Fall 1997

Design of a communications interface for a very high performance computer

Jesse Boyd Zydallis
New Jersey Institute of Technology

ABSTRACT

DESIGN OF A COMMUNICATIONS INTERFACE FOR A VERY HIGH PERFORMANCE COMPUTER

by
Jesse Boyd Zydallis

PetaFLOPS computing power is the newest goal of Federal Government agencies, in the increasingly active supercomputer field. To obtain this performance goal by the year 2007, sophisticated parallel processing designs are required. To effectively create network interfaces/routers for interprocessor communications in such computer systems, it requires optimal hardware and software codesigns.

An interface is presented for the NJIT New Millennium Computing Point Design, a system that targets 100 TeraFLOPS performance by the year 2005. The router handles store-and-forward switching and wormhole routing for the system.

DESIGN OF A COMMUNICATIONS INTERFACE FOR A VERY HIGH PERFORMANCE COMPUTER

by
Jesse Boyd Zydallis

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

January 1998

APPROVAL PAGE

DESIGN OF A COMMUNICATIONS INTERFACE FOR A VERY HIGH PERFORMANCE COMPUTER

Jesse Boyd Zydallis

Dr. Sotirios G. Ziavras, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering, and Computer and Information Science, NJIT

Dr. John D. Carpinelli, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, and Computer and Information Science, NJIT

Dr. Edwin Hou, Committee Member — Date
Associate Director for Computer Engineering Program and Associate Professor of Electrical and Computer Engineering and Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author:	Jesse Boyd Zydallis
Degree:	Master of Science
Date:	January 1998

Undergraduate and Graduate Education:

- Masters of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1998

- Bache!or of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1997

Major: Computer Engineering

This thesis is dedicated to my wife, my family, and the
United States Air Force who have all made this opportunity possible for me.

ACKNOWLEDGMENT

The author wishes to express his sincere thanks to his advisor, Dr. Sotirios G. Ziavras. His guidance, support, and insight have made this research possible. Special thanks to the United States Air Force, the Aerospace Education Foundation, and the Air Force Association who have assisted me throughout the past year. Additional thanks are given to Dr. John D. Carpinelli, and Dr. Edwin Hou for participating in my committee.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Architectures for High Performance Computing

The objective of this thesis is to present the design of an interprocessor communication interface that, in the near future, could aid in the creation of a near PetaFLOPS scalable parallel computer (i.e., capable of performing close to 10^{15} floating-point operations per second). This design will cover the details of the interface to be placed among the processor, communication coprocessor, and output buffers of this parallel computer. This design includes both the software and hardware aspect of the interface that allows it to be scalable and easily modified in the future.

The increasing demand for high performance scalable computer systems is the motive for this thesis. As time progresses, so does the need for a larger amount of computational power. Even today, we have applications that may take weeks or longer to run on the fastest parallel machines in the world. With no end in sight for the desire to increase computational power, new parallel designs are needed to allow computer systems to surpass current obstacles in speed and performance.

Near PetaFLOPS and more performance is a necessity for many of the applications that scientists and the government are currently pursuing. A few of the leading applications are weather modeling, physical phenomena simulation, aerodynamic simulation and testing, and nuclear weapons testing. Also encryption techniques, neural network simulation, simulation of computer chips, structural analysis, real time image processing, robotics and artificial intelligence, among others, all require more computational power in order to advance to the next level above where we are currently at. In November 1996, the fastest parallel computer became operational in Sandia Labs, New Mexico. This computer was designed by Intel and uses 9,072 of the Intel P6 Pentium Pro processors to obtain its 1.8 TeraFLOPS peak performance [4, 5].

A PetaFLOPS machine appears to be something of the distant future considering that it would operate at approximately 1000 times the speed of the current fastest parallel computer in the world. Previous to Intel's release of the 1.8 TeraFLOPS Paragon supercomputer, the world record for computing speed was achieved by a team of scientists from Sandia and Intel Corp. They were able to link two of the largest Intel Paragon supercomputers, in the world, together to achieve a speed of 281 GigaFLOPS [4, 5].

A major problem encountered when attempting to develop a near PetaFLOPS parallel computer is the difficulty in developing low-complexity, high-bisection bandwidth, and low-latency interconnection networks to connect the thousands of processors that are required to achieve the performance measures sought. There are several different architectures, each of them with its own problems, which could be used in the design of the system.

To fully understand this thesis, there are several terms that must be defined. The number of edges or links that are connected to a specific node will be referred to as the *node degree d*. The node degree is an important parameter since this basically indicates how many lines enter and leave each node, thereby directly affecting the cost of the system. However, to get good performance, the *diameter* of the interconnection network should be as small a number as possible. The maximum shortest path between two nodes is referred to as the *diameter* of the network D. This is essentially the maximum number of hops between two nodes. It is obvious that a small node degree often results in a large diameter. The *bisection width* of an interconnection network is obtained by cutting a particular network into two equal halves. The minimum number of edges that are severed by this cut is the channel bisection width b. Evaluating these three parameters is very important in determining a good solution to a problem through the use of a number of different architectures [6].

1.1.1 Mesh and Torus Architectures

The regular mesh and other versions of mesh architectures have been widely used in the past. The physical qualities of the mesh are displayed in Figure 1. In the general case, a symmetric k-dimensional mesh with n nodes in each dimension contains $N = n^k$ nodes and has a diameter of $D = k*(n - 1)$ and a degree of $2*k$ (excluding the outer rows and columns of nodes in the mesh). A 2-dimensional mesh is actually an n x n array; where each processor, except for the ones on boundaries, has four neighbors.

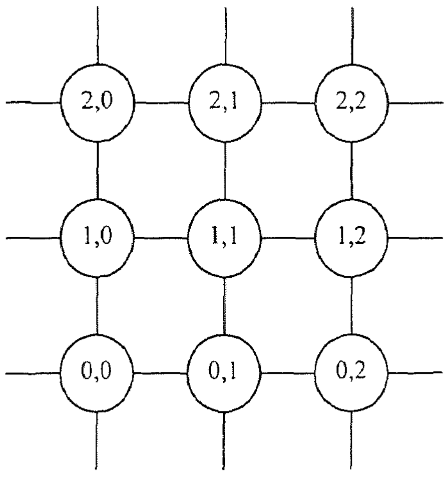

Figure 1 Mesh Architecture (3-D)

Assume a 9 x 9 mesh, where the processor addresses are P(i,j), for $0 \leq i \leq 8$ and $0 \leq j \leq 8$, where i and j represent the row and column number, respectively. The problem arises when processor P(0,0) wants to communicate with processor P(0,8), for example. There is no direct connection from processor P(0,0) to P(0,8); instead the message must travel from processor P(0,0) through processor P(0,1), and continue passing through processors until it reaches the destination processor P(0,8). This problem exists both in the rows and in the columns, leading to bottlenecks on transmission lines. The worst case scenario occurs when the two communicating processors are at the opposite corners of the mesh. This situation occurs when P(0,0) wants to send a message to P(8,8). This communication would require 16 hops in the best case scenario. Since the goal here is to use an architecture that provides the smallest delay time between message transmissions, the mesh is not optimal and will not be used.

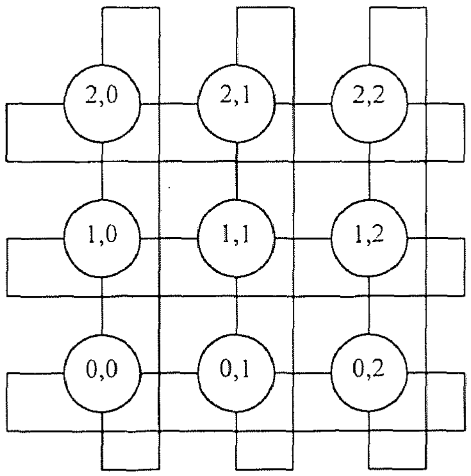

Figure 2 Torus

A variation of the mesh, namely the torus in Figure 2, is another example of a possible architecture to use for parallel computing. The torus can be viewed as a mesh architecture with an increased number of communication lines. The structure of the torus is the same as that of the mesh, with the exception that each of the end nodes in all of the rows and all of the columns connects to each of the beginning nodes in their corresponding rows or columns. The torus also keeps a constant node degree of four for all nodes due to the wrap around connections. For example the first row in the 9 x 9 torus contains nine nodes, P(0,0) to P(0,8), where all are connected in a ring configuration in the same dimension. P(0,0) connects to both P(0,1) and P(0,8). This will also be true in the columns. In an 9 x 9 torus, node P(8,0) will connect to nodes P(7,0), and P(0,0). Essentially each node connects to its two immediate neighbors in each dimension.

The torus is not the preferred architecture for our system either. The problem encountered with the torus is that its diameter is large. More specifically, the diameter of the k-dimensional torus is $k*(n/2)$, while the diameter of the corresponding mesh is $k*(n-1)$.

1.1.2 Hypercube Architecture

The hypercube is also referred to as the binary n-cube architecture since there are a total of 2^n nodes along n dimensions of a cubic structure. The node degree is $d = n$ and so is the diameter $D = n$. Each node is connected to exactly n neighbors, as can be seen in Figure 3. A property of the n-cube is that the n-bit binary addresses of any two neighboring nodes differ by only one bit position.

A simple routing technique would be to take the exclusive-OR of the source address and the destination address. This will provide the dimensions to be traversed for the message to reach the destination. This way, the intermediate nodes that the message must pass through can be determined [7].

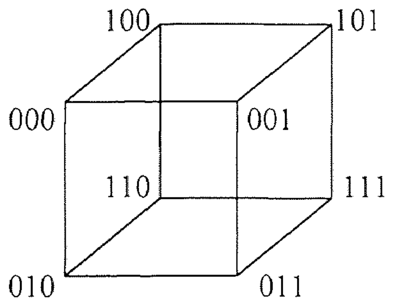

Figure 3 Binary Hypercube

Considering the physical properties of the hypercube, especially the fact that the node degree d increases linearly with the number of nodes, the hypercube is not viewed as a good architecture for scalability. As the number of dimensions increases, so does the wiring complexity of the n-cube. In real situations with a large number of nodes, the complexity becomes so great that it is not feasible to implement the n-cube.

1.1.3 Generalized Hypercube Architecture

The generalized hypercube is a modified form of the binary hypercube [1]. The symmetric n-dimensional, k-ary, generalized hypercube GH(n,k) has N = k^n nodes, with each one labeled by a n-digit number in radix-k notation. Each node is connected to *n*(k-1)* other nodes, where all differ in just one of the n digits in the node's binary label. The diameter is only n in the GH architecture. Any two given nodes in a given dimension of the hypercube are replaced by a fully-connected system with k nodes to obtain the GH(n,k). Figure 4 shows the GH(2,4). Because of its very high complexity, the generalized hypercube has not been implemented in a large scale.

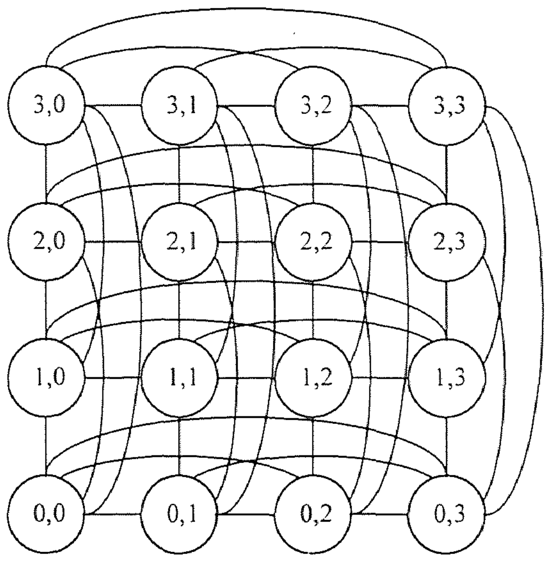

Figure 4 Generalized Hypercube GH(2,4)

1.2 Communications Operations

There are three important communications operations that are relevant to the work being done here. These communications operations fall into two major groups, broadcasting and multicasting. The broadcasting operations attempt to send the same message to one or all destination nodes, thereby giving us the one-to-one broadcast and one-to-all broadcast. The multicast operation is just a case of a limited one-to-all broadcast. The multicast operation attempts to send the same message to specific destination nodes but not all of the destination nodes. Essentially this is a subset of all of the nodes in the system and therefore, it can be considered a limited broadcast.

1.2.1 One-to-One Broadcasting

One-to-one broadcasting is the process of sending a message from a source node to a destination node. Since this case has only one destination, that greatly simplifies the algorithm or hardware that must handle it. Since the respective hardware is simple in nature, we should also expect the communication process to be rather quick. The one-to-one case must not be omitted since frequently a node needs to communicate with only one other node. This communication is done very quickly so as not to waste precious processor cycles.

1.2.2 One-to-All Broadcasting

In this procedure, a single node wants all of the other nodes in the system to receive the same copy of a particular message. For example, this technique is useful for updating caches of the changes in certain variables or entries in distributed shared-memory (DSM) systems.

Channel traffic and communication latency are two important factors to consider when developing a broadcasting technique. In order for the technique to be optimal, both the channel traffic and communication latency must be minimized. The channel traffic is the number of links that must be utilized in order to complete the desired broadcast. The communication latency is the longest packet transmission time in the system. These parameters affect systems differently. Store-and-forward systems react differently than wormhole routing systems to these two parameters. The actual routing techniques will be discussed later.

Another broadcast communication operation is the all-to-all broadcast. This is another important operation. The procedure presented here is to execute a one-to-all broadcast at every single node in the system at the same time instant. In this case, every node has a distinct message to send to all of the other nodes. The complexity increases greatly when attempting to complete this operation. The problem arises when multiple nodes attempt to utilize the same lines to broadcast their messages. The result is multiple collisions on the communication lines in the system. There is no easy solution to this problem but one proposed optimal solution for the generalized hypercube appears in [1].

1.2.3 Multicasting

Multicasting is another important communication operation that appears in many parallel algorithms. This operation can be viewed as in between the one-to-one broadcast and the one-to-all broadcast operations. The multicast operation takes place when a single node has a single message to send to multiple destination nodes in the system, but not all of the n nodes in the system. This operation is identical to the one-to-all multicast if all processors become destinations. This is the worst case for the multicast operation since it has the same complexity as the one-to-all broadcast.

Multicasting is very important because, otherwise, a node would be forced to send multiple messages using the one-to-one broadcast operation. This situation would not yield the best case transmission time. A multicasting algorithm is also present in [1].

1.3 Packet Switching Techniques

Routing is one of the most important areas in parallel processing. The main reason is that the router(s) must efficiently handle all of the messages that are transmitted across the system. These routers will determine the path that the message will take either dynamically, as the messages arrive, or statically by hardwiring specific schemes into the system. The objective of the particular system being discussed will determine which routing technique will have an optimal effect on the transmission time.

Routers can be very simplistic or complex and can be implemented in software, hardware, or in hybrid form. They are one of the most important units of a parallel system. The routers and routing techniques control the bandwidth for communication and hopefully avoid bottlenecks and minimize collisions. The static, fixed, routers that were mentioned before make poor use of the available bandwidth. This is caused by their inability to adapt to the changing transmission load that the system must handle. In times of low load, a static router may perform very well since there is most likely more bandwidth available than is currently needed. In the case of a parallel system, the goal is to have all of the nodes operating 100 % of the time thereby leading to a greater magnitude of messages being transmitted across the system to keep up this optimal performance with no down time. Since the number of communications would be very large when the system is operating at its peak, a static algorithm would almost certainly fail here. The load would be greater than the static router can handle, and this would lead to multiple collisions and retransmissions.

A dynamic router would be able to better "control" the load on the system by allocating all of the bandwidth to few transmissions during low loads and lesser amounts of the bandwidth to a large number of transmissions during peak load times. The high load times occur when there is a high load on the entire system or even just on a small section of the system. A high load on a small section of the system can drastically slow the communication time down, if that is the area where the majority of messages are trying to pass through. The dynamic router would then reroute some of the messages to a path that may incur more hops but cuts down the total transmission time. The total transmission time would be lower on these other paths if more of the bandwidth could be allocated to the messages passing through these areas. The dynamic router will not be collision free, but instead will better utilize the bandwidth to lead to fewer collisions. In a massively parallel system, fewer collisions will increase the performance of the system [8].

One aspect of the dynamic router is the ability to change the path due to a high load on the entire system or parts of the system, but another important reason to change the path is when a link or node is faulty. The router must have the ability to notice the bad link and/or node and accordingly adjust all of the transmissions that would normally have passed through that area. That is not the end though, the router must also be able to notice when the link goes back up so that communications can resume across that area of the system. If the router does not reincorporate the link when it becomes available again, then the system cannot operate at its optimal potential.

1.3.1 Store-and-Forward Routing

Long messages are divided into fixed-sized packets in parallel systems, and the packets are routed individually. Without loss of generality, we assume that each message is one packet long. One widely used packet switching method is the store-and-forward method. The router uses this communication method when one node wants to communicate with another node in the system. The source node, Ps, sends a message to the destination node, Pd, but the message may have to pass through one or more intermediate nodes, including, say, Pi. Ps sends the message to the intermediate node Pi. Pi receives the whole message and then sends it to the next node on the path or to Pd, if Pd is the next node on the path. If there were more intermediate nodes to receive the message, store-and-forward is used for all of them until it reaches Pd[9].

In the store-and-forward communication technique, the latency time may be significant. The latency time, for a transfer, is defined as the amount of time required for the message to reach its destination. The latency has a linear dependence on the number of hops that a message makes between the source and the destination nodes. The message length also has the same relation with the latency.

A necessary part of the message is the message header. The router will use the header to determine where to route the current message. Typically the message header will contain the source and destination node information. The message length should not be too long so as not to tie up a node for a long period of time. We assume the all-port model here, where a node has the ability to send or receive messages through all of its communication ports in the same time instant. If we further assume a bi-directional channel, then this means that a node can send and receive on the same line during each time step.

1.3.2 Wormhole Routing

Another important routing method is that of wormhole routing. Wormhole routing accomplishes its objective by using a pipelining method of transmission involving flits. All packets are divided into flits (flow control digits). A message header is still necessary but not in each flit. The first flit normally serves as the header and opens the path for the remaining flits in the packet to follow.

Assume that a source node sends a packet to an intermediate node. The first flit contains the complete message header so that the intermediate node knows where to route the particular flit and the others, if any, that follow. If there are no more flits, then the flit eventually makes its way to the destination in a similar way to the store-and-forward system. In the case where there are flits that follow the initial flit from the same source node, then these flits do not need the complete message header, as the header opens the path for them.

An intermediate node receives the first flit and then routes all of the following flits of this message along the same path as the first flit. The intermediate nodes do not store each of the flits as they arrive, if there is no channel contention. This fact makes the latency of wormhole routing highly independent of the distance between the source and destination nodes. Because of the pipelined message transmission, wormhole routing is a very efficient method.

The disadvantage is that during the transmission phase, of a message packet, intermediate channels cannot be interrupted. This means that the channels are "dedicated" to the current message transmission so as not to lose any of the flits. Any other messages or flits originating at different nodes that must use some of the channel(s) being used by the current message transmission must wait until that transmission is done using the intermediate nodes.

1.4 Limitations of Current Designs

Current supercomputer designs are limited by a number of factors. Currently, there is great difficulty in developing low complexity, high-bisection bandwidth, and low latency interconnection networks to connect thousands of processors and memory systems. Many different designs for interconnection networks, to obtain the previous attributes, have been proposed. These include the mesh, enhanced mesh, fat tree, binary hypercube, generalized hypercube, and many more.

The binary hypercube dominated the field in the late 1980's and early 90's due to its low diameter and ability to emulate many topologies that are used in the development of algorithms. The limiting factor here is in the VLSI area. Hypercubes often have a very high VLSI wiring complexity due to the large increase in the number of communication channels required when there is an increase in the number of nodes (processors) in the system. This increase in the wiring complexity questions the scalability of the hypercube architecture. The high VLSI complexity prevents the hypercube from being used as a powerful, massively parallel system. The advantages of the hypercube encouraged others to develop modified forms of the hypercube, with lower complexity, which kept some of the advantageous features.

One of the greatest limiting factors is in the wiring of the system. This is especially even more dramatic in the generalized hypercube case where any node connects to any other node that differs in just one dimension of the node's label.

Essentially the wires connect each node to every other node along the same dimension, for all dimensions. Our current VLSI technology limits the number of nodes that we can connect together in this fashion in a single chip.

Current approaches to scalability in parallel processing do not achieve a good performance level when a comparison is made with the hypercube structure. The current use of bounded-degree networks, such as meshes and k-ary, n-cubes with a low degree of connection result in large diameters, and average internode distances, and small bisection bandwidth. These properties result in the secondary performance of the systems in relation to the hypercube.

The generalized hypercube would increase the performance to a better level than hypercubes, but its VLSI complexity prevents it from achieving this currently. The generalized hypercube implements a fully connected system with k nodes in each dimension, contrary to the binary hypercube which can only implement two nodes per dimension. The increased complexity and VLSI cost do however increase the performance of the generalized hypercube greatly. This increase permits optimum emulation of hypercubes and k-ary, n-cubes [2].

Distributed shared memory (DSM) systems play a major role in parallel processing. DSM systems are already present in the massively parallel processing area. These systems add versatility by incorporating the message-passing and shared memory systems simultaneously. However, limitations with current designs appear in the memory access area. Processor speeds have increased much faster than memory and interconnection network speeds. This brings about the need for advanced memory systems that have the ability to hide their latency times. Such systems could achieve this through advanced pipelining, prefetching, cache coherence, multiple contexts, and relaxed memory consistency [6]. The good properties of the generalized hypercube could mitigate these problems.

1.5 Motivations and Objectives

The main objective of this thesis is to develop and evaluate an interface for a scalable, high performance parallel computer that could be feasible in less than ten years and be capable of obtaining 100 TeraFLOPS performance. This goal can be further expanded to develop a system that will be scalable to obtain PetaFLOPS performance. A specific design, namely the New Millennium Computing NJIT Point Design, is assumed here; this design would require advanced, yet readily available, electronic and optical technologies. The system's scalability is a major concern and must be addressed throughout the design phase. The scalability factor would ensure that the lifetime of the system would extend into the next decade and beyond. The system must remain feasible and cost effective based off of the performance objectives.

A goal of this work would be to persuade other scientists and engineers to adopt the outlook that PetaFLOPS computational power is something of the near future and not of the distant future. This would hopefully bring about an increased awareness and interest in massively parallel computer systems. Further more, to increase the research done in this field and possibly bring about new ideas and techniques for handling current difficulties. This work could also fuel interest in generalized hypercubes, possibly increasing the popularity of the architecture. This interest could even lead to new VLSI research that would open the door to using high complexity generalized hypercubes in more systems.

The main objective is to create an interface for a high performance computer that implements a generalized hypercube. This objective could lead to new innovative designs being created and further inspire others in the parallel processing field to build upon these designs. These designs, or subsequent ones, could be used to further increase the performance of high performance computer systems.

CHAPTER 2

THE NJIT POINT DESIGN

2.1 The Architecture

The architecture of the NJIT point design uses some innovative approaches to obtaining the 100 TeraFLOPS performance objective by the year 2005 [2,3]. Based upon the SIA projections, commodity microprocessors will be capable of 10 GFLOPS in the year 2005 which would require at least 10,000 processors in the system to obtain the 100 TeraFLOPS performance objective. For a system of this magnitude to be viable, its physical volume must be relatively small, and its communication and I/O capabilities should match the speed of its computational engine within a few orders of magnitude. The wiring complexity is dramatically reduced by the use of free-space optical technologies. This optical technology would be used in the implementation of most of the communication channels [2,3].

This design uses the free-space optics to produce a 1-D fully connected, scalable building block (BB). The complete system will be a 2-D configuration of 8-processor cards, with 40 cards in each dimension, for the implementation of a 2-D 40 x 40 generalized hypercube of 8-processor cards. This would yield a total of (40 cards) * (40 cards) * (8 processors per card) or 12,800 processors (yielding a peak performance of 128.00 TeraFLOPS). The BB is a fully connected system of 320 processors. This is obtained from the (8 processors per card) * (40 cards per dimension). The 8 processors on each card are fully connected via an electronic crossbar network. The crossbar network was chosen due to its high efficiency with a low number of nodes. This crossbar works in conjunction with the optical interface to allow all of the 8 processors to communicate simultaneously in inter-card data transfers without any loss in performance. All of the specified data transfers are across bit-parallel communication channels.

The system referred to here will perform far better than any interconnection network that has ever been built for massively parallel processing. The small diameter obtained with this system, of two, the large bisection width, and the high-speed network properties allow the advanced cache system to be a viable option for implementation. In systems that would obtain similar performance objectives, these tasks would not be as viable an option. The complete distributed shared memory system design supports scalability and simplicity in the mapping of application tasks to the system [2,3].

Other designs have limited bandwidth and substantial latencies that result in unpredictable performance under changing loads. This design differs from others in its efficient, uniform interconnection of resources in the system. These qualities allow the system to be more predictable in its performance measurements and allow for easier development of algorithms [2,3].

An intelligent high performance decoding system has been designed for directing the data to/from the communications coprocessor and to/from the correct set of optical detectors. This design will include hardware and software support at different levels of the memory hierarchy. This support will allow for performance monitoring and data reassignments that can be used by the operating system.

For this system to have an edge above other systems in the parallel-processing field, it must provide a way to cut the cost on certain key operations. The most costly operations are those that must be repeated time and time again. These operations waste bandwidth as well as valuable communication cycles. Some examples of these operations are the previously mentioned multicast, and broadcast operations, as well as reduction using associative operators, prefix computations, and barrier synchronization. The operations that are not as critical, since they do not occur with the same frequency as the previous operations, are the one-to-all personalized and the single node gather operations. These operations can be carried out efficiently throughout this design. The key to cutting the cycles for the costly operations will appear later [2,3].

2.2 Communications Interface: A First Look

The interprocessor communications interface is one of the most significant pieces of hardware to design. This interface must be powerful and efficient in order to obtain the performance power that is sought. The communications interface can bottleneck the entire system very quickly since communication is a high priority item in parallel machines and is executed frequently. As mentioned earlier, the goal of a parallel system is to have all nodes operating close to 100 % of the time. This leads to an optimal system performance. If all of the nodes are operating at 100 % of their possible performance, then there will also be a large number of communication messages traveling on the system. This would be necessary since a parallel system would divide its jobs between multiple nodes for a faster completion time. All of this essentially means that the communications interface must be optimal in order to allow the whole system to perform optimally and not to be a bottleneck.

A first look at the communications interface requires a full understanding of the NJIT Point Design. The system is based upon a GH(2,40) generalized hypercube. Each of the nodes in the x dimension, which will be referred to as dimension 0, and each of the nodes in the y dimension, which will be referred to as dimension 1, contain eight processors. These eight processors are connected on the card via a crossbar network. Each processor is also connected to its own dedicated coprocessor. This yields a total of 40 x 40 x 8 = 12,800 processors and coprocessors.

Each coprocessor is divided into ten ports. Five ports of the coprocessor are allocated for each dimension, with ports 0 to 4 allocated for dimension 0 and ports 5 to 9 allocated for dimension 1. Each port connects to 8 destination cards in the specified dimension. The coprocessor is divided into ports to increase the speed of the routing that must take place. Internal to the coprocessor, the algorithm creates the initial message and then outputs the message through one or multiples of the ten ports. Each of the ten ports can therefore route a message to a maximum of eight destination cards.

Table 1 Coprocessor Port Allocation

Port Number	Destination Dimension	Destination Cards
0	0	0,1,2,3,4,5,6,7
1	0	8,9,10,11,12,13,14,15
2	0	16,17,18,19,20,21,22,23
3	0	24,25,26,27,28,29,30,31
4	0	32,33,34,35,36,37,38,39
5	1	0,1,2,3,4,5,6,7
6	1	8,9,10,11,12,13,14,15
7	1	16,17,18,19,20,21,22,23
8	1	24,25,26,27,28,29,30,31
9	1	32,33,34,35,36,37,38,39

In the coprocessor, ports 0 to 4 route the messages to destination cards 0 to 39 in dimension 0 and ports 5 to 9 route the messages to destination cards 0 to 39 in dimension

1, as illustrated in Table 1. Upon leaving the coprocessor port(s), the message(s) then encounter the 3x8 decoder(s), which route the message(s) to the correct destination processor queue, as illustrated in Figure 5.

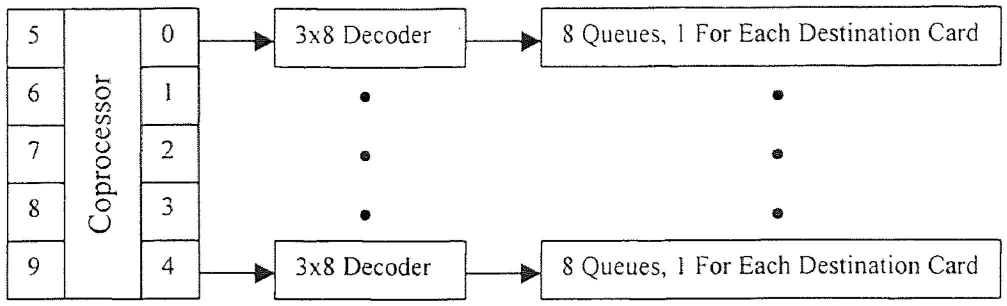
5
6
7
8
9
Coprocessor
0
1
2
3
4
3x8 Decoder
3x8 Decoder
8 Queues, 1 For Each Destination Card
8 Queues, 1 For Each Destination Card

Figure 5 Initial Communications Interface Design

Each node contains eight processors, each of these eight processors has its own coprocessor and each of the coprocessors connects to a total of 79 queues. Any specific queue directly connects to its corresponding destination card. There is one queue for each of the 40 destination cards in the same 0 dimension as the source card and a queue for each of the 40 destination cards in the same 1 dimension as the source card. These queues buffer the messages that are ready to be sent to the destination cards until the processor is able to output the messages. The actual time that the message will wait is dependent upon the load on the system, and the priorities of the messages.

We assume the all port model for the queues, which means that the all of the queues can both receive messages and output messages at the same time instant. The queues follow a FIFO (First In First Out) queuing format. Any specific processor receives messages, where it is the recipient, through the queue that is allocated for itself out of the 79 queues. This queue receives inputs from other processors and itself and outputs directly to the processor.

2.3 Communications Interface Requirements

The requirements for the communications interface are important since they affect the performance of the entire system. The interface must have the ability to output messages to all of the destination cards at the same time instant. This is accomplished through the design of the coprocessor and the queuing system. All of the queues have direct links to all of the destination cards in the same 0 and 1 dimension as the source processor. This is illustrated through the generalized hypercube (Figure 4). Since the queues are all linked by a separate, dedicated line, they all have the ability to transmit at the same time instant. Another important feature is the separate ports of the coprocessor. The separate ports allow the coprocessor to output the same message to all of the queues, or more importantly, to output a different message to each of the ten output ports. This functionality greatly increases the performance of the system.

Other requirements are that the processors can receive and transmit messages at the same time instant. This is accomplished through the all port model. Looking deeper into the coprocessor, a limit must be set on the size of the messages in bits. All messages to be transmitted will be 128-bits long. This limit is set to avoid bottlenecks and overloading the system. The 128-bit limit provides an optimum message length. The message contains 64-bits for the data and 64-bits for the message header. The details of the message creation will be discussed in the next chapter.

CHAPTER 3

DETAILED STUDY OF THE COMMUNICATIONS INTERFACE

3.1 Refinement of the Coarse Design

The original design specifications required a GH(2,40) generalized hypercube, with each node representing a card. Connected to each card would be eight processors, eight coprocessors, a number of decoders, and 79 queues per processor along with memory elements. These specifications would result in a grand total of 40 x 40 = 1,600 cards, 1,600 cards x 8 processors per card = 12,800 processors and coprocessors, and 12,800 processors x 79 queues = 1,011,200 destination card queues. This is a good design but modifications can be made to improve its performance.

The problem encountered with the current design appears when the coprocessor attempts to send a message to more than one of the destination cards within the same port, as is the case in a multicast or a broadcast to all nodes. The coprocessor opens the communication port and sends the message. The 3x8 decoder in Figure 5, however, will decode the three control bits and only be able to send the message to one of the eight destination queues. This will cause a reduction in the number of messages that can be transmitted from the communications interface per unit of time. This will impact the performance of the system considering that multicasting and broadcasting operations are very common in parallel systems and would have to be completed in multiple cycles.

A simple solution to the decoder problem would be to increase the speed of the decoder to compensate for the inability to perform multicasting and broadcasting operations in a single cycle. While a one-to-one broadcast operation within each port can be completed in a single clock cycle, the multicasting and all-to-all broadcasting must also have the ability to be completed in one cycle. The one-to-one broadcast operation occurs when any port wants to output a message to only one of the eight possible destination queues. While increasing the decoder speed will work, a change to the hardware design would provide a better solution to the problem at hand.

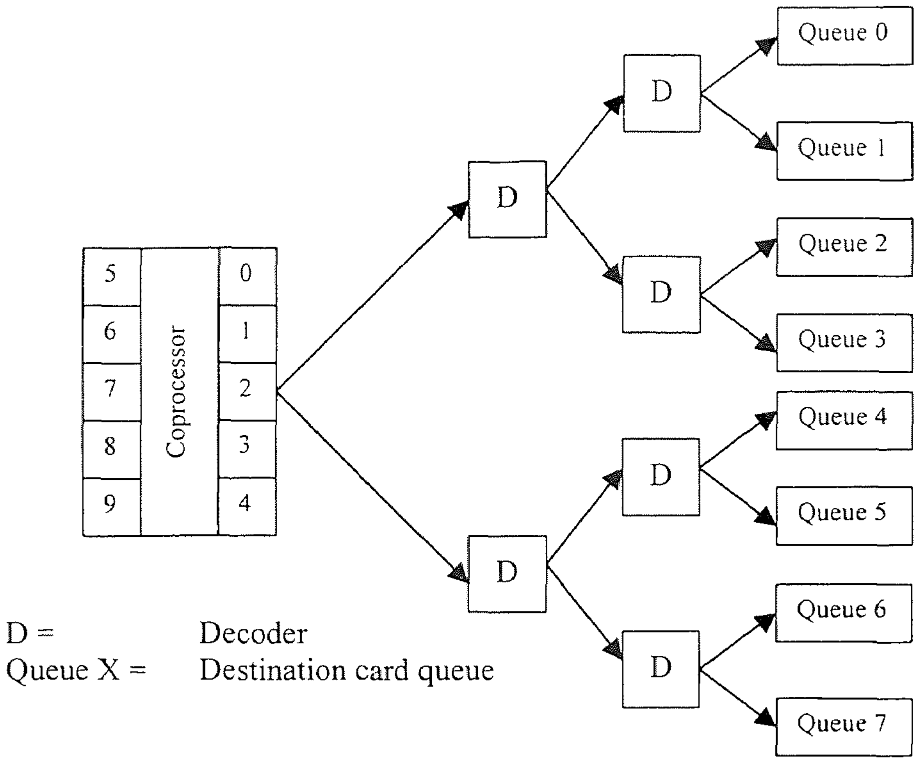

Figure 6 Binary Tree Communications Interface for Each Port

To allow all of the broadcast and multicast operations to be completed in one clock cycle, modifications must be made to the original communications interface design. Figure 6 illustrates one solution to the problem. This design uses the binary tree structure, emanating from each port of the coprocessor, to complete the broadcast and multicast operations. Each of the D boxes, in Figure 6, represent simple decoders and the queues in the leaves of the tree represent the destination queues for the messages. This design requires additional hardware components, and contains more levels than the original 3x8-decoder solution, present in Figure 5. Even though there in an increase in the number of hardware components and levels, the decoders in Figure 6 are simpler and can be operated at higher speeds. The ability to operate the simple decoders at higher speeds allows this design to outperform the previous design even when that one is operating at very high speeds. The difference is that the solution in Figure 6 will allow multicasting and all-to-all broadcasting at the higher operational speeds, whereas the solution in Figure 5 does not allow these operations at the higher speeds, but over a period of time completes these functions. An overall better performance is achieved with the binary tree structure.

In Figure 6, each port of the coprocessor is connected to six decoders and eight destination message queues arranged in a binary tree structure to improve the delay time. At each level of the binary tree the decoders use different bits of the message header to determine if the lower levels of the tree should receive the message. If the message should be delivered to the next level, then the decoders allow the message to be passed through on the correct output line. Otherwise the decoders do not pass the message on to the next level of the tree.

Initially this design was going to incorporate a number of codes that the decoders would receive to determine which destination cards would receive the messages. The codes would specify which subset of destination cards would receive the messages. An example of this solution would be that one code would specify all even numbered destinations would receive the message, another would specify all destinations that are a power of 2 would receive the message. This is a valid solution and has the advantage that fewer bits could be used to specify the destination cards of the message. The main disadvantage is that this method does not cover all of the possible destination patterns. Many of the multicasting possibilities would not be covered and would force multiple messages to be sent, thereby lowering the throughput of the system. In reality there are $2^8 = 256$ different combinations of destinations that could receive the message. The final design uses all eight bits to represent the destination card, but this method of codes will be used later for a different section of the message header. The use of these codes will be described in Section 3.2.2.

Since each of the ports output the message to a binary tree, this allows for each port to implement the multicast and all-to-all broadcast operations. In either of these operations, the decoders determine which of their children should receive the message as they receive it. The binary tree allows for just one of the eight destination queues, all eight destination queues, or any subset of the eight destination queues to receive the same message in one clock cycle. One objective for the communications interface has been achieved.

3.2 Message Formats

One communications interface objective is to process the messages as fast as possible and output them to the queues. This is accomplished through the design of the message header. The communication latency is a very important issue here. The message length must be chosen so as not to slow the system down due to their long lengths, but at the same time to be long enough to provide a good transfer rate. In both the store-and-forward routing and wormhole routing techniques, the message lengths must not be extremely long. In the store-and-forward technique, a long message length would tie up a node for a long period of time thereby cutting the performance of the system. In the wormhole routing technique, once the first flit passes through the nodes on the way to the destination node, all of the intermediate nodes are reserved for that message transmission. If the message length is long, then the whole path cannot be used by other messages until the first is completely sent. This is another reason for finding an optimal message length.

The initial requirements for the messages specify that they are all to be 128-bits long. This was the length determined to provide the best usage of the available bandwidth. The message is split into a 64-bit header and a 64-bit data field, as shown in Figure 7. The 64-bits of the header field will be further broken down to describe what each bit's function is.

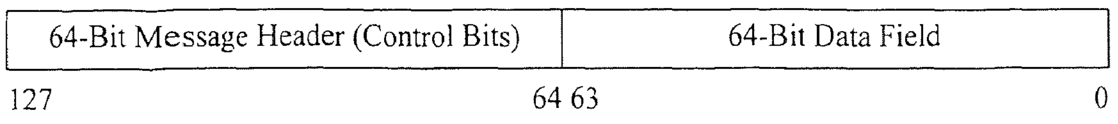

Figure 7 Generic Message Format

The message header will provide a number of different routing functions. First off, the decoders of the binary tree communications interface will use some of the bits to determine which of the queues will receive the message. In addition, some of the bits of the header will be used to specify the source and destination cards and processors. This is not an easy design to deal with. The reason is that there are a number of fields that are necessary for fault tolerance in the system but there are only a limited number of bits that can be utilized for each of these fields.

In creating the messages, there became a need for three different message formats. The reasoning behind this is that there are so many different fields in the header that are needed at different stages of the communications process that the best solution was to change the message header at each particular stage. The first message format is used by the processor to communicate with the coprocessor. This format will be referred to as the Processor Coprocessor Transfer Format (PCTF message). The next message format is used by the coprocessor to output to the binary tree interface; this will be referred to as the Coprocessor Interface Message Format (CIMF message). The last message format appears at the last level of the binary tree interface, before the messages reach the destination queues. These messages are changed by special purpose hardware inside the decoders from the CIMF message to the format that will be received by the destination processors. This last message will be referred to as the Destination Message Format (DMF message). The next section will explain the differences among these different formats.

3.2.1 Processor to Coprocessor Message

In order to understand the reasoning behind the different message formats, the definition of a CPU and a coprocessor must first be understood. The central processing unit, or CPU, is basically a scalar processor with the capabilities to compute arithmetic, and logical functions, and much more. Coprocessors are attached to the CPU and aid in completing some of the tasks that are submitted to the CPU. These tasks include, but are not limited to, floating point operations, vector processing, digital signal processing (DSP), and much more. The coprocessor is essential since it aids in the processing of tasks sent to the processor and frees up the processor for additional tasks. This thesis deals with the design of a communications coprocessor. It processes the data sent to it for communication.

The processor must pass to the coprocessor the data to be transferred to the destination, or the memory location of this data, and the address of the destination processor(s). This comprises the Processor Coprocessor Transfer Format message (PCTF message). The goal for this message format was to keep the message length small, so that the time spent communicating from the processor to the coprocessor is not extensive, but at the same time to include as many destinations as possible.

The actual format of the message is shown in Figure 8, and the descriptions of the different fields are shown in Table 2. There are a number of factors that affected this design. First, we assume that the memory is local to both the processor and coprocessor. In addition, the memory is 2-ported, which means that both the processor and coprocessor can access the memory at the same instant in time. The messages here are consistent with all of the messages in the system, in the fact that they are 128-bits long. The 64-bits of the header contain the destination processor addresses for the coprocessor to send the message to, along with other control bits that are necessary to decode the header. The last 64-bits of the message contain the data that the source processor is passing to the destination processor(s).

Due to the 128-bit limit on the message size, the PCTF message can transmit a maximum of two destinations per transmission. This may appear to be a small number of destinations per message but this message is sent at a very high frequency, thereby preventing starvation of the coprocessor. The coprocessor will not have to wait a significant amount of time for the list of destinations to arrive from the processor.

(15-bits)	(15-bits)	(30-bits)	(1-bit)	(1-bit)	(1-bit)	(1-bit)	(64-bits)
Dest0	Dest1	SMMA	C0	C1	D	MD	Data
127 113	112 98	97 68	67	66	65	64	63 0

Figure 8 Processor Coprocessor Transfer Format Message (PCTF message)

Table 2 PCTF Field Descriptions

Field	Description
DestX	The destination processors of the messages (15-bits each dest, 30-bits total).
SMMA	Memory address specifying the beginning location of the data in the destination processor's memory space (30-bits).
C0	Specifies if the DP field in Dest0 is a listing of the actual destination processor or if it is a code used to denote multiple processors (1-bit).
C1	Specifies if the DP field in Dest1 is a listing of the actual destination processor or if it is a code used to denote multiple processors (1-bit).
D	Specifies if DMA Control is being used (1, if yes; 0, otherwise) (1-bit).
MD	More destination field; specifies if there are more destinations for the current message in the next source processor's memory location (1-bit).

The D field specifies if DMA control is being used. In this design, the coprocessor also acts as the DMA controller. A DMA controller allows other devices to directly access the memory without having to ask the CPU for permission and without being limited by the speed of the CPU. This was stated earlier in the definition of the 2-ported memory. In the case where DMA control is utilized, the PCTF message contains a 1 in the D field. DMA control is useful in the cases where either the data to be sent is over 64-bits long or when there are a large number of destinations to send to the coprocessor.

Once the coprocessor receives the message and sees the 1 in the D field, it will ignore the entire message with the exception of the SMMA field and the Data field. In DMA control, the processor will list the destinations in the memory, immediately followed by the data. By allowing the coprocessor to access those destinations directly from memory, the processor does not have to waste processing time in sending multiple messages containing destinations or a large amount of data to the coprocessor. The coprocessor will proceed to look at the SMMA field of the message to find the correct memory location and find the list of destinations followed by the data to be sent. Since this system is a distributed-shared memory system, the coprocessor can directly access the list of destinations with the memory address (SMMA).

In the other case, when we are not using DMA, the coprocessor must decode the entire PCTF message that it receives. The D field will be 0 in this case and the coprocessor will know that this is not a DMA situation. Once this is known, the coprocessor retrieves the destination(s) from the message. The DestX fields have not been described up to this point. These fields represent the destination card and processor information. They are essential for achieving the performance that is needed. The DestX field contains the 0 and 1 dimension information for the destination card and the actual destination processor on that card. These fields are further illustrated in Figure 9 and Table 3.

(6-bits)	(6-bits)	(3-bits)
DD0	DD1	DP
14 9	8 3	2 0

Figure 9 Destination Fields (DestX)

Table 3 Destination Field Descriptions

Field	Description
DD0	Destination card dimension 0 (x-axis) (6-bits).
DD1	Destination dimension 1 (y-axis) (6-bits).
DP	Destination processor(s) (3-bits).

The DD0 field represents the dimension 0 term of the destination card. As stated in Chapter 1, there are 40 cards in dimension 0 and 40 cards in dimension 1. This means that the only valid binary values for these cards will range from 000000 to 100111, or 0 to 39 in decimal. All of the values from 40 to 63 are not being used and can be allocated for other uses. In the case where only one of the DestX fields has a destination listed, the other field must contain a code representing no information in this field. The code of 111111 was placed in the DD0 field for that destination. This means that the coprocessor will look at the DD0 field of a DestX and if the code 111111 appears, then the message contains only one destination in it. The bits in the CX field are used to specify if the corresponding DestX field is using a code in its DP field. The DP field denotes the destination processor on the card with 0 and 1 dimensions of DD0 and DD1. The CX field will be described in Section 3.2.2.

In the DMA controller case, a similar method is used. In the memory location used for the source processor, the processor stores all of the destinations one after the other, following the format of Figure 9. After all of the destinations are listed in the memory, the next DestX field must specify to the coprocessor that the data will begin. The processor then stores the value 111111 in the next DD0 field. This signifies to the coprocessor that all of the destinations have been listed and the next memory location contains data. . This memory location, to the end of the memory space is allocated for data.

If more destinations exist in the non-DMA case, besides the two in the first PCTF message, there must be a method for informing the coprocessor that there are more messages to be sent. This is the case when the MD bit is set to 1. The coprocessor already has the SMMA field and the data field; therefore, those fields do not need to be sent again. The internal algorithm that the coprocessor runs specifies that all messages received after a PCTF message with the MD bit set to 1 contain destinations only, as illustrated in Figure 10. This will increase the number of destinations that can be transmitted per clock cycle. Once all of the destinations are sent to the coprocessor, the processor sends a 128-bit message containing all ones. This would normally be considered an error message since both the DD0 and DD1 fields cannot contain all ones, but this is a control message for the coprocessor. This message specifies that all of the destinations for the data previously received have been sent and any additional messages will follow the original PCTF format containing only two destinations and the other fields necessary and will contain a new set of data to broadcast or multicast.

(15-bits)	(15-bits)	(15-bits)	(15-bits)	(15-bits)	(15-bits)	(30-bits)	(6-bits)	(1-bit)	(1-bit)
Dest0	Dest1	Dest2	Dest3	Dest4	Dest5	ND	C0-C5	D	MD
127 113	112 98	97 83	82 68	67 53	52 38	37 8	7 2	1	0

Figure 10 PCTF Destinations

The messages containing only destinations use the 128-bit format in Figure 10. There are 6 DestX fields and 6 CX bits allocated for the C field. Each CX bit specifies if the corresponding Destx field is using a code or a destination processor listing; this code will be explained shortly. There are six bits allocated for the CX field since there are six possible destinations included in this message. Bits C0 through C5 are used to state if destinations Dest0 through Dest5 are using a code or not. If a code is being used, a 1 is found in the CX bit position, else a 0 is present there. There is no SMMA field in this case; instead there is a ND field. This ND field lists the total number of destinations that will receive the message so that the coprocessor can differentiate between where the destination addresses end and where the data that is stored in memory begins. The MD field is not necessarily needed here, but can be used for double-checking.

Table 4 PCTF Destinations Field Descriptions

Field	Description
DestX	The destination processors of the messages (15-bits each dest, 30-bits total).
ND	Number of destinations field that specifies the total number of destinations to be sent to the coprocessor (30-bits).
CX	Specifies if the DP field in DestX is a listing of the actual destination processor or if it is a code used to denote multiple processors (1-bit each, 6-bits total).
D	Specifies if DMA Control is being used (1, if yes; 0, otherwise) (1-bit).
MD	More destination field specifies if there are more destinations for the current message in the next source processor's memory location (1-bit).

The DMA case is more efficient when there are a large number of destinations to send to or there is a lot of data to send. In either case, the D bit is set to 1. Following this, the coprocessor proceeds to retrieve the destinations and data directly from memory. This allows the processor to utilize its processing power for other tasks.

3.2.2 Coprocessor to Interface

Once the coprocessor has received all of the destinations and data to be broadcast or multicast, it must then create the messages that will be sent to the destination processor(s). If there is only one destination card and processor, which means a one-to-one unicast, then the message format is simplistic. If, however, there are multiple destination cards or processors, the message format becomes complex. At this point the coprocessor has all of the destinations stored, and either has the data to be sent stored or has the memory location of the data stored. This is important since the messages to be sent out have to contain the list of destinations in them but can either contain the data or the address of the data stored in memory. As previously stated, that is the DMA controller case.

(6-bits)	(5)	(1)	(1)	(2)	(6-bits)	(6-bits)	(3-bits)	(2-bits)	(2-bits)	(30-bits)	(64-bits)
BC	U	C	D	CDP	DD0	DD1	DP	XDD	CDD	MA	Data
127	121			114	112	106	100	97	95	93	63 0

Figure 11 Coprocessor Interface Message Format (CIMF message)

In creating the message to be sent out of the ports of the coprocessor, there are a number of factors to consider. The one-to-one unicast message differs from the one-to-all broadcast message and both of these differ from the multicasting message. All of these messages have the same general format, of Figure 11; they just differ in the bits that are placed in the fields of the message. The messages follow the standard 128-bit length and the fields are described in Table 5.

Table 5 CIMF Field Descriptions

Field	Description
BC	Broadcasting code field used to denote an all-to-all broadcast (6-bits).
U	Unused bits (5-bits).
C	Code bit used in the DMF message of the next section (1-bit).
D	Specifies if DMA Control is being used (1, if yes; 0, otherwise) (1-bit).
CDP	Code destination processor bits used with the DP field in the DMF message of the next section (2-bits).
DD0	Destination card dimension 0 (6-bits).
DD1	Destination card dimension 1 (6-bits).
DP	Destination processor (3-bits).
XDD	Extra destination dimension bits used for multicasting (2-bits).
CDD	Control bits for the destination dimension fields (2-bits).
MA	Memory address field for the destination processor. The total memory for each processor is $2^{30} = 1$ Gword. The local memory is part of the DSM (distributed shared memory). Each word is 64 bits long. Since we are making the memory word accessible only, in actuality we have 8 times as much space per processor, which equals $2^{33} = 8$ Gbytes of DSM (30-bits).
Data	Data field of the message (64-bits).

The one-to-many broadcast message utilizes the greatest amount of bits in the CIMF message. In this case, both the BC and the U fields are not used. The DD0 and DD1 fields are both six bits long and specify the 0 and 1 dimensions of the destination card. Since these fields are only six bits long, they can only specify one of the 40 different destination cards in each dimension. This presents a problem since the multicast operation needs to send the message to multiple processors that may be on different cards.

One solution to this problem is to implement codes into the remaining unused values in the DD0 and DD1 fields. Both of these fields do not use the binary values 101000 to 111111, which are 40 to 63 in decimal. This solution would work but could not cover every case of the multicast operation. Another disadvantage is that this would require more software code in order to decode these fields. The final solution calls for the use of two additional bits to be added onto the end on one of the DD fields. These additional bits appear in the extra destination dimension bits used for multicasting, the XDD field. The purpose of these bits is to allow a total of eight bits to be allocated towards one of the two DD fields.

Our system uses the GH(2,40) generalized hypercube. This topology allows for a maximum of two hops between any two nodes. For instance, if node (0,0) wants to transmit to any other node in the system, it will take a maximum of two hops to get there. If the destination node is in either the same 0 or 1 dimension as the source, then only one hop must be made; otherwise, the message will take two hops to reach the destination.

Once eight bits are allocated to one of the two DD fields, by using the XDD field as the least significant bits of the DD field, then a multicast operation can be executed as long as all of the destinations contain the same DD0 or DD1 field. In essence, this means that a source can multicast to any card that is in the same 0 or 1 dimension as it is. This is further simplified since each coprocessor port outputs to only eight different destination card queues. Subsequently, this approach leads to a better multicasting ability than originally perceived. The message does not need to multicast to all 40-destination cards in one dimension but instead to only eight of the 40 destination cards in a single dimension. The reason behind this is that the coprocessor is divided up into ports and each port can only output messages to eight different destination card queues. An example of this is when source (0,0) wants to send a message to cards [(0,1), (0,2), (0,3), and (0,25)]. In this example, destination cards [(0,1), (0,2), and (0,3)] will all appear in the same message leaving port 0, but destination card (0,25) will leave port 3, as specified in Table 1. A portion of the actual message leaving port 0 would look like 00001110, with the most significant bit being the leftmost bit. The six most significant bits would be stored in the DD1 field while the two least significant bits would be stored in the XDD field.

With the 64-bit limit on the message header, it is not possible to complete a multicast operation if both of the destination dimensions differ from the source. The next problem to address is how to specify whether the XDD field should be added onto the end of the DD0 or the DD1 field, or if the XDD field is not used, as is the case with a one-to-one unicast. The solution to this problem can be found in the control bits for the destination dimension fields, the CDD field. Table 6 describes what the different bit combinations in the CDD field correspond to.

Table 6 CDD Field Descriptions

Bit Combination	Output Operation	Description
00	One-to-One Unicast	Send to the card specified in the DD0 and DD1 fields
01	Multicast	XDD field is added on to the end of the DD0 field since all destinations contain the same DD1 field.
10	Multicast	XDD field is added on to the end of the DD1 field since all destinations contain the same DD0 field.
11	One-to-All Broadcast	Sends the message to all destination cards in the system.

The MA field specifies the memory address that is allocated to the processor. The one bit D field specifies if DMA Control is being used. If the D bit is set to 0, this means that the data to be sent to the destinations is not more than 64-bits long and fits in the Data field of the CIMF message. If the D bit is set to 1, then DMA control is being used and the data can be found in the memory location specified by the MA field. DMA control is essential here. If the data to be sent is over 64-bits, without DMA control it could take multiple messages to send all of the data to the destination. DMA control allows the destination processor to access the DSM to obtain the data without the additional overhead of traffic on the system. When a particular processor accesses the memory, I/O lines are used instead of the lines used to send the messages to the destinations. The load on the I/O lines will not directly affect the system's ability to send and receive messages. This load will only affect the retrieval rate of data from the memory.

The CDP field has the same function as the CX bits in the PCTF message of Figures 8 and 10. The bits in the CX field are used to specify if the corresponding DestX field is using a code in its DP field. The DP field denotes the destination processor on the card with dimension 0 and dimension 1 of DD0 and DD1. If the code destination processor bit field (CDP field) is set to 00 in the CIMF message of Figure 11, then the DP field contains the destination processor and not a code.

There are eight destination processors on each card, hence there are only three bits needed to represent them in the DP field. These three bits correspond to destination processors 0 through 7 (000 through 111 in binary). This listing works fine as long as there is only one destination processor on the card. In the case where there are multiple destination processors on a single card, the source would be forced to send a separate message to each of the destination processors. This is not an optimal solution since multicasting can be done at the card level. Through the use of codes, when the CDP field is set to 01, multicasting can also be achieved at the processor level. Table 7 lists the different codes that are implemented in the DP field and which destination processors receive the message when each code is used.

Table 7 DP Code Description

Code	Description
000	All destination processors receive the message.
001	All even processors 0, 2, 4, 6 receive the message.
010	All odd processors 1, 3, 5, 7 receive the message.
011	All processors that are a power of two (0, 1, 2, 4) receive the message.
100	All face processors of a cube (0, 1, 2, 3) receive the message.
101	All face processors of a cube (4, 5, 6, 7) receive the message.
110	All face processors of a cube (0, 1, 4, 5) receive the message.
111	All face processors of a cube (2, 3, 6, 7) receive the message.

The DP code allows for the implementation of multicasting operations. Only eight codes appear in Table 8. This is because the eight combinations of processors with the highest probability of selection for a multicast operation were chosen. The coprocessor now has the ability to directly list the destination processors or implement a code. The field that allows the destination to distinguish between a direct processor listing in the DP field or a code listing is the CDP field. All of the possible entries for the CDP field are contained in Table 8.

Table 8 CDP Field Code Description

Code	Description
00	The 3-bit DP field represents the actual processor number in binary.
01	The 3-bit DP field represents a code for multiple destinations.
10	The 3-bit DP field also uses the C bit to allow 4-bits for the code and double the number of codes from eight to sixteen.
11	The 3-bit DP field is not needed since this CDP code specifies that all destination processors will receive the message.

The multicasting operation was just described but the one-to-one and one-to-all broadcasting operations are capable of execution also. In the one-to-one unicast operation, the BC field would be set to 000000 so that a code of 111111 does not accidentally appear and cause the decoders to think that a one-to-all broadcast is being executed. In the one-to-one unicast operation, the DD0, DD1, and DP fields will all contain the destination card and destination processor information. The DP field will be set to whatever the actual destination processor is, and the CDP field will be set to 00 to specify that a destination processor is listed in the DP field and not a code. In the other case of the one-to-all broadcast operation, the BC field will be set to 111111 to denote that all destination cards in each dimension should receive the message. This will also cause the DD0 and DD1 fields to be set to 111111. The 111111 code specifies that all the destination cards in that dimension should receive the message. The DP field is set to 000 to further specify that all of the destination processors should receive the message and the CDP field is set to 01 to specify that the DP field contains a code. The setting of the DP and CDP field would also allow a one-to-all broadcast to all of the destination cards but not all of the destination processors. This would be useful if only a specific processor on each card needed to receive the data.

The CIMF messages just described are created by the coprocessor and sent to the binary tree structure of the communications interface for output to the destination card queues. The decoders of the interface will look at specific bits of the message header to determine if their children (destination queues) should receive the messages. If the DD0 and DD1 fields along with the XDD and CDD fields specify that their children are destinations for the message, then they will pass the message along to the corresponding queues. Otherwise, the message will not reach the queues. At the lower level of the binary tree structure, the message will be modified for the final time. This is where the decoders along with some additional logic will add to the message the source card dimensions and the source processor listing for fault tolerance. These fields are needed in the case that a particular card or processor goes down and the message must be retransmitted. This modification of the message will be described in detail in the next section.

3.2.3 Interface to Destination Card

The interface to destinaton card message is the final message that is created in this system. Once the message reaches the last level of the decoders in the communications interface, the special purpose logic in the decoder chips modifies the CIMF message to the format that will be sent to the destination queues. This modification must be made prior to the message being passed to the destination card queues. The new message that is created will be referred to as the Destination Message Format (DMF message). The DMF message appears in Figure 12.

(6-bits)	(6-bits)	(3-bits)	(6-bits)	(6-bits)	(3-bits)	(1)	(1-bit)	(2-bits)	(30-bits)	(64-bits)
SD0	SD1	SP	DD0	DD1	DP	C	D	CDP	MA	Data
127	121	115	112	106	100	97	96	95	93	63 0

Figure 12 Destination Message Format (DMF message)

The DMF message is similar to the CIMF message in that many of the fields are the same except they appear in a different location in the message. The DMF message fields are all described in Table 9. The first three fields, SD0, SD1, and SP are all information about the source card and processor. This information is known by the current card and can easily be inserted into the CIMF message. The data that was in the corresponding fields is moved to a different location. The CDP field is moved from bit location 114 to bit location 95 and overwrites the CDD field. This CDD field is no longer needed since the decoders have placed the messages on the correct lines to go to the correct destination card queues. This information was only used so that the decoders could determine which queue to place the corresponding messages into.

Table 9 DMF Field Descriptions

Field	Description
SD0	Source card dimension 0 (6-bits).
SD1	Source card dimension 1 (6-bits).
SP	Source processor (3-bits).
DD0	Destination card dimension 0 (6-bits).
DD1	Destination card dimension 1 (6-bits).
DP	Destination processor (3-bits).
C	Code bit used in addition to the DP field when necessary (1-bit).
D	Specifies if DMA Control is being used (1, if yes; 0, otherwise) (1-bit).
CDP	Code destination processor bits used with the DP field to specify if a code is being used (2-bits).
MA	Memory address field for the destination processor. The total memory for each processor is $2^{30} = 1$ Gword. The local memory is part of the DSM (distributed shared memory). Each word is 64 bits long. Since we are making the memory word accessible only, in actuality we have 8 times as much space per processor, which equals $2^{33} = 8$ Gbytes of DSM (30-bits).
Data	Data field of the message (64-bits).

Once the queue is determined, the CDD field is no longer needed. What the decoder will do here is place the correct destination card dimension in the corresponding DD field in the DMF message based off of the output line that the message is currently on. This line specifies which destination card the message will travel to. The reason that the decoder will make this modification is that the CDD and the XDD fields are no longer used. Now the decoder will not put the eight bit multicasting code (DD field plus XDD field) in the corresponding DD field of the DMF message, but instead will place the actual destination card dimension of the queue into the correct DD field. The correct DD field is coded directly into the hardware, just like the source information. In simpler terms, the decoders know that they are either outputting messages to the 0 or 1 dimension section of the coprocessor, and therefore know that the DD field that they need to modify is the one that is opposite of the dimension section they are part of. For example, if source (0,0) wants to send a message to cards [(0,1), (0,2), and (0,3)], the decoders for the message are on the dimension 0 side (port 0) of the coprocessor. The fields that are changed in the message are the DD1 fields since those are the ones that are multicast fields and use the extra two bits from the XDD field.

The C and D fields are moved and overwrite the XDD field which, as stated before, is no longer needed. The D field is especially important here since it informs the destination processor if the data field actually contains the data to be sent or if the MA field contains the address of the data in memory. The DD0 and DD1 fields are either changed corresponding to the previous explanation, or left as they are if the message is not a multicasting message. In the case of a one-to-all broadcast, the BC field states that all of the destination cards would receive the message. The 111111 code that is contained in field BC is also contained in fields DD0 and DD1. This allows for the removal of the BC field. Since the DD fields contain the information for the one-to-all broadcast operation, the fields do not need to be modified. The same is true for the DP field. It must be left unchanged. The CDP field, that specifies if the DP field is a code or not, must be moved from its current bit location of 114 to location 95. These are all of the fields that need to be moved. The MA and Data fields will stay as they are. Now the source information can be added to the message. The upper 15-bits of the message is replaced with the SD0, SD1, and SP fields. These fields are known by the hardware and are easily inserted into the message.

3.3 Initial Design

The initial design of the communications interface calls for the creation of the coprocessor interface system previously discussed. The design calls for a processor to be locally attached to its communications coprocessor. The processor and coprocessor will both be able to access the memory locally and at the same time. Since we are dealing with a distributed shared memory system (DSM), all processors will be able to access the memory of all of the other processors. The memory for a processor can be accessed by another processor through the use of the processor's label as a part of the memory address. With the address of the processor whose memory is needed to be accessed, another processor can read the data that is stored there. This is extremely useful in the case where the data to be transferred to the destination processor is larger than 64-bits and would require multiple messages to be sent. With a single message, the source can allow the destination to access data that is greater than the 64-bit maximum of a single message and without tying up the message transmission lines.

The coprocessor will run code that will allow it to complete all of the message transformations and transmissions as specified in the previous sections. The coprocessor must be able to acquire the destinations and the data, and further be able to combine all of the information into the least number of messages as possible. The goal of as few messages as possible is important to retain the performace of the system. With fewer messages, that means a lesser load on the system, which in turn leads to fewer collisions and a greater total throughput.

Each of the 12,800 coprocessors in the system must have the same design of ten ports with eight connections per port and run the same code. The eight lines leaving each of the ports must connect to a binary tree of six decoders. These decoders will view different fields of the message header and proceed to distinguish which of the messages are meant for which destination queues. The lower level of decoders will also contain additional special purpose hardware that will allow for the modification of the message. The modifications can be done with a shift register for the fields that need to be moved and the insertion of 15 bits to add fault tolerance to the message, the additional hardware will be minimal. Upon leaving this layer of the decoders, the messages will proceed directly to the destination queues. These queues are connected to the decoders by direct links. Therefore, the destination card information that each of the messages contains in it is present for fault tolerance and not necessarily for the first layer routing of the messages. After arrival at the destination card queues, the queues will handle when the messages will be put onto the medium and transferred to the actual destination cards. The description of the inner workings of the queues will be presented in Chapter 4.

3.4 Software Design for the Coprocessor

The coprocessor contains extensive code to process the incoming messages, retrieve information from the memory, and output additional messages to the queues. The software design of the coprocessor has the goal of alleviating some of the tasks from the processor. The coprocessor's software must provide for optimal execution of all tasks.

CHAPTER 4

INTRODUCTION TO MESSAGE PRIORITIES

4.1 Queue Design

The queuing system of the cards is the last major design area to be discussed. The performance of the system discussed in this thesis is partially dependent on the performance of the queuing system. The queues directly affect system performance due to the fact that all destination messages must pass through them. If the queuing system is poorly designed it will lead to multiple collisions upon leaving the processor and will drastically reduce the throughput of the system. A design for the queues must achieve a number of goals.

The major goals of the queuing system are to allow all of the queues to output destination messages at the same exact time instant, to implement priorities into the messages, to be quick and efficient, and to avoid collisions as best as possible. The initial design of the queues was to implement a basic first-in-first-out (FIFO) structure. After evaluating this design it was deemed inappropriate since it does not allow for the priority schemes to be introduced.

The initial hardware design calls for the queues to be hardwired directly to the communications interface and further to connect to the lowest level of decoders in the binary tree structure. The original design specifications required a GH(2,40) generalized hypercube, with each node of the hypercube representing a card. Connected to each card would be eight processors, eight coprocessors, a number of decoders, and 79 queues per processor along with memory elements. The value of 79 queues was chosen since this is equal to having one queue for each card in the same 0 and 1 dimension as the source card. There are 79 and not 80 queues since the source card's queue appears in both the 0 and 1 dimension but only needs one queue for itself.

The queues receive the messages directly from the communications interface and must output the messages directly to the destination cards that they correspond to. This means that the queue's have direct links to the destination cards that they are designated to send messages to. This is an optimal design and allows all of the queues of a particular processor to transmit their messages to all of the destinations at the same time instant. The actual time instant that the queue will output messages to the destination will vary based on a number of factors. A problem is encountered if all of the processors on one card or even if processors on different cards all try to send messages to the same destination cards at the same time instant. This could lead to collisions on the transmission lines leading to the destination card.

In order to prevent as many collisions as possible, the system will have to be synchronized. This synchronization will allow for the standardization of the exact instant that the queues will output their messages to the destinations. For example, the goal is to have all of the queues send their messages at the beginning of every clock cycle. This will prevent collisions that would occur if some queues sent their messages at the beginning of the cycle and others sent their messages at the end of the cycle. This is a very important property since the algorithm to send the messages is based upon a standardization of when all the queues output their messages. This must be standardized so those different queues can output their messages to their destinations without incurring collisions. This solution will be further discussed later as it relates to reference [1].

A second design of the queuing system calls for a reduction in the total number of queues. These modifications were made to reduce the amount of total hardware in the system and further reduce the cost of the system. This design calls for b queues per processor where $b<d*k$. The variable k represents the total number of cards in one dimension of the system and the variable d represents the number of dimensions in the system. Essentially in this system the formula $b<d*k$ means that the number of queues (b) will be less than 2x40, or $b<80$. The advantage to using fewer queues per processor is in reducing the total amount of hardware in the system and moving more of the complexity away from the hardware and into the software design of the queuing system.

This design of using less that dk queues requires that messages for different destination cards be stored in the same queue. Storing the messages for different destination in the same queue is accomplished by logically splitting the output queue into many separate "mini queues," which will be referred to as mq. There are $mq=(d*k)\div b$ "mini queues" per processor. Once the number of queues and "mini queues" has been determined, the next major design issue is the priority scheme to use within the queues.

4.2 Priority Policies

The policy for assigning priorities is not a simple one. The scheme must either be implemented in hardware, software or a combination of the two. The 100% hardware approach to priorities is not an optimal one in this system. The reason is that the load on different areas of the system may change frequently and one the priority scheme is hardwired, it cannot be changed at runtime. The 100% software approach can be optimal since the priorities are dynamic in that the system has the ability to change the priority of specific messages to fit the state of the system. The optimal solution, which is discussed in the next section, uses a hybrid approach.

The need for different priorities arises in the case when control messages must be sent immediately. If these messages have no way of overriding the current data messages to be sent, the performance of the system will be affected negatively. Priorities are also needed to allow information needed immediately to be sent to the destination prior to information that is not needed immediately. Some possible priority schemes include, but are not limited to the following.

One scheme would be to send the messages that have the furthest distance to travel in the 0 or 1 dimension prior to those that must travel a shorter distance. This is useful in cases where the distance directly affects the propagation time. Another scheme would be to send the messages to the destination that has the most messages waiting to be sent. This scheme is useful since in many cases the destination with the most outstanding messages may be idle or waiting for additional information to complete its computations.

Another priority scheme would involve the creation of a history table. This table could be used to track either the destination with the greatest probability of receiving messages. This information could further be used to help alleviate the amount of traffic going to that destination by sending messages to other destinations first or it can be used to determine which messages will probably flow to that destination. Yet another scheme would be to assign a higher priority to the messages that have already traveled one hop and are one their way to the destination. Since these messages have already passed through one priority mechanism at their source, they do not need to be reshuffled into another queue before being sent. These messages would therefore receive a higher priority. All of the above methods are useful in various cases and there are many more useful methods. The optimal solution for this system is to combine the above priority schemes into the software of the system, since this system will contain a hybrid design. This makes the priority mechanism dynamic and capable of producing a higher throughput for the system. The many different priority designs will be addressed in Section 4.3 [8].

4.3 Priority Designs

The main issues to contend with when designing the queues are the priorities of the messages and the speed of transmission of the queues. In order for this method of "mini queues" to be as effective as the method when the number of queues, b, equals the number of cards in all connected dimensions $d*k$, it must run at a faster clock speed. This will be necessary in the case of one-to-all broadcast operations. In the first case when the number of queues equals the number of possible destination cards, all of the $d*k$ queues could output the messages during the same clock pulse. In this second case, it would take $mq * b$ clock cycles to obtain the same effect since there are less total queues present. Assuming that all queues have the same number of output lines, the "mini queue" design with fewer total queues would require a single queue to send messages during multiple clock pulses in order to output the same total number of messages as the prior case. This is not a major design issue since the case containing fewer queues could be operated at a much faster clock speed to account for this extra time delay.

The new design assumes a queue that contains 2^s "mini queues." To determine the "mini queue" with the highest priority of transmission, a code of length s would be used. The process to determine this code will be described shortly. The first issue that we must realize is that the "mini queues" use a FIFO queuing system with a pointer to select the next message to be sent out. This is important since the pointer will take the place of the traditional counter in determining how many messages are currently stored in that particular mini queue. Since for each "mini queue", we know its starting location within the queue and the location that the pointer is currently pointing to, the number of messages waiting can be found by subtracting the one number from the other.

There are two different ways to proceed with the "mini queue" design. The first design would be to use comparator circuits, as illustrated in Figure 13 with eight "mini queues" per queue, to determine which pointer is the farthest from the beginning of the "mini queue" and, therefore, contains the most data and encoders. The second hardware implementation, discussed later, would be the more efficient one resulting in the highest operating speed.

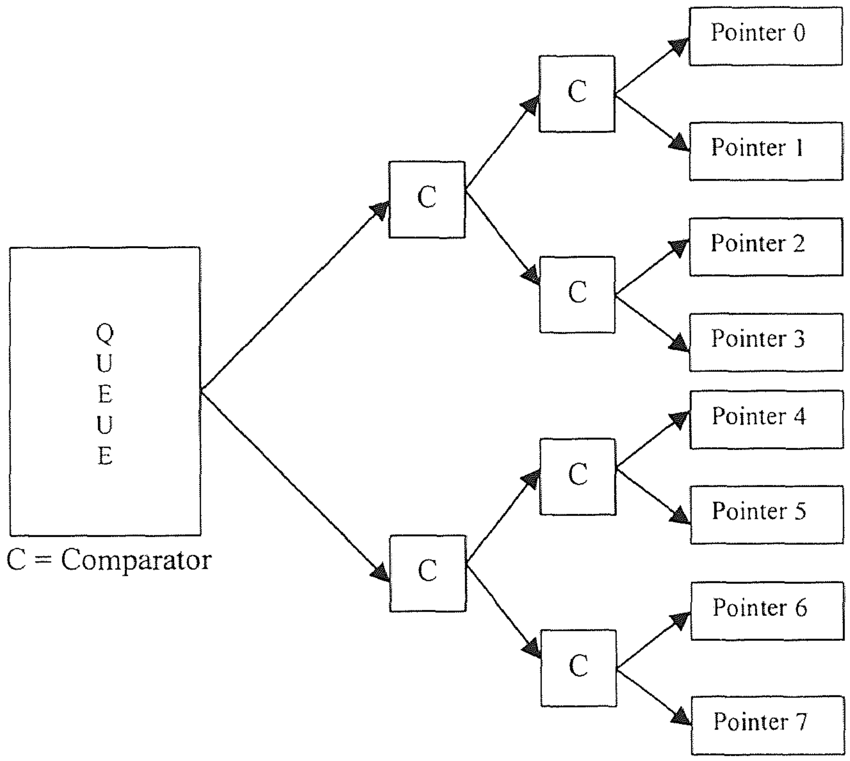

Figure 13 Queue Comparator Design

The comparator circuit is not a very complex design but does require a large amount of hardware as the size of the "mini queue" increases. The complexity is already high once the number of queues is less than the possible number of destination cards. In this case, which we will explain in great detail, problems arise with the "mini queues" inside of each of the queues. The problem of sending the data from all of the "mini queues" of a queue at the same instant (i.e. within a single regular clock cycle) is solved, as mentioned before, by increasing the speed of the components. This problem assumes the same priority of transmission for all of the destination cards). In contrast, the next problem to be addressed is in determining which one of the "mini queues" inside a queue has the highest priority of transmission each time.

The priority problem could be corrected by adding a priority bit to each of the "mini queues" and a priority code to each of the queues. This code would depend upon the number of "mini queues" in the queue. That is it was stated that if the number of "mini queues" increases exponentially, so would the number of bits for the priority code. The "mini queues" would be labeled with binary numbers, which is identical to the priority code's labeling. For example if there are 10 queues, each containing 8 "mini queues," then the "mini queue" priority code would be 3 bits long (we will refer to this as example 1, corresponding to 79 possible destination cards, 10 queues, and 8 "mini queues" per queue).

This priority code would specify the "mini queue" with the largest number of messages waiting to be sent as having the highest priority. Each of the "mini queues" would have its own priority bit and its own timeout bit in order to prevent starvation. This would lead to eight priority bits (*pb*) and eight timeout bits (*tb*) here. Each of the eight priority bits would be connected to a priority encoder that encodes the eight bits into three bits. This does not handle the case when all of the queues have a priority code of 0. This can be corrected by adding another bit to the output of the encoders, using a second encoder, or using some additional logic to detect that case. We will assume the last solution of using additional logic to bypass the code that is created. The timeouts will be handled in the same exact fashion. The difference is that the timeouts will be able to override the priority signals to specify which "mini queue" will be allowed to send its message, this is illustrated in Figure 14.

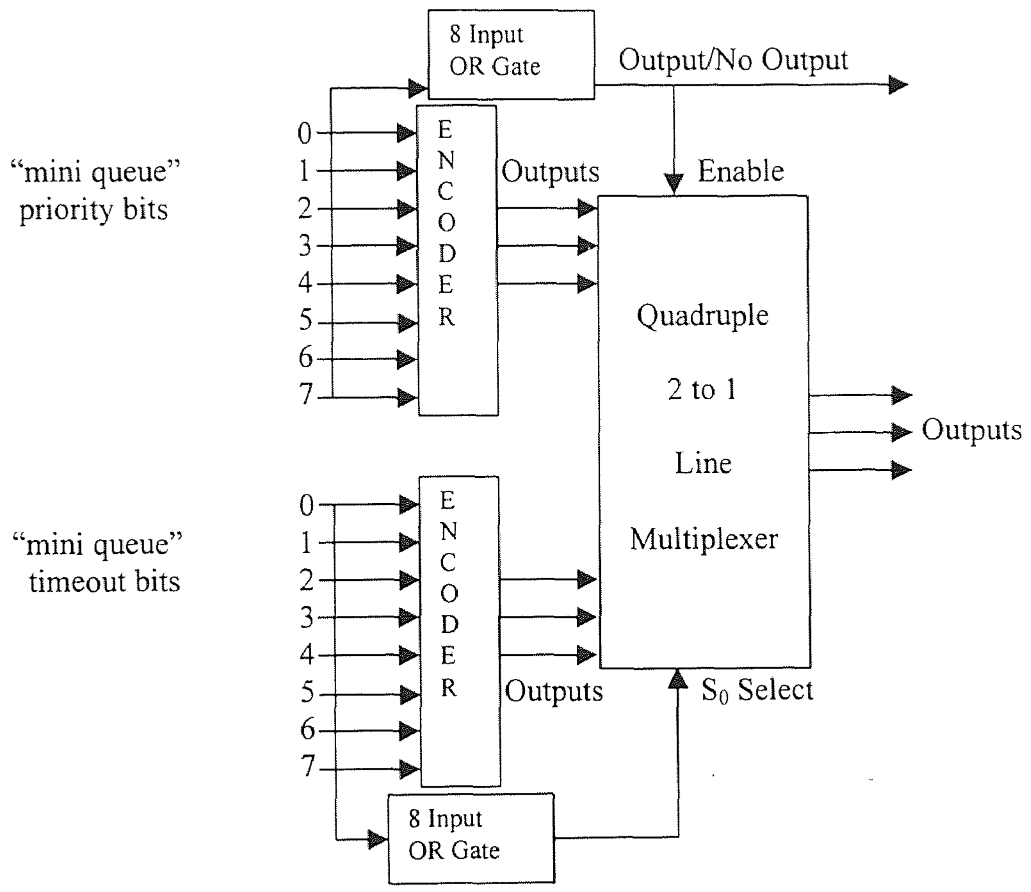

Figure 14 "Mini Queue" Encoder Design

Figure 13 illustrates the design that I was just discussing, right before the priority bits are set. The comparators are in a tree-like structure and check the pointers for the largest value. The pointer(s) with the largest value will have its corresponding priority bit(s) set to 1. Once the priority bits are set, the circuit in Figure 14 will analyze them.

In Figure 14, the eight priority bits and timeout bits are connected to their respective encoders. The priority bit and timeout bit encoder's data is contained in Table 9. Each of these encoders is actually a priority encoder. By this I mean that if more than one input has a 1 set in it, then the higher one has the priority. To better explain this, look at Table 9. In the last line of Table 9, if all of the inputs (I0 to I7) have a 1, only the last input (I7) matters and the code is set for that input having the highest priority. The other inputs are don't care's symbolized by the X. The encoders in Figure 14 will encode the eight input bits and convert them to a three bit code to be sent to the quadruple 2-to-1 line multiplexer. The Enable line to the multiplexer is just the logic OR operation of the eight input bits to the priority bit encoder. The purpose of this is to handle the case when none of the "mini queues" have any messages waiting to be sent. In this case the multiplexer is not enabled and the Output/No Output line is set to 0. This 0 reflects the fact that nothing needs to be sent from this queue. The select line is the logic OR operation of the 8 input bits to the timeout bit encoder. When there is at least one timeout signal on the input lines, then the timeout code must be sent, otherwise, the select is set to 0 and the code from the priority encoder is sent out.

Figure 15 illustrates where the 3 output bits of the 2-to-1 MUX will be connected. These 3 bits will be connected to the 8-to-1 MUX and the 1-to-8 DEMUX. The input line to the 8-to-1 MUX are the output lines from the queue containing the 8 "mini queue"s. There is one line for each of the 8 "mini queue"s. The output from the 8-to-1 MUX then connects to the input of the 1-to-8 DEMUX, along with the 3 select lines. The outputs of the 1-to-8 DEMUX in turn connect to each of the 8 destination cards for this specific queue.

Table 10 Priority Bit Encoder and Timeout Bit Encoder Table

Input								Output		
I0	I1	I2	I3	I4	I5	I6	I7	O0	O1	O2
1	0	0	0	0	0	0	0	0	0	0
X	1	0	0	0	0	0	0	0	0	1
X	X	1	0	0	0	0	0	0	1	0
X	X	X	1	0	0	0	0	0	1	1
X	X	X	X	1	0	0	0	1	0	0
X	X	X	X	X	1	0	0	1	0	1
X	X	X	X	X	X	1	0	1	1	0
X	X	X	X	X	X	X	1	1	1	1

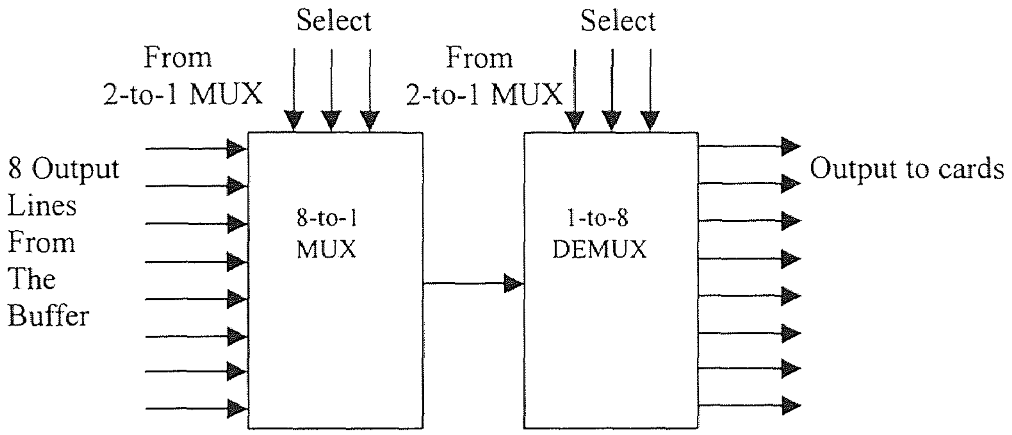

Figure 15 All Hardware Priority Design

The circuit will function as follows. The 2-to-1 MUX will send the 3-bit code to the 8-to-1 MUX. This MUX will use the code to select which of the messages from the "mini queue" has the highest priority and will activate the corresponding line. The DEMUX will receive the message and with the code, determine which of the destination cards it should be sent to. Then the corresponding line will be activated. A second possible solution to this same circuitry would be to connect the 2-to-1 MUX to the decoder of Figure 16. This 3-bit code is then decoded into 8 bits to select which of the 8 "mini queues" has the highest priority and therefore selects its corresponding register to transmit the message to the destination card. Each of the lines of the decoder in Figure 16 corresponds to a register connected to the queue. This is a 100% hardware approach to the problem. It is not the best solution but does show that there is a purely hardware solution available.

A different solution to the priority problem would be to use software in addition to hardware. This would allow for the priorities to be changed according to the situation at hand and not to be hardwired. One possible design for this type of system would require the software surrounding the "mini queues" to determine the priorities and to set the priority bits. This would allow the priorities to be shifted from one priority scheme to another. The possible schemes are the "mini queue" with the largest number of waiting messages, or the one that currently has the message with the largest amount of data waiting to be sent. Another is the one that has the farthest distance to travel, or the "mini queue" with the largest number of messages going to the same destination card. Yet another would be the one that is sent to most often as dictated by a history table, and so on. Basically this design allows the priorities to be dynamic. This is very important since the need for different types of messages could depend greatly on the amount of traffic on the system. This dynamic priority scheme can alleviate certain problems through the software surrounding it.

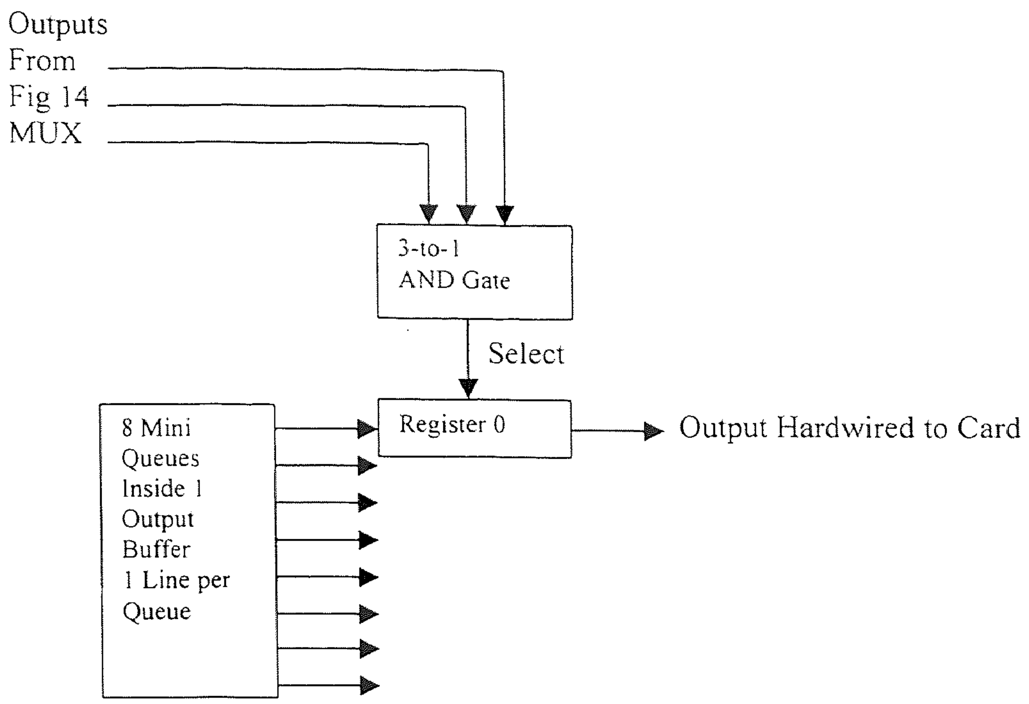

Figure 16 All Hardware Priority Design with Registers

A more efficient solution to this problem would be a combination of hardware and software. The software will handle the first level priority assignments, deciding which of the messages in each of the "mini queues" has the highest priority for that queue and then sets its corresponding code. In addition the software handles the second level priority assignments and determines which of the queues within the queue has the highest priority and sets the corresponding 3-bit code.

The best solution is to do the following. The 3-bit software determined code from the queue would be sent to the decoder in Figure 17. This 3-bit code is then decoded into 8 bits to select which of the 8 "mini queues" has the highest priority and therefore selects its corresponding register to transmit the message to the destination card. Each of the lines of the decoder in Figure 17 corresponds to a register connected to the queue.

The previous analysis provides a number of different designs that incorporate hardware, software, and a combination of the two in solving the priority scheme problem. The final design of Figure 17 would provide the optimal solution to the problem.

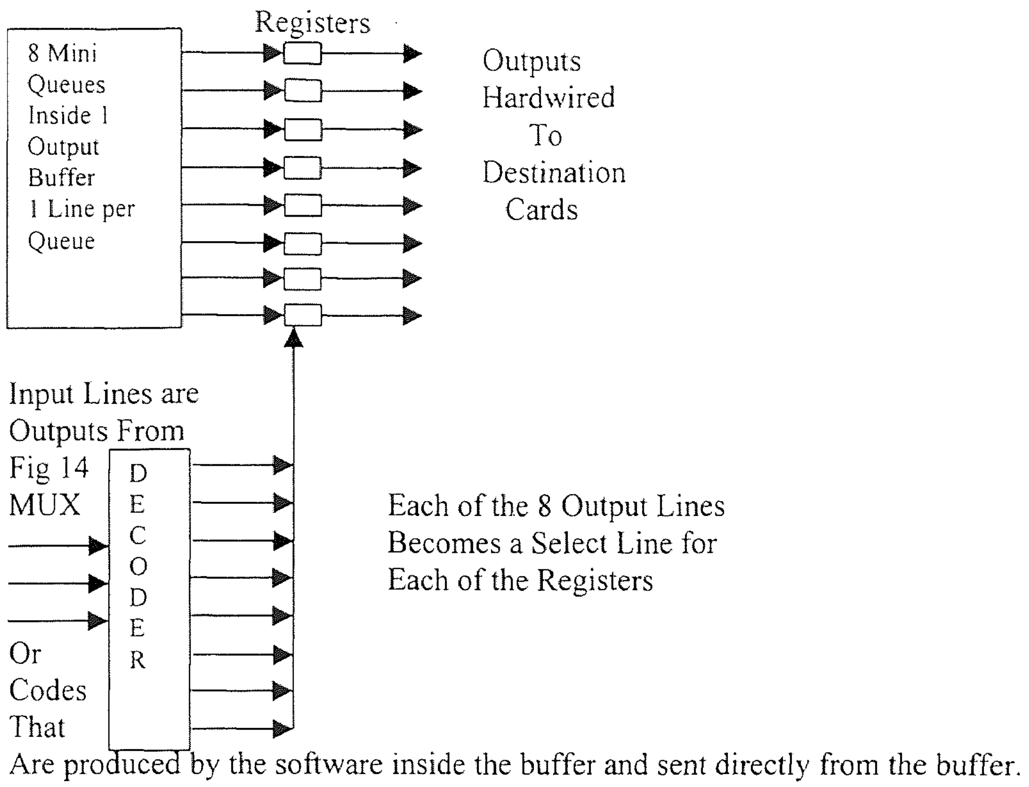
Registers
8 Mini Queues Inside 1 Output Buffer 1 Line per Queue
Outputs Hardwired To Destination Cards
Input Lines are Outputs From Fig 14 MUX
DECODER
Each of the 8 Output Lines Becomes a Select Line for Each of the Registers
Or Codes That Are produced by the software inside the buffer and sent directly from the buffer.

Figure 17 Software Design

4.4 Priority Software for the Coprocessor

The software support for the priority scheme of Figure 17 is necessary to achieve the desired throughput of the system. The code will be fully integrated with the hardware and allow the priority schemes to be efficiently implemented.

CHAPTER 5

SUPPORT FOR WORMHOLE ROUTING

5.1 Store-and-Forward vs. Wormhole Routing

Store-and-forward routing is one of the most basic routing methods. This method sends packets to their destinations by storing the entire packet in intermediate nodes before forwarding it. A full explanation of this routing method is contained in Section 1.3.1.

The advantage of using this routing technique is that it is simplistic in nature and often performs satisfactorily. The all-port model in our architecture allows each card to receive messages from all 79 input lines at the same time instant and store the messages in the buffers for each line. The internal buffer must be large enough to support this and the card must support outputting on all lines at the same time instant. In the system that we are discussing, each card has a buffer to store this data. The buffers hold the information until it can be passed on to the next node in the system.

The disadvantage of the store-and-forward technique is that the intermediate cards must store every packet that is received and then forward it to the destination. Since each card stores the arriving packet, the latency time is linearly dependent upon the number of hops from the source to the destination. The internal code will take the messages that are in the buffers and attempt to send the messages out as soon as possible. In our system this is a "worst-case" scenario where a message must travel a maximum of 2 hops between the source and the destination.

Wormhole routing on the other hand reduces the time that is needed to transmit messages from a source to a destination processor. The source will send the initial flit, containing the message header to the intermediate card first. This flit will be read by all of the intermediate nodes so that they know which node to pass on the following flits to. The following flits are not stored in the intermediate nodes as they arrive, instead they are just passed on to the next node until they reach the destination. As soon as the flit begins to arrive at the intermediate node, it can start passing through the card and leave on the output lines. This passage of flits without storing decreases the amount of time that it takes for a message to travel from the source processor to the destination processor.

The disadvantage of wormhole routing is that the path must be fully dedicated to the packet being sent, until all of the flits are done transmitting. This can cause degradation in the performance of a system with multiple hops. However, that is not the case in our system. In our system, a message will have a maximum of two hops between any two cards in the system. This makes the wormhole routing design feasible and effective.

Even though the channel must be dedicated to the source until all flits are transmitted, this will not affect the performance of our system significantly. The intermediate nodes must not receive flits or messages sent from other nodes on the same lines, or else the intermediate card will assume that they are part of the message being sent by the original card sending the message. After the first flit passes through, since the intermediate card knows that all messages are a total of 128-bits long, it routes the following flits to the same destination until 128-bits have been sent to that destination. Because of this, any flits from a different card could be sent to the wrong destination if they were allowed to arrive in the middle of a transmission of another message.

In this wormhole routing design, the intermediate card does have the ability to accept flits on all input lines of the card, at the same time instant, and then route the flits to the correct destinations. This is feasible as long as the arriving flits on each line are from their respective cards that initiate the transmission. If another card attempts a transmission by sending its flits before the initial card finishes sending its flits, the flits for the former transmission will be stored temporarily in flit buffers.

5.2 Design

The design of the store-and-forward system and the wormhole routing system will differ in a number of aspects. The first design to be discussed will be that of the store-and-forward routing method. The arriving messages will automatically be placed into an arrival buffer. There will be an arrival buffer attached to each input line to buffer the arriving messages. The arriving messages must be buffered so that the card can determine if the destination of the messages is that card that just received them or another card in the system. The buffer is also needed in the case where too many messages arrive for the same output and the card cannot process them all at the same time instant.

Once a message arrives, the hardware will determine if the message must be sent out immediately or routed to the correct processor on the card for processing. If the arriving message specifies the current card as the destination, the message header is then processed and the destination processor is determined. Once the processor is determined, the message is sent via the crossbar network to the corresponding destination processor. At the destination, the message is put into the destination queue for the processor that it corresponds to. As was stated earlier, each processor has 79 queues where one of those queues is dedicated to arriving messages for the current processor. The process of analyzing the message header and sending it to the correct queue is completed quickly due to the great processing power of the interface and destination card. Once the message is placed into the destination processor's queue, the card is finished with it.

The other case to contend with is that of the arriving messages that must be sent out again and routed to their corresponding destination cards. The procedure is similar to that of the messages that arrive and must be routed to their corresponding destination processors. The arriving message headers are analyzed and routed to queues that correspond to each destination card. There are 79 destination queues. The arriving messages are then sent out of the queues as soon as possible. This will yield a best case processing time of one clock cycle to process the header and route it to the correct queue. Once at the queue, if there are no other messages waiting to be transmitted, the message will be immediately transmitted. The worst case time will depend upon the size of the queues.

The wormhole routing method differs in its design. Initially, the process is similar in that the first flit to arrive will be analyzed and placed into an arrival queue. The header information of this first flit will be analyzed to determine if the destination is on the current card or on a different destination card. The case of the destination processor being on the current card will be analyzed first.

Once it is determined that the destination processor is on the current card, the arriving flits will be placed in an arrival buffer that corresponds to the correct processor. Therefore, there will be eight arrival buffers for the processors on the current card. Once the initial flit specifies the processor as the destination processor, all of the flits that arrive from that input line will be placed into the buffer until all 128-bits are received. The total number of flits that make up a packet is a set value, and therefore the card knows how many flits will be sent to the buffer. The buffer sets a flag bit once enough flits have been received to constitute a message. This flag bit is scanned and, once set, the processing power of the card will then reassemble the message from the flits and place it in the corresponding destination queue of the destination processor on the current card as was done in the store-and-forward method.

One problem with this method occurs in the case when more than one input line has the same destination processor on the current card as the destination card. This presents a problem since the flits cannot be mixed up because only the initial flit contains the header information. Another solution to the problem, which requires more hardware would be to have an arrival buffer on each input line for destinations that are on the current card and one arrival buffer per input line for destinations that are on different cards. The first case will be analyzed now.

The arriving flits will be analyzed to determine if the destination is on the current card. If the destination is on the current card, then it and the following flits will all be placed into the buffer that corresponds to the input line that it arrives on. Once the whole message is received, a flag bit is set and then the processing element of the card will reassemble the message and place it into the destination processor's queue that it corresponds to. For example, if the destination processor is processor 0 on card 0, then the message will be placed into processor 0's queue corresponding to card 0.

The more complicated case of the arrival of flits to a card that must be outputted to another card will be analyzed now. As each flit arrives, the header information is analyzed. Once it is determined that the destination is on a different card, the flits will pass through an output port for the corresponding line of the destination. Once the first flit arrives at the port, a flag bit is set to 1 to specify that a message is in the process of passing through the port and arriving at the destination. The process is completed instantly since the output port immediately sends the flits out. From outside of the card there is no delay and as a packet arrives it is sent out to the next card. This works optimally as long as two input lines do not want to pass their flits to the same destination card.

In the case where two of the input lines want to pass their flits to the same destination card, a method must be presented to prevent the flits from getting mixed up. This was the purpose of the flag bit present in the output port. Once a flit is passed through the buffer, the buffer will only accept flits from the same input line until all of the flits for the message have been outputted. In the meantime, if a flit arrives on a different line to be sent out to the same destination, its flits are temporarily stored in a separate input buffer at the corresponding input line.

In reality there are two sets of buffers per transmission line in the card. The first buffer is the arrival buffer for the flits that have the current card as a destination. The next buffers are those for the case when the output port is in use but more flits arrive from a different input line with that card as the destination. In this case the arriving flits are temporarily stored in the input buffer for outputting to a different card. Once the first flit is stored, a 1 is placed in a list of all destination lines corresponding to the destination line that this group of flits must travel on. Once the output port is done transmitting, the processing element of the card scans the input buffers for different destinations for a 1 placed in the list corresponding to the open output port. Once a 1 is found, the flits present there are transmitted one by one to the destination. The input buffers can store more than one group of flits since the buffer knows how many flits correspond to a message. This case is not optimal since the more flits what are stored in the buffer, the greater the transmission time for that message.

To be optimal, the arrival buffer must have a maximum capacity of 1 message. In this optimal case, the input buffers would not exist. This will have the effect that the arriving flits will either be immediately transferred to the output port or the flits must be resent. This is optimal since the arriving flits do not have to wait for an available output port to continue on the path to the destination. Instead the arriving flits will immediately be sent to the next intermediate node or else they must be resent. The key here is that the next immediate node that the flits go to may not be the original node intended to receive the flits.

The routing and congestion algorithm will be responsible for deciding the path that the flits will take This decision will be based on the current load on the system, bad links, and other congestion factors. This routing decision will be made instantly by the dynamic algorithm being used. The advantage of this method is that there is no buffering of the nodes to wait for a line to become free. This factor cuts down the total time that the flits spend in intermediate nodes and thereby enhances the performance of the system.

This wormhole routing method will perform optimally in the system that we are discussing. This optimal performance will be achieved with the presence of load detectors in the routers that will have the ability to shift the load on the system if it is high in one area of the system, and through the use of buffers that are not too large. With small buffers, if there is more than one set of flits stored (each set corresponds to one message), then other arriving messages will be turned away and the source will have to route them differently. Many of these improvements can be made through additional software support.

CHAPTER 6

CONCLUSIONS

The thesis just presented is the culmination of a year's worth of work in the area of parallel processing. The designs that have been presented here were done with the intention of improving the performance of a massively parallel processing machine that is still in its own design phase. This paper has proven the suitability of the author's work for use in the NJIT New Millennium Computing Point Design. The proposed modifications will increase the performance objective of the proposed system, thereby bringing the system closer to the PetaFLOPS performance objective.

REFERENCES

[1] Fragopoulou, Paraskevi, Selim G. Akl, and Henk Meijer, "Optimal Communication Primitives of the Generalized Hypercube Network," *Journal of Parallel and Distributed Computing.* 32, 1996, pp. 173-187.

[2] Ziavras, Sotirios G., Haim Grebel, and Anthony Chronopoulos, "A Low-Complexity Parallel System for Gracious, Scalable Performance. Case Study for Near PetaFLOPS Computing," *6th Symposium on the Frontiers of Massively Parallel Computation,* 1996, pp. 363-370.

[3] Ziavras, Sotirios G., Haim Grebel, Florent Marcelli, and Anthony Chronopoulos, "A New-Generation Parallel Computer and its Performance Evaluation," *Submitted for Publication.* 1997.

[4] Grimes, Vincent P, "Quantum Leap In Computing Speed Likely Within Months," *National Defense.* 81(521), 1996, pp. 16-17.

[5] Miller, Chris, "Sandia Helps Break The Supercomputer Speed Barrier," *Sandia Lab News.* December 20, 1996.

[6] Hwang, Kai, Advanced Computer Architecture. McGraw-Hill, New York, 1993, pp. 77-89, 161, 376, 387-391, 475-483.

[7] Mano, M. Morris, Computer System Architecture. Prentice Hall, Englewood Cliffs, NJ, 1993, pp. 415-416, 498-500.

[8] Gaughan, Patrick T., and Sudhakar Yalamanchili, "Adaptive Routing Protocols for Hypercube Interconnection Networks," *Computer.* May 1993, pp.12-23.

[9] Akl, Selim, Parallel Computation. Prentice Hall, Upper Saddle River, NJ, 1997, pp. 50-69.