

StarT-jr: A Parallel System from Commodity Technology

by

Michael S. Ehrlich

Submitted to the Department of Electrical Engineering and
Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

© Massachusetts Institute of Technology 1997. All rights reserved.



Author
Department of Electrical Engineering and Computer Science
March 10, 1997



Certified by
Jonathan Allen
Professor
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 24 1997 Eng.

StarT-jr: A Parallel System from Commodity Technology

by

Michael S. Ehrlich

Submitted to the Department of Electrical Engineering and Computer Science
on March 10, 1997, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

StarT-jr is a parallel machine with support for both active user-level messages and cache-coherent distributed shared memory and can be assembled from mostly off-the-shelf commodity technology. The architecture, design, and implementation of the custom hardware is described in this thesis from both high and low level perspectives. Separate sections describe the message passing and shared memory hardware respectively. The custom hardware of the StarT-jr project consists of a network switch fabric and a network interface card (set of cards) that plugs into the I/O bus of a stock X86 PC. The most important component on the network interface card is the protocol processor which communicates with other protocol processors in order to maintain cache coherency of the globally shared memory space and off-loads much of the application processor's overhead during message passing. A high level description of some of the support software including the coherency protocol code is given. Performance data is reported including a measured 50MB/sec sustainable block data transfer rate.

Thesis Supervisor: Jonathan Allen

Title: Professor

Acknowledgments

Several people were particularly helpful to me in my thesis work and I would like to take this opportunity to thank them. First of all, I would like to thank my thesis advisor Professor Jonathan Allen and the other readers of this document including in particular Professor Arvind of CSG and George Hadjiyiannis of RLE. A special thank you has to go to James Hoe of CSG who wrote software for and was my partner on the StarT-jr project. I would like to thank all the other members of CSG for being so helpful on a daily basis, and mention in particular Boon Ang, George "Andy" Boughton, Derek Chiou, and Jack Costanza. Larry Blackledge of Texas Instruments helped the project financially and technically by arranging the cooperative TI/MIT Comdex effort. Victor Luchangco helped me with the theory of cache coherency protocols. My girlfriend Susie Hilton was a constant source of support in a countless number of ways.

Contents

- 1 Introduction**
 - 1.1 Project Goals**
 - 1.2 Related Work**
- 2 Project Overview**
 - 2.1 Software / Hardware Overview**
 - 2.2 Implementation Strategy**
- 3 Message Passing Hardware Details**
 - 3.1 Service Processor's View of Common Message Passing Interface**
 - 3.2 High Speed Serial Bus Interface Card**
 - 3.3 Arctic Switch Fabric Interface Card**
- 4 Shared Memory Support Details**
 - 4.1 Global Shared Memory Reads**
 - 4.2 Global Shared Memory Writes**
 - 4.3 Split-Phase Transactions**
 - 4.4 Coherency Protocol**
- 5 Experimental Results**
- 6 Future Work**

List of Figures

figure 1	page 9	System based on Arctic Switch Fabric
figure 2	page 15	Common Message Passing Interface
figure 3	page 20	A StarT-jr parallel machine
figure 4	page 26	1394 High Speed Serial Bus NIC block diagram
figure 5	page 29	Arctic Switch Fabric NIC block diagram
figure 6	page 35	Dual Ported SRAM space
figure 7	page 37	Shared Memory Data Paths
figure 8	page 39	Shared memory registers
figure 9	page 41	Coherence boundary
figure 10	page 51	Digital Video Processing Demo

List of Tables

table 1	page 48	Low level performance measurements
table 2	page 52	Support software performance measurements

Glossary of Acronyms

ACD:	Address Capture Device (first occurrence on page 17)
AP:	Application Processor (first occurrence on page 18)
Arctic:	A Routing Chip That Is Cool (first occurrence on page 8)
CMOS:	Complementary Metal Oxide Semiconductor (first occurrence on page 20)
CPU:	Central Processing Unit (first occurrence on page 48)
CRC:	Cyclic Redundancy Check (first occurrence on page 26)
DMA:	Direct Memory Access (first occurrence on page 25)
DPSRAM:	Dual-Ported Static Random Access Memory (first occurrence on page 17)
DRAM:	Dynamic Random Access Memory (first occurrence on page 12)
ECL:	Emitter Coupled Logic (first occurrence on page 20)
FIFO:	First-In First-Out (first occurrence on page 15)
FPGA:	Field Programmable Gate Array (first occurrence on page 14)
GRF:	General Receive FIFO (first occurrence on page 26)
GSM:	Global Shared Memory (first occurrence on page 17)
GSMC:	Global Shared Memory Cache (first occurrence on page 17)
HDD:	Hard Disk Drive (first occurrence on page 50)
HDL:	Hardware Description Language (first occurrence on page 21)
HPRF:	High Priority Receive FIFO (first occurrence on page 15)
HPTF:	High Priority Transmit FIFO (first occurrence on page 15)
I/O:	Input / Output (first occurrence on page 10)
L1:	Level 1 (of the cache) (first occurrence on page 17)

LifFPGA:	Link Interface FPGA (first occurrence on page 23)
LPRF:	Low Priority Receive FIFO (first occurrence on page 15)
LPTF:	Low Priority Transmit FIFO (first occurrence on page 15)
MESI:	Modified, Exclusive, Shared, Invalid (first occurrence on page 45)
MPHI:	Message Passing Hardware Interface (first occurrence on page 14)
MPI:	Message Passing Interface (first occurrence on page 52)
MPP:	Massively Parallel Processor (first occurrence on page 12)
NIC:	Network Interface Card (first occurrence on page 16)
NOW:	Network of Workstations (first occurrence on page 7)
NVM	Network Virtual Memory (first occurrence on page 11)
OS:	Operating System (first occurrence on page 7)
PCB:	Printed Circuit Board (first occurrence on page 14)
PROM:	Programmable Read Only Memory (first occurrence on page 23)
RTL:	Register Transfer Level (first occurrence on page 21)
SMP:	Symmetric Multi-Processor (first occurrence on page 7)
SP:	Service Processor (first occurrence on page 14)
SRAM:	Static Random Access Memory (first occurrence on page 17)
SSI:	Small Scale Integration (first occurrence on page 20)
TI:	Texas Instruments Inc. (first occurrence on page 23)
X86:	The Intel line of microprocessors 8086, 80286, 80386, etc. (first occurrence on page 18)

1 Introduction

In recent years, the Network of Workstations (NOW) has become a common platform for running parallel applications. Powerful uniprocessor workstations or SMPs connected via ethernet can provide a large computational resource to simultaneously support many user applications. In most implementations the Operating System (OS) is involved in communication between sites (uniprocessor or SMP) and so the communication latency can be quite large.

When an application requires only course-grain communication then this disadvantage is acceptable because the OS can change context (temporarily suspend the job that requires communication and perform useful work on another application) while waiting for the network communication to complete, but if the goal is to run a single application as fast as possible then the latency and overhead introduced by the operating system and network hardware that is not optimized for frequent low-latency communication are difficult obstacles to overcome.

Nevertheless, several successful projects have made use of standard NOWs as parallel machines; a few of them are briefly described in the “Related Work” section of this thesis. The StarT project is an attempt to improve the performance of standard NOWs through the use of fast networks and the addition of minimal hardware support that will reduce the network latency and improve parallel performance while running programs that require fine-grain parallelism.

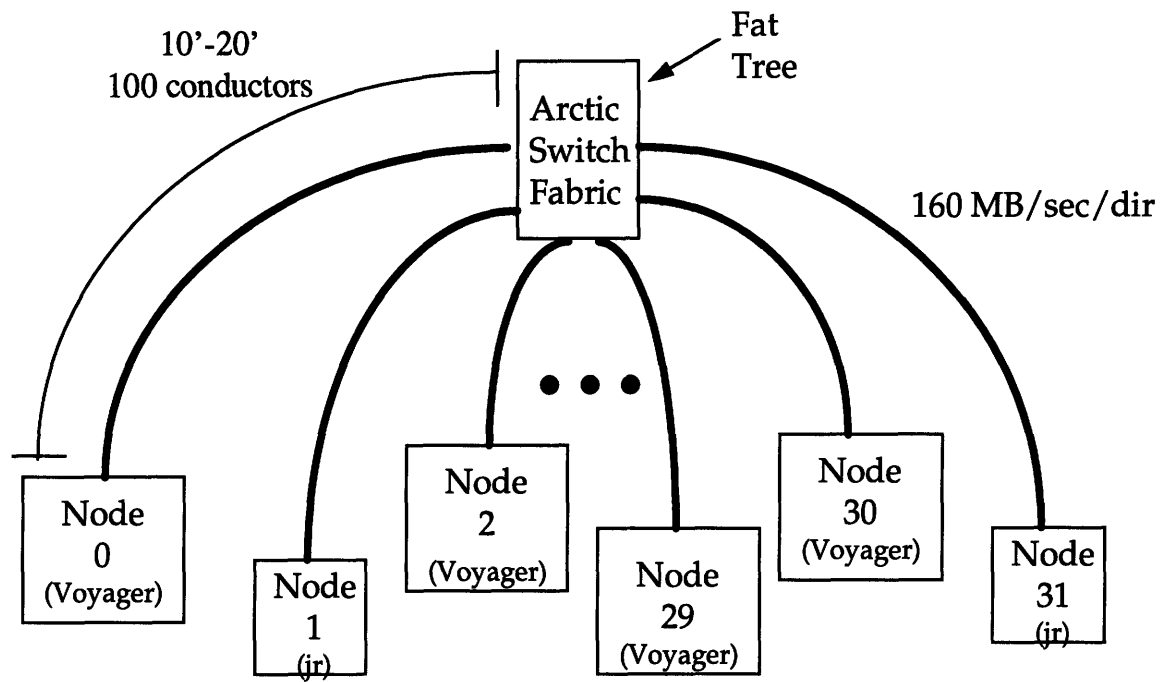
1.1 Project Goals

StarT-jr is part of the StarT project currently underway in the Computation Structures Group of the Laboratory for Computer Science. The goal of the StarT project is to design hardware and software that: presents a familiar (but scaleable) memory model to the user, provides fast user-level message passing, and maintains acceptable performance over a wide range of applications.

We provide support for a global shared memory space because it looks the same to the user as the single memory space provided to applications by uniprocessor hardware. Both StarT-jr and StarT-Voyager (StarT-Voyager is a higher performance multiprocessor that is also part of the StarT project) attempt to provide fast user-level message passing, although StarT-jr sacrifices low latency in order to minimize host overhead [1][2]. Both machines have programmable network adapter cards whose software can be fine tuned to a particular application; this should result in acceptable performance on varying applications. Also, both machines are based on unmodified microprocessors and so should be able to “ride the technology curve” to greater performance with few future design changes.

We will use StarT-jr as an experimental platform for testing user-transparent global shared memory coherency protocols as well as message passing libraries. In addition to its value as a research machine, because it has been functional since late 1995 StarT-jr will be used as a stable test fixture to evaluate other StarT hardware. StarT-jr will be the only host available for testing the Arctic Switch Fabric when prototype chips are evaluated in April '97 [3]. A heterogeneous Arctic network system made up of StarT-

jr and StarT-Voyager nodes will be used to evaluate StarT-Voyager hardware when it is available (see figure 1). The reprogrammability of the StarT-jr hardware also allows evaluation of potential StarT-Voyager features before they need to be committed to silicon.



System based on Arctic Switch Fabric

(figure 1)

The StarT-jr system was implemented in a very short amount of time. The first prototypes were delivered utilizing only five man-months of hardware design effort. This necessitated the use of as much commercially available hardware as is possible. Because we needed StarT-jr to be compatible with the Arctic network and the Arctic Switch Fabric

was not available at the time, it was necessary to make the system flexible enough to interface to multiple network mediums.

We are also experimenting with different host architectures. Since the memory bus of most systems does not adhere to a universal standard, it was necessary for the StarT-jr network adapter to reside on the system's I/O bus for the reason of portability.

1.2 Related Work

As stated above, one of the goals of the StarT project is to improve the performance of standard NOWs through the addition of minimal hardware support; Less elaborate custom hardware will be less expensive to the user. Some systems have implemented distributed global shared memory on a NOW without any custom hardware.

Network Virtual Memory (NVM) provided user transparent distributed shared memory and was described by Kai Li in 1986 [4]. NVM used operating system calls to map pages as owned, read-only, or invalid for each processor in the NOW and was able to present a common global shared memory space to all of the processors running a parallel application. The Treadmarks project improved upon this by allowing multiple simultaneous writers to the same page and later merging together the changes thus achieving higher performance through eliminating most false sharing communication [5]. More recently, the Shasta project performed by researchers at Digital Equipment Corporation has implemented software-only shared memory by inserting extra cache coherency instructions into existing binary code; no OS involvement is required [6].

The Berkely NOW project utilizes a building-wide network of workstations made from all stock hardware to run small interactive jobs and large parallel applications simultaneously [7]. In addition to using commercially available hardware, the researchers at Berkely are altering the industry-provided OS as little as possible. Necessary kernal changes include low-overhead communication software for user-level active messages and possibly an altered swap device driver to allow the virtual memory system to make use of a remote node's RAM as a disk cache. Using a fast commercially available network called

Myrinet, the Berkely NOW project has been able to rival the performance of much more costly MPPs.

GLUnix (the NOW project's parallel operating system layer built on top of UNIX) searches the NOW for idle processors and unused DRAM and uses the resources to perform useful work on parallel applications. Because of the fact that workstations are often idle, this parallel processing power is available almost for free in most NOWs. When an interactive user begins to make use of a resource, GLUnix will attempt to migrate the non-interactive job to another unused workstation.

Implementing hardware support for shared memory can result in higher performance. One of the earliest directory-based parallel machines with hardware support was the Dash machine built at Stanford University [8]. All of the coherency protocols were implemented in hardware and so were fast but not flexible. MIT's Alewife machine utilized limited hardware support with a software extension for "limitless directories" which added flexibility [9]. Fast context switches were key in allowing the system to tolerate network latency. The Alewife machine was based on a customized version of a standard SPARC microprocessor.

The Computer Architecture Group at MIT is currently implementing the M-Machine which is a full-custom quad processor [10]. All global shared memory coherency operations are performed by one of the custom processors. The M-Machine is a very aggressive design and will offer quite a bit of flexibility to customize protocols to a particular application.

2 Project Overview

Our custom network interface provides two basic services for use by the application processor. Support for message passing and user transparent global shared memory is provided.

2.1 Software/Hardware Overview

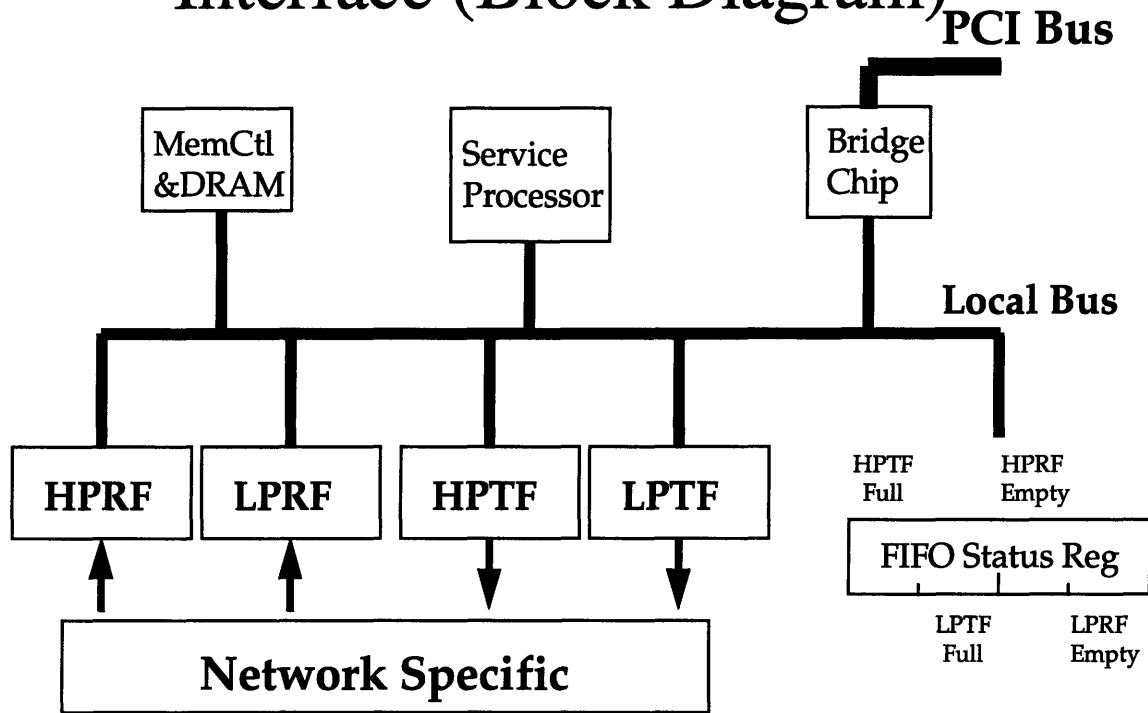
The most important components of the StarT-jr system are the transmit/receive queues supported by James Hoe's lightweight active message communication library. In order to minimize message passing overhead for the host processor, the queues are resident in the host's local memory. The processor can send messages using cached writes and need only communicate to the network adapter over the PCI bus after it has composed a series of messages by updating a pointer on the Cyclone card.

The Service Processor (SP) will then notice that message passing service has been requested and will flush the messages out of the host microprocessor's cache and send them over the network. At the other end, the corresponding SP will write the incoming message directly into the host processor's local memory where it can be retrieved by user software [1].

Service Processor's View of Common Message Passing Interface:

The message passing hardware interface (MPHI) provides the service processor with access to the Modular Network Customization Interface (two separate 1394 High Performance Serial Busses or the Arctic Switch Fabric). All of the silicon used to build the MPHI can be bought "off-the-shelf"; only the PCB and the programmable logic inside the FPGAs are custom to our application (see figure 2).

Common Message Passing Interface (Block Diagram)



Common Message Passing Interface

(figure 2)

The MPFI presents a simple 32-bit FIFO interface to the service processor. A High Priority Transmit FIFO (HPTF) and a Low Priority Transmit FIFO (LPTF) pass commands and packets from the network processor to the MPFI, while a High Priority Receive FIFO (HPRF) and Low Priority Receive FIFO (LPRF) pass command responses, packet acknowledgments, and packets from the MPFI to the network processor. During normal operation, writing/reading packets from all four FIFOs and reading acknowledgments from the HPRF are the only necessary service processor actions.

In order to speed implementation and minimize engineering effort, we chose to define a standard interface between the Message Passing Hardware Interface (MPHI) and the Network Interface Card (NIC).

The first network specific NIC interfaced to a pair of P1394 High Speed Serial Busses. The two busses implemented a physically separate two-priority network to assist in deadlock avoidance. P1394 is an emerging commercial standard targeted at multimedia applications. We used off the shelf 100Mb/sec physical layer silicon. Because the P1394 network can drop packets during normal operation (due to overflow conditions), it was necessary to implement a graceful error recovery scheme in software/hardware.

The second NIC interfaces to the Arctic Switch Fabric. Arctic implements a virtual two-priority network and is reliable, so the NIC checks for errors but can consider it a disaster if an error occurs. Therefore, error detection rather than error recovery is supported in hardware.

Support for Shared Memory:

A user can implement any parallel algorithm on a machine that supports user-level message passing, however, implementing programs via user-level message passing requires the programmer to be aware of low level details that may not be of particular interest. The addition of hardware support to provide a global shared memory space that the user can address in the same manner as all other memory can make the task of programming a machine much less tedious. The data that resides in this global shared memory will be spread out over all the nodes in the parallel system, so as a practical matter, it is necessary to be able to take advantage of data locality and move data in

reasonably sized blocks from one node to another and keep it there for some time in case other data in the block is subsequently requested.

We have implemented a two-level Global Shared Memory Cache (GSMC) that is accessible by a combination of hardware and software resident on the Cyclone EP Module. The memory for the first level is Dual Ported Synchronous SRAM (DPSRAM) resident on the Squall Module attachment to the Cyclone Module. The second level cache is a portion of the i960 bus DRAM on the Cyclone module.

Upon detecting an attempted access to a shared memory location, the Address Capture Device (ACD) hardware will attempt to service the request without the aid of the SP. If the data is not available in the L1 GSM cache, then an interrupt is posted to the SP which takes appropriate action. Action might include sending data requests to other processors in order to retrieve data or ownership of data.

2.2 Implementation Strategy

The main driving force behind our choices with respect to implementation strategy was our desire to produce a working system quickly. We used commercially available components as often as possible and implemented a bare minimum of features. This allowed us to produce a platform for software development in a very short amount of time.

Rationale for Using X86:

Intel microprocessors are the most widely used processors in the world. They are also among the most powerful. A parallel processing architecture based on X86 microprocessors can depend on future performance increases from new processor versions.

Rationale for Using SP:

Message passing overhead is the amount of processing required of the Application Processor (AP) to send/receive a message. The main reason for including a Service Processor (SP) in the message passing architecture is to decrease the message passing overhead incurred by the AP.

When accessing Global Shared Memory space (GSM space), the SP helps to provide shared memory service that is transparent to the AP. The SP implements the coherency protocol which negotiates temporary ownership of data with other SPs in order to service the requests of its AP. Because the SP is itself a microprocessor and is programmable, our design is flexible enough to allow us to experiment with different protocols and optimize the system performance for a wide variety of applications.

Rationale for Using PCI:

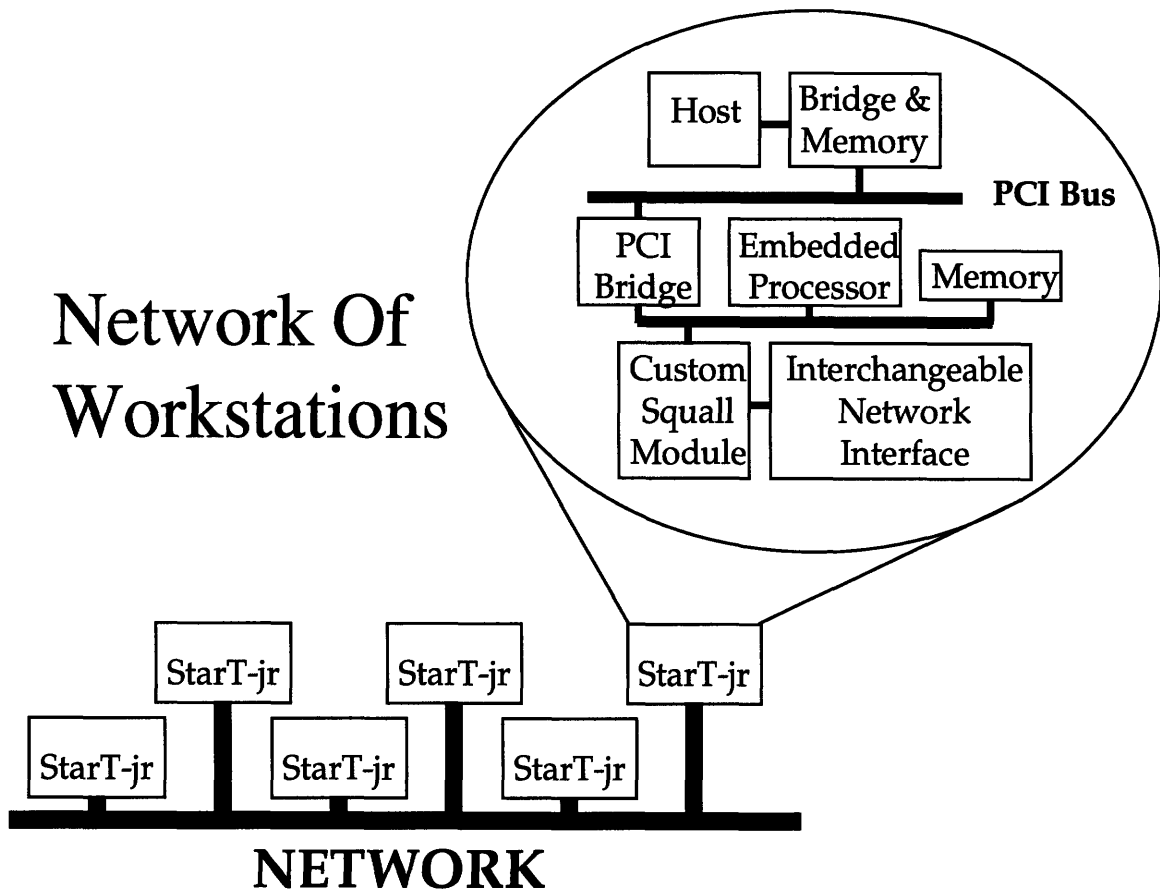
Making hardware that is PCI Local Bus compatible assures that it can be used in future machines with little or no customization.

Rationale for Building Custom Network Interface:

As was mentioned in the introduction, we are building custom network hardware in an effort to reduce the network latency and improve parallel performance while running programs that require fine-grain parallelism.

System Components:

In order to bypass much of the design effort required to produce a microprocessor-based PCI compliant peripheral, we decided to base our network adapter hardware around a commercially available PCI add in card built by Cyclone Microsystems Corp. The Cyclone card allows i960 processor module upgrades so we will be able to increase SP performance as new versions of the i960 become available. The Cyclone card also defines a “Squall Module Interface” that allows our custom hardware direct access to the i960’s local memory bus. The i960 is our SP running coherency protocols and assisting the host X86 processor with message passing (see figure 3).



A StarT-jr parallel machine

(figure 3)

Our Squall module was implemented with off the shelf parts in order to speed the design process. Xilinx FPGAs implemented most of the logic in the design; SRAMs, FIFOs, and CMOS SSI parts made up the remainder of the design. The Squall module connects to a network specific card that plugs directly to the network medium. This network specific card can be replaced by different network specific cards in order to interface StarT-jr to varying network technologies. Because of the high speed of the Arctic Switch Fabric, much of that network specific card was implemented using ECL off the shelf parts.

All of the design was implemented at the register transfer level (RTL) using Verilog HDL. FPGA implementations were created using the Synopsys synthesis tools. In order to verify the logic design before fabrication of the custom printed circuit boards, the RTL logic was surrounded by a structural/behavioral simulation environment that included a bus-cycle model of the i960 processor as well as all other relevant components on the Cyclone, Squall, and network modules.

3 Message Passing Hardware Details

Our custom network interface provides two basic services for use by the application processor. Support for message passing and user transparent global shared memory is provided.

3.1 Service Processor's View of Common Message Passing Interface:

The message passing hardware interface (MPHI) provides the service processor with access to the Modular Network Customization Interface (two separate 1394 High Performance Serial Busses or the Arctic Switch Fabric). All of the silicon used to build the MPHI can be bought "off-the-shelf"; only the PCB and the programmable logic inside the FPGAs are custom to our application. The digital off-the-shelf components include four TI 723631 synchronous FIFOs, and three FPGA/PROM pairs from Xilinx (4005PQ160/XC117) which are used to implement the LIfFPGA (Link Interface FPGA) and daughter card logic. The FIFOs reside on the daughter card as they are used regardless of the choice of network (1394 vs Arctic Switch Fabric). The rest of the components on the "Grand Daughter Card" are also available off-the-shelf from various electronics vendors (see figure 2).

The MPHI presents a simple 32-bit FIFO interface to the service processor. A High Priority Transmit FIFO (HPTF) and a Low Priority Transmit FIFO (LPTF) pass commands and packets from the network processor to the MPHI, while a High Priority Receive FIFO (HPRF) and Low Priority Receive FIFO (LPRF) pass command responses, packet acknowledgments, and packets from the MPHI to the network processor. Single-word and four-word (four word bursts provide more than double the bandwidth of single-word reads/writes) reads and writes are supported. A single-word write to a special address signals that the write in progress is the last word of a packet/request. This notifies the LIfFPGA that there is data in the transmit FIFO.

The network processor performs two different types of writes to the FIFO interface: 1) register requests and 2) packet requests. Many register/packet requests can be written to the transmit FIFOs. The hardware responses (if any) will be queued up in the HPRF, which can hold at least sixteen responses and packets.

The control register on one of the daughter card FPGAs contains a full flag for each transmit FIFO and an empty flag for each receive FIFO. The empty flag must be consulted only once per packet/response, as the flag is only updated when the last word of the packet/response is written/read to/from the FIFO by the LIfFPGA/network-processor. In all cases, the type of packet/response can be determined by looking at the first word. The length of a received packet is contained in the length field defined by the network standard. Register requests are used to read and write registers on the LIfFPGA and other chips resident on the NIC.

The register request method is used to initialize the card and in error recovery. Posting a read register request to the HPTF will cause a request response to appear in the HPRF (an identification word/bit and the read data). Register requests have a bit set in the first word that differentiates them from a packet request. During normal operation, writing/reading packets from all four FIFOs and reading acknowledgments from the HPRF are the only necessary service processor actions.

Modular Network Customization Interface:

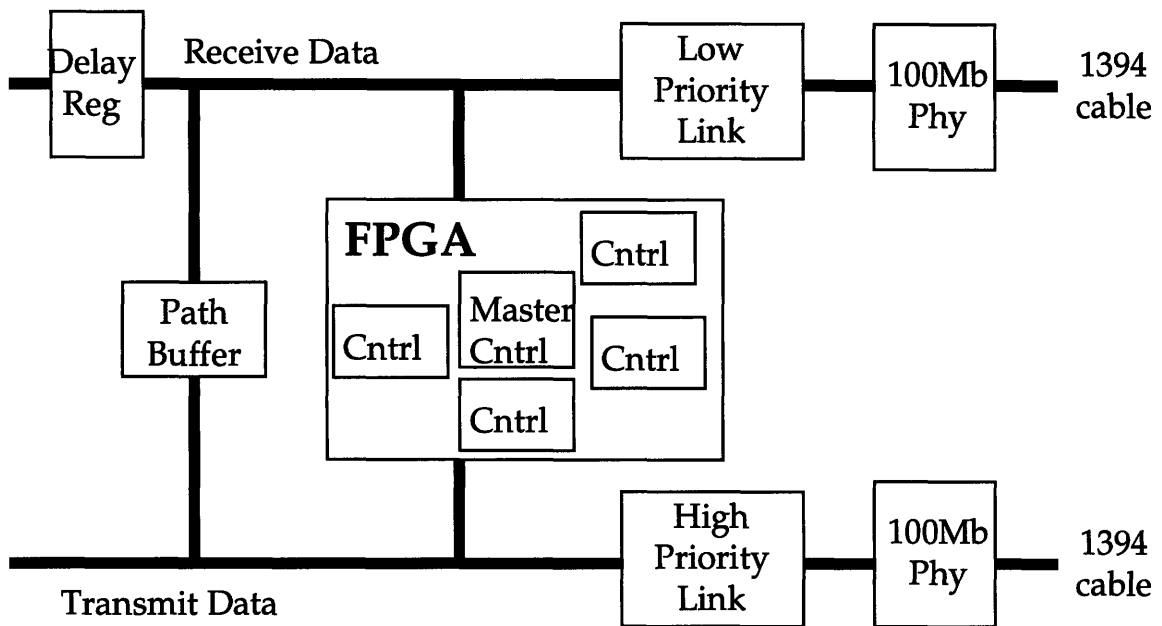
In order to speed implementation and minimize engineering effort, we chose to define a standard interface between the Message Passing Hardware Interface (MPHI) and the Network Interface Card (NIC).

3.2 High Speed Serial Bus Interface Card:

We chose to start with a 1394 High Speed Serial Bus NIC because it would take the shortest amount of time to implement. Most of the network specific actions are handled by the Link-layer controller (TI 12C01A), so it was only necessary to design an interface to the Link chip and a DMA engine to move data from the MPHI FIFOs to the Link's on-chip FIFOs; Texas Instruments (TI) provided engineering expertise and a design for the analog portions of the NIC (TI also had the boards manufactured for us). Register read/write request control was designed into the NIC's FPGA so that direct manipulation of the Link chip registers could be directed by SP software. The length of register read responses is two 32-bit words (with the exception of a LIFPGA read-enable register read response which is one 32-bit word): an identification word followed by the data word. The length of a hardware packet acknowledgment is one 32-bit word.

Packet requests result in transmission across either the high or low priority 1394 Serial Bus and cause a hardware acknowledgment to be written over the "Receive Data" path into the HPRF of the sender (see figure 4). On the other side of the network, the packet will be placed into the appropriate receive FIFO of the receiver. The hardware acknowledgment can be associated with the corresponding packet by examining a 4-bit transaction label that is part of both the packet and the returned hardware acknowledgment word (which includes the "ack" code). In most cases, the ack will be 0x1 which means that the packet was received and will be processed by the network processor on the other node. If any other ack is returned, it will be necessary for the network processor to resend the packet. If a "busy" ack is returned, it is likely that the

other node's receive FIFO has overflowed and that node is currently in the error recovery state. If "ack missing" is returned then no node responded to the packet which is normal for broadcast packets, but probably is the result of a header CRC error on a node-to-node transmission.



1394 High Speed Serial Bus NIC block diagram

(figure 4)

Error Recovery:

A General Receive FIFO (GRF) in each Link chip buffers packets until they can be moved by the LiffPGA into the HPRF or LPRF. The size of a GRF can be adjusted using a register write request to the appropriate Link chip. However, regardless of the size of the GRF, it is possible for it to overflow if packets are not read out of the Squall Module

receive FIFOs and they become full thus prohibiting continued movement of data from GRF to HPRF/LPRF.

In the event of a GRF overflow, an interrupt is signaled to the network processor and data movement from that GRF is disabled. Data movement from the other GRF (high priority Link GRF after a low priority Link GRF overflow or low priority Link GRF after a high priority Link GRF overflow) can continue. The interrupt routine should perform a LIFFPGA read-enable register read request to determine which GRF overflowed.

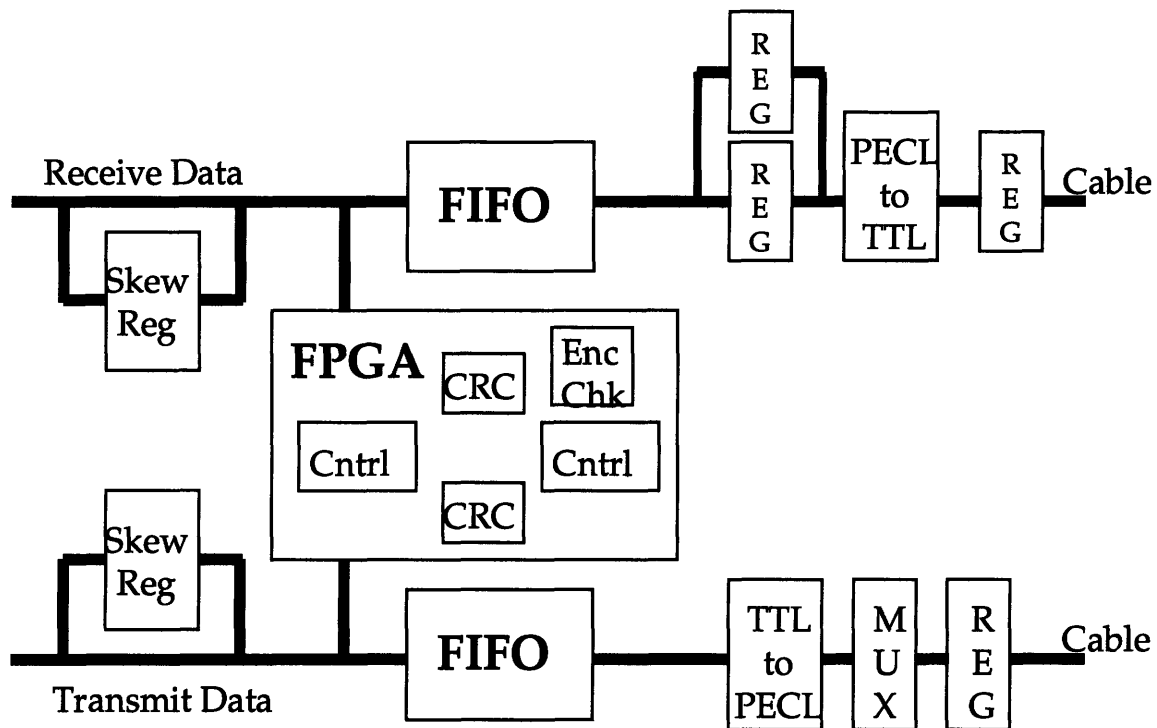
A lower bound for the number of "good" packets (packets which were acknowledged by hardware with $ack=0x1$) in the overflowed GRF is a function of the GRF size and the maximum packet size. For compatibility with the Arctic Switch Fabric, our chosen maximum packet size is 96 bytes (plus a 32-bit "ack sent" word makes a total of 100 bytes). The LIFFPGA can be commanded to transfer this lower bound number of packets into the receive FIFOs with a LIFFPGA read-enable register write request. The LIFFPGA will interrupt the network processor after the packets have been moved. Because some or all of the received packets might be smaller than the maximum allowable size, another LIFFPGA read-enable register write request might be appropriate.

When it is possible that the next packet to be read out of the GRF is a partially received packet, the network processor must use register read requests to inspect the remaining contents of the GRF and service any other packets for which hardware returned $ack=0x1$ to the sender. The partial packet or packets (if any) can be discarded as it/they will be resent. At this point, the auto-receive function can be reenabled and normal operation can continue.

3.3 Arctic Switch Fabric Interface Card:

The target network medium for the StarT-jr project is the Arctic Switch Fabric. The switch is based on the Arctic routing chip which was designed by a team headed by Dr. George Andrew Boughton in the Computation Structures Group. Arctic specifies a high bandwidth interface. A bandwidth of 320MB/sec (concurrent 160MB/sec transmit and 160MB/sec receive) can simultaneously be available to all nodes in the system. The StarT-jr Arctic NIC hardware is cable of utilizing a little less than half of this performance, while the Squall Module hardware is capable of using less than 25% of it. The rate at which SP software can format and send packets is even lower and so therefore it would not make sense to try to upgrade the Squall/NIC until we make progress in the packet formatting area (StarT-Voyager's SP can potentially use 100% of the available Arctic bandwidth).

The Arctic specification calls for ECL differential drivers/receivers at the cable interface for electrical (signal integrity) reasons. Because of the high frequency of the cable clock, it was convenient to use ECL logic to double the width of the data path on the NIC in order to halve the clock frequency for some of the control sections of the Arctic NIC control FPGA (see figure 5). The width of the data paths on the cable (for separate transmit and receive data paths) is sixteen bits, while the FIFOs shown in the block diagram are thirty-two bits wide.



Arctic Switch Fabric NIC block diagram

(figure 5)

There are three separate clock domains on the Arctic NIC. An 80MHz clock is generated on the NIC and is sent over the cable along with transmit data. This same clock is used to produce a slightly phase-shifted 40MHz clock which is used to move data out of the NIC's transmit FIFO so that it can be multiplexed and fed into the cable driving register. A second clock domain exists on the receive data path. An 80MHz clock signal is received from the cable and is used to capture data. This same clock is used to produce a 40MHz clock that moves the data into the NIC's receive FIFO.

The third clock domain is used to move data from the NIC FIFOs to and from the Squall module FIFOs. The CRC calculation being done by the Arctic NIC could not

easily be performed at 40MHz with the chosen FPGA technology and so the NIC FIFOs and this third clock domain were added. Both the transmit and receive CRC calculations can be performed simultaneously (the data paths are separate) but for convenience are clocked by the same 20MHz crystal.

Several error conditions are flagged by the Arctic NIC FPGA's error checking circuitry. Standard CCITT-16 CRC is generated and checked by the NIC FPGA. If an incoming packet fails CRC then an interrupt is posted to the SP and the NIC stops accepting incoming packets. The flawed packet is not confirmed in the Squall Module FIFO for the convenience of the SP. The SP can depend on the fact that any packet detected in a receive FIFO is valid.

A "PHASE" signal is sent across the cable along with the data. The logic level of PHASE toggles after every 80MHz clock. The PHASE signal allows the receiver to group 80MHz data into pairs; PHASE is low during the first half of a pair and is high for the second half. If PHASE ever fails to toggle after some 80MHz clock then an interrupt is posted to the SP and the receiver disables itself.

Control signals that are sent over the cable are Manchester encoded. One bit of information is sent for every pair of 80MHz clock periods. The exclusive OR of the pair corresponding to a control bit must always be true or an error is flagged. The FRAME bit set to true indicates that a packet transmission is in progress and information other than the idle pattern is present on the data lines (actually, the FRAME bit becomes false on the last pair of 80MHz clocks corresponding to the packet transmission so that the receiver

can prepare to terminate packet reception). A FRAME error (exclusive OR of a pair is false) will cause an interrupt to be posted and will disable the receive logic.

Each receive section in the Arctic routing chip has three packet buffers. When all three buffers are full, the chip cannot accept another packet on the corresponding link. When a buffer is no longer occupied because data has been forwarded, the Arctic receive logic notifies the sender that it has freed up a buffer and that it can accept another packet. It is the responsibility of the sender to keep track of how many free buffers are available in the receiver. A virtual two-priority network is implemented by requiring the last free buffer to be filled by a high priority packet. Because of this restriction, an Arctic network that is congested by low-priority packets will continue to allow high-priority packets to flow through the network.

The receive logic notifies the sender that a new buffer is available by asserting the BUFFER_FREE signal. The BUFFER_FREE signal is sent in the opposite direction of the data flow on this half of the bi-directional link. BUFFER_FREE is Manchester encoded and an error detected in the receive section of the NIC will cause an interrupt to be posted to the SP, but will not terminate packet reception. The pair that caused the error will be ignored (assumed to be deasserted) and it is possible that the NIC will no longer use the third of its three available input buffers in the Arctic router across the cable. This would cause serious performance degradation which would persist until the SP could reset the NIC (only one available low-priority input buffer would prevent pipelining of packets through this hop over the network).

The Arctic router connected to the NIC will be programmed to assume that the NIC has three available input buffers. When a packet has been received, the NIC FPGA moves it from the NIC receive FIFO to either the HPRF or LPRF on the Squall Module and immediately signals BUFFER_FREE to Arctic. The fact that HPRF and LPRF can hold much more than three packets allows temporarily continued packet reception while waiting for SP software to empty out the Squall Module FIFOs. However, once HPRF and/or LPRF are full the NIC will stop sending BUFFER_FREE to Arctic which will eventually halt the flow of packets out of the network to this node. When room becomes available in HPRF/LPRF a BUFFER_FREE will be sent and transmission can continue.

NIC Initialization:

After the SP commands the NIC to come out of the reset state by clearing a bit in the control register on the Squall Module, it must prepare the NIC to send and receive packets. The NIC drives the output of the NIC transmit FIFO across the cable and the contents of the FIFO are unknown after power-up. Before enabling the corresponding Arctic router's receive section it is therefore necessary to take some action which guarantees that the idle pattern and deasserted FRAME are being driven across the cable. This is accomplished by sending any valid packet (which will be ignored by the not yet enabled router chip receive logic). The NIC FPGA inserts the idle pattern in the FIFO at the end of each packet and so once a packet has been sent no further special action must be taken.

After all transmitters are sending the idle pattern the receive sections can be enabled. The NIC receive logic is enabled by a two word command written by the SP into

the HPTF. This command also clears the error register and causes its previous contents to be written to the HPRF so that it can be examined by the SP. The error register contains four bits: CRC error, BUFFER_FREE error, PHASE error, and FRAME error. After the SP reads the contents of the error register from the HPRF the hardware is in normal operating mode.

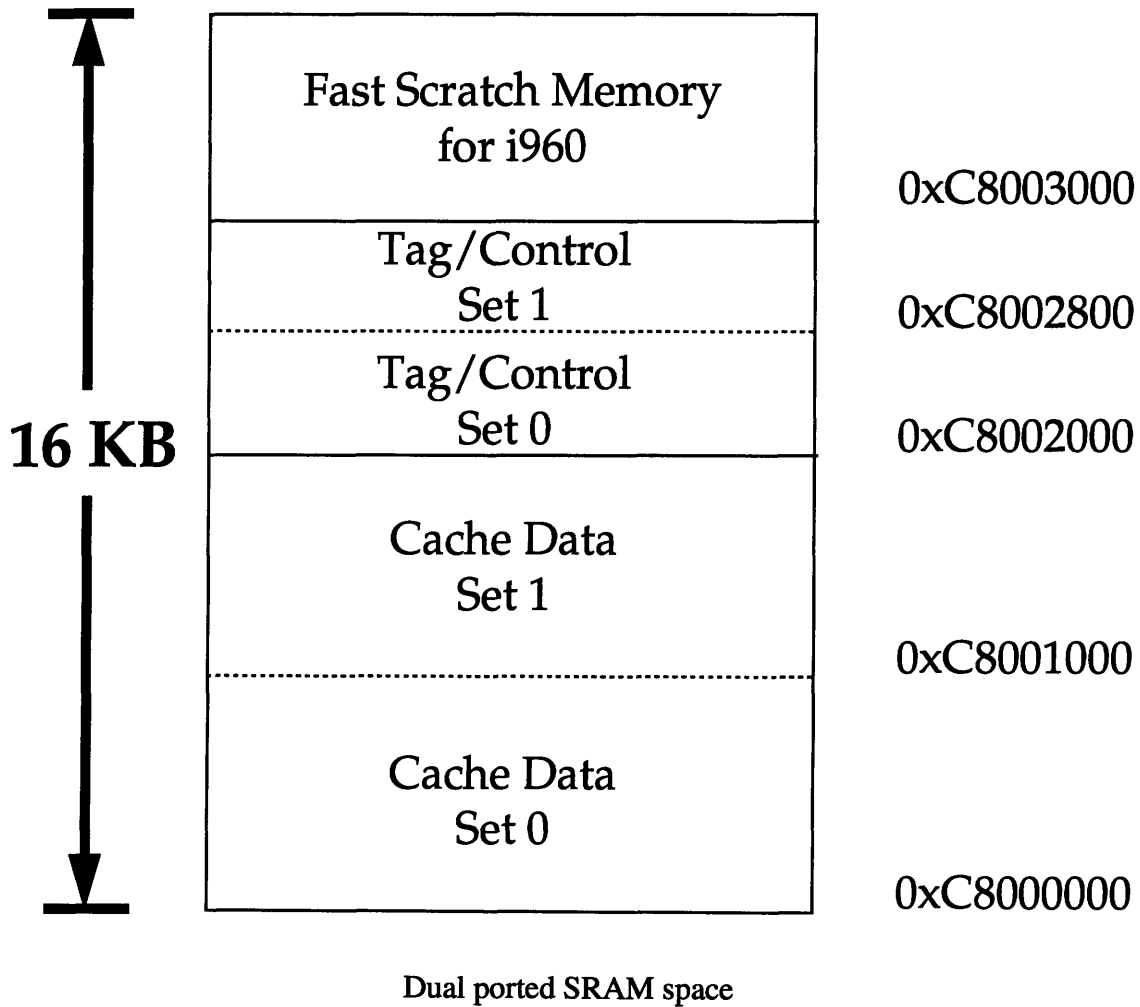
A similar method is used to initialize the NIC after an error. First, the SP should read all valid packets out of the HPRF/LPRF. Second, the SP should reset the Squall Module FIFOs in case a partial packet is present in HPRF/LPRF. Then the SP can take the actions described above to drive the idle pattern, recover error information, and reenables the receive logic.

4 Shared Memory Support Details

User transparent global shared memory can be used by a parallel application in nearly the same way that a uniprocessor application uses its own memory space. A system that supports such a shared memory space makes it easier for the programmer to write his/her application in a familiar style and does not require him/her to deal with low-level details such as race conditions in memory references (the protocol software running on the SP can implement sequential consistency or some other weaker memory model and will take care of many of the low-level details, however, there is always the danger in a parallel system that the user might implement an algorithm in such a way that network latencies introduce some amount of non-determinism in the application).

Common operations such as a “sync” operation can be implemented using sequentially consistent global shared memory with very little trouble by the programmer. The details of what message is sent when and to whom are taken care of by the protocol processor (in our case, the SP). We have implemented global shared memory on StarT-jr with a combination of custom hardware and software running on the SP. It is obviously advantageous for performance reasons if most memory references can be serviced by the hardware and so we decided to make cache hits more likely by implementing multi-set associativity in hardware. N-set associativity allows the hardware N choices as to where in the cache to place a line of data while casting out the least valuable of the N lines that are already in the cache (as opposed to a direct-mapped cache where N is one and there is only one place in the cache where a particular piece of data can reside).

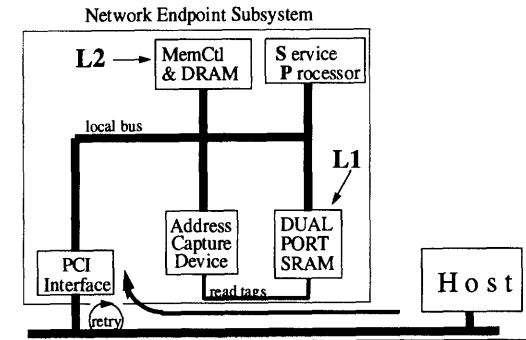
We have implemented a two-level Global Shared Memory Cache (GSMC) that is accessible by a combination of hardware and software resident on the Cyclone EP Module. The memory for the first level is a 16KB Dual Ported Synchronous SRAM (DPSRAM) resident on the Squall Module attachment to the Cyclone Module. One half of the DPSRAM is used for the first level GSMC (half of that half is used for each of the two “sets”), while one quarter is used for GSMC tag space. The remaining 4KB is reserved for miscellaneous control functions and fast external scratch memory for the i960 (see figure 6).



(figure 6)

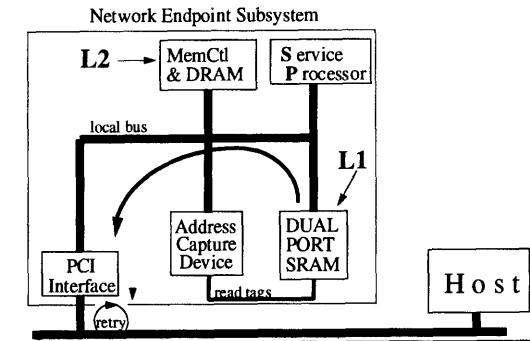
The hardware relevant to the GSMC is shown in figure 7. On a read access from the host, the PCI interface chip requests ownership of the local bus and places a global shared memory address on the address lines (see figures 7a and 7b); the 128MB of globally shared memory is addressed with 32-bit addresses from 0xC0000000 to 0xC7FFFFFF on the Cyclone EP Module, however, the PLX Technology PCI Interface Chip can translate other address spaces to the required range. All global shared memory accesses must be 32-bit aligned 32-bit word accesses.

Shared Memory Data Paths



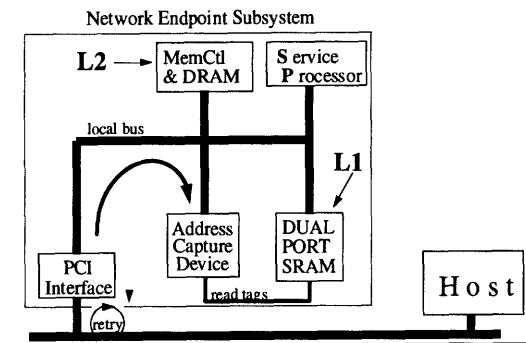
(figure 7a)

Shared Memory Data Paths



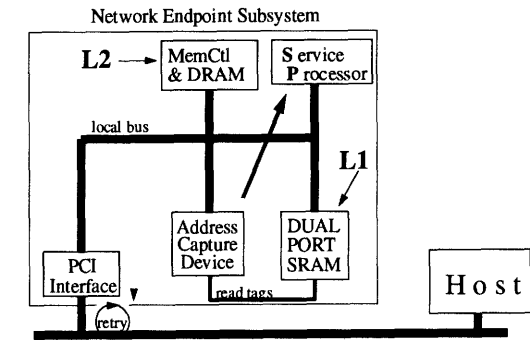
(figure 7d)

Shared Memory Data Paths



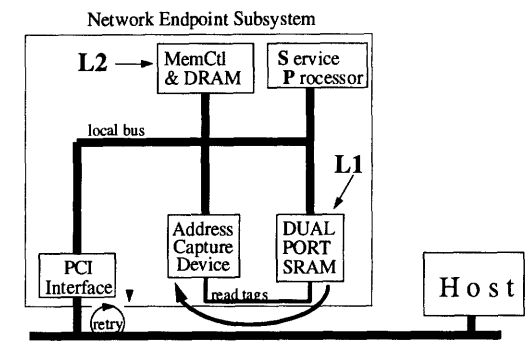
(figure 7b)

Shared Memory Data Paths



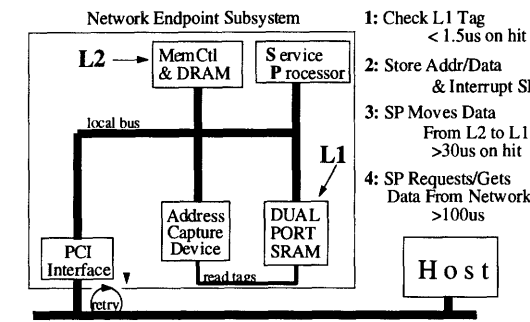
(figure 7e)

Shared Memory Data Paths



(figure 7c)

Shared Memory Data Paths



- 1: Check L1 Tag < 1.5us on hit
- 2: Store Addr/Data & Interrupt SP
- 3: SP Moves Data From L2 to L1 >30us on hit
- 4: SP Requests/Gets Data From Network >100us

(figure 7f)

4.1 Global Shared Memory Reads

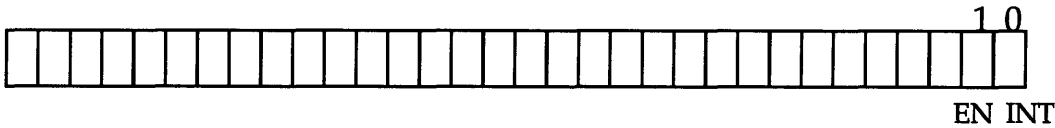
Upon detecting a read of a shared memory location, the ACD (Address Capture Device) hardware will proceed to read two tag/control words from GSM Tag Space via the left side port of the DPSRAM (see figure 7c). If the ACD determines that the requested data is available in the DPSRAM, it will allow the PCI Interface chip to access the DPSRAM and complete the transaction (see figure 7d). If the ACD determines that the requested data is not currently accessible, it will force a PCI bus retry by not responding during the PCI Interface chip time-out period (this time-out period is software programmable and is set to be slightly longer than the worst-case response time of the GSMC control hardware to a hardware-completed GSM operation). At the same time, the ACD will signal an interrupt to the network processor (see figure 7e). Until the interrupt has been serviced, the ACD will not respond to any global shared memory access and so any such access will result in a retry. The reason for this is so that the interrupt routine can read information about the access that caused the interrupt from ACD registers (shadow copies of these registers would allow operation to continue, however, the interrupt is serviced so quickly in comparison to PCI bus transaction timing that such duplication would yield relatively little performance improvement - see figure 8).

The network processor's interrupt routine will hopefully be able to retrieve the requested data from the level two cache. The memory for the level two cache is some portion of the 32MB DRAM resident on the Cyclone EP Module (16MB for most applications). The network processor copies the data from the level two cache into the level one cache and updates GSM tag space to reflect the change so that the next GSM

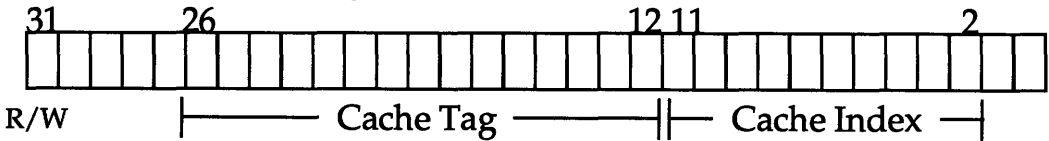
access to this location can be completed by the ACD. It is possible that the data replaced in the level one cache will require cache coherency operations by the network processor before the data copy can proceed.

If the level two cache does not contain the required data, then the network processor utilizes the message passing hardware interface to send a message over the network to another node requesting the data. After receiving a read response (with data included), it can then proceed to update the level two cache and the level one cache as above. In either case (level two hit or messaging required), the interrupt routine will also enable the ACD to accept GSM accesses when future level-one hits are possible. The amount of time it takes to return data to the host is dependent on the location of the valid data (see figure 7f).

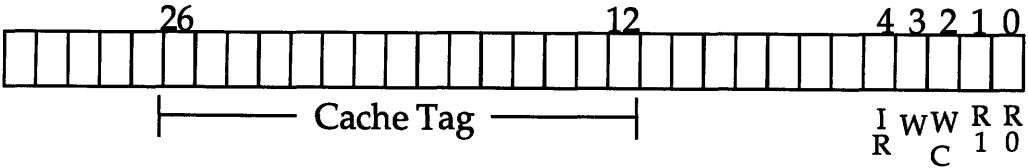
Control/Status Register:



Address Capture Register:



Tag/Control Word (in SRAM):



Shared memory registers

(figure 8)

4.2 Global Shared Memory Writes

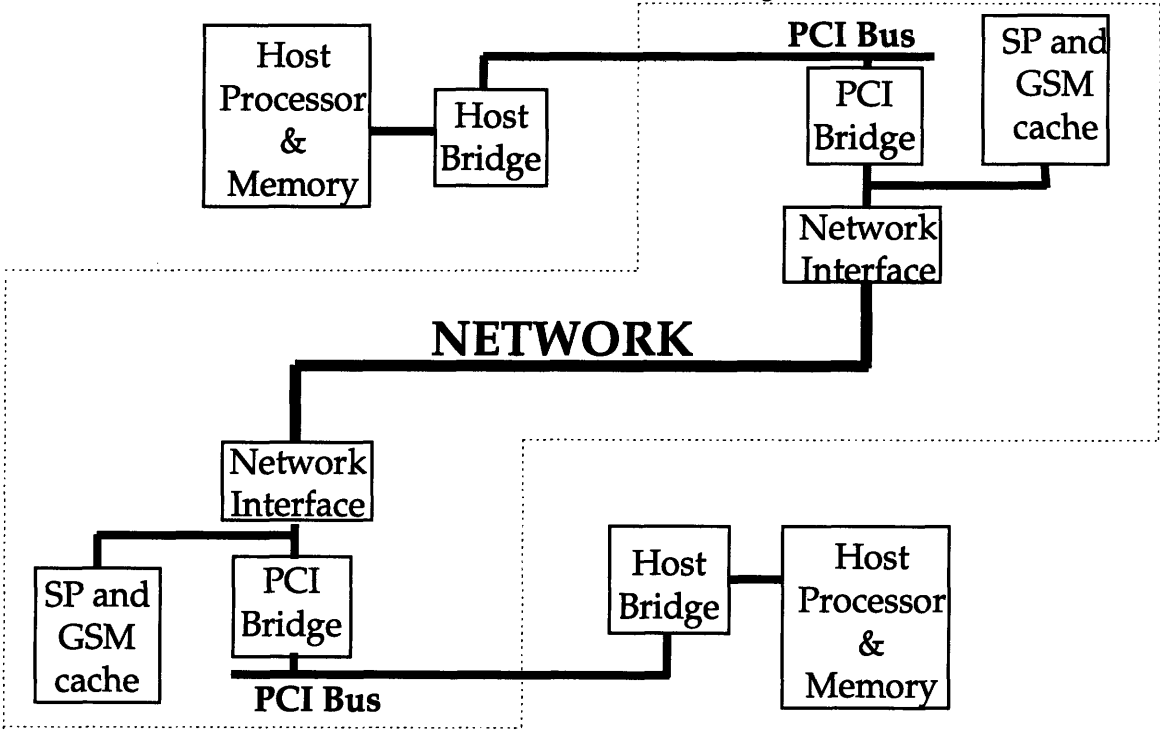
The actions required to complete a write access to GSM space are similar to those required to complete a read access to GSM space. The hardware reads two tag/control words from GSM tag space and evaluates them in reference to the requested write access. The two tag/control words correspond one to one to the two sets of our two-set-associative level one cache. If the control portion of either of the two tag/control words indicates that a write can proceed to the corresponding cache location (in the event that both locations are "eligible", the more appropriate one is chosen), then the ACD allows the data to be written to the DPSRAM and completes the transaction (from the point of view of the host processor). Otherwise, the ACD does not respond and interrupts the network processor.

Even if the data can be accepted thus completing the transaction, it is likely that the ACD will interrupt the network processor in order to manipulate tag/control words in GSM tag space. Several bits of the control portion of every tag/control word are used to indicate to the ACD when such an interrupt is appropriate.

Both coherent write and non-coherent write modes are supported. In coherent write mode host processors do not cache GSM locations, but coherency is maintained between the two-level GSM caches on all nodes in the system up to the PCI bus boundary (see figure 9); a GSM write will always fail unless this node's host processor has obtained ownership of the write destination. When the host processor attempts to write to an unowned location, the ACD recognizes that the location is not owned because either the tag will not match or control bits will have marked the location as "not writable". The

interrupt routine will subsequently gain ownership of the cache block, alter tags, and finally will mark the block as "writable" so that the ACD will be able to complete future write transactions to that cache block.

Coherence Boundary



Coherence boundary

(figure 9)

In non-coherent write mode, a write access to GSM space will be accepted by the ACD before ownership of the block has been acquired. The interrupt routine will then be called upon to update the tag and control bits in tag/control space. As with all other

interrupt situations, no other GSM accesses will succeed until the interrupt routine has had a chance to read the access information captured by the ACD.

The PCI interface chip will complete a write transaction before it is completed on the local bus, so for GSM writes to be transparent to host software it is necessary to dedicate one of the two level-one cache sets to acceptance of GSM writes. In this mode, all of the control words for the dedicated set are marked as "non-coherent" and are issued a tag that corresponds to an unmapped page of GSM. This guarantees a place to put any write data and causes an interrupt to the network processor due to the fact that the tag cannot match (a matching tag to a non-coherent location would not cause an interrupt).

The processor responds to the interrupt and immediately causes the PCI chip to retry all further accesses while it reads access information from the ACD and retrieves the data from the DPSRAM. If the GSM access tag matched the tag for the other set and the location was writable, the ACD would have completed the transaction without signaling for an interrupt. We have not yet verified that the network processor will be able to service this type of interrupt quickly enough in a worst case scenario (before another GSM write occurs). If the network processor is not fast enough, then two GSM writes would have to be separated by a GSM read (to any address) as seen on the PCI bus after processor reordering which is, admittedly, a rather difficult requirement.

If transparent operation is not required, each write to a new GSM block entails two write-read pairs. If the first write attempt fails, the second try will always succeed because the first read attempt will be retried until the cache block is owned. A second read of the location is necessary to signal that ownership of the block can be relinquished

if necessary. This four access protocol is only necessary to gain ownership of the block/page, and so further writes to the block/page can be single transparent writes until ownership is (possibly) relinquished by the second read to the original location. A pair of "SYNC" operations should surround the transparent write operations.

When the network processor is interrupted due to a write access that will be retried, it prepares one of the two possible cache locations to receive the data. This may require over-the-network cache coherency operations and/or movement of data between the level one and level two caches. At some point, the network processor alters the appropriate tag/control word in GSM tag space so that the ACD will see an available spot to accept data when the write request is retried. The interrupt routine also enables the ACD to accept another GSM access.

4.3 Split-Phase Transactions

Split-phase operations can hide the latency of network coherency operations. The host processor performs a split-phase read by reading a GSM location and then comparing the returned value to a "Miss Pattern". If the compare succeeds, then the data is discarded and the host should put the current process to sleep. The network processor will wake up the sleeping process after retrieving the requested data; the process will then retry the read and continue. If the requested data happened to match the Miss Pattern, then the network processor can immediately wake up the process and the second read is redundant but not harmful.

Our ACD provides support for split-phase reads in the form of an "interrupt on any read access" bit present in each tag/control word. When a GSM read access misses and is retried on the PCI bus, if the data is not present in the level-two cache the network processor can place the Miss Pattern in the GSMC location, set the "interrupt on any read access" bit, and adjust the tag and read access bits accordingly. The PCI bus retry will succeed, disable further GSM accesses and interrupt the network processor. The network processor can then revoke read access from the GSM location and enable the ACD.

4.4 Coherency Protocol

Caching for the purpose of minimizing network traffic and/or latency (prefetching can reduce average latency) is critical in any implementation of distributed shared memory. I wrote a sequentially consistent coherency protocol to support global shared memory on StarT-jr in order to prove that all required hardware support was present. Three of the four MESI states were used to describe each cache block: Modified, Shared, and Invalid. The protocol was greatly simplified by the restriction that each node was allowed at most one outstanding request. Memory resources at each node were abundant enough compared to the total number of nodes so that it was not a concern.

Each node is responsible for maintaining directory information related to part of the global memory space and is considered the home node for the region. A request for exclusive or shared access to data would be sent to the appropriate home node. The home node maintains two lists of pending requests: the active list and the sleep list.

The active list contains requests that are currently being serviced. When a request is received by a home node, if there is no other request on the active list pertaining to the same cache block then the SP will immediately take some action to service the request. For example, if the request is for a shared copy and no other node has a modified copy then the home node can immediately return the data and the new request need not be added to the active list (in my protocol, if any node has a shared copy then the home node must have one also). However, if another node has a modified copy of the data then the home node must send a request for the modified data and put the incoming request on the active list so that it can complete the transaction at a later time.

The sleep list contains requests that cannot be serviced at this time. When a request is received by a home node, if there is another request on the active list pertaining to the same cache block then the SP will place the incoming request on the sleep list. Whenever a request is deleted from the active list (service has been completed), the sleep list is scanned for requests to the same cache block. If one is found then the SP attempts to service the request. Partial service might result in a request being moved from the sleep list to the active list.

In order for a protocol to be considered sequentially consistent one must be able to build some model of the entire global shared memory region and have it agree with each node's global shared memory cache (all processors agree on the contents of the shared memory). Proof of the following two theorems is sufficient proof that a protocol is sequentially consistent: **1)** If any node has a cache block in the modified state, then all other nodes have that block in the invalid state. **2)** If any two nodes have a shared copy of the same cache block, then both nodes' blocks contain the same data.

The first theorem is true for my protocol because when the home node receives a request for a modifiable copy of a cache block it will send invalidate messages to all nodes with shared or modified copies and place the request for the modifiable copy on the active list. Only after receiving confirmation of invalidation from all nodes that had a shared or modified copy will the home node respond to the original requester and retire the request from the active list.

The second theorem is true because a cache block must transition to the modified state before the data in the cache block can change. Because there can be no shared

copies until there are no longer any nodes with the block in the modified state (the home node will not respond to a request for even read-only data until no modified version exists), any copy in the shared state must have been acquired after the most recent data update. Therefore, all such copies must be equivalent.

We can build a model of the global shared memory region as follows: If some node has the cache block in the modified state then we choose that node's copy of the data as part of our model of memory. This will not disagree with any other node's cache if the protocol satisfies the first theorem. If one node has the cache block in the shared state then we choose that node's copy of the data as part of our model of memory. If more than one node has the cache block in the shared state then we can choose a copy from any one of those nodes because they all are equivalent (if the protocol satisfies the second theorem).

5 Experimental Results

As was mentioned above, the most important goal in the StarT-jr project was to minimize the overhead incurred by the host while sending messages. The message passing overhead for applications running on StarT-jr is favorable compared to that of other machines based on similar network technology [1]. The overhead is defined as host CPU time used to send a message. This would include packet-header/data formatting and communication with our network adapter card amortized over the average number of packets sent per adapter card handshake. StarT-jr's message passing overhead is independent of the network medium and is slightly less than 2usec/packet (see table 1). The overhead number presented is for a single packet launch (one packet and one handshake).

	Arctic	1394
Overhead	2 usec	2 usec
Round-trip Latency	35 usec	170 usec
Hardware Bandwidth	65 MB/sec	7 MB/sec
Actual Bandwidth	10 MB/sec	5 MB/sec

Low level performance measurements

(table 1)

The round-trip latency depends on the network medium as can be seen in table 1. Because the Arctic Switch Fabric was not available at the time that these measurements were taken, the 35usec number for Arctic was measured using two StarT-jr nodes connected directly together using a cable similar to the one which will be used to attach a node to the switch. The round-trip number includes host overhead at both ends, SP software processing at both ends, and network hardware delay in both directions which is part of the reason that the response time is so large. SP processing is more complicated for the 1394 NIC because of the additional support required to present the user with the abstraction of a reliable network, while the Arctic switch hardware is assumed to be reliable.

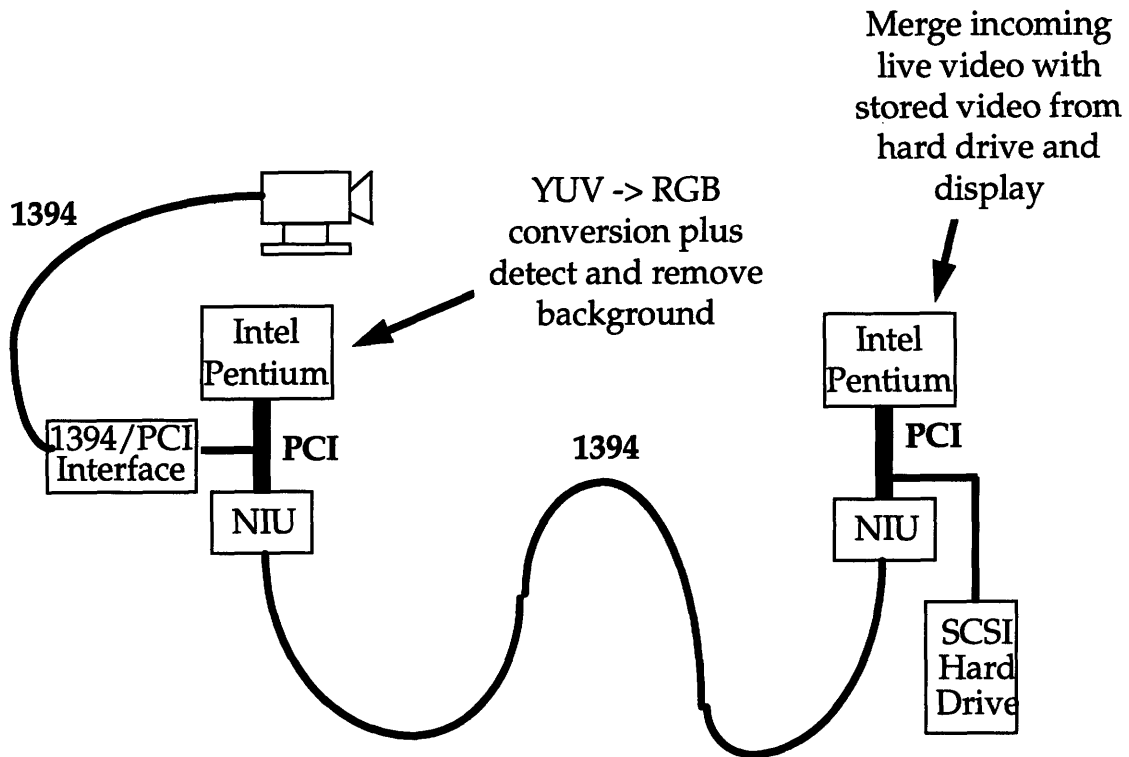
The hardware bandwidth numbers pertain to the limits of our custom hardware. These numbers were obtained by counting clock cycles required to send/receive maximum length packets (24 four byte words). In the case of Arctic, the limiting factor was the speed at which the SP can write/read data to/from the FIFOs on the Squall Module. In the case of 1394, the limiting factor was the speed at which the NIC FPGA state machines could use DMA to move data from/to the Squall Module FIFOs to/from the TI 1394 Link level controllers.

The actual bandwidth numbers were obtained by James Hoe using his lightweight active message communication library running on the SP. These numbers are far below the hardware limits and illustrate the fact that SP message processing is the bottleneck for our system.

Digital Video Processing Demo:

The first application to be run on StarT-jr was the demo we presented at the 1995 Fall Comdex show in Las Vegas, Nevada in cooperation with Texas Instruments (TI). TI viewed StarT-jr as an interesting application of their 1394 silicon. The demo made use of our custom Squall module and 1394 NIC. James Hoe was responsible for the lightweight active message communication library software running on the SP (Cyclone Card) and for the video demo software running on the host machines.

We digitally recorded video at the show using a Sony 1394 compatible desktop camera and stored the data on a hard disk drive (HDD). Afterward we used the same camera to digitize live video of a subject standing in front of a solid colored background. The image of the background was detected and removed by the host processor of the first machine and then sent through our custom message passing hardware to the other machine where the image was merged in real time with the stored video playing from the HDD (see figure 10).



Digital Video Processing Demo

(figure 10)

One Intel Pentium machine provided insufficient processing power to convert the data format, filter out the background, and merge the video streams in real time and so this demo was a good candidate for a parallel processor. The message passing bandwidth obtained while performing the demonstration can be seen in table 2. The 6MB/sec number exceeds the "actual bandwidth" number from table 1. This is because the communication was nearly unidirectional with only occasional flow control packets in the reverse direction. Also, data loss in such a multimedia application is not a catastrophe and so SP software was written accordingly in order to optimize bandwidth. As seen in table 1, a

typical data application with bi-directional communication would likely top out at 5MB/sec using our 1394 NIC.

	Arctic	1394
Distributed Video Processing	10 MB/sec	6 MB/sec
MPI	20 MB/sec	not measured
Matrix Multiply linear up to	2.5 MB/sec	not measured
Barrier (two nodes)	1ms/barrier	not measured

Support software performance measurements

(table 2)

Host Message Passing Software:

The MPI number in table 2 was obtained by Andy Shaw and Adam Meyerson, both members of the Computation Structures Group. I worked with them to implement hardware support for DMA operation that moved data directly from sending host memory to receiving host memory without SP involvement (using the DMA engines on the PCI interface bridge). We were able to exercise the hardware at rates of nearly 50MB/sec for large data blocks. As of yet, they have not been able to realize all of this bandwidth using their MPI library but are still investigating this issue.

Matrix Multiply Application:

Matrix multiply is an embarrassingly parallel application that requires relatively little interprocessor communication. The far from optimal coherency protocol I wrote for the SP can sustain about 2.5MB/sec bandwidth over the PCI bus to the host when most accesses result in L1 global shared memory cache hits on the Squall module. Until the required communication exceeds that bound, the matrix multiply algorithm is limited by processor FLOPs rather than our shared memory implementation. Linear speedup can be sustained up until that limit is exceeded.

Synchronization Barriers:

I tested barrier performance in order to verify the robustness of the coherency protocol. Barriers are communication intensive, especially for the type of conservative sequentially consistent protocol I had implemented. For a two node system, each barrier requires four round-trip latencies as well as other protocol processing. The round-trip latency from table 1 would suggest a barrier time of less than 200usec, about a factor of five less than what was measured.

This is mostly due to the fact that the coherency protocol used polling rather than interrupts for servicing local host global shared memory requests for the purpose of simplifying verification of the protocol. While protocol processing was being done, the PCI interface chip would use most of the Cyclone card's local bus bandwidth to continually retry the failed global shared memory read until the SP could disable it.

Our hardware's normal method for causing a retry is to time-out on the local bus, which is then occupied for a significant fraction of a microsecond. It is likely that hundreds of such retries were encountered with the PCI chip enabled during each barrier.

6 Future Work

As was shown in section 8 “Experimental Results”, StarT-jr is a reasonably high performance message passing machine. The almost 50MB/sec sustainable bandwidth of the hardware is more than that of most other PC-based machines of which we are aware. The use of cached reads and writes in message passing results in extremely low host overhead. Still, there are improvements that we intend to make in the near future.

Requiring the host processor to format each packet is contrary to one of our original goals which was to minimize host overhead in message passing; SP involvement in message passing is a bottleneck. One solution is to design hardware that will add headers to the packets as the DMA operation moves them out of host memory and into one of the transmit FIFOs. Analogous hardware would strip the headers off as the DMA operation moves the packets into host memory at the receiving node. This design work is currently under way in our group.

Much improvement can be made in the hardware support for shared memory. The chosen line size of the level one GSM cache is two four-byte words. This fine level of granularity did not prove to be necessary and so might be eliminated in future implementations in favor of a larger line size. In the meantime, the network processor is being used to experiment with replacement algorithms that work with varying replacement block sizes; nothing in the hardware precludes replacement of several lines as a single unit. The small line size only requires the update of multiple tag/control words to emulate a larger line size.

The design is not fundamentally limited to two associative cache sets. Four (or more) set associativity can be added by reprogramming the FPGAs on the Squall Module if performance degradation due to tag checking is deemed to be acceptable after further experimentation. This reprogrammability makes StarT-jr a more useful research tool.

Also, the flexibility of individual coherency strategy and interrupt policy on a line by line basis did not prove to be useful, but because the tag/control words reside in off-chip memory it was not costly. The overhead of manipulating these bits by the network processor suggests the future implementation of universally relevant register bits in the ACD to store this information.

As was described in section 4 “Detailed Hardware Description”, there are two major problems with using the current StarT-jr implementation as a distributed global shared memory machine. The fact that global shared memory writes are not quite user-transparent is the most crippling of the two. A solution to this problem would require the use of a different PCI interface chip that would allow our custom hardware to cause a host-initiated write transaction to be retried over the PCI bus. Our group is currently looking at other off-the-shelf silicon and also considering designing our own FPGA-based custom PCI interface using Verilog HDL.

The other major problem does not affect correctness but has a large performance impact. The fact that the host processor is not caching the global shared memory data is costing us orders of magnitude in performance when accessing global data. Each time a global location is requested by the processor, the host is retrieving it across the PCI bus.

The PCI specification does allow for maintaining cache coherency across the bus,

and so our own PCI interface or a different off-the-shelf part might solve this problem (we must also identify a host machine whose memory controller/PCI bridge will cache I/O locations). If we can move the coherency boundary so as to include the processor's on-chip cache we will likely be able to outperform software-only systems and be competitive with other more costly hardware solutions (see section 3 - "Related Work").

We plan to use StarT-jr to connect various hosts together in a heterogeneous system. Any machine that uses PCI as its I/O bus is a candidate for inclusion. Sometime this summer we will connect Sun SMPs, Digital SMPs, and Intel Pentium machines using the Arctic Switch Fabric and a descendant of the existing StarT-jr hardware.

References

- [1] James C. Hoe and Mike Ehrlich. StarT-JR: A Parallel System from Commodity Technology. CSG-Memo-384, Proceedings of the 7th Transputer/Occam International Conference, Tokyo, Japan, November '96.
- [2] Boon S. Ang and Derek Chiou. StarT-Voyager. The Proceedings of the MIT Student Workshop for Scalable Computing, 1996.
- [3] G. Andrew Boughton. Arctic Routing Chip. MIT Laboratory for Computer Science, CSG Memo 373, April, 1995, In Proceedings of the 1994 University of Washington Parallel Computer Routing and Communication Workshop, May, 1994.
- [4] K. Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Yale University, Sept. '86. (Also as YALE/DCS/RR-492)
- [5] P.Keleher, A.L.Cox,S.Dwarkadas, and W.Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In Proceedings of the 1994 Winter Usenix Conference, January 1994.
- [6] Daniel J. Scales,Kourosh Gharachorloo and Chandramohan. Shasta: A Low Overhead, Software Only Approach for Supporting Fine-Grain Shared Memory. In the Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge MA, October '96.
- [7] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). IEEE Micro, Feb. 1995.
- [8] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In Proceedings of the 19th International Symposium on Computer Architecture, pages 92-103, Gold Coast, Australia, May 1992.
- [9] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiawicz, B. H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995.
- [10] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. Appears in the proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI 1995.

3233-11