of large data blocks (up to 2560 bytes per initiation). To further reduce the communication overhead, FUNi, with the general-purpose processing power of i960, can be programmed to off-load some of the simpler message processing from the host processor. For example, memory access requests from a remote node could be satisfied by FUNi directly without ever disrupting the host processor. Active messages[20] processing can also be off-loaded from the host processor.

In addition to message passing, FUNi's hardware can also support globally coherent shared memory by capturing and responding to memory accesses within a designated global address range. After an address capture, the hardware first tries to satisfy the request by consulting a globally coherent shared memory cache. If the request cannot be satisfied directly, the FUNi coprocessor is interrupted to perform the necessary coherence processing and communication to complete the

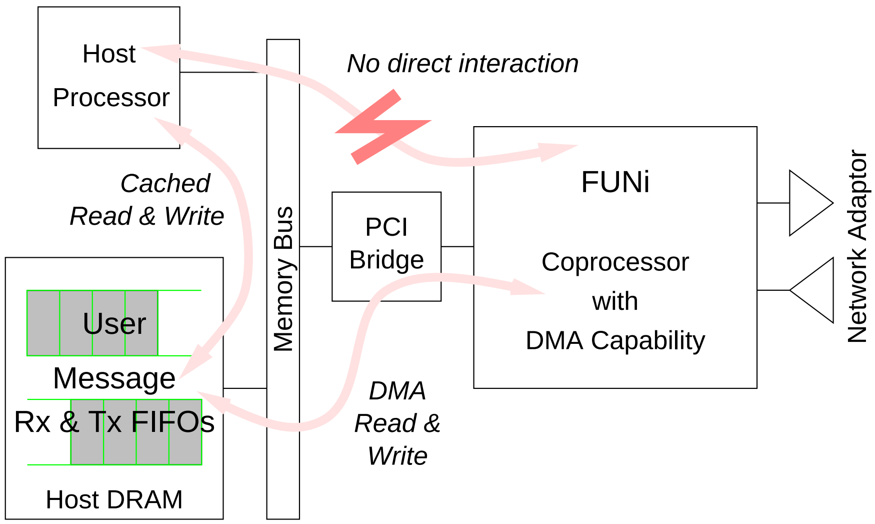

Figure 1: A Message Interface based on Software Queues in User's Virtual Memory

memory request. The programmability and processing power of the FUNi coprocessor allow for experimentation with memory coherence protocols. Unfortunately, in the current implementation, host processors are unable to cache the shared memory regions, thus limiting the performance of shared memory applications.

To obtain a meaningful performance estimate, a light-weight communication library (for the C programming language) based on active messages has been developed for StarT-jr. The library efficiently exposes FUNi's message-passing and DMA features to user-level applications. Preliminary measurements of the communication library demonstrated overheads of 1$\sim$4 μsec for sending or receiving small ($\leq$ 40 bytes) messages, and an user-to-user latency of 85 μsec. Direct memory-to-memory transfers can sustain 3.4 MByte/sec on an unloaded network. The low overhead is attributed to the fact that the host processor only has to access the host memory for message passing. The latency and bandwidth is below the capability of the raw hardware since the coprocessor firmware is designed to minimize overhead. By reprogramming the FUNi firmware for an alternative interface style, the low overhead can be traded for lower latency and better bandwidth. Currently, no software has been developed for shared memory. However, based on low-level experiments, a single-word read of a locally-cached shared-memory location takes approximately 2 μsec.

In this section, we have given an introductory overview of StarT-jr and FUNi. The remainder of this paper presents the details of the StarT-jr system and pays particular attention to the interprocessor communication mechanism that FUNi provides. The next section describes the current implementation of StarT-jr using stock PCs and IEEE 1394 interconnections. Section 3 discusses FUNi, the interprocessor communication mechanism in StarT-jr. Section 4 presents the results acquired from the prototype system. Section 5 briefly relates StarT-jr to other projects in the area of NOW and network interface design. This paper concludes with a summary and a brief discussion in Section 6.

2 A StarT-jr Implementation

The StarT-jr system is not fixed to any one specific implementation. Following our objective to construct the most efficient parallel system from the best suited commercial technology, we have left the design open to adopt new implementation technologies. In this section, we outline the implementation of our current IEEE 1394-based StarT-jr. We first describe the commodity components (i.e. stock, commercial portion) of our system and how each is selected. In the current StarT-jr implementation, a portion of FUNi does contain semi-custom hardware because no adequate substitute existed. The end of this section is devoted to the design and implementation of this semi-custom IEEE 1394 adaptor module.

2.1 Commodity Components of StarT-jr

The StarT-jr system can be broken into three components: the processing nodes, the interconnect substrate, and the network interface. Each of the components are specified or designed for trade-offs between cost and performance, together with the added constraint of flexibility and upgradability. The goal is to achieve the widest range of commercial options for implementation.

2.1.1 Processing nodes

The current StarT-jr system is composed of eight stock PCs with 120 MHz Pentium processors. These commodity-like PCs, with 256 KBytes of cache and 32 MBytes of main memory, is estimated at 172.2 SPECint92 and 108.4 SPECfp92. Even when fully configured with disk drives and I/O peripherals, the system costs under US$2000. This level of price-performance ratio, made possible by the market volume of the PC industry, is precisely what StarT-jr wants to exploit.

StarT-jr currently employs Linux, a PC version of the Unix operating system. StarT-jr only requires the addition of a FUNi device driver to provide mapping and protection for the interface memory region in the user's virtual address space. The standard OS offers the familiar suite of software development tools (compilers, debuggers, windowing systems, etc.) to reduce software development overhead. Equally importantly, pre-exiting sequential applications make StarT-jr immediately useful as a cluster of powerful stand-alone workstations.

With PCI-compliant FUNi hardware, StarT-jr can adopt any PCI-equipped platform running a selection of operating systems. This option ranges from entry-level PCs to high-end SMP servers, and allows for a heterogeneous StarT-jr system. Furthermore, StarT-jr can automatically track the technology curve by adopting to faster, and even cheaper, base platforms as they emerge.

2.1.2 Interconnect Substrate

The IEEE 1394 High Performance Serial Bus standard [9] is intended for multimedia applications with real-time and bandwidth requirements. This high performance technology is available as a ready-to-integrate chip set at a negligible cost. Chip sets for 100 and 200 Mbit/sec networks are already available, and 400 Mbit/sec and 1 Gbit/sec are being developed. IEEE 1394 has already found applications in consumer electronics such as disk drivers, printers, VCRs and video cameras.

Currently, two 100 Mbit/sec IEEE 1394 High Performance Serial Busses constitute the interconnect substrate for its price-performance ratio and its ease of implementation. However, the multimedia lineage of IEEE 1394 presents a problem in the case of network buffer overflow. Whereas it is of little concern to discard overflowing packets in most multimedia applications, parallel processing systems have traditionally relied on a reliable, loss-less network. In Section 3.2.4 we will explain how FUNi maintains the loss-less abstraction on top of the lossy underlying IEEE 1394 substrate.

A number of interconnect technologies can be incorporated into StarT-jr with minimal modifications to the remainder of the system. For example, a network module for the Arctic Switch Fabric (160 MBytes per link)[4] has been developed concurrently with the IEEE 1394 adaptor

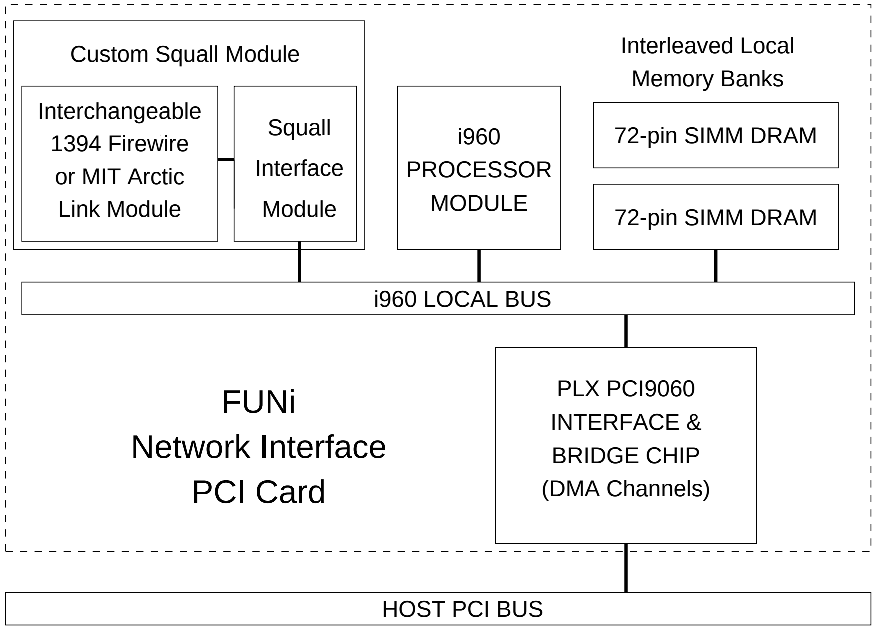

Figure 2: FUNi PCI Card

module. The Arctic module will allow us to construct a larger, higher performing StarT-jr using mostly the same hardware and software.

2.1.3 FUNi **Hardware**

Although FUNi contains semi-custom hardware, it is mostly based on a commercial embedded system. The custom hardware development is limited to the IEEE 1394 adaptor module that plugs into an existing interface. The details of this custom module is presented at the end of this section. The following paragraphs describe the commercial embedded system.

FUNi is based on Cyclone Microsystem's PCI80960 *Intelligent Communication Controller*[10]. Packaged as a plug-in PCI card, PCI80960 is a general-purpose embedded system, complete with a 33 MHz Intel i960CF (a 32-bit superscalar RISC processor) and 2 MBytes (upgradable to 32 MBytes) of local DRAM. A bus bridge provides i960 with direct load/store access and a DMA engine to the host memory. Eight 32-bit mailbox registers, visible to both the host and i960 by memory-mapped reads and writes, are also available to implement handshakes and synchronizations.

PCI80960 is engineered with an open-standard, Squall I/O adaptor interface on the i960 local bus. A variety of network adaptors, such as Ethernet, ATM, etc., are commercially available. FUNi employs a custom IEEE 1394 adaptor module for the Squall interface to provide i960 with access to the interconnect. Figure 2 illustrates the datapath of the FUNi PCI card. In StarT-jr, i960 serves as the intelligent network coprocessor. Section 3 describes how FUNi makes use of this embedded processing to implement an efficient low-overhead user-level message-passing interface despite being physically located on the peripheral PCI bus.

To help track the microprocessor performance curve, PCI80960's modular design packages the i960 processor in an interchangeable module. The binary compatibility and standardized bus interface within the i960 family allow transparent upgrades to upcoming generations of i960. By upgrading the host system and network coprocessor accordingly, we will be able to maintain the balance between computation and communication performance.

2.2 Custom IEEE 1394 Interface Module for FUNi

The Squall IEEE 1394 adaptor module for FUNi provides the FUNi coprocessor with access to two separate IEEE 1394 High Performance Serial Busses. All components on the adaptor can be purchased "off-the-shelf"; only the PCB layout and the logics inside two Xilinx 4005 FPGAs are custom to our system. The IEEE 1394 adaptor module for FUNi is made up of two separate printed circuit boards, joined by a connector. The IEEE 1394 Physical Link Module (1394PLM) PCB is specific to IEEE 1394 and utilizes commercial 1394 chip sets. The other PCB, Squall Interface Module (SIM), provides a generalized interface between the FUNi i960 coprocessor and PLMs.

2.2.1 Squall Interface Module

SIM presents a simple generic 32-bit FIFO-based transmission and reception interface to the FUNi coprocessor. The same SIM interface can be fitted with different specifically designed PLMs to support different network technologies. To support two physically prioritized networks, four hardware FIFOs are built from TI723631 synchronous FIFOs. A high and a low priority transmit FIFO pass commands and out-going packets from the FUNi coprocessor to PLM. A high and a low priority receive FIFO provide FUNi with access to hardware responses and in-bound packets from PLM. The SIM interface supports both single-word and four-word-burst accesses from the i960 FUNi coprocessor to the FIFOs. The optimized burst transactions allow more than double the bandwidth of single-word reads and writes.

SIM also includes the hardware for implementing globally coherent shared memory on StarT-jr. During shared memory operations, the shared memory region of the user's virtual address space is mapped to an Address Capture Device (ACD) on SIM. ACD is backed by a two-way set-associative Global Shared Memory Cache (GSMC) that is managed by a combination of hardware and FUNi firmware. GSMC is based on a 16KB Dual Ported Synchronous SRAM (DPSRAM) where one half of the DPSRAM is used for the data store, and one quarter is used for the tag store. The remaining 4KB of DPSRAM is available to FUNi as fast scratch memory.

When an access to a shared memory location occurs, ACD checks the two tag/control words in the GSMC tag store that correspond to the memory reference. If ACD determines that the memory requested can be completed according to the tag/control words, it will allow the bus transaction to access the corresponding GSMC data location to complete the transaction. If the transaction cannot be satisfied in hardware, ACD will force the memory transaction to retry indefinitely and interrupt the FUNi coprocessor for assistance. Once interrupted, the FUNi coprocessor queries ACD for the cause and performs the necessary coherence protocol to satisfy the memory request. This may require communication with FUNis on other nodes over the IEEE 1394 interconnect. The FUNi coprocessor can also maintain a large software cache in its local memory to reduce network traffic. Once the FUNi coprocessor is ready to complete the memory transaction, the FUNi coprocessor updates the corresponding data and tag location in GSMC. ACD is re-enabled to complete the retried memory transaction.

2.2.2 1394 High Performance Serial Bus Physical Link Module

1394PLM is built from a ready-to-integrate IEEE 1394 Physical Layer and Link Layer chip sets from Texas Instruments. Connections to two independent IEEE 1394 busses are provided; one is given a higher priority. A custom FPGA-based controller interfaces with the SIM FIFOs and carries out the corresponding low-level handshakes with the IEEE 1394 chip set. The specifics of the network implementation is contained within PLM. This modular design allows StarT-jr to adopt a variety of different interconnect technologies with only limited modifications.

3 FUNi

The basic FUNi message-passing interface was first proposed for SBus-equipped workstations[8]. The design proposed a way to implement a fine-grain message-passing interface on a peripheral bus without a penalty in communication overhead from the long access latency. The design required more intelligence in the interface hardware to manage a message-passing interface in the host memory. Interaction between the host processor and the network interface is achieved indirectly through these shared memory locations. The original proposal asked for implementation of custom logics in FPGAs. Due to the limited logic density of FPGAs, only the very basic message-passing mechanisms were supported.

This initial study has led to the current embedded processing-based FUNi implementation. The new design still retains the same low-overhead since the memory-based interface paradigm is unaffected. Embedded processing does slightly increase communication latency over the FPGA-based design. However, newly available processing power and programmability enable richer features and open opportunities for experimentation. Furthermore, by eliminating much of the custom logics, embedded processing has also significantly reduced the design and implementation complexity. Logic designs are replaced by firmware development using a standard C compiler and interactive debugger.

In the following paragraph, we describe the communication mechanism currently supported by FUNi. FUNi is composed of two parts: an user-visible part that implements the user-level message interface, and an internal part that processes network events such as transmission and reception, and the necessary network protocols. We first describe the basic interaction between the FUNi firmware and host software at the message passing interface. Next, we will examine the internal part of FUNi. Finally, possibilities to extend the FUNi feature set is briefly explored.

3.1 Memory-based User-level Message-Passing Interface

FUNi externally presents an abstract network interface to the user. Due to the FUNi coprocessor, the user interface can include more complicated services not directly supported by the underlying network hardware. The interface abstraction off-loads the tasks of dealing with the particularity of the network implementation to the FUNi coprocessor. This does not only streamline the host processor's communication task, but the generalized abstraction also allows the same user software to be carried over among different network implementations.

The basic message sending and receiving interface currently implemented for StarT-jr is based on a set of message queues for transmission (Tx) and reception (Rx). Figure 3 logically depicts this message interface in the user's virtual address space. The message queues are jointly maintained by the host processor and FUNi as circular buffers in the host memory. Two sets of Tx and Rx queue pairs are implemented to support two message priorities. Each queue is logically divided into fix-sized message slots, each large enough to hold the largest message. A set of memory-mapped registers (the PCI80960 mailbox registers) holds the head and tail indices associated with each circular buffer. During the initialization call to the device driver, the user application supplies a region of virtual address space to be mapped to the Tx and Rx queues pinned in the host physical memory. The index registers are also mapped into the user's virtual address space.

The circular buffers use the standard convention of head and tail indices to implement FIFO queues. The head index points to the next free slot, whereas the tail index points to the next valid

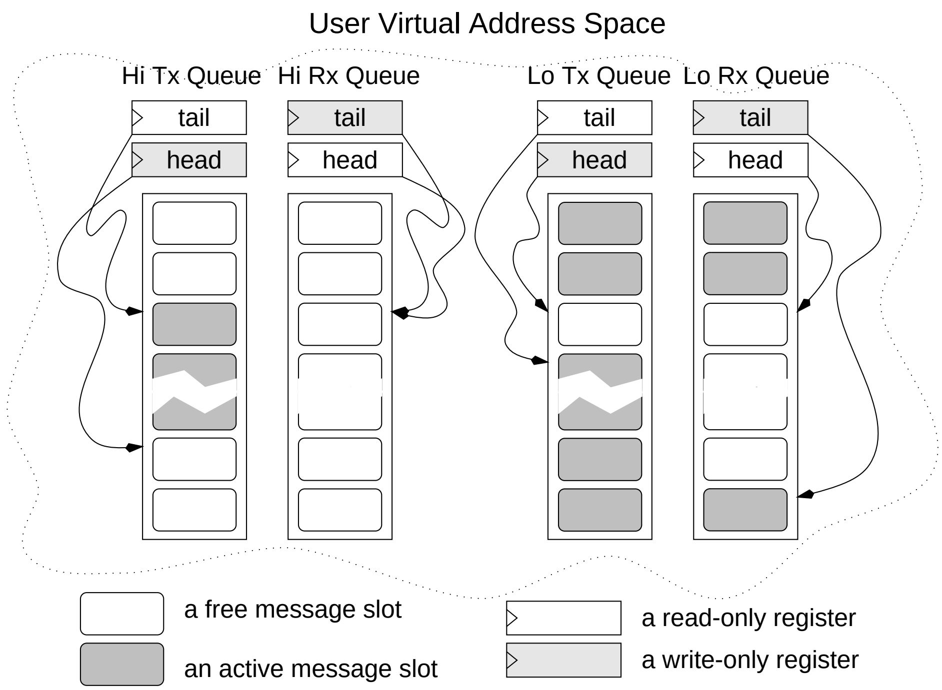

Figure 3: FUNi User Message Interface

slot. A queue is empty when the head and tail indices both point to the same slot, and full when the head is immediately before the tail. The producer of a queue increments the corresponding head index register to inform the consumer about new slots containing messages. The consumer increments the tail index to indicate which slots are ready for reclaim. Each index register is only incremented by either the user or FUNi, and therefore does not require atomic operations to modify.

3.1.1 User-level Message Passing: Sending

As the consumer, the FUNi firmware compares the head and tail index registers to poll the Tx queues for pending out-going message. To send a message, the user process directly composes the message in the next free slot in the Tx queue. The user-level interface uses very simple message formats (simpler than an actual network packet). For normal message passing, the message header, containing the logical destination address, message length and message type, is written to the first word of the message slot. The message payload is written to successive addresses following the message header. These writes to cacheable memory locations incur minimum overhead. Afterwards, the user process increments the content of the head index register to make the current message slot visible to FUNi. Once notified, FUNi issues cache-coherent read requests on the PCI bus to retrieve the user message, presumably still in the data cache, and composes the corresponding network packet for transmission. After each successful retrieval from a slot, FUNi will increment the tail index to release the slot.

3.1.2 User-level Message Passing: Receiving

When a packet arrives, FUNi, acting as the producer, enqueues the message to one of the two Rx queues depending on the message's priority. Then, FUNi increments the corresponding head index register to indicate the arrival of a new message. Prior to enqueuing, the contents of the tail and the head index registers are compared to detect queue overflow. Section 3.2.2 explains how an overflow is handled by FUNi.

The user process detects the presence of an incoming message by comparing the head and tail indices of the Rx queue. However, reading the head index register involves costly physical I/O accesses to the FUNi device. Therefore, the same information is redundantly represented as a `message-valid` bit in the header word of each Rx message slot. When FUNi enqueues a message, aside from incrementing the head index, it also sets the `message-valid` bit in the message slot. Thus, to poll for new messages, the user process should check the `message-valid` bit of the next message slot using a cached memory read. As long as the Rx queue remains empty, the header word is unchanged, and each failed poll only incurs the overhead of a cache hit. This continues until FUNi flushes and updates the header word to reflect a new message arrival.[1]

When the user process locates an unreceived message, the user process first extracts the message type and length from the message header. Then, the message payload is read from successive addresses. After the message is received, the user process releases the message slots by incrementing the tail index register. According to the FUNi Rx handshake, the user also needs to clear the `message-valid` bit afterwards.

3.1.3 Remote DMA Transfers

With a passive network interface, a block memory transfer requires the sending process to explicitly copy, in verbatim, each byte of transfer from the source to the interface. Similarly, the receiving process must later copy each byte from the interface to the destination location. To eliminate the data copying overhead, the active FUNi network interface is extended with two types of remote DMA transfer mechanisms. In either DMA modes, the user process only needs to enqueue a header that specifies the source and target virtual addresses and the length of the data block; FUNi will automatically move the data block across the nodes.

The first DMA mode, intended for medium sized blocks (a few hundred bytes), is only a zero-copy variant of message passing where FUNi operates on the source and target location directly instead of the message queues. The second type of DMA can transfer up to 2560 bytes of data per initiation. In this case, payload is moved to and from FUNi using a DMA engine. FUNi automatically packetizes the data block for transfer over the network. The trade-off between the two DMA modes is bandwidth versus latency and granularity of control. The overhead of initiating a DMA transfer, regardless of size, is comparable to normal message passing.

3.2 Internal FUNi Functions

In addition to providing the user interface, the FUNi coprocessor is responsible for interfacing with the network adaptor hardware as dictated by the user application. At the highest level, this simply involves translating between the abstract user-level messages and the appropriate network packets, according to a fixed rule. However, FUNi presents an abstraction of a secured, multi-prioritized,

[1]The `message-valid` scheme is not used to eliminate the need to increment the Tx head index by the host processor. For FUNi's coprocessor, reading from the Tx queue in the host memory is a much more costly operation than reading the index registers. Since the FUNi coprocessor is time-shared among many tasks, better overall performance is achieved when the index registers are used for the Tx queues. The host overhead of writing to the index registers can be partially hidden by the write-buffer of the host processor. Furthermore, the head and tail index registers do not need to be updated immediately after each message for correct operations. Thus, the overhead of updating the index registers can be amortized over many messages.

loss-less and deadlock-free network to simplify the work of the host processor. If these characteristics are not directly supported by the physical network substrate, FUNi must enforce these characteristics through network protocols. In the paragraphs below, we describe the light-weight protocols for enforcing these assumptions and how they apply to the IEEE 1394 implementation in particular.

3.2.1 Protection and Security

StarT-jr is designed to allow multiple parallel applications to time-share the network and processor resources while providing each application the illusion of a private and reliable network. FUNi maintains this abstraction by automatically tagging network packets with Group Identifiers (GIDs).

Each parallel application on StarT-jr is assign an unique GID. When the process of a parallel application is active on a workstation, the operating system makes the corresponding GID available to FUNi. Every out-going packet is automatically tagged with the GID. When the packet arrives at the destination, the receiving FUNi compares the GID tag of the in-bound packet against the local GID. The in-bound packet is delivered to the current process only if a match is made. A mismatched GID indicates the correct process is not presently executing, and the in-bound packet is not delivered. (Under the flow control protocol of StarT-jr, FUNi will discard the undeliverable packet and return a negative acknowledgment to the packet's originator.) Thus, a process is only able to communicate with its cooperating peer processes of the same application whom all share the same application GID.

3.2.2 Network Flow Control for Performance Guarantee

When a parallel system is shared among different users, another concern is gross performance degradation, or even deadlock, of the network due to misuse by a user. This issue is addressed in StarT-jr by the Acknowledgment/Retry End-to-End Flow Control Protocol carried out by the FUNi coprocessor.

The Acknowledgment/Retry Protocol in StarT-jr is a simplification of the Selective Repeat Protocol[19]. In this sender-buffering protocol, FUNi is responsible for buffering its out-going packets until the packet is accepted and positively acknowledged by the receiver. When FUNi absorbs an in-bound packet from the network, a positive acknowledgment is sent if the packet is accepted. If FUNi cannot accept the packet, a negative acknowledgment is returned to the originating FUNi to request for a retry.

With the ability to discard incoming packets, FUNi at each node can continuously absorb packets from the network even with only finite local storage. Since network packets are continuously absorbed, the network will not deadlock regardless of the individual behavior at each node. Thus, FUNi can never be indirectly blocked from communication by other misbehaving nodes. Thus, the protocol can guarantee the worst-case performance of the network. FUNi is stopped from transmitting only when its own receivers cannot absorb and acknowledge the packets fast enough. Thus, this protocol also serves as an automatic rate control for throttling network activities of over-active sender nodes.

The Acknowledgment/Retry Protocol needs to be implemented on top of two physically separate or prioritized network where the higher priority network traffic cannot be blocked by lower priority traffic. In IEEE 1394-based StarT-jr, this is accomplished by using two separate IEEE 1394 busses. In the protocol, the payload packets are sent on the low priority network, and the acknowledgments are sent on the high priority network to avoid deadlock within the protocol itself.

3.2.3 Virtually Prioritized Networks

On top of the Acknowledgment/Retry Protocol, any number of virtual network priorities can be extended to the user. Currently, FUNi has chosen to implement two levels of priority to support the popular request-reply based user protocols. Two sets of Tx and Rx message queues as described in Section 3.1 are provided to handle the two priorities of messages independently. FUNi preferentially services the higher priority queue first. Parts of the buffering resources at each FUNi are also reserved for exclusive use by high priority message traffic. This is sufficient to guarantee the flow of high priority messages in the system regardless of lower priority traffic since the Acknowledgment/Retry Protocol already guarantees that the network itself will not block.

3.2.4 Packet Loss Recovery

Although the Acknowledgment/Retry Protocol allows FUNi to continuously absorb packets from the network, under an extreme many-to-one communication pattern, FUNi's hardware buffer can become full because the FUNi coprocessor cannot keep up with the influx of packets. In a traditional loss-less MPP network, this condition would temporarily block the network, but eventually it will become freed again without extra handling.

The Acknowledgment/Retry Protocol on a loss-less network can expect every payload packet transmitted to be matched with a returning acknowledgment, whether positive or negative. However, on IEEE 1394, when the hardware buffer overflows, the partially transferred message is discarded. In this case, a discarded payload packet would never be acknowledged. (Due to the small size of acknowledgment packets, the acknowledgment receive buffer is guaranteed not to overflow.) Thus, for IEEE 1394-based StarT-jr, the Acknowledgment/Retry Protocol is augmented with an additional *round* counting scheme to restart packets with lost acknowledgments. All packets destined for a particular node is tagged with a round identifier associated with that node. When FUNi detects a situation where packets are lost on the network, it broadcasts a new round identifier for itself and discards any future in-bound packets tagged with the old round identifier. After receiving a restart broadcast, the other FUNis will treat unacknowledged packets as if they were negatively acknowledged and retry them with the new round identifier.

3.3 Coprocessing-based Extension to FUNi

Higher-level communication features can be developed to further off-load network processing tasks from the host processor to the firmware programmable FUNi. We are currently working on a version of the FUNi firmware with integrated MPI[13] functions to reduce host processing overhead during communication. Processing of light-weight active messages and remote fetches are also candidates for transfer into the FUNi coprocessor. Other possibilities include scheduled prefetches, conditional prefetches and advanced synchronization services. Additional performance for mission-critical applications can be obtained through fine-tuning the firmware to the specific traffic pattern and communication needs.

FUNi's embedded processor-based implementation also encourages research in network flow control protocols. Even non-trivial protocols can be quickly tested by recoding the firmware. Performance monitors can also be included during system development. Future work in protocols can extend to address fault tolerance and network load balancing issues.

Table 1: Examples of JAM Primitives

Message-passing Primitives		
primitive	argument	description
`JAM_4()` `JAM_reply_4()`	`dest, *func, arg1, arg2, arg3, arg4`	Sends an active message containing the pointer `func` and the 4 integer arguments to `dest`. `*func(arg1,arg2,arg3,arg4)` is invoked at `dest` upon arrival.
`JAM_n()` `JAM_reply_n()`	`dest, *func, arg[], size`	Sends an active message containing the pointer `func` and up to 20 integer arguments from arg[] to `dest`. `*func(arg[0],arg[1],...,arg[n-1])` is invoked at `dest` upon arrival.

Block Data Transfer Primitives		
primitive	argument	description
`JAM_xfer_4()` `JAM_reply_xfer_4()`	`dest, seg_addr, word0, word1, word2, word3`	Transfers 4 integer data to `dest`. The transfer segment ID and address at the destination are encoded in `seg_addr`. This primitive uses FUNI's standard message passing mechanism.
`JAM_mfer_n()` `JAM_reply_mfer_n()`	`dest, seg, *source, *target, nword`	Transfers up to 16 4-byte data to `dest`. The payload is moved from local `source` to remote `target` using FUNI's 0-copy message passing mechanism.
`JAM_DMA_n()` `JAM_reply_DMA_n()`	`dest, seg, *source, *target, nword`	Transfers up to 640 4-byte data to `dest`. The payload is moved from local `source` to remote `target` using FUNI's direct memory-to-memory transfer.
		NOTE: Like CMAM, data transfers first involve registering a transfer segment and a handler at the destination node. Subsequent transfers decrement a transfer counter. When the transfer is completed, the registered handler is invoked.

4 Current Status and Results

A two-node StarT-jr prototype has been completed as a prelude to the eight-node final system. A light-weight communication library, Jam, has been developed to assist the software development for SPMD message-passing programming. The library is based on University of California at Berkeley's version of the Connection Machine Active Message (CMAM) library[20]. The CMAM library is ported to StarT-jr by rewriting the low-level primitives that deal directly with the network interface[2]. New primitives are added to take advantage of the features of StarT-jr and FUNi. A new set of primitives to support larger messages, up to twenty arguments, is added. A set of data transfer primitives is also added to take advantage of FUNi's low-overhead remote DMA transfer. Examples of the Jam primitives and a brief description are given in Table 1. As in CMAM, each Jam primitive is supported in two priorities; high priority primitives have the additional suffix `_reply`.

A benchmark suite to assess the performance of FUNi in conjunction with Jam is executed on the two-node StarT-jr prototype. Communication overhead, one-way user-to-user latency and sustained throughput of the basic Jam primitives are measured. The send and receive overheads measure the average execution time of the corresponding primitives on the sending and receiving host processors respectively. User-to-user latency includes all the processing by the host and FUNi on both the sending and receiving ends, plus the one-hop network transit latency. In addition to

[2]Currently, all software and firmware additions are coded in C and compiled with GCC with optimization options.

Table 2: Benchmark Summary of JAM on StarT-jr

Message-passing Primitives							
	send overhead	receive overhead	send throughput		receive throughput		latency
	μsec	μsec	MB/sec	μsec	MB/sec	μsec	μsec
`JAM_4`	2.2	1.1	0.5	29	0.5	29	85
`JAM_reply_4`	1.6	1.1	0.5	29	0.5	29	85
`JAM_n=1`	2.0	1.3	0.1	27	0.1	26	
`JAM_n=20`	5.4	3.5	1.9	42	1.9	42	
`JAM_reply_n=1`	1.5	1.3	0.1	27	0.1	27	
`JAM_reply_n=20`	4.0	3.6	1.9	43	1.9	42	

Block Data Transfer Primitives							
	send overhead	receive overhead	send throughput		receive throughput		latency
	μsec	μsec	MB/sec	μsec	MB/sec	μsec	μsec
`JAM_xfer_4`	2.7	1.5	0.5	48	0.5	48	
`JAM_reply_xfer_4`	1.8	1.4	0.5	49	0.5	50	
`JAM_mfer_n=16`	1.9	1.2	1.5	17	0.4	69	
`JAM_reply_mfer_n=16`	2.0	1.1	1.5	17	0.4	69	
`JAM_dma_n=640`	2.1	1.1	3.4		3.4		
`JAM_reply_dma_n=640`	2.1	1.1	3.4		3.4		

Table 3: Communication Performance Comparison

		TCP/IP and Ethernet	CMAM and CM5	JAM and StarT-jr (IEEE 1394)	JAM and StarT-jr (MIT Arctic)	FM and Myrinet
overhead	(μsec)	27	2.9	3.3	3.6	4.4
	(cyc)	3240	92.8	396	432	330
latency	(μsec)	1289	12.5	85	18.0	13.1
	(cyc)	154K	414	10200	2160	982
bandwidth	(MB/sec)	0.8	10.0	3.4	9.1	17.5

MByte/sec, sustainable throughput is also given in terms of μsec, which roughly corresponds to the occupancy of each JAM primitive on the FUNi coprocessor. Table 2 gives a summary of the measured results.

Table 3 compares the performance of FUNi against other message passing systems. The first column lists the typical performance of TCP/IP on a Linux PC with a 120MHz Pentium processor and 10 Mbit/sec Ethernet. The second column shows the performance of the original CMAM library on a 32-node CM-5 with 32 MHz SPARC processors. The performance of StarT-jr connected by IEEE 1394 is given in column three. For comparison, the performance of StarT-jr connected with MIT's Arctic Switch Fabric[4] is given in column four. Finally, a comparison is made against a pair of Myrinet-equipped 75 MHz SPARCstation20s running UIUC's FM library[15]. The overhead and latency are measured using minimum size packets in each system. The overhead shown includes both the send and receive overheads, and the latency is for one-way user-to-user. In Table 3, overhead and latency are also given in terms of native processor cycles to give an approximation of the communication cost in terms of number of instructions.

The comparison shows that FUNi can give nearly an order-of-magnitude improvement in performance relative to normal Unix IPC mechanisms over conventional Ethernet. FUNi's memory-based, decoupled user-level interface achieves lower communication overhead in comparison with the contemporary Myrinet-based system. However, Myrinet-based systems achieved nearly five-times the bandwidth and one-sixth the minimum latency as compared to FUNi. Section 5 further discusses the comparison against Myrinet-based systems.

FUNi's performance, in terms of bandwidth and latency, is restricted by our design choices to exchange latency and throughput for lowered overhead. The user-visible latency and bandwidth of FUNi is well below the capability of the raw hardware. An alternate FUNi firmware can be used to improve the bandwidth and latency of the system, but with a penalty in overhead and overall system performance.

FUNi's performance is also limited by the processing rate of the i960 FUNi coprocessor. As described in Section 3.2, additional protocols have to be layered on top of the IEEE 1394 High Performance Serial Bus to present a network abstraction more suitable for parallel computing. The amount of protocol required directly affects each message's occupancy on the network coprocessor, which translates to lower bandwidth and longer latency in the processing-bound FUNi. For example, by switching to the Arctic Switch Fabric which is designed from the ground-up for parallel processing, FUNi, using more minimal protocol, can gain a factor of two to three improvement in bandwidth and latency. We hope FUNi's performance will increase with the availability of new generations of i960 processors. Meanwhile, FUNi has to make use of its general-purpose processing power to assist the host processor in more sophisticated message processing to compensate for its shortcomings.

Lastly, one should also notice that both StarT-jr and the Myrinet system perform poorly against CM-5 in terms of native processor cycles. This exemplifies the two-fold problem that hinders the communication performance of NOW systems. First, unlike the two-generation-old SPARC processor in CM-5, modern microprocessors have greater bias for number crunching rather than I/O performance. Secondly, unlike CM-5's proprietary hardware, the NOWs do not have the luxury of a customized network interface tightly coupled to the microprocessor on the memory bus.

5 Related Research

The parallel processing community has long recognized the importance of low communication latency and overhead for good parallel performance. The StarT project[14] integrated an user-level network interface directly into ISA and the datapath of a RISC microprocessor. However, the time and effort required for such a hardware-intensive project not only introduces great risks but also often results in out-dated hardware. To close this technology gap, later projects such as StarT-ng[5] and Flash[12] rely on stock mainstream commercial microprocessors but with tightly coupled external hardware (using a direct connection to the processor). In the near term, these projects may still be less practical because their overall message-passing performance remains hindered by the internal design of the microprocessor, despite extensive efforts in optimizing the external hardware and its interaction with the microprocessor. Nevertheless, this tight-integration of computation and communication will become essential as the performance between the microprocessor and the remainder of the computer system continues to widen.

Meanwhile, the network interface design for a practical NOW system must work within the limitations of existing system constraints, such as the bias for normal memory accesses versus I/O accesses. The SHRIMP multicomputer project[1, 2] specifies another memory-based interface to achieve low communication overhead in a stock workstation environment. The SHRIMP network interface is designed for a network of Pentium PCs with Xpress Bus and EISA Bus. Communication between any two PCs is logically an uni-directional mapping of a source virtual memory region on one PC to the target memory region on another. The network interface on the source PC snoops the bus for writes into the mapped area of memory and automatically transmits a message to the target PC. The target network interface then writes the data to the corresponding location using DMA. To support this communication paradigm, SHRIMP's hardware needs to support specialized functions such as bus snooping and memory-management. SHRIMP's hardware implementation overhead is higher than FUNi's, and SHRIMP's association with a specific processor-memory bus limits its upgrade path. SHRIMP also does not have the benefit of a network coprocessor.

Several academic and industry NOW projects[1, 7, 11, 15, 16, 18] have based their research on a cluster of workstations interconnected by Myrinet[3]. Derived from Caltech's ATOMIC project[6], Myrinet is available commercially as a ready-to-go high-performance NOW interconnection package. The Myrinet package comes complete with network routers, end-point adaptors for SBus and PCI, standard IP layers, and a custom low-level interface layer.

Myrinet's network adaptors are controlled by Myricom's custom LANai embedded processor. Much of the current research has involved taking advantage of the programmability of the LANai processor to implement fast user-level message-passing on NOW. Both LANai and the host processor can access the LANai local memory on the adaptor. However, unlike the FUNi coprocessor, the LANai processor does not have access to the host memory except using a DMA engine. Thus, in fine-grain messaging, either the host processor has to pay the overhead for writing messages all the way out to the adaptor, or to pay the latency penalty for data movement by the DMA engine. This could possibly explain the higher communication overhead compared with FUNi. However, Myrinet's integrated custom implementation yields far better latency and bandwidth (less than 10 μsec latency and greater than 30 MByte/sec are reported[3]).

6 Discussion

The goal of StarT-jr is to achieve usable performance comparable to hardware-intensive designs while maintaining the price-performance advantage through the use of low-cost commodity components. The system incorporates standard hardware and software subsystems to achieve state-of-the-art performance with minimal cost and effort. By designing for commodity technology, StarT-jr is general enough to adopt to emerging technologies. This flexibility will enable StarT-jr to maintain up-to-date performance despite the rapid turn-over of technology.

StarT-jr employs FUNi's memory-based network interface paradigm to overcome the two disadvantages facing NOW development:

1. Stock microprocessors and workstations are optimized with a bias for processing rather than non-cached, non-burst I/O operations, and

2. Stock workstations lack a generic/standard interface that is more tightly-coupled to the processor than the peripheral bus slots.

FUNi overcomes these problems by decoupling the network interface hardware from the host processor during communication. The FUNi coprocessor also off-loads much of the communication overhead from the host processor. By relying on latency-hiding software techniques and concentrating on minimizing communication overhead, StarT-jr can provide good support for fine-grain message passing.

[3]Latest performance figures for Myrinet are posted on **http://www.myri.com/myrinet/performance/**.

This paper describes the system design and development of the StarT-jr project. Continuing StarT-jr research will focus on developing additional memory-based interface paradigms. The research will take full advantage of the programmable FUNi coprocessor to compensate for FUNi's hardware tradeoffs. Although FUNi is able to reduce the overhead of message passing, its latency is not optimal. The success of FUNi and StarT-jr must rely on latency-hiding techniques such as split-phased transactions and software pipelining. Future research will involve analyzing existing programs/algorithms' communication pattern and transforming them into more latency tolerant ones. The analysis of communication patterns will also help in designing more suitable message interface services. Given the programmable FUNi, it is possible to fine-tune the FUNi firmware on a program-by-program basis for maximum performance.

Relying on standard technologies from the PC and the workstation industries, the bus-based symmetric multiprocessing workstations (SMPs) have solved the chicken-and-egg problem of large market volume versus lowered production cost. By providing good performance for both existing sequential binaries as well as new parallel programming paradigms, SMPs have also resolved the paradox of creating a software base without a user base, and vice versa. By touching off this seamless transition, SMPs have finally given parallel processing a solid foothold in mainstream computing. Nevertheless, the scalability of these bus-based parallel systems will inevitably be challenged. Sharing the same two traits that made SMPs successful, NOW systems such as StarT-jr will play a crucial role in resolving the scalability bottleneck and further extending this seamless transition into parallel computing.

7 Acknowledgments

This research is supervised by Professor Arvind and is supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-1310 and Ft. Huachuca contract DABT63-95-C-0150. Special thanks to Intel for their donation of the Pentium PCs and to Texas Instruments for their assistance in manufacturing the 1394 interface cards. Many thanks to our colleagues, B. S. Ang, M. J. Beckerle, A. Boughton, D. Chiou, R. Greiner, J. E. Hicks, L. Rudolph, and X. Shen on the StarT-ng and StarT-voyager sister projects.

References

[1] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Two virtual memory mapped network interface designs. In *Proceedings of Hot Interconnects II*, August 1994.

[2] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of 21st ISCA*, April 1994.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet – a gigabit-per-second local-area network. *IEEE Micro*, February 1995.

[4] G. A. Boughton. Arctic routing chip. In *Proceedings of Hot Interconnects II*, August 1994.

[5] D. Chiou, B. S. Ang, Arvind, M. J. Beckerle, A. Boughton, R. Greiner, J. E. Hicks, and J. C. Hoe. StarT-NG: Delivering seamless parallel computing. Technical Report CSG Memo 371, MIT Laboratory for Computer Science, February 1995.

[6] R. Felderman, A. DeSchon, D. Cohen, and G. Finn. Atomic: A high-speed local communication architecture. *Journal of High Speed Networks*, 3(1), February 1994.

[7] M. D. Hill, J. R. Larus, and D. A. Wood. Tempest: A substrate for portable parallel programs. In *Proceedings of COMPCON Spring 95*, March 1995.

[8] J. C. Hoe. Network interface for message-passing parallel computation on a workstation cluster. In *Proceedings of Hot Interconnects II*, August 1994.

[9] IEEE standard of a high performance serial bus, 1995.

[10] Intel. *i960 Microprocessor User Guide for Cyclone EP and PCI-SDK Platform*, 2.0 edition, 1995.

[11] K. K. Keeton, T. E. Anderson, and D. A. Patterson. LogP quantified: The case for low-overhead local area network. In *Proceedings of Hot Interconnects II*, August 1995.

[12] J. Kuskin, D. Ofelt, M Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of 21st ISCA*, April 1994.

[13] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1.1 edition, June 1995.

[14] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of 19th ISCA*, May 1992.

[15] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and massively-parallel processors. submitted for publication, April 1996.

[16] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[17] PCI Special Interest Group. *PCI Local Bus Specification*, 2.1 edition, June 1995.

[18] M. R. Swanson and L. B. Stoller. Low latency workstation cluster communications using sender-based protocols. Technical Report UUCS-96-001, Department of Computer Science, University of Utah, January 1996.

[19] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 1989.

[20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of 19th ISCA*, May 1992.