A DISTRIBUTED TRANSPUTER-BASED ARCHITECTURE USING A RECONFIGURABLE SYNCHRONOUS COMMUNICATION PROTOCOL

Marie Josette Brigitte Vachon

B.A.Sc. Royal Military College, Kingston, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
August 1989

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of ELECTRICAL ENGINEERING

The University of British Columbia
Vancouver, Canada

Date 16 AUG 1989

DE-6 (2/88)

Abstract

A reliable, reconfigurable, and expandable distributed architecture supporting both bus and point-to-point communication for robotic applications is proposed. The new architecture is based on Inmos T800 microprocessors interconnected through crossbar switches, where communication between nodes takes place via point-to-point bidirectional links. Software development is done on a host computer, Sun 3/280, and the executable code is downloaded to the distributed architecture via the bus.

Based on this architecture, an operating system has been designed to provide communication and input/output support. The message passing communication protocol uses circuit switching and a centralized reconnection control strategy. The communication protocol is composed of two modules: A Local Bus Interface (LBI) that runs on every processing element and a Central Switch Controller (CSC) which executes on a bus master and reconfigures the network topology as required by the user program. The LBI is small, simple, and deadlock free. User reconfiguration requests are interrupt driven, and the CSC can support real-time reconfiguration of the topology without interfering with other communications. An Input/Output Controller (IOC) process runs on the host computer and provides standard library support to each processor in the network.

An important feature of the circuit switching communication protocol is that it preserves synchronous communication where processors do not need to store messages and no buffer management is required. Routing overhead occurs only when the circuit is set up, so subsequent messages may flow through the network with a guaranteed bandwidth and a maximum communication latency, which is an important consideration in real-time systems.

The routing mechanism is adaptive where communication hot spots may be detected and bypassed. The centralized reconfiguration control strategy and the adaptive routing mechanism provides a basis for a reliable architecture. In the case of a link or processor failure, the routing mechanism is capable of bypassing a faulty component without affecting the application program, and can also redirect messages to a backup processor. Knowledge of the faulty component is required

at the CSC only, in contrast to most routing algorithms where either the local or global state of the network is essential at every node.

The performance of this communication protocol is compared with Helminen's store-and-forward model [1] running on the FPS T-series hypercube. The results show that the centralized circuit switching protocol provides better performance when a message crosses multiple hops, large message length, and when the algorithm contains temporal locality properties. Dynamic protocols, however, are less desirable for short messages due to the initial connection latency.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisors, Dr. Lawrence and Dr. Ito for their pertinent suggestions and continuous encouragements throughout the course of my thesis.

Special thanks to my colleagues from the transputer group, Real Frenette, Benjamin Lau, and Sanjay Chadha for their pertinent discussions and great help in the rewrite of the Occam software to Parallel C and Joseph Poon for his support in the testing of the path planner.

I would also like to specially thank my friends Ron Jeffery, Mark Miller, and Kevin O'Donnell for their patience in the laborious task of correcting my English writing. Thanks again to Ron for his help in the last phase of my thesis writing.

Most of all, I must express a special gratitude to my closest friend, Joel Bisson, for his warmth and invaluable support during these last two years.

I would also like to thank The Department of National Defense for their financial support throughout the duration of this thesis.

Chapter 1

Introduction

Advancement in technology has led to many novel robotic applications. For example, robots are now found in military sites, hospitals, nuclear power plants, and, in the near future, the space station. Simpler teach-playback robots with no more than six degrees of freedom are being superceded by high-performance robots incorporating recent advances in vision, optimal and adaptive controls, and artificial intelligence. Since sophisticated robots will be used in hazardous and unstructured environments, highly complex sensor-based control problems must be solved in **real-time.**

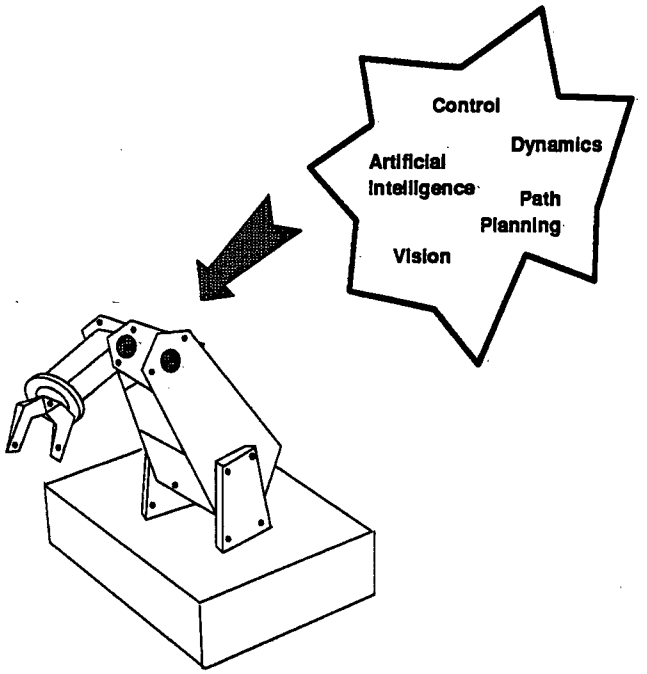


Figure 1.1: Computational Demand Of Advanced Robots

As illustrated in Figure 1.1, robotic algorithms will require high computational power that is beyond the capabilities of modern conventional computers. To illustrate the computational effort needed, consider processing of a 512 $\times$ 512 pixel image. On a conventional computer such as the VAX 11/780, which is capable of 1Million Instructions Per Second (MIPS), processing time will vary from 100 seconds at 100 operations per pixel to 2.78 hours at 10,000 operations per pixel for every frame [2].

To support adaptive control algorithms, sensor information may have to be sampled at rates up to 5 KHz [3]. The necessary arithmetic operations for these computer intensive adaptive control equations must be performed at a throughput of approximately 30 MFLOPS to meet a real-time deadline of 200 us [4] [5].

The demand for computing power coupled with the continuing advances in hardware technology have motivated researchers to investigate highly parallel computing to attain acceptable real-time performance.

1.1 Highly Parallel Computing

The two driving forces in the field of parallel computing are the recent advances in **VLSI technology** and the improvements in the state of **computer architectures.**

Improvements in **VLSI technology** such as the advances in semiconductor materials and fabrication techniques have increased the density of wafer-scale integration. Today's supercomputers as the CRAY X-MP and Fujitsu VP-200 [6] are limited in its performance by off-chip delays. The use of higher density packaging may reduce the propagation delays, but is limited by power dissipation requirements. Researchers, however, have overcome this limitation by improving the performance of computer architectures.

The conventional single processor is characterized by the Von Newmann model of a serial processor. The main performance limitation of the Von Newman architecture is the memory access time. To overcome this limitation, the Von Newmann architecture was enhanced with pipeline techniques in instruction units and vector processing as used in the Cray X-MP [6]. A pipeline divides operations into a sequence of simpler operations called stages. Operations at each stage are performed by individual hardware units which act in similar fashion to an assembly line. A K stage pipeline should theoretically improve the system throughput by a factor of K. But the number of stages is limited to the number of partitions that a given operation requires, and hence this defines the task parallelization limit. Also, the performance of the pipeline is dependent on keeping the pipeline full, data dependencies between stages will limit the propagation of data at the speed of the slowest stage. A vector processor will execute an instruction on a vector of data. If a computation cannot be

vectorized, the performance of a vector computer is rapidly reduced to its scalar equivalent. Therefore, pipelining and vector processing can only be applied to a subset of the overall application domain.

A more general approach to pipeline and vector processing is to subdivide the application program into several parallel jobs. Each job is executed on separate processors which are combined to form a **parallel architecture.** In the spectrum of parallel architectures, there are three primary divisions which are based on the **number** and **complexity** of the processing elements [7]. Parallel architectures may be designed with:

1. Very large number of simple processors:

The processing elements are very simple and comparable to bit serial processors. Each processing element is useless by itself but the computing power may be large when many are coupled together. Reeds [7] compares this type of architecture to a large colony of termites devouring a log. A representative example of this approach is the Thinking Machine which uses 65,536 1–bit processing elements [8].

2. Small number of powerful processors:

Each processing element is very complex and use the fastest available VLSI technology. The processors are equipped with pipeline and vector processing units. To pursue Reed's analogy, this design is similar to four woodsmen with chain saws. As an example, the Cray 3 architecture is composed of 16 processors designed with Gallium Arsenide technology and processes data at a speed greater than 10 GFLOPS [9].

3. Large number of microprocessors :

This approach is an intermediate between the extremes described above. This is similar to a small number of hungry beavers. The development of this type of architecture was led by the advent of fast microprocessors and inexpensive memory. In many of these systems, each node contains a processor with some locally addressable memory and a small number of links to connect to other nodes. Examples of such systems are the FPS T-series [10], the iPSC hypercube [11], and the NCube [7]. This category also includes architectures such as Condor [12] where several microprocessors communicate via a shared bus. Parallel architectures with a number of microprocessors greater than

two that are sharing the computational load of an application will be referred in this thesis as a **distributed system.**

1.2 Levels of Parallelism

Increasing the computational power with parallel architectures involves parallel execution of the problem. The **granularity** of a parallel architecture defines the size of the processes which constitute a parallel computation. The difficult question is where to extract the parallelism? Parallelism may be found at the following levels :

1. job or program level (large grain)
2. task or procedure level (medium grain)
3. instruction level (fine grain)

The granularity of a system is closely interrelated with the complexity and the number of processing elements which form a parallel architecture. A problem that is parallelized at the instruction level usually requires a very large number of simple processing elements. At the other extreme, if the level of parallelization of a problem is at the program level, a few powerful processors would be best suited for this type of application.

There are compromises between the extremes of granularity. Fine grain architectures involve overhead to support communication between small processes and a too large a process size will reduce the scope of parallelism. The extraction of parallelism, however, greatly depends on the problem to be solved and the most efficient solution may not be unique.

Amdahl concluded that the speedup is limited by the amount of parallelism inherent in the algorithm [13]. Speedup (S) is defined as the ratio between the execution time using a single processor and the execution time using multiple processors. The maximum Speedup S_{max} of P processors executing an algorithm is expressed by:

$$S_{max} = P/(fP + 1 - f)$$

where factor f represents the fraction of the computation that cannot be parallelized. Note that when f=1, S_{max}=1 which indicates that the problem cannot be partitioned. On the other hand if the problem is completely parallelizable (f=0), the speedup S_{max}=P.

Amdahl defined the effectiveness of a parallel architecture E by: $E = S/P$. This last equation shows that the efficiency of a parallel architecture is limited by the fraction of the computation that cannot be parallelized.

The level of parallelism that best fits the problem cannot be selected without a careful examination of the application. The following section presents a computational analysis of various robotic algorithms.

1.3 Computational Analysis of Robotic Algorithms

This section briefly examines areas of robotic application such as control, vision, sensing and planning. An overview of proposed parallel architectures, necessary to solve these functions is also presented.

1.3.1 Algorithms for Control

Today's industrial robots are usually limited to position and velocity control. Tomorrow's state-of-the-art robots will interact with the environment through vision or other external sensors. More complex control algorithms based on position and force are constantly being developed. A typical control system must calculate computationally intensive algorithms for the **inverse kinematics** and the forward and inverse **dynamics.**

Inverse kinematic equations are dependent on the manipulator position. These equations involve many trigonometric calculations. Given an end effector position in environment coordinates (usually Cartesian), the required joint angles to achieve this position must be determined. This relationship is a one to many mapping which uses a Jacobian between Cartesian and rotational coordinates.

Dynamic equations are tightly coupled and non-linear. Forward dynamic equations calculate the position of the end effector given a set of joint forces or torques. The inverse dynamics, however,

finds the joint forces or torques which produce the desired trajectory of the end effector. The Newton-Euler equations are generally regarded as the most efficient method for the evaluation of dynamics [3]. Various levels of parallelism have been proposed in this area.

Nigam and Lee [14] have developed an algorithm which extracts parallelism at the job level based on a **multiprocessor** robot controller. A multiprocessor architecture is composed of processors connected to memory boards via a common bus.

Medium grain parallelism has been explored by Wang [3] where independent parts of the Newton-Euler equations are processed concurrently. A similar approach has been proposed by the Oak Ridge Laboratory where the Newton-Euler inverse dynamic equations are computed on a distributed system configured in a **hypercube network** [15]. Each processing element in a hypercube network has local memory and they are connected in a binary n-dimensional cube topology using external links. The hypercube architecture is described in more detail in the next chapter.

Geffin [16] proposed to extract parallelism of the Newton-Euler state space formulation at the instruction level. The concurrent processes would execute on a **dataflow architecture** which, in this case, consists of 2000 processing elements. The Dataflow computer is usually labelled as a non-Von Newmann architecture. An instruction is executed only when all operands are ready within an operation. When many operations of a program are ready for execution, they are assigned to individual processing units for execution. A good survey of dataflow computing is given by Dennis [17]. The main disadvantage of this architecture is that the overhead, needed to distribute the tasks to the processing elements, is too demanding when the computer contains a thousand or more processing elements. In addition, the fine granularity of the computation implies that many processing elements are needed to achieve speedup.

Lee and Chang [18] [19] have derive algorithms for forward and inverse dynamic to be implemented on an **SIMD** computer. In an SIMD parallel model, a single instruction is executed in parallel by several processing elements in lock-step. Multiple arithmetic units operate in parallel and perform the same operation on a different sets of data. The Thinking Machine [8], for example, is classified as a SIMD architecture. The main drawback of the SIMD architecture is that the system can only execute one instruction stream at any time. If a program does not have enough data streams,

some processing elements in the system remain idle, and hence, may lead to severe degradation of the system performance.

1.3.2 Algorithms for Vision and Sensing

The main task of robotic vision is to provide information about the position of objects in space for path planning and control. The task of a robot extends over time, consequently a sequence of images must be sampled at a rate greater than external events. The robot must also capture the spatial occupancy of objects in its environment in order to avoid collisions. Mudge [2], highlights robot vision algorithms and classifies them into three categories: low, intermediate and high level processing.

In **low level** processing, the image is enhanced and features of the image are detected. Deterministic operations are applied to each pixel in the image, whereby the value of each pixel is dependent of the values of the surrounding pixels. When the image is subdivided into equal areas, each area requires the same processing time. Low level vision processing is well suited for fine grain computations where each processing element computes a small area of the image.

Systolic arrays and array processors are special purpose architectures well suited to low level processing algorithms. Systolic and array processors are processors which precede the host computer. **Array processors** are commercially available and are generally high speed processors which are optimized for vector and array operations. A **Systolic Array** is a parallel architecture where all processing elements are interconnected in a two-dimensional grid, and the output results of the computation are passed to one or more neighbouring processors. All processing elements execute synchronously. Kung presents an overview of systolic arrays [20]. The major drawback of a systolic array is that an adequate amount of data from the outside world must be fed into the array in order to achieve a high computational bandwidth. When implemented in hardware, the I/O of a Systolic array limits its performance.

The **intermediate level** processing extracts features from the enhanced low level image. Algorithms in this category extract the edges and organize them in connected segments. The last stage, **high level** processing, involves algorithms for object recognition.

The data representation of the image is not uniform. Edges and objects to be extracted from the image and may be found at any part of the frame. Array processors and systolic arrays are not efficient for this type of problem due to the uneven workload. Mudge [2] and Jones [21] extracted parallelism at the job and procedure level and distributed the jobs to a small number of microprocessors connected in a hypercube topology.

The University of Pennsylvania REPLICA project investigated the performance of a special-purpose computer for sensing applications [22]. Eva Ma concluded that the best solution to parallelize the problem is not unique. Consequently, a dynamically reconfigurable and partitionable computer system called REPLICA was designed. When a problem similar to low level vision processing involves the execution of a sequence of similar operations on different subsets of input data points, the processing elements are partitioned into an SIMD computer. To perform edge detection or object recognition algorithms, the REPLICA architecture can be reconfigured as a distributed system. A **distributed system** is more efficient than SIMD architectures for processing non-uniform workloads. The operating system overhead to partition REPLICA architecture dynamically is high, and hence its use in real-time application is not feasible at the moment.

1.3.3 Algorithms for Path Planning

The workspace of a robot may contain obstacles. The robot must avoid these obstacles when executing a task. The path planner ensures that the robot takes a collision free path when moving from a starting position to a target position. Poon [23] proposed to extract parallelism of the path planner at the procedure level. The path planner program is represented by a coordinator process and a set of collision detector processes. Each collision process knows the position of a subset of obstacles and subset of robot surfaces. Each time the the robot must move, the collision processor must check if the path is collision-free with its subset of objects. The structure of the computer architecture proposed by Poon is a distributed system where each processing element has local memory and communicates via communication links.

From the above computational analysis of various robotic applications, researchers have extracted the parallelism of robotic algorithms at all three levels: program, procedure and instruction. In light of the above, we can see that there is not a unique solution to the problem of mapping robotic algorithms to a parallel architecture. However, it seems that a distributed system is the most suitable architecture, at least for program and procedural level of parallelism.

A very high speed parallel architecture with a small number of of powerful processors is not a feasible cost effective solution since they are extremely expensive to buy and maintain. Also, real-time computing is not necessarily equivalent to fast computing [24]. Supercomputers, for example, may have difficulty to respond to many external events or I/O operations and still meet the real-time constraints of a robotic system.

Architectures which extract parallelism at the instruction level have also major drawbacks. Dataflow architectures suffer large performance degradations when there are many processing elements. The formulation of robotic algorithms which deal with matrix computation appear to be favorable for Systolic Arrays, however, the I/O of the Systolic Array severely limits its performance. The fact that all processing elements of a SIMD architecture can only execute the same instruction

in lock-step is clearly limited to a small subset of the application domain. Consequently, a very large number of simple processors is not suitable for general robotic applications.

Powerful distributed systems may be built with low cost microprocessors. They are simple to implement and easy to maintain. A distributed system efficiently maps algorithms that are parallelized at the procedural and program levels. For fine grain computation, a distributed system may simulate an SIMD architecture where each processing element has an identical program running in parallel. As an example, Mudge [2] developed an algorithm which calculates low level vision problems with a distributed system programmed as a SIMD architecture. The performance results were comparable to an array processor.

1.4 Distributed Architecture: A Definition

A distributed architecture consists of:

1. a large number of microprocessors connected in a given topology,
2. processing elements which are programmable and can execute their own programs asynchronously,
3. a parallel architecture well suited for medium grain computation,
4. processing elements which communicate either via shared memory or message passing.

Currently, application programs must be manually decomposed in concurrently executing tasks. There is a large spectrum of distributed architectures. Distributed architectures may be classified in two main categories depending on how the data is exchanged between microprocessors. Processors may communicate either via message-passing or shared-memory.

Shared-Memory

In a shared-memory architecture, the entire information is directly accessible by every processor. A shared-memory distributed system contains typically less then ten processing elements connected to shared memory via a high speed bus such as VME or Multibus.

Communication over shared-memory provides an excellent medium for broadcasting information. This simple architecture, however, is efficient only for small numbers of processing elements.

Memory contention becomes a significant bottleneck as the number of processing elements increase. To alleviate the memory contention problem, the single bus is replaced by multiple busses or a high speed interconnection network. But, these enhancements increase greatly the cost and the complexity of the architecture. In robotics, shared-memory distributed systems were used by [12] and [3]. Because of these limitations, a shared-memory system was not considered in the present study.

Message-Passing

Each node of a message-passing architecture contains some locally addressable memory, a communication controller capable of routing messages without delaying the processor, and a small number of connections to other nodes. Asynchronous tasks executing on different nodes communicate via message passing. The message is transferred via a communication network. The complete message is transferred explicitly from one processor to another. Notice that, unlike the shared memory technique, the message-passing mechanism does not require arbitration or synchronization scheme to ensure the validity of the data. Examples of message-passing architectures used in robotics may be found in [15], [2], and [21].

A message-passing architecture has the important advantage of not being limited by the number of processors that can be added to the system. As the number of processing element increases, the communication bandwidth of the system also increases. **Expandability** is a desirable feature for an architecture designed for robotic applications.

The computing power required to control next generation robots is still unknown. For instance, the original version of the architecture for the Utah-MIT hand consisted of six Motorola 68000 based technology linked via a tightly-coupled Multibus. Narassimhan [25] stated that the architecture was already saturated with low level hand servo calculations. Hence a second version of the system called Condor, was based on Motorola 68020 processors, which resides on a VME bus and has floating point support [12]. This architecture has also reached its limit and the architecture will be expanded with microprocessor nodes that will communicate with Condor via a message passing protocol called Ganglia.

Although shared-memory bus based systems are attractive for their simplicity, problems arise in

the form of limited scalability, contention for accessing shared-memory, and rising cost for overall speed gain. Since the absolute computational requirements for newly developed robotic applications are still uncertain, a message-passing distributed architecture represents a better cost-performance parallel system. A processor of a message-passing architecture may also be directly connected with external sensors. This arrangement decreases communication latency between the I/O boards and the processors which process the I/O data. Message-passing distributed systems have other important advantages which are **reconfigurability** and **fault-tolerance.** These issues will be discussed in greater detail in the following chapter.

1.5 Interconnection Network

An important component of a message-passing distributed system is the interconnection network which supports communication among the processing elements. Since the interconnection network determines the communication delays among the processing elements, it directly affects the system performance. Maximum system throughput is achieved if each processor receives the data it requires with minimum delay. The interconnection network also plays an important role in system reliability. The architecture of an interconnection network is categorized by the network topology, the control strategy and the switching method [26].

A network topology may be either **static** or **dynamic.** In a static structure, links between two processors are permanent. Therefore, a message may have to be routed through intermediate nodes if a processor needs to communicate with another processor not directly connected. Alternatively, some systems may allow the topology to be reconfigured to provide direct connection. A dynamic topology is implemented by connecting communication links to switching elements. A dynamic topology is clearly the more powerful topology.

The setting of the switches may be either **centralized** or **distributed.** If control-setting of the switches is executed by a centralized controller, the control strategy is centralized. A fully distributed control strategy is where the setting of the switches is made by the individual switching elements.

The two major switching methodologies are **circuit switching** and **packet switching.** In circuit switching, a physical path is established between the sender and the receiver. In packet switching,

no physical path is established in advance between the sender and the receiver. Instead, the message is divided in packets, and routed to its destination via intermediate nodes. Thus, a circuit switching strategy is attractive to real-time systems since it provides a maximum communication latency and guarantees a minimum bandwidth once the circuit is established.

1.6 Scope of the Thesis

This thesis investigates the design options for the hardware and software of a distributed system for robotics and automation and evaluates the performance of an implementation.

1.7 Thesis Overview

Chapter 2 discusses design objectives, such as predictability, reliability, reconfigurability, expandability and development support, to be achieved in an implementation of a message-passing distributed system for robotics. A discussion on the various hardware and software design options to meet these objectives is then presented. Hardware issues such as interconnection networks, node architectures, and reconfigurability are discussed. An overview of the software design options is also presented. These options are: synchronous versus asynchronous communication, centralized versus a distributed control strategy, packet switching versus circuit switching, reconfiguration modes, addressing modes and finally software support to achieve reliability. This chapter concludes with an outline of the desirable hardware and software design options selected for an implementation of a distributed architecture.

In Chapter 3, hardware and software implementations of a message passing distributed architecture is presented. The hardware architecture is based on transputer modules interconnected via a reconfigurable topology. The interconnection network is composed of both bus and point-to-point communication links. Inter-node messages are routed via point-to-point communication links and the bus architecture is mostly used for operating system functions and reliability. The implementation of an operating system which provides communication and input/output support is presented. The operating system communication protocol is synchronous, the selected switching mechanism is circuit switching and the reconfiguration control strategy is centralized.

The communication performance of the architecture is evaluated in Chapter 4. Performance tests which isolate hardware and software overhead are presented and the results are compared with Helminen's store and forward model [1]. Helminen's communication protocol is asynchronous and runs on the FPS-T series hypercube, which is a transputer-based distributed system.

Chapter 5 concludes the thesis with a summary of results and suggestions for future areas of research.

Chapter 2

Design Requirements of a Message-Passing Distributed System for Robotics

There are various requirements on the design of an architecture for robotic control. The previous chapter discussed the need for substantial computing power and proposed a message-passing distributed architecture as a suitable cost-performance solution. A distributed architecture, however, should also includes the following features:

1. Predictability;
2. Reliability;
3. Reconfigurability;
4. Expandability, and
5. Development support.

This chapter will discuss the above architectural design objectives and will evaluate various hardware and software approaches to meet these objectives. For the remainder of this thesis, the term **distributed system** will refer to a message-passing distributed system.

2.1 Design Objectives

2.1.1 Predictability

The objective of real-time computing is to meet the individual timing requirements of each task. A real-time system must be predictable, that is, the timing behavior of concurrent processes must be deterministic [24]. The predictability of a system is determined by the algorithm, the architecture, and the operating system.

An **algorithm** is difficult to trace when it is parallelized into concurrent processes that interact with external events. A parallel **architecture** provides a higher throughput than today's sequential computers, but this does not guarantee that the timing constraints will be met. Other factors such as interrupt latencies, communication delays and input/output support must also be considered. The **operating system** of a parallel architecture should also provide fast and deterministic scheduling

support. Fast scheduling requires efficient context switching capabilities. A context switch occurs when a running process is descheduled to allocate the cpu resource to another concurrent process. A scheduling algorithm is deterministic when it guarantees that tasks are scheduled within a maximum time delay.

2.1.2 Reliability

Robots may have to execute tasks in hazardous environments. Failure to meet the real-time constraint can result in dangerous situations where massive loss of equipment or even human life may occur. The architecture must provide capabilities for fault detection and recovery or shut down of the system in a controlled, fail-safe manner.

2.1.3 Reconfigurability

In the previous chapter, various algorithms to parallelize robotic applications such as control, vision and path planning were presented. The algorithms require different interconnection networks to support efficiently their communication pattern requirements. To achieve efficient mapping of the communication pattern of an application to an interconnection network, it is desirable that the interconnection network be reconfigurable. A reconfigurable system can be matched to the problem to be solved.

Reconfigurability is also important for fault tolerant computing. The failure of a component in an interconnection network can bring down the entire system or cause severe performance degradation unless sufficient measures such as redundant communication paths or reconfiguration are provided.

2.1.4 Expandability

Adaptive control, development of flexible joint robots, vision, and artificial intelligence are active research areas in robotics and automation. For applications such as these, a parallel architecture should be easily expandable in order to support an increased demand in computing power.

2.1.5 Development Support

A good development environment is necessary for efficient software design, implementation and testing. The development environment of a distributed system should provide:

1. access to high level languages and debugging tools;
2. concurrent execution of programs from multiple users;
3. an easy to use programmer's library for inter-process communication; and
4. concurrent input/output capabilities where each processing element may directly access secondary storage and be connected to a terminal.

2.2 Survey of Multiprocessor Hardware

There are two major hardware issues in the design of a distributed system. The first is the design of the **node architecture** which supports communication among the processing elements and the second is the design of the **interconnection network** architecture. This section will begin with a description of a node architecture followed an overview of static and dynamic interconnection networks. The architecture must be resilient to failures, and hence techniques and hardware enhancements for achieving fault-tolerance in distributed systems will be presented. Following this discussion, the node organization and interconnection network of existing distributed systems will be described and advantages and disadvantages of the various designs will be discussed.

2.2.1 Node Architecture

A typical **node** in a distributed system contains:

1. a microprocessor;
2. locally addressable memory;
3. a communication controller to support routing of messages;
4. communication links connected to other nodes; and
5. optional hardware such as a vector processing unit or a floating point co-processor.

There are two main approaches to the design of a node for a distributed system. The first approach is to use an existing microprocessor such as a Motorola 68020 as in the Mark III hypercube [27], or an Intel 80286 as in the iPSC hypercube [11], and then enhance the processor with additional logic to manage communication. The second approach is to integrate communication capabilities in the design of the microprocessor as in the Inmos Transputer [28] and the NCube [7]. For example, the T800 Inmos transputer is a **single chip** microprocessor containing 4k bytes of memory, a 32 bit external memory interface, four full duplex communication ports, and a 1.5 MFLOPS floating-point co-processor.

A detailed description of the node architecture of distributed systems with each node design approach is presented in section 2.2.4. It will be shown, that the main advantages of a processor with integrated communication facilities is the simplicity of the node design and the compactness.

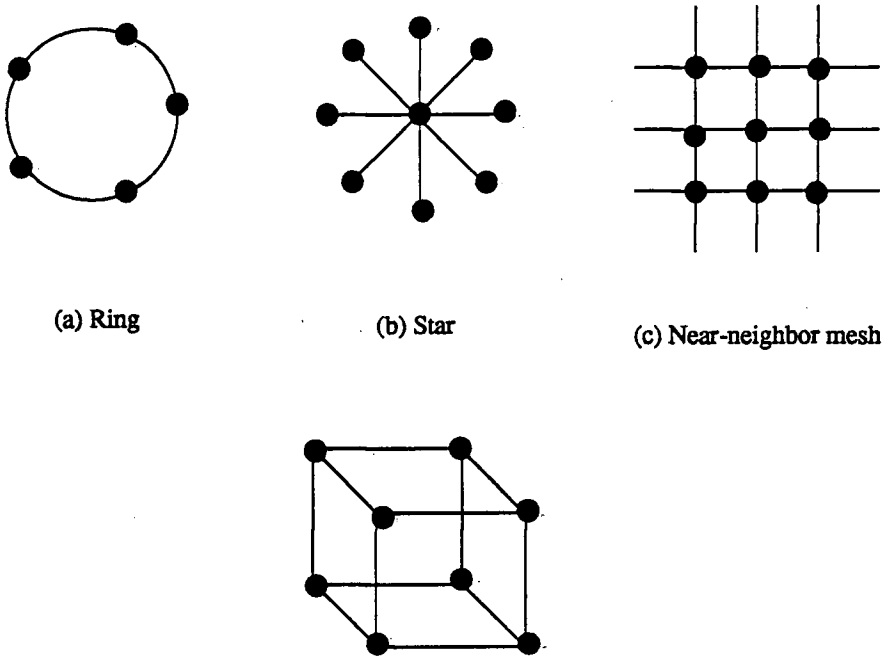
2.2.2 Interconnection Networks

2.2.2.1 Static Topology

In a static topology, links between processors are fixed. Static topologies are classified according to the dimension of the network [26]. One of the simplest two-dimensional interconnection schemes is a ring in which each node is connected to two others in a circular topology. Other two-dimensional topologies include star, tree, and near neighbor mesh as presented in Figure 2.2(a)(b) and (c).

The main drawback of these simple topologies is that the **network diameter** is relatively large. The network diameter is the maximum number of links that must be traversed to transmit a message to any node along a shortest path. For example, a message routed on a network of P processors connected in a ring topology may have to traverse P/2 hops before reaching its destination. Each hop crossed by a message involves some intervention by the processor at that node and this consequently decreases the throughput of the system and increases communication delays.

To decrease the network diameter, a n-dimensional network topology known as n-cube or hypercube has been developed. The dimension of a hypercube determines the number of nodes in the network, the network diameter, and the number of communication links connected to each node. A hypercube of dimension d has a network diameter d, and is composed of 2^d nodes, where each node has d connection links. A three dimensional hypercube is presented in Figure 2.2(d). Examples of commercial hypercube distributed systems are the Intel iPSC [11], the Amteck System/14 [7], the FPS-T Series [1], and the NCube/ten [7]. The hypercube topology, however, has one major drawback: the number of connections required at each node grows with the the hypercube dimension. Consequently, the expansion of a distributed system connected in a hypercube is limited by VLSI packaging technology.



(a) Ring (b) Star (c) Near-neighbor mesh

(d) 3 dimensional hypercube

Figure 2.2: Static topologies

2.2.2.2 Dynamic Topology

A reconfigurable network is composed of communication links connected to switching elements. A switching network may be either blocking or non-blocking. A communication network is called **blocking** when messages contend for communication resources. A **non-blocking** network provides the resources to handle all possible connections between any node pairs with no conflicting path.

A **crossbar network** is an example of a non-blocking reconfigurable network. In a crossbar network, all communication links are connected to a switch and a non-blocking path between any pair of nodes may be created with a maximum of one switch delay. The hardware cost, however, grows as $O(N^2)$ for a system with N inputs and N outputs. Examples of architectures using crossbar networks are the Supernode [29] and Parsifal [30] computers.

In order to reduce the switching complexity, communication links among the processors can be shared. The penalty for this reduction in complexity is that the network becomes blocking. A blocking network suffers longer communication delays than a non-blocking network as a result of contention

for link resources. There are two main approaches in the design of blocking interconnection networks. The first approach is to design a **multistage network** composed of a large number of simple 2×2 switches. Multistage networks provide interconnection of N devices at a cost of switching circuitry that grows as $O(N\log_2 N)$. The Star architecture developed at the University of Texas is an example of an architecture design with a baseline multistage network [31]. A baseline multistage network is illustrated in Figure 2.3(a).

A second method of reducing the switching cost of a crossbar network is to design a **distributed crossbar** communication structure as illustrated in Figure 2.3(b). Distributed crossbar networks support non-blocking communication when there is no inter-cluster communication and become blocking when messages are sent between clusters. This interconnection scheme was used by the Connection Machine. In the Connection Machine, a cluster is composed of 16 processors connected to a switch. Each switch has twelve connections for inter-cluster communication and the switches are interconnected in a 12 dimensional hypercube. This distributed cluster interconnection network can support communication among 65,536 processors [8].

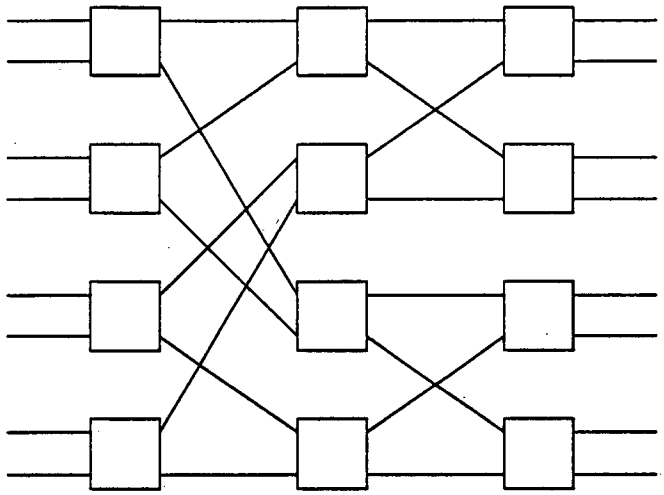
A distributed crossbar network is well suited to applications such as robotics where the problem can be subdivided into functions and the majority of interprocess communication messages are within processes of the same function. In robotics, functions such as sensing and vision, control and path planning may be executed on a cluster of processors where each cluster will be interconnected in a distributed crossbar configuration. With the high communication locality pattern within each function, the majority of messages will not contend for shared resources.

The main disadvantage of a dynamic topology over static topology is the switching hardware complexity. A dynamic topology, however, has the following advantages.

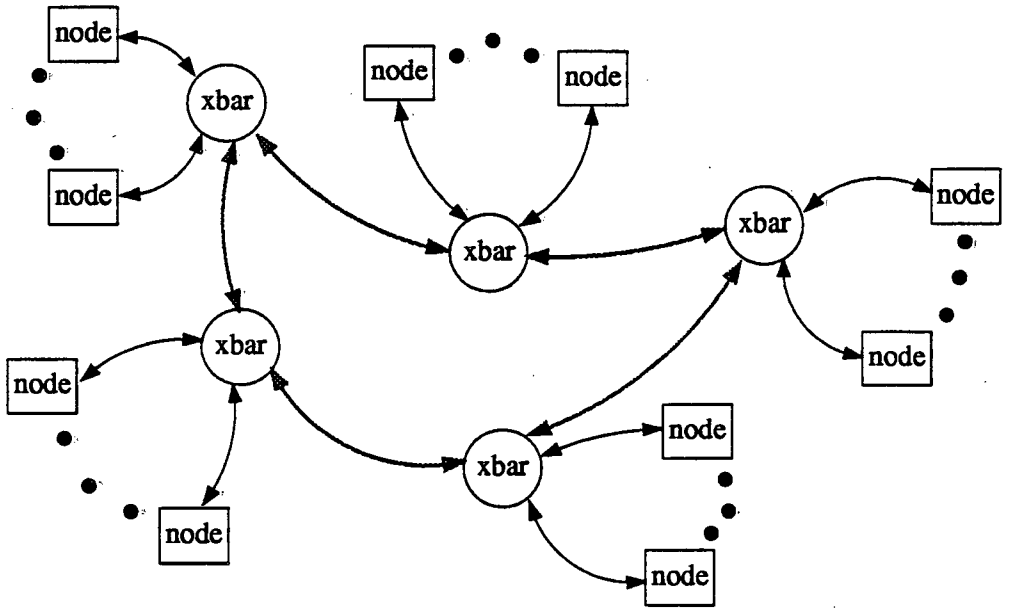
Increased performance:

A dynamically reconfigurable topology may efficiently support communication patterns which are either local or non-local. A communication pattern is said to be local if most messages are sent between neighbor processors.

If the communication pattern becomes non-local in a static topology, a message will have to traverse many hops, prior to reaching its destination which increases communication latency. Also,



(a) 8 x 8 Baseline Network



(b) Crossbar Network

Figure 2.3: Blocking interconnection networks

as the anti-local message traffic increases, the nodes which are in the path of the messages would tend to work primarily for message routing at the cost of all other local processes.

In a dynamic topology, however, the communication network may be reconfigured to provide a route for any messages without requiring any processing support of other nodes in the network.

Design flexibility:

If the interconnection network is static, programmers must develop algorithms where the communication pattern maps into the interconnection network. This limitation decreases design flexibility [32]. A flexible interconnection topology, however, will alleviate such problems by matching the topology to the communication pattern of the algorithm.

Message-passing flexibility:

A reconfigurable network can provide a direct communication path between any node pairs. If a message has to traverse many hops to reach its destination, a dynamic topology provides the flexibility to either reconfigure the network to create a direct connection or to simply forward the message through other nodes as in a static network.

Fault-tolerance:

If a link or a node fails, a dynamic topology provides fault recovery through physical reconfiguration of the network.

2.2.3 Hardware Organization for Fault-Tolerant Architecture

In this section, a review of techniques for achieving a desirable level of hardware fault-tolerance in a distributed system is presented. Fault masking and being able to detect, diagnose and recover from a failure are two methods currently used to attain fault-tolerant hardware designs [33]. The efficiency of some techniques will depend on the interconnection network characteristics.

2.2.3.1 Masking

Masking is a method which achieves hardware fault-tolerance by adding either hardware or software redundancy to the system. Masking is an active form of redundancy where a faulty system is transparent to the user. To mask a fault, a n-modular redundancy technique based on a voting scheme is widely used. The redundancy may be implemented at the hardware level with the use of redundant nodes or at the application level where processes are replicated.

The main advantage of this technique is that a fault may be corrected with minimal disruption of the system. It is well suited for a static interconnection network. On the other hand, masking introduces a large overhead if used in a large multicomputer network. For example, to design a fault-tolerant network at the node level with a 3–modular hardware redundancy technique, each node must be replicated 3 times. The overhead is not only hardware, but also a large number of messages must be replicated which increases the communication load. Masking techniques may become very complex and costly for large systems. They require a large amount of resources during fault-free operation in order to ensure minimal disruption of the system if a fault occurs.

2.2.3.2 Detection, Diagnosis and Recovery

To achieve a greater cost efficiency, the amount of resources dedicated to fault-tolerance activities could be reduced and a larger overhead after the occurrence of a fault could be accepted. This option is called fault detection, diagnosis and recovery. Using this scheme, a failure must be detectable. Once detected, the fault is located, the system is repaired and recovery procedures take place.

The **detection** of a failure may be either internal or external to the system. The internal detection scheme places the failure detection mechanism for each node within that node. For example, a node may contain hardware for memory error detection and correction. Using internal detection techniques to detect all possible failures would greatly increase the node complexity since processing elements are complex entities, with a large number of internal states and many potential failure modes.

The alternative approach, external detection, implies that a facility external to the node is responsible for detecting a node failure. One method is to have each processor communicate with a neighbor processor which performs testing and status monitoring. Although this method does not require any additional hardware, it has one major disadvantage. In a real-time system, it is crucial that a fault be detected rapidly to prevent the fault from propagating through the system and creating a disastrous situation. Consequently, a node must execute fault-detection checking at frequent intervals. This will decrease the throughput of the distributed system.

Another method of implementing external fault detection was proposed by Wong [34]. The interconnection network would be enhanced with a global bus which would provide access to

each processor's memory. A central diagnostic processor would be able to test network processors independently from the application processor. Ideally, each application processor should be designed with dual-ported memory to allow the testing to be completely transparent to the application node. The main disadvantage of this fault detection approach is the increased complexity of the hardware. This option, however, is well suited to a distributed real-time system where fault detection algorithms may be executed independently of the application program.

Also, the central diagnostic processor with a shared bus has complete knowledge of the overall state of the system which will allow it to compute a single, centralized **diagnosis** for the system. Diagnosis refers to the process of determining the cause and location of the fault. The maintenance processor may directly access any node in the network which increases the confidence in the testing information. If testing data had to traverse a faulty intermediate node, it is highly probable that the testing information would be corrupted. This effect greatly increases the complexity of the diagnosis algorithm[33].

Recovery from a failure may be performed either by hardware reconfiguration or logical reconfiguration. Logical reconfiguration of the system implies that each fault-free processor is aware of the faulty processor, switch or link and avoids interaction with the faulty component. This method is usually used for static interconnection networks and involves broadcasting information about the faulty component to a large number of nodes. The broadcasting algorithm must ensure that all destination nodes received the message properly which increases the reconfiguration latency time [35].

Hardware reconfiguration involves reconfiguration of the interconnection network to bypass faulty components. A good survey of various reconfiguration strategies has been written by Yala [31]. A centralized hardware reconfiguration strategy is able to reconfigure the network efficiently without any broadcasting of data in order to recover from a fault. Consequently, hardware reconfiguration is attractive for use in real-time systems.

In summary, the above discussion shows that dynamic reconfiguration of an interconnection network provides various advantages ranging from increased performance through design flexibility and programming ease to greater fault-tolerance.

Achieving fault-tolerance using masking techniques is the most efficient recovery technique but the hardware cost is too high when applied to a distributed system. Minimizing the resource redundancy in the design of a hardware fault-tolerant architecture induces a longer recovery period. It can be seen that a distributed crossbar network enhanced with a bus, a central maintenance diagnostic controller, and a centralized reconfiguration strategy represents an efficient approach for fast error detection, diagnosis and recovery as required in real-time systems.

2.2.4 Research and Commercial Distributed System Architectures

2.2.4.1 NCube/Ten

The commercial NCube/Ten [7] is a ten–dimensional hypercube architecture which contains a microprocessor designed expressly for distributed systems. Each node consist of an NCube microprocessor with 256K byte of memory, and eleven communication channels. The main advantages of this architecture is the simplicity and compactness of the node design which make the system easily expandable. The main disadvantages of this architecture, however, are:

1. The interconnection network topology is not reconfigurable.
2. Due to the pin limitations, NCube only supports 17–bit physical address space. Also, 32 bit data must be accessed as 16–bit half words. The maximum speed and accuracy at which a robot can move is highly dependent of the speed and word width of the computer architecture. Wang [3] emphasised the importance of microprocessors with 32–bit data width for robot control computers.
3. A maximum of 256K of memory is available at each node. In robotics, there are applications where this small amount of memory may not be sufficient. For example, in a path planning algorithm, a processor may have to store object positions that are part of the robot environment. If the environment is complex, the memory required to store all the information is likely to exceed 256K.

2.2.4.2 JPL Mark-III

Mark III is a research hypercube architecture developed at Caltech [27]. The Mark III node architecture uses a standard MC68020 microprocessor and enhances it with communication links. The node architecture is illustrated in Figure 2.4. Each node utilizes two Motorola MC68020 microprocessors, a MC68882 scalar floating-point co-processor and a Weitek 8000 floating point chip. One MC68020 serves as the application processor and the other MC68020 is dedicated to communication.

The main problem of this architecture is the complexity of the node design. Each node is composed of three circuit boards. Distributed architectures with special microprocessors with on-chip communication capabilities like NCube and Transputer can package multiple nodes on each board. The Mark III topology is also static and is limited to 128 nodes.

2.2.4.3 Supernode

The Supernode [29] is a research architecture which consists of 16 Inmos T800's where each node is connected to a bus and to a switch. All four links of each node are connected to a 72×72 crossbar switch, and the switch is controlled by an additional transputer. The shared bus is currently used as a debugging facility. A program running on the switch controller transputer can set up any interconnection network in a non-blocking mode for all links. Each transputer has 256 Kbyte of external memory and each Supernode also contains a transputer with 16 Mbytes of memory which can be used as a host processor for storage and distribution of data and code. The Supernode architecture is illustrated in Figure 2.5.

A block diagram of the Inmos T800 transputer is shown in Figure 2.6. The T800 includes 4K bytes of on-chip static RAM, a hardware timer, a floating point unit conforming to IEEE standards and four link interfaces. The link interfaces use direct-memory-access controlled, and can operate up to 20 megabits per second.

The notions of concurrency and processes are supported by the transputer instruction set. The scheduler is a hardware facility that supports two levels of process priorities. The micro-coded

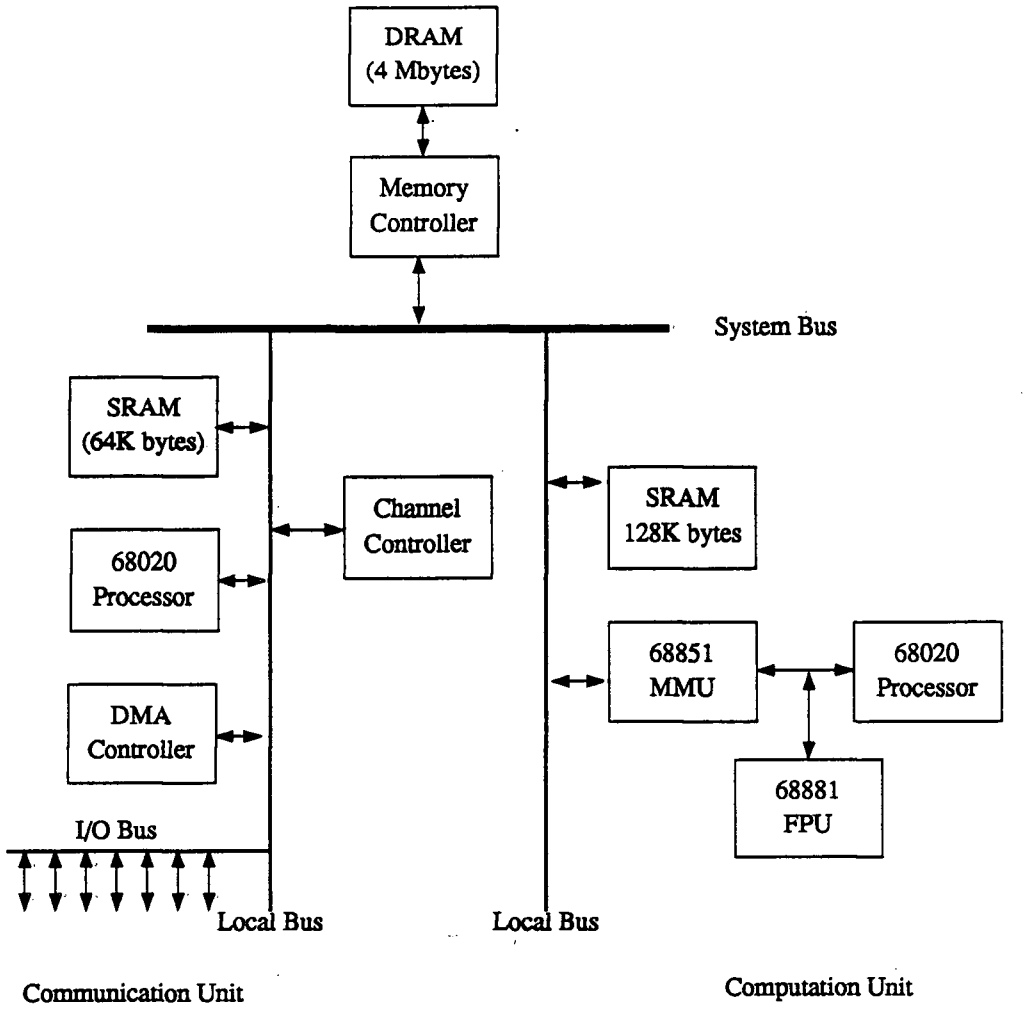


Figure 2.4: JPL Mark III node architecture

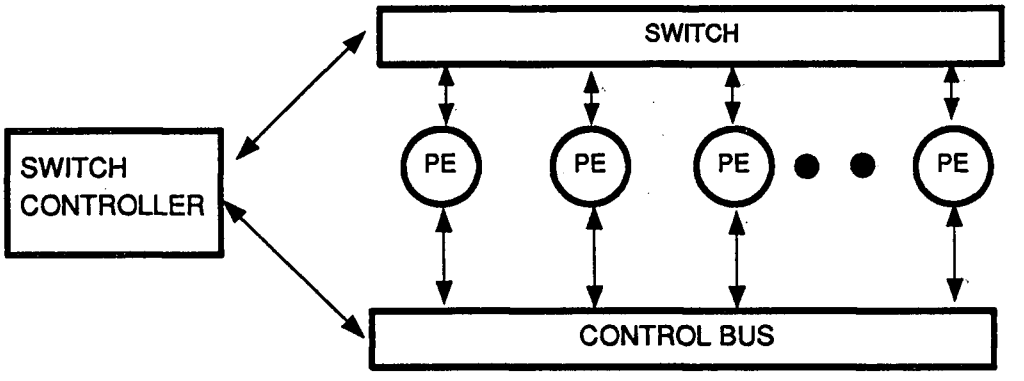


Figure 2.5: Supernode reconfigurable crossbar network

scheduler can execute a context switch in 1 to 2.5 microseconds [36]. For comparison, the Unix operating system has a context switching time of few milliseconds. The transputer also provides

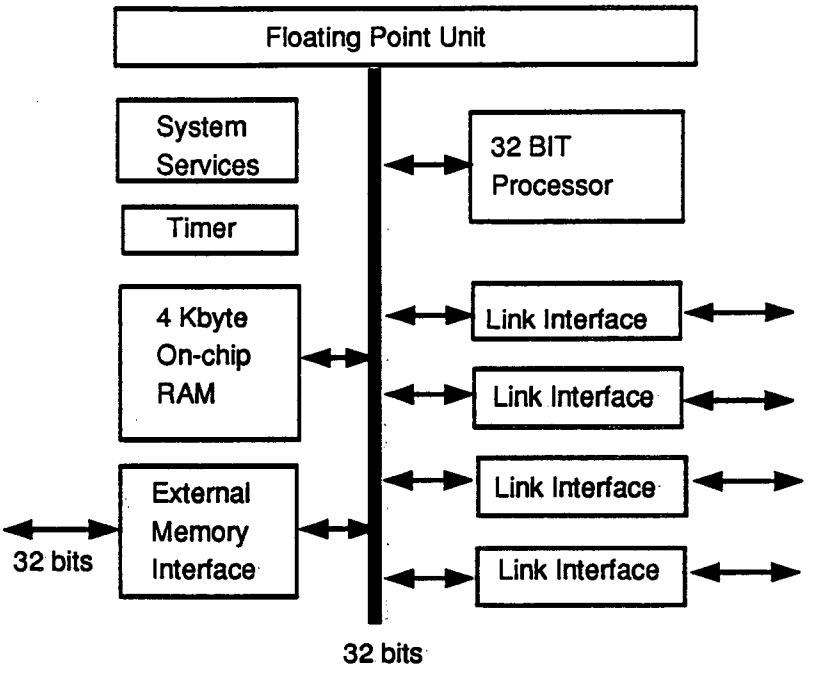


Figure 2.6: T800 transputer hardware

features which enhance the predictability of real-time systems. Scheduling delays may be calculated as follows:

1. An upper bound delay of 40 processor cycles is specified before a low priority process reaches a point where it can be suspended.
2. The maximum time it takes to execute a context switch from a low priority process to a high priority process is 18 cycles [37].

The above figures give a measure of the maximum delay expected when servicing an interrupt. In real-time systems, it is important that the operating system be expressive enough to prescribe certain timing behaviors. For example, Stankovic [24] reported that the language Ada is troublesome for real-time applications since upper bound timing delays cannot be guaranteed.

Supernode architecture is flexible and the control bus is important in the design of a fault-tolerant architecture. T800 microprocessors are powerful and well suited to real-time applications. However, the Supernode architecture has the following disadvantages:

1. each node has 256 Kbyte of external memory. When accessing the 10 MByte memory bank of the host transputer, large memory latency times are incurred;

2. the system supports only one user per Supernode; and
3. all software development must be done on the host transputer. Operating system facilities such as those available under Unix or VMS cannot be used.

2.2.4.4 FPS T-Series

Each FPS T-Series module contains eight nodes interconnected in a hypercube topology, a system board and a system disk [1]. A link bus provides communication between the system board and the processors. All the modules are interconnected in a ring. The ring network and the link bus provides the connection between the transputers and the host computer. The host machine is a DEC MicroVAX II. The architecture is presented in Figure 2.7.

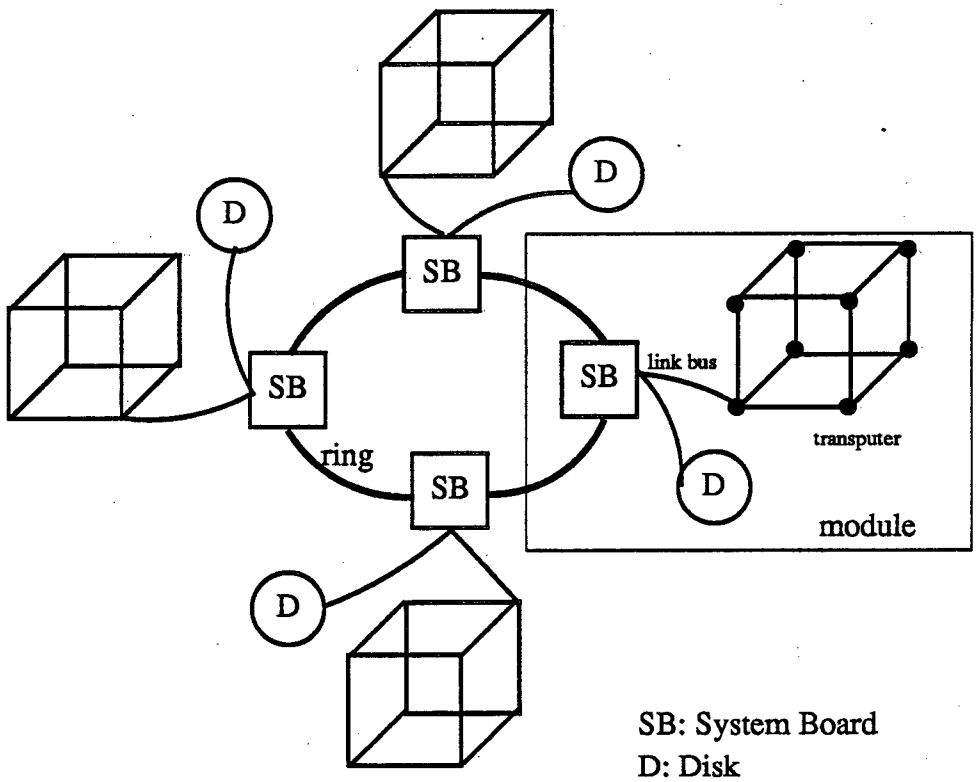


Figure 2.7: Architecture of the FPS T-Series hypercube

Each node of the FPS architecture contains a 32 bit Inmos T414 control processor, a 64–bit floating point vector processor, 1 MByte of dual-ported memory and four hardware communication links. Each link is multiplexed four ways and is used to connect to adjacent nodes.

The control processor is an Inmos T414 transputer which is similar to the architecture of the T800 without the floating point co-processor. The floating point vector processing unit incorporates a pipeline adder and a pipeline multiplier. All the memory on each node is divided into groups of 256 words called a slice. Four shift registers which can hold one slice of memory are used to transfer data to and from the vector unit. The contents of these shift registers becomes unreliable after approximately 20 microseconds.

There are two major drawbacks in the vector processing unit: the volatility of the shift registers and the absence of scalar floating point support. The volatility of the shift register implies that a process which executes vector operations cannot be context switched. If a process was context switched while using the vector processing unit, the result of the operation may become invalid by the time the process is rescheduled.

Because the T414 transputer supports only integer arithmetic, the vector floating point unit processes scalar as single element vectors. Unfortunately, the overhead for scalar floating point operations is as large as for vector operations due to the fact that a shift register can only manipulate 256 word blocks of data. The result of any vector operation destroys the previous contents of an entire 256 word block of memory, even if the specified vector is shorter than 256 words.

The four physical links of each node are each multiplexed four ways to create sixteen logical links. The logical links are mapped to physical links such that logical link L corresponds to physical link L mod 4. Consequently two logical links L1 and L2, where L1 mod 4 = L2 mod 4 cannot be used simultaneously. One physical link is reserved for input/output with the host processor. The host processor is a DEC MicroVAX II computer. The connection control for the reminder 12 logical links is done through software and are used for hypercube connections.

A major disadvantage of this system is that I/O routines or disk transfer routines destroy all established connections between nodes. Software connection calls must be executed by the user to reestablished the connections before continuing inter-node communication.

2.2.5 Hardware utilized in the project

The Transputer with its excellent support of concurrency and inter-process communication

enhanced with a deterministic hardware scheduler represents a useful design of a distributed system. The discussion of software design options in the next section assumes that the system hardware contains:

1. a distributed crossbar interconnection network;
2. a global bus which provides the capability to communicate directly with any processors; and
3. a node architecture composed of a Transputer with locally addressable memory.

2.3 Software Design Options

Optimizing distributed system performance requires a judicious combination of node computation speed, message transmission latency, and operating system software [7]. The Transputer has an efficient communication engine as a result of built-in support for inter-process communication and direct-memory-access link interfaces that can transmit data at a speed of 20MBits/sec. This communication engine, however, does not support communication between transputers that are not directly connected and assumes that the interconnection network is static.

This section begins by describing the built-in communication support and the communication limitations of the Transputer. Following this description, design options in the development of an operating system to enhance the transputer communication facility are examined. These design options are:

1. communication protocol (synchronous versus asynchronous);
2. control strategy (centralized versus distributed);
3. switching methodology (packet switching versus circuit switching);
4. reconfiguration mode (off-line, breakpoint or on-line);
5. addressing mode (broadcasting versus direct-addressing); and
6. reliability.

2.3.1 Transputer Communication Support

The Transputer's instruction set provides operations for inputting and outputting messages. Processes exchange messages over **channels**. In this thesis, a channel will refer to a communication connection between processes. There is a distinction between the implementation of channels within a single transputer, called soft channels, and channels for communication between two transputers, called hard channels.

2.3.1.1 Soft Channels

A soft channel allows two processes running on the same transputer to communicate. Figure 2.8 illustrates communication of processes A and B over a soft channel. A memory word is allocated

at compile time (Figure 2.8 a). Assume that Process A is ready to communicate first. The channel word is empty, and therefore process A must wait. A pointer to the workspace of process A is stored in the channel word by the transputer micro-coded scheduler (Figure 2.8 b) and process A becomes suspended. Once process B is ready to communicate, the channel word, which already contains the address of process A, indicates that process A is also ready to communicate. Process B initiates the communication (Figure 2.8 c), and the message is transferred directly from process A to process B. Once the message transfer is completed, process A becomes ready and process B continues its execution. Notice that when a process must wait to communicate, the transputer's micro-coded scheduler suspends it and performs a context switch automatically.

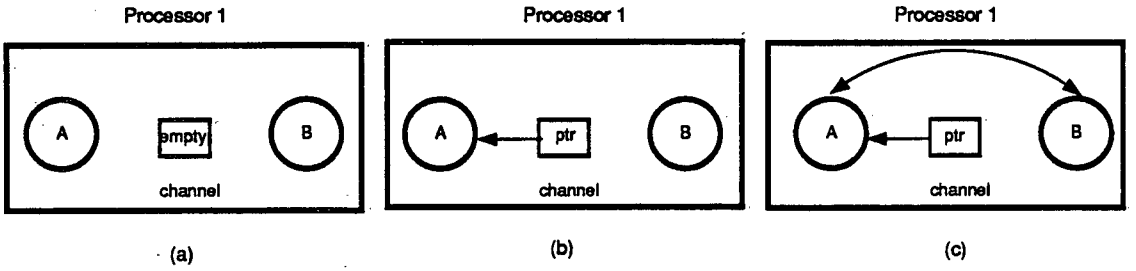


Figure 2.8: Processes communicating over a soft channel

2.3.1.2 Hard Channels

A hard channel allows two processes running on different transputers to communicate. If the address of a channel is mapped to an external link, the processor delegates the responsibility for the communication to an autonomous link interface. As presented in Figure 2.9, when communication takes place, Process A and Process B are suspended and the two link interfaces are exchanging data.

Each link interfaces has three registers that are initialized prior the start of a communication. These registers contain a pointer to the workspace of the communicating process, a pointer to the message and a count of the number of bytes to be transferred. The link interface stores by Direct Memory Access (DMA) an incoming message in the workspace of the destination process and reschedules the suspended process once the communication has completed. These last two actions are executed independently of the processor.

The use of DMA by the link interface is reliable only if the memory locations used by the DMA link interface are accessed by only one process. If concurrent processes need to communicate over a single hard channel, arbitration is required.

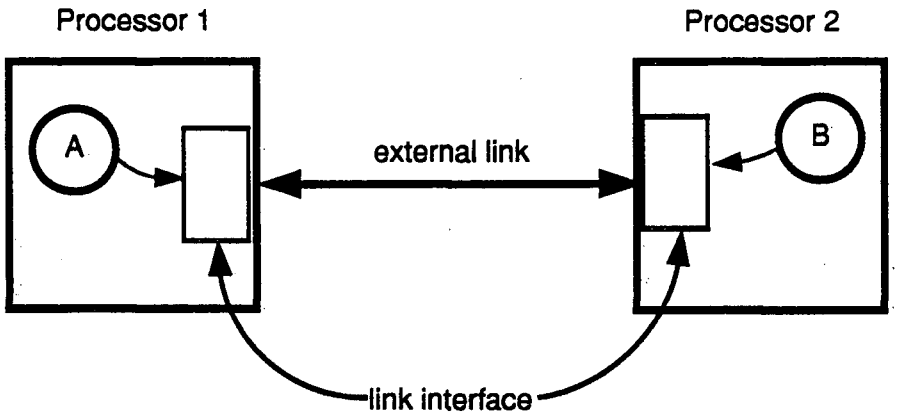


Figure 2.9: Hard channel communication

2.3.1.3 Communication Limitations

A transputer link supports two autonomous half duplex channels, one in each direction. As discussed above, only two processes, a sender and a receiver, can share a half duplex channel. Consequently, hard channel resources can supports no more than eight communicating processes. A deadlock will occur if more than one process access the same hard channel concurrently. The communication mechanism of the transputer has a static nature where soft and hard channels must be declared at compilation time. Also there is a maximum number of four hard channels that can be created since the number of hard channels is limited by the number of communication links. Consequently a transputer cannot communicate with more than four other neighbors.

The communication facility of the transputer should be enhanced with operating system primitives which support routing of messages between any node and multiplexing of the hard channels. Multiplexing hard channels represents an important advantage for input/output operation when only one transputer link may be connected to the host computer.

2.3.2 Communication Protocol

The communication protocol used by an operating system can be classified as synchronous or asynchronous. In a **synchronous** communication protocol, the process that reaches the communication point first must wait for the other process before it can continue. Prior to starting the communication, the transmitter requires an acknowledgement from the receiver as an indication that both processes have reached the communication point. The transmitter then sends the message. On receipt of this message, the receiver will issue an acknowledgement. The communication terminates when the transmitter receives the end of message acknowledgement. Both processes can then continue autonomously. In this manner, process synchronization is enforced through communication. This type of communication is used in CSP [38], Occam [37], and Parallel C [39].

Asynchronous communication does not require any acknowledgement. The transmitter issues an initiating message and then continues its operations. Since synchronization is not enforced by communication, buffering capabilities must exist at the destination node in case the receiver is not ready to accept the message. The CrOS-III operating system for the Mark-III hypercube supports asynchronous communication [27].

The main disadvantage of a synchronous communication protocol is a decrease in process concurrency. A transmitter process is always blocked while waiting for a receiver. In an asynchronous communication protocol, a transmitter is not blocked and can continue its task autonomously even if the receiver is not ready. Also, a receiver process may not have to wait for the transmission of a message if the message has already arrived and is stored at the destination node. Synchronous communications, however, has many advantages. The following sections will discuss these advantages and will demonstrate the importance of a synchronous communication protocol for real-time applications.

No Buffer Requirements

One advantage of a synchronous communication protocol is its bufferless property. Prior to sending a message, a transmitter must wait for the receiver to become ready. The message is sent directly from the transmitter to the receiver process. Consequently, the message does not need to

be buffered at the destination node.

An asynchronous communication protocol, however, requires buffers at the receiver and intermediate nodes. Complex algorithms must be designed for allocating and managing buffers in order to avoid **deadlocks**. A deadlock occurs when no message can advance toward its destination because the buffers are full. Consider the example shown in Figure 2.10. The communication pattern forms a cycle. The buffer of each node is filled with messages destined for the opposite node. No message can advance toward its destination; thus the cycle is deadlocked.

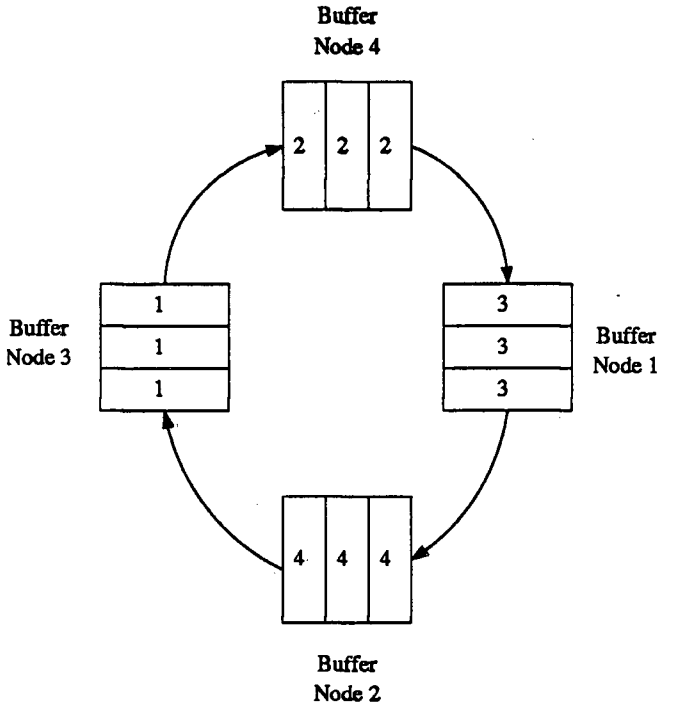


Figure 2.10: Example of a deadlock

Deadlock-free message-passing algorithms have been developed [40] and [41]. Designing an algorithm which prevent deadlocks greatly increases the complexity of the communication algorithm and also increases the communication transmission time. To prevent deadlocks, a transmitter may have to wait prior to transmitting its message or a message may have to be routed via a longer path. Some deadlock-free algorithms also limit the size of messages by requiring messages to be broken down into packets prior to transmission. This scheme introduces an overhead of reconstructing the message at the destination.

Some asynchronous message-passing protocols may also require buffers at all the intermediate nodes where messages may temporarily be stored prior reaching their destination. The allocation of buffers and the copying of the message from one node to another creates delays.

Deterministic Communication Delays

A synchronized communication protocol provides the advantage of more determinism in communication delays. Synchronized communication requires that a direct communication path be created between the sender and the receiver. Therefore an initial delay exists to create a communication path. The delay is stochastic and depends on the availability of resources. Once the path is created, however, all subsequent communication are deterministic.

In an asynchronous communication protocol the message latency time is indeterminate for every transmission. Message delays vary with:

1. The network traffic: If the network traffic is high, a message may wait at an intermediate node until communication resources become available.
2. The computational load of intermediate nodes: Routing messages may not be the highest process priority of a node, therefore a message may have to wait at an intermediate which is busy processing another task.
3. The availability of buffers: A message may be blocked if the destination node has no buffering space to store the incoming message.

Greater Process Traceability

In a synchronous communication protocol, a sender cannot continue autonomously until the message has been received by the receiver process. The receiver can assert that the state of the sending process is consistent with the message just received.

In an asynchronous communication protocol, a sender process will be able to proceed arbitrarily far ahead of the receiver. A message received at the destination node, therefore, may not reflect the

current state of the sending process. The following example will show that determinism in message transmission as well as traceability of the system state are critical for real-time control.

An autonomous robot which performs its task in a hazardous environment is controlled by a distributed system connected in a hypercube topology as discussed by Einstein [42]. The communication protocol of this distributed system is asynchronous. Due to the indeterminate communication delays, the path planner must send command messages to the end effector task in advance of the times at which these commands are to be carried out to allow for maximum possible delays. This planning ahead scheme will probably create queues of input messages at the end effector task. When unpredictable changes take place in the environment, the sensor task communicates the new information to the planner, which may have to change its previous plan.

As a result of the planning ahead method, there is a high probability that messages based on the old plan have already been sent to end effector tasks. These old messages must now be eliminated from the system. The Time Warp operating system has been developed by Einstein [42] to provide emergency messages and cancellation of old queued messages. The performance of Time Warp in real-time robotics application is under investigation. Einstein suggested that Time Warp poses difficult theoretical and practical problems which are still unsolved. Therefore, the use of asynchronous communication protocols leads to difficulties in robotic systems requiring rapid response to unpredictable events.

Temporal Ordering of Message Arrival

In synchronous communication, messages always arrive in temporal order. A message is not sent unless the receiver is ready to accept it. The receiver process controls the temporal arrival of messages. In a real-time environment it is important that messages that are sent to a node are processed in a specific order. The order in which the messages arrive at destination is not necessarily similar to the order in which they must be processed. In asynchronous communication, messages may arrive out of sequence for the following reasons:

1. The network load may not be equally distributed. Messages routed through busy network

channels may suffer longer delays than others.

2. In a real-time application, the processing load placed on each node depends greatly on external events. The transmission of a message may be delayed depending on the processor load.
3. In a real-time application, various processes may execute at different frequencies and may also transmit messages at different rates making it unlikely that the messages will arrive strictly in order of generation.
4. If a fault occurs, error recovery may induce considerable message delays.

Therefore, the overhead of time stamping every message becomes necessary in a real-time application. If a message is broken into packets, the message header of every packet should include both a sequence number and a timestamp. None of these overheads are necessary if the communication protocol is synchronous.

Capability to Emulate Asynchronous Communication

When required, a synchronous communication protocol can emulate an asynchronous communication protocol. Buffering processes can be created and interposed between communicating processes. To increase process concurrency, it may be advantageous to create a bounded buffer where the sender can proceed ahead of the receiver but become blocked if the buffer becomes full.

From the previous discussion, we conclude that a synchronous communication protocol has many important advantages particularly in the area of real-time control.

2.3.3 Reconfiguration Mode

There are three reconfiguration modes that can be implemented in a dynamic topology. In an **off-line** reconfiguration mode, the interconnection network is configured prior the start of a program and remains static during program execution. Programs are written for a fixed topology that matches the algorithm. The setting of the switches may be defined by the user or derived automatically by software tools. Software tools have been developed to automatically extract the communication graph of a program and creates a configuration table for the switch setting. The efficiency of this mode

depends on the flexibility of the architecture. An architecture may be capable to map only a small subset of communication graph. The Supernode architecture supports this mode of operation.

The second reconfiguration mode is called **breakpoint.** The topology may be changed at predetermined synchronization points. When a synchronization point is reached, all network communication is stopped and the topology is reconfigured. This mode of operation may be well suited for image processing applications. The architecture may be configured in a mesh topology for low-level processing of the data such as convolution. After the completion of the low-level operations, the architecture may be reconfigured in a tree topology for pattern recognition algorithms. The CHIP architecture developed by Snyder [43] supports breakpoint reconfiguration. This reconfiguration mode is efficient only if multiple nodes require reconfiguration at the same time. The overhead of interrupting all communication in the network becomes significant if only a minority of nodes require reconnection.

The last mode, **on-line** reconfiguration, is the most flexible approach. Using on-line reconfiguration, any link can be connected to any other link at any time during program execution if the communication resources are available. On-line reconfiguration involves greater software overhead than the other two reconfiguration modes, but provides important benefits in flexibility and fault-tolerance. Any communication patterns may be mapped to the architecture. On-line reconfiguration also provides the capability of reconfiguring the system to bypass a failed component when a failure occurs during program execution.

In summary, communication software for a real-time distributed system should support on-line reconfigurability to achieve greater flexibility and fault-tolerance.

2.3.4 Control Strategy

The control of the interconnection network switches may be either centralized or distributed. If all switches are set by a central controller processor, the control strategy is **centralized.** The main disadvantage of a centralized control strategy is a the possibility of a bottleneck in the switch controller. A bottleneck occurs if the rate of reconfiguration requests is high or if many processors request reconfiguration simultaneously.

To prevent a bottleneck at the switch controller, a **distributed** control strategy may be used where the task of reconfiguring the switches is shared by multiple controllers. For example, in a distributed cluster interconnection network, a fully distributed control strategy would utilize one crossbar controller for each cluster. All controllers must be connected together to handle topology reconfigurations which involve setting of multiple crossbar switches. The efficiency of a distributed controller depends on the volume of inter-cluster communications. If a large number of messages are sent between processors in different clusters, the overhead due to controller to controller communication may be greater than for a central controller.

In order to provide fault-tolerance and reliability, redundancy at the switch controller must exist. The failure of a switch controller may lead to a catastrophic situation in system using either either breakpoint or on-line reconfiguration where communication between processors relies on reconfiguration. Therefore, a distributed system with a distributed crossbar interconnection network designed for real-time applications should have the following characteristics:

1. The control software should be capable of supporting either a centralized or distributed control strategy:

The efficiency of the control strategy depends on the size of the network and the communication pattern. The control strategy should be flexible and capable of adapting to various communication requirements.

2. A minimum of two crossbar controllers:

Redundancy in crossbar controllers provides two important advantages. The first advantage is fault-tolerance since the reconfiguration task may be supported by a redundant controller in case the primary controller fails. The second advantage is flexibility, since either a centralized or distributed control strategy may be implemented.

2.3.5 Switching Methodology

The two main switching methodologies are circuit switching and packet switching.

Circuit Switching

In circuit switching a physical circuit between the source and the destination node is established. The end-to-end path must be set up before any data is sent and is held for the duration of the transmission. Circuit switching has the following advantages:

1. it allows the system to operate synchronously or asynchronously;
2. messages are not routed through intermediate nodes, they are always sent directly from the transmitter to the receiver; and
3. the network appears to be fully connected from the user's stand point which greatly simplifies the mapping of applications.

The disadvantages of circuit switching, however, are that if a circuit is allocated and not used, the bandwidth is wasted. Also, the overhead cost of creating a connection path for small messages is high.

Circuit switching may be implemented either in hardware or in software. An example of a hardware circuit switching implementation is the Direct-Connect Router [44] developed by Intel Scientific Computers for the iPSC hypercube. The Direct-Connect Router is a hardware module which communicates with each node over two unidirectional parallel busses. The communication protocol is asynchronous and the reconfiguration mode is distributed. Hardware circuit switching is fast but the major drawbacks are:

1. the hardware complexity of the system is increased;
2. the Router is based on a e-cube routing algorithm and cannot support any other routing schemes; and
3. the Direct-Connect Router does not support re-routing of messages if a hardware module fails.

The Supernode architecture is an example of a circuit switching software implementation [29].

Packet Switching

Packet switching does not create a direct path between the sender and the receiver. Instead, when the sender is ready to transmit, a block of data is sent to an intermediate node. When the entire block has been received at the intermediate node, it is forwarded to the next node in the path until it reaches the destination node. This is also called store-and-forward message passing. Messages may also be broken into multiple packets to decrease the buffering requirements at each node.

The main advantage of this switching methodology is that circuit are never dedicated to a sender/receiver pair. The bandwidth may be utilized by packets from unrelated sources going to unrelated destinations. Packet switching is efficient for short message transmission. The disadvantages, however, are:

1. only asynchronous operation is possible;
2. a sudden burst of input traffic may cause the buffer capacity to be exceeded which could cause either deadlocks or loss of packets;
3. the arbitration overhead to design a deadlock free system is significant; and
4. packets may be delivered in the wrong order.

Packet switching have been implemented on a transputer network by Roscoe [41] and by Helminen [1]. This message passing mechanism is also supported by many operating systems such as Helius, Trolius, and Vertex.

Hybrid switching schemes implemented with a mixture of circuit and packet switching properties have also been developed. Some examples of hybrid protocols are: whormhole [40], stagged [45], and Virtual Cut-Through [46]. These hybrid switching schemes are implemented with an asynchronous communication protocol.

The importance of a synchronous communication protocol for real-time control have been discussed. Synchronous communication protocol, however, may be implemented only with a circuit switching methodology. A hardware implementation of circuit switching may be very efficient but a software implementation is more flexible. Therefore, an implementation of a software circuit switching methodology is well suited to support communication in a real-time environment.

2.3.6 Addressing Mode

The four main addressing modes used in communication protocols are direct naming, mailbox, port, and broadcast mode [30].

Direct naming is the technique used in the CSP and Occam models. It is very simple and efficient but it is not as flexible as the other protocols. This technique explicitly identifies the sender and receiver processes. The resulting map of logical process interconnection is expected to be static. Direct naming requires that every communication channel which might be required must be declared at compile time.

The **mailbox** addressing mode is the most flexible. Mailboxes allow messages to be sent by any number of processes to any number of processes. The versatility of mailboxes is achieved at the cost of a significant arbitration overhead. For example, if multiple receivers are waiting for a message at the same mailbox, arbitration among the receivers is required.

The remaining two modes represent compromises between direct naming and mailboxes. The **port** mode maps multiple senders to one receiver and the **broadcasting** protocol maps one sender to multiple receivers.

The transputer instruction set provides extensive support for the direct naming mode of addressing. The transputer provides operations based on direct naming for inputting and outputting messages with a synchronous communication protocol. Broadcast, port and mailbox addressing modes must be implemented with some form of broadcasting, which would create considerable overheads in a transputer environment. If possible, the communication protocol should preserve the message passing engine of the transputer.

2.3.7 Reliability

From the standpoint of fault-tolerance and reliability, robustness and reconfigurability are two important properties required in the communication software [33]. System communication software is **robust** if it is able to maintain reliable communication between processors in the presence of failed nodes, links, busses or switches. Robust communication software should use redundant communication paths provided by the interconnection network to minimize the effect of a failure. The Direct-Connect Router designed by Intel does not have the property of robustness.

If the software overhead due to routing of messages is low with and without the presence of failures, the communication software is said to be **reconfigurable.**

Reconfigurability and robustness are two important properties in the design of reliable distributed system communication software.

2.4 Summary

A distributed crossbar interconnection network enhanced with a shared bus is well suited to robotic applications. A distributed crossbar interconnection network is expandable and reconfigurable (Section 2.2.2). The global bus provides a communication link to each node which is an important feature for fault detection, diagnosis and recovery (Section 2.2.3). The global bus also provide the capability of having an efficient multi-user system where the code of a program may be downloaded to a transputer without affecting programs running on other nodes. In addition, the bus is a valuable communication media between the host computer and the network nodes for input/output support .

The node architecture is based on Inmos T800 transputers which are simple and compact compared to the Mark III hypercube node design (Section 2.2), and have on-chip floating point support. The hardware scheduler of the transputer supports fast context switching and permits the prediction of an upper bound in scheduling delays. Scheduling predictability is important in real-time system designs.

From the standpoint of software, the communication protocol should be synchronous. Synchronous communication increases the system traceability and avoids buffer requirements (Section 2.3.2). The switch controller should support on-line reconfiguration. On-line reconfiguration provides two functions: faulty components can be bypassed and any two network nodes which need to communicate can be connected. It is proposed that the switching strategy be circuit-switched. Circuit switching guarantees a certain bandwidth and maximum communication latency once the connection is established which is important for real-time environments.

Currently high level languages such as Occam, Parallel C and Parallel Prolog cross-compilers are available for transputers where the software may be developed on a host computer and downloaded to the network. Also debugging tools similar to Unix DBX are commercially available. Consequently, good support for software development exists for transputer architectures.

Chapter 3

Implementation of a Real-Time Distributed System

This chapter describes the design of a real-time distributed system which was assembled to implement the objectives defined in the previous chapter.

3.1 Hardware Organization

The hardware used to realize the real-time distributed system resides in two VMEbus card cages. The first, the distributed architecture, contains eight transputer cluster module boards, and two transputer bus master modules. The modules communicate with each other via a point-to-point interconnection network or via the VMEbus. This card cage is connected to a second card cage through a VME to VME bus extender card. The second VME card cage contains a Sun 3/280 computer which functions as a host.

3.1.1 Overview of System Architecture

Details of the hardware are shown in Figure 3.11. The host computer is a Sun 3/280 68020 based cpu equipped with a terminal board, an ethernet interface, and a disk storage subsystem. The distributed system is consists of the following modules:

Bus Masters (Parsytec-BBKV2)

A transputer bus master is composed of one T800 transputer with 2 MBytes of dual-ported memory which is asynchronously accessible by the transputer and the VMEbus. Three transputer links are hardwired to external serial ports which can be accessed from the interconnection network. The last transputer link is connected to the VMEbus via a link adapter. The link adapter, an IMS C012 chip, transforms data received serially from a transputer link to a 8 bit wide parallel data word that can be read from the VMEbus. The bus master also performs automatic conversion of data format allowing compatible messages to be exchanged efficiently with the host computer. This module is a commercial BBK-V2 board procured from Parsytec [47].

Clusters (Parsytec-VMTM)

Each cluster module consists of four Inmos T800 transputers each with 1 MByte of local memory, nine external serial ports, a 32×32 crossbar switch, and four link adapters for interfacing a transputer connection link the a VMEbus. Each transputer is a VMEbus **slave processor**. A slave processor cannot initiate a data transfer on the VMEbus. A data transfer must be initiated by a bus master which is capable of acquiring the bus and transferring data between itself and a slave processor [48]. A VMTM cluster module was also purchased from Parsytec [49].

Interrupt requester board (UBC-DPIRB)

This functional module can generate an interrupt on the bus. This board has been designed at UBC to provide efficient communication between transputers on the cluster modules and the transputer bus masters. Communications between a slave processor and a bus master can be initiated either via polling or interrupt handling. With the use of polling, the communication delay would quickly reach an unacceptable level as the number of nodes in the network increased. Therefore, the interrupt requester board has been designed to provide interrupt driven communication between master and slave processors.

When a slave transputer transmits a byte to a link connected to a link adapter, a hardware signal is generated. This signal is sent to the interrupt board via a user defined connection available on the VMEbus backplane. The interrupt requester then, generates a VMEbus interrupt. The interrupt is handled by a bus master which takes appropriate actions. The schematics of the board may be found in Appendix A.

I/O boards (UBC-DPIO)

The architecture can be equipped with special I/O boards which may be directly connected to the node that processes the data. Robot control involves interaction with serial and parallel ports, a/d and d/a converters, and various sensor devices. With this architecture, interaction between the control processor and these devices may be performed either via the VMEbus or a communication link. The bus structure allows commercial VMEbus boards to be easily added to the system. Direct I/O channels between sensors/actuators and controllers provide an efficient I/O handling scheme without causing bus saturation.

VMEbus repeater

The VMEbus repeater links the two card cages so that bus masters, slaves, interrupt handlers and interrupters located in the host and the distributed architecture card cages functionally appear to be on the same bus. In addition, it provides both hardware and software switches which can be used to isolate the host from the distributed architecture. With this isolation capability, boards may be removed from or added to the card cage without affecting the host computer. Specifications of the card may be found in [50]

The system is presently configured in an integration mode where the distributed system and the host share the same bus. The integration mode is an excellent software development environment since software can be designed on the Sun 3–280 running Unix, cross-compiled for T800 transputers, and downloaded directly to the various transputer boards via the VMEbus. Once the software development is completed for a particular application, the distributed system can be isolated from the host and all transputers booted from EPROM.

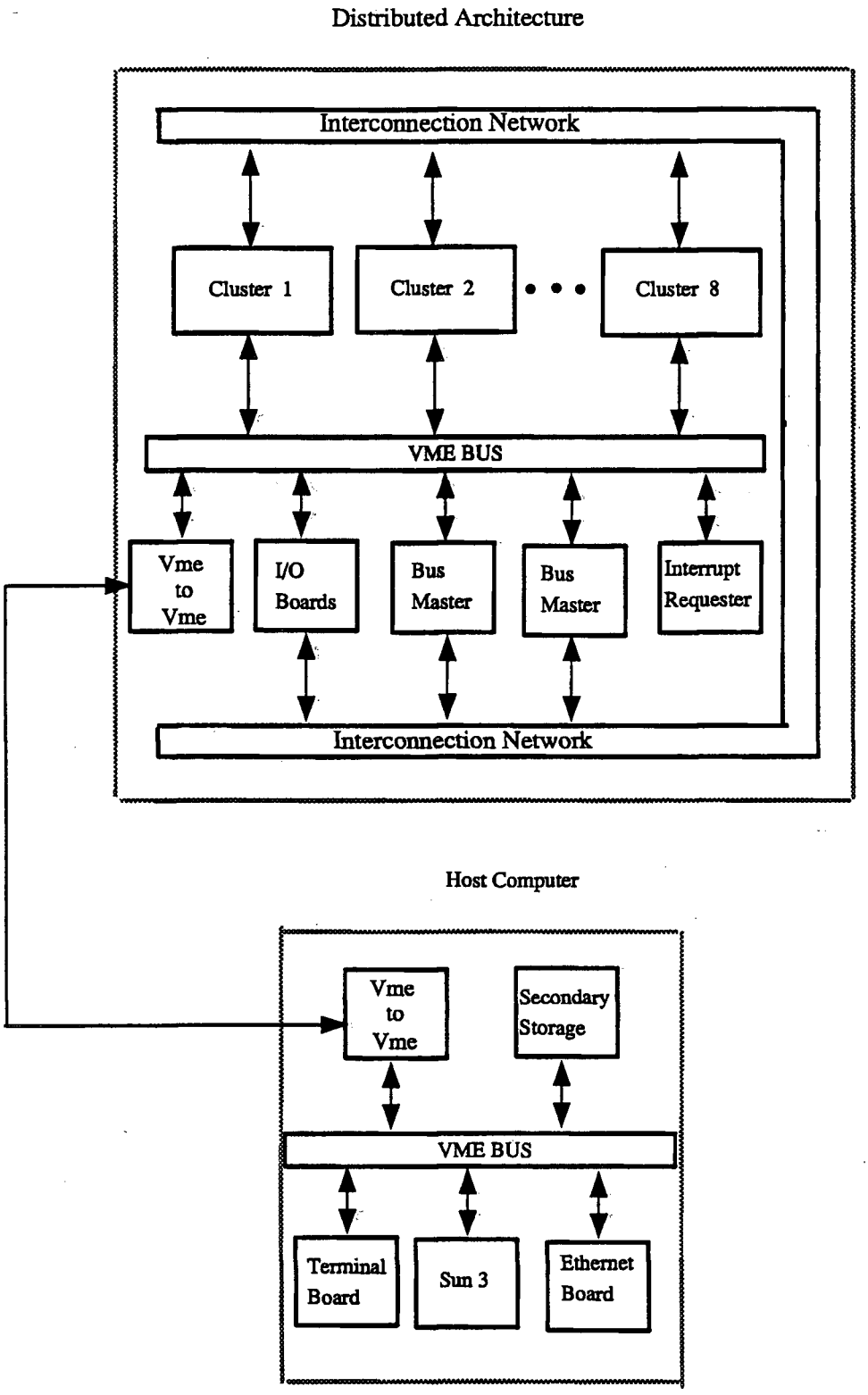


Figure 3.11: System Architecture Overview

3.1.2 Cluster Communication Architecture

A cluster module is composed of four transputers. Each cluster has 29 software reconfigurable channels connected to crossbar switch.The crossbar switch, an IMS C004/B, is a 32×32 multiplexer with the following features:

1. The crossbar switch is **centralized** which means that the switch is programmable from a single source.
2. The crossbar switch is controlled via a configuration link in order to modify the crossbar connections. The configuration link may be connected to a transputer link or a C012 link interface.
3. The crossbar may be programmed without interrupting existing communications.
4. The status of the crossbar connections may be read from the configuration link.

The 29 software reconfigurable channels are illustrated in Figure 3.12. All 16 transputer links are hard-wired to the crossbar switch. Four link adapters are available to implement communication channels between a transputer and the VMEbus. Furthermore, a cluster module may be integrated into a transputer network via 9 external channels which are brought out to the front panel of the module. For example, Figure 3.14 shows five cluster modules interconnected into a ring topology.

3.1.3 Interconnection Network

In order to support dynamic reconfiguration, some cluster connections of Figure 3.12 are set at initialization and remain fixed for the duration of the user program. As illustrated in Figure 3.13, the configuration link of the switch and link zero of three transputers are connected to link adapters. With this arrangement, one or more bus masters may control all of the network switches and reconfigure the network dynamically. A bus master may also communicate directly with each cluster transputer via the VMEbus. Due to the limitation of having only four link adapters available on each cluster module, one transputer on each cluster will not be connected to the bus and its link connections will not be reconfigured. A detailed description of the switch controller process is presented Section 3.2.4.

On-line dynamic reconfiguration can allocate a communication path between any two transputers. In order to support synchronous communication between processors, a circuit switching communi-

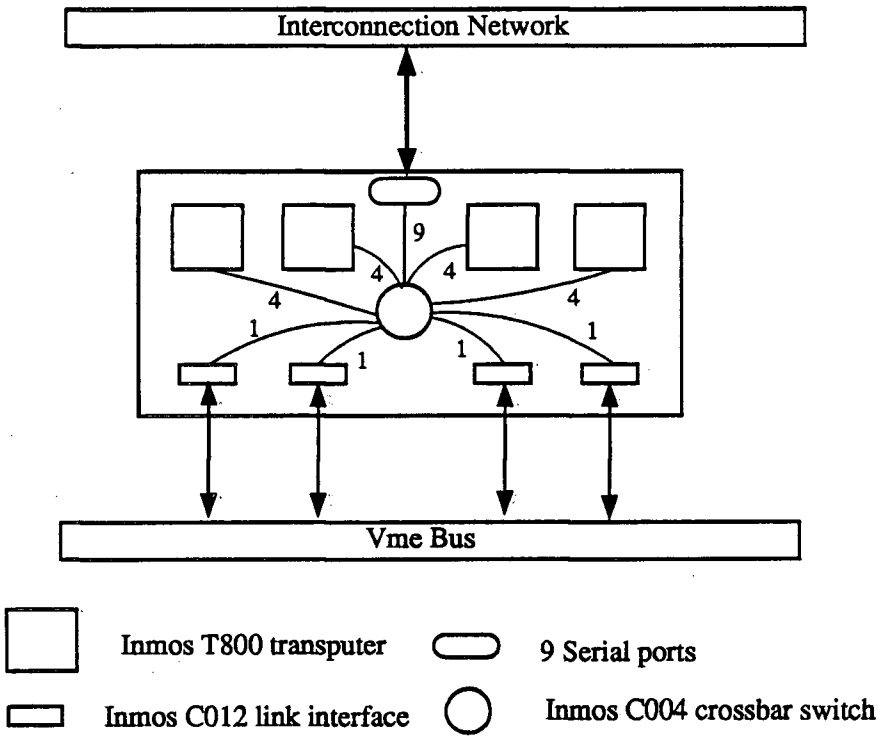


Figure 3.12: Cluster Architecture

cation strategy is implemented. Creation of a circuit switch requires resources such as transputer links and inter-board cables. As illustrated in Figure 3.14, the interconnection network created by the ribbon cables is fixed. The switches, however, may be programmed by a switch controller to connect a cluster transputer link to any external port. Consequently, a circuit switch may be created between any nodes as long as the interconnection network created by the inter-board cables provides a path between any two clusters. For example, the ring network of Figure 3.14 provides a route between any two boards. In this project three types of interconnection networks have been used, which are a tree, a ring and an hypercube configuration.

One major disadvantage of globally accessing the crossbar switches via a shared bus is that a failure of the bus may disable all network reconfigurability. This single point of failure could be avoided if the control of each crossbar was distributed [34]. A **distributed switch** is designed with more than one configuration link. Redundant configuration links allow programming of the crossbar from two different sources. It is intended to enhance the existing architecture with distributed switches where each switch point could be configured from the bus as well as from a point-to-point connection.

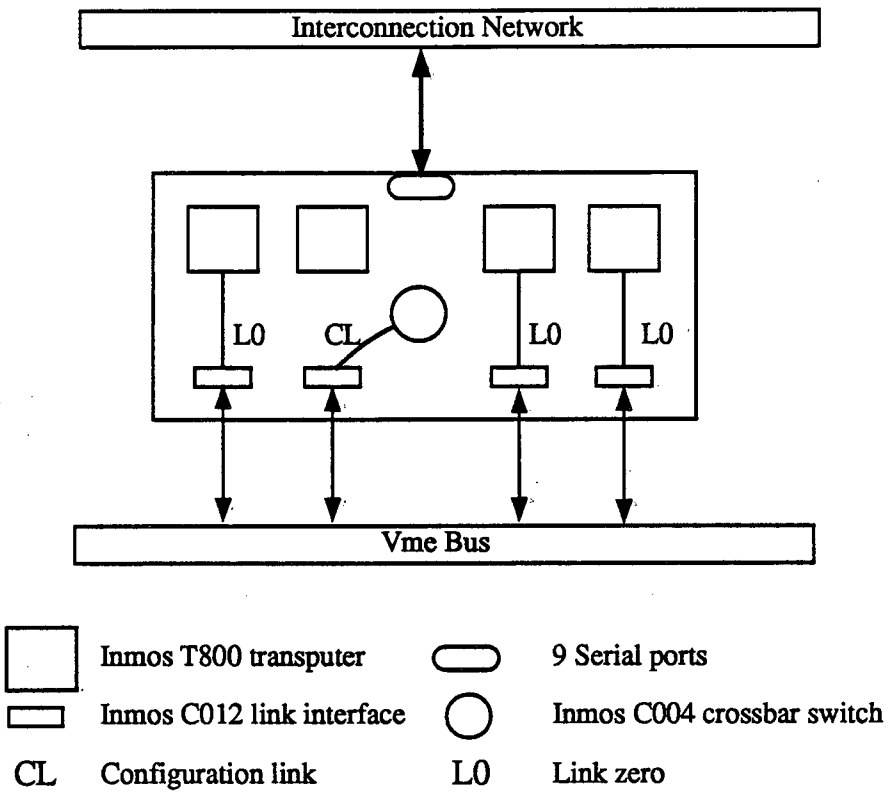


Figure 3.13: Cluster interconnection network to support dynamic reconfiguration

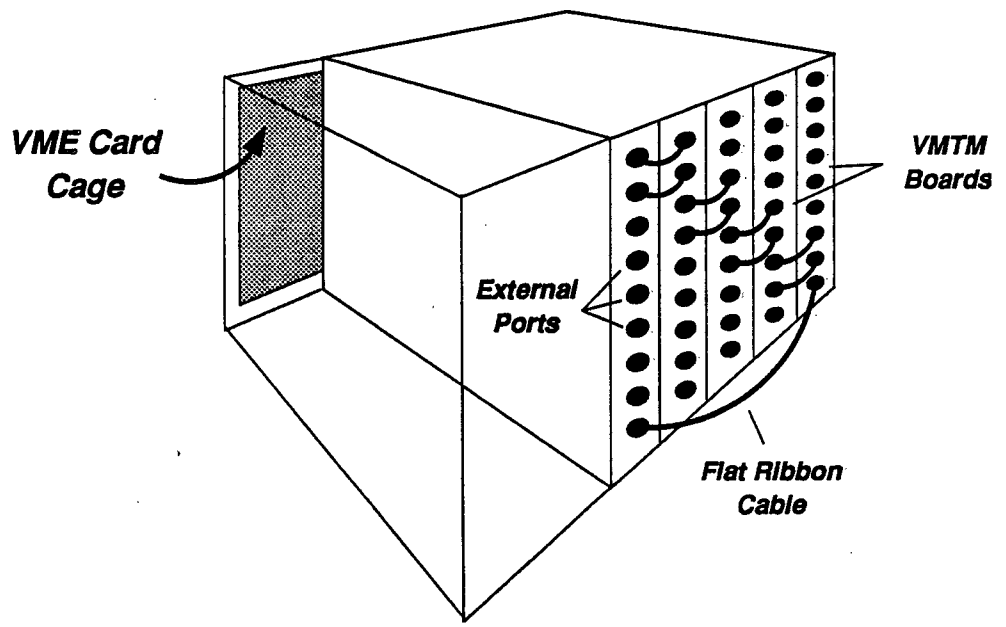


Figure 3.14: VMTM boards interconnected into a ring topology

For the purpose of **fault-tolerance**, one link of each transputer in the network may be connected

to a link adapter to create a direct connection path to each processor from the bus. An autonomous diagnostic/maintenance processor may then access transputer data for on-line fault detection, diagnostics and recovery.

Also, cluster modules and bus master modules are designed with a reset line parallel to each transputer link. This line provides the capability of resetting individually any transputer in the network from the bus is case of an error. When a transputer is in a reset mode, its local memory is still be accessible via the bus. This allows a maintenance processor to examine the memory for fault diagnosis and reload new program code before restarting the processor. The maintenance processor function has not yet been implemented and is beyond the scope of this thesis.

One disadvantage of the design of the cluster module is that all four transputer links are hardwired to the crossbar switch (Fig. 3.12). A failure of a switch would create a cluster failure. The cluster boards should be modified to provide a hardwired connection directly from a transputer to a link adapter. Doing so, the global bus could be used as a communication link to access the processors connected to the failed switch.

This architecture represents an excellent software **development environment.** Programs may be developed under the Unix environment and downloaded to any transputer in the network. Each transputer may be allocated to a different user and booted from the bus without interfering with other users. Each transputer may be connected to a terminal and have access to external storage via the global bus.

The architecture is **expandable** by adding new clusters. In order to avoid bus saturation, communication over the bus should occur mostly for operating system functions and fault-tolerance activities. All inter-transputer application messages should be sent via point-to-point communication links.

3.2 Software Organization

Two software versions of an operating system shell which includes communication an input/output primitives have been developed as part of this research. The first version was written in Occam [37] with the support of the Transputer Development System (TDS) tools and a folding editor developed by Inmos. The parallel structure of Occam provides a powerful and efficient tool for designing distributed system software. The main disadvantage of Occam, however, is its static nature. No dynamic memory allocation or process creation is provided. Also the input/output libraries provided by the TDS environment are less developed than those provided by high level languages such as "C". In addition, the source code of the compiler was not available which made it difficult to interface the developed software with the host operating system.

Consequently, the second version was written in Logical System's Parallel C [39]. This will be the version referred to in this thesis. The software developed may be divided into four major sections:

1. **Operating System Primitives**: These user system calls provide operating system support for communication between arbitrary processes. System primitives have also been developed to provide communication between any transputer process and the host computer for input/output functions.
2. **Central Switch Controller process (CSC)**: This programs runs on a transputer bus master and reconfigures the network topology as required by the application. The switch controller process that has been implemented is centralized.
3. **Local Bus Interface process (LBI)**: This process is an operating system process that runs in parallel with a user's program. This processes represents an interface between the user processes and the Central Switch Controller. This process must be linked with all user programs that make calls to the operating system primitives.
4. **Input/Output Controller (IOC)**: This software package runs on the host computer and communicates with the user programs to provides input/output support.

Figure 3.15 illustrates the communication channels between the user processes, the LBI, the CSC and the IOC. Each cluster transputer has multiple user processes running concurrently with the LBI.

Connection requests called by the user processes are first transmitted to the LBI and forwarded to the CSC via the VMEbus. The CSC will then create a circuit switch and send a reply back to the user via the LBI.

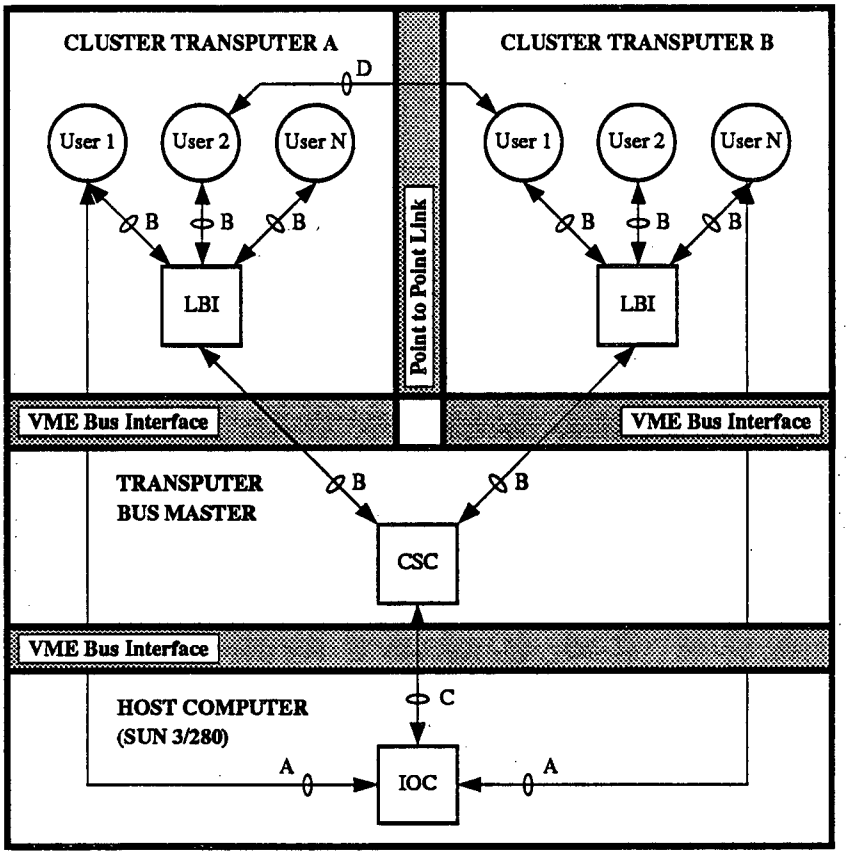
There are two categories of connection requests. The first will create a circuit switch between two user processes so that they can exchange messages. Operating system primitives of this category are: *FixLink*, *ReleaseLink*, *OpenChannel*, *CloseChannel*, and *ChanAlt*. The second category of connection request, *ConnectHost* and *ReleaseHost*, creates a communication channel between the host computer (Sun 3/280) and a user process for input/output support. Operating system calls are described in details in Section 3.2.2.

Figure 3.16 shows the interactions between user processes (Process A and Process B), the LBI processes and the CSC when a circuit switch need to be created. In this example, both processes are running on different transputers. When Process A or Process B reaches the communication point, it requests a communication channel by making an operating system call *FixLink*. This connection requests is received by a LBI process running on the same transputer. The LBI process tries to communicate with the switch controller by loading a byte into a link adapter interfaced to the bus. The interrupt requester board catches a hardware signal caused by loading the link adapter and interrupts the CSC. The CSC retrieves the connection request message from the link adapters, and creates a circuit switch by configuring the switching elements. Note, that both processes must be ready to communicate before a circuit switch is created. The communication protocol is synchronous.

Once the circuit switch is created, the CSC sends a hardware link address to the user process via the LBI. At the reception of the CSC reply, the user processes may then communicate using standard communication primitives supported by the Parallel C Language[1]. At the completion of the communication, a user process frees the circuit switch by calling *ReleaseLink*. All communications from the LBI to the CSC are initiated by a hardware interrupt.

---

[1] Parallel C Compiler designed by Logical System provides the following communication primitives: ChanOut, ChanIn, ChanOutInt, ChanInInt, ChanOutChar, and ChanInChar.



	Type of Messages	Operating System Calls
A	IO Data	*printf, gets, fopen, fclose...*
B	Connection Request	*ConnectHost, ReleaseHost, FixLink, ReleaseLink, OpenChannel, CloseChannel*
C	Start and Finish IO Messages	
D	User Messages	*ChanOut, ChanIn, ChanOutInt, ChanInInt...*

Figure 3.15: Communication channels between user processes and operating system processes

The communication channel, illustrated in Figure 3.15, between the CSC and the IOC process is only used when a host connection request is received. An input/output scenario is illustrated in Figure 3.17. A connection with the host computer is initiated from the user process by calling *ConnectHost.* The message is transmitted from the LBI to the CSC. At this point, the LBI stops sending messages to the CSC. The CSC must then reply with a start IO signal to advise the user process that I/O calls may be executed and to block the LBI from receiving messages from the bus. Then, the IOC process

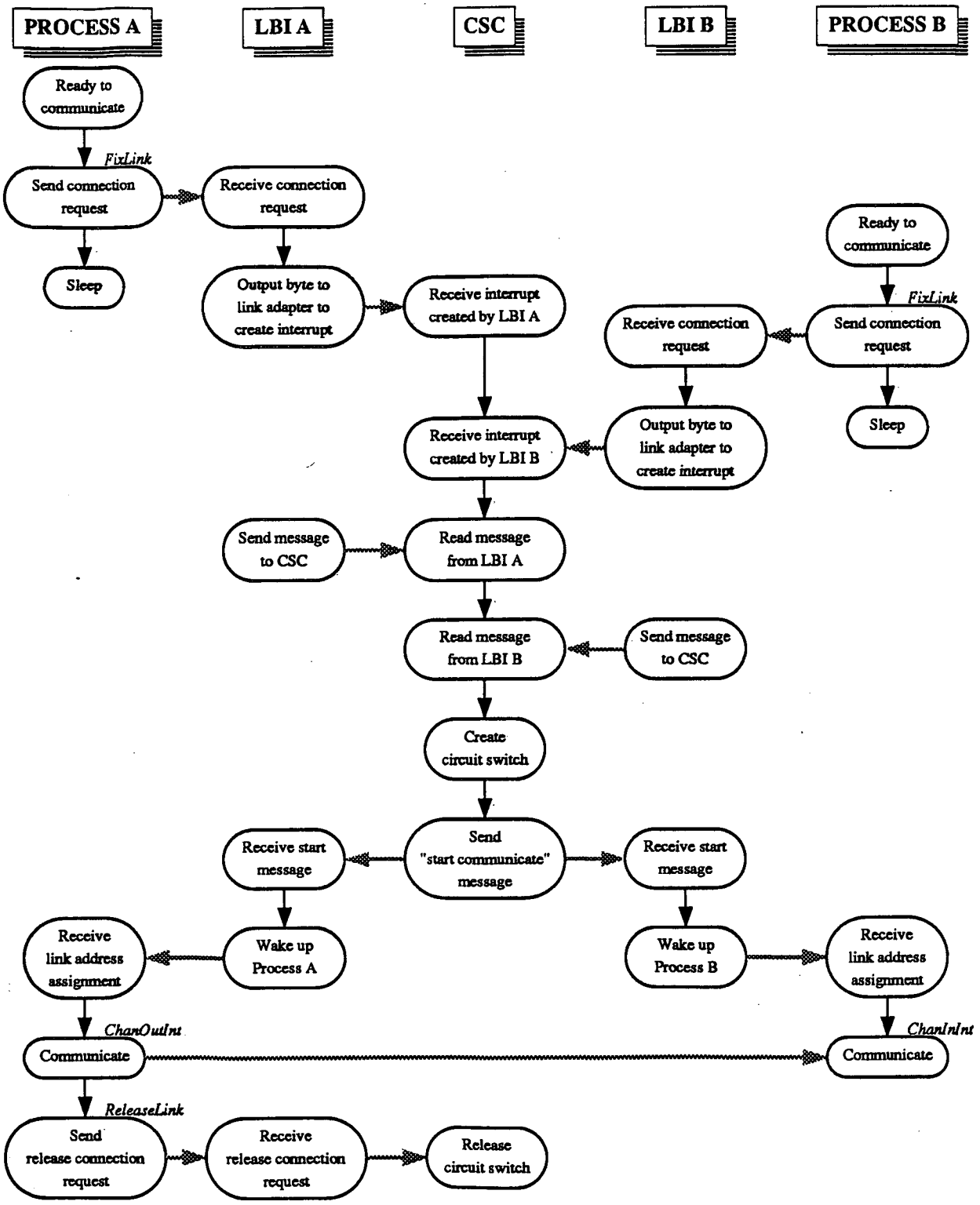


Figure 3.16: Scenario of the creation of a circuit switch

running on the host computer is started and executes I/O requests from the user process[2].

The LBI must stop receiving and sending messages to the bus in order to dedicate the transputer link connected to the bus for I/O operations only. Once the LBI has freed the VMEbus channel, the user process and the IOC may communicate directly. The IOC starts polling the transputer link adapter and executes the user request until it receives a *ReleaseHost* message. Once the IOC has stopped, the LBI restarts communicating with the CSC.

The CSC and IOC processes are both running on bus master processors. Conflicts may occur between the two masters if both processes try to access the same VMEbus address simultaneously. Consequently, the interrupt mode is disabled at the transputer that has requested an I/O connection, until the IOC receives a *ReleaseLink* message.

---

2 The Parallel C compiler support all standard C input/output system calls

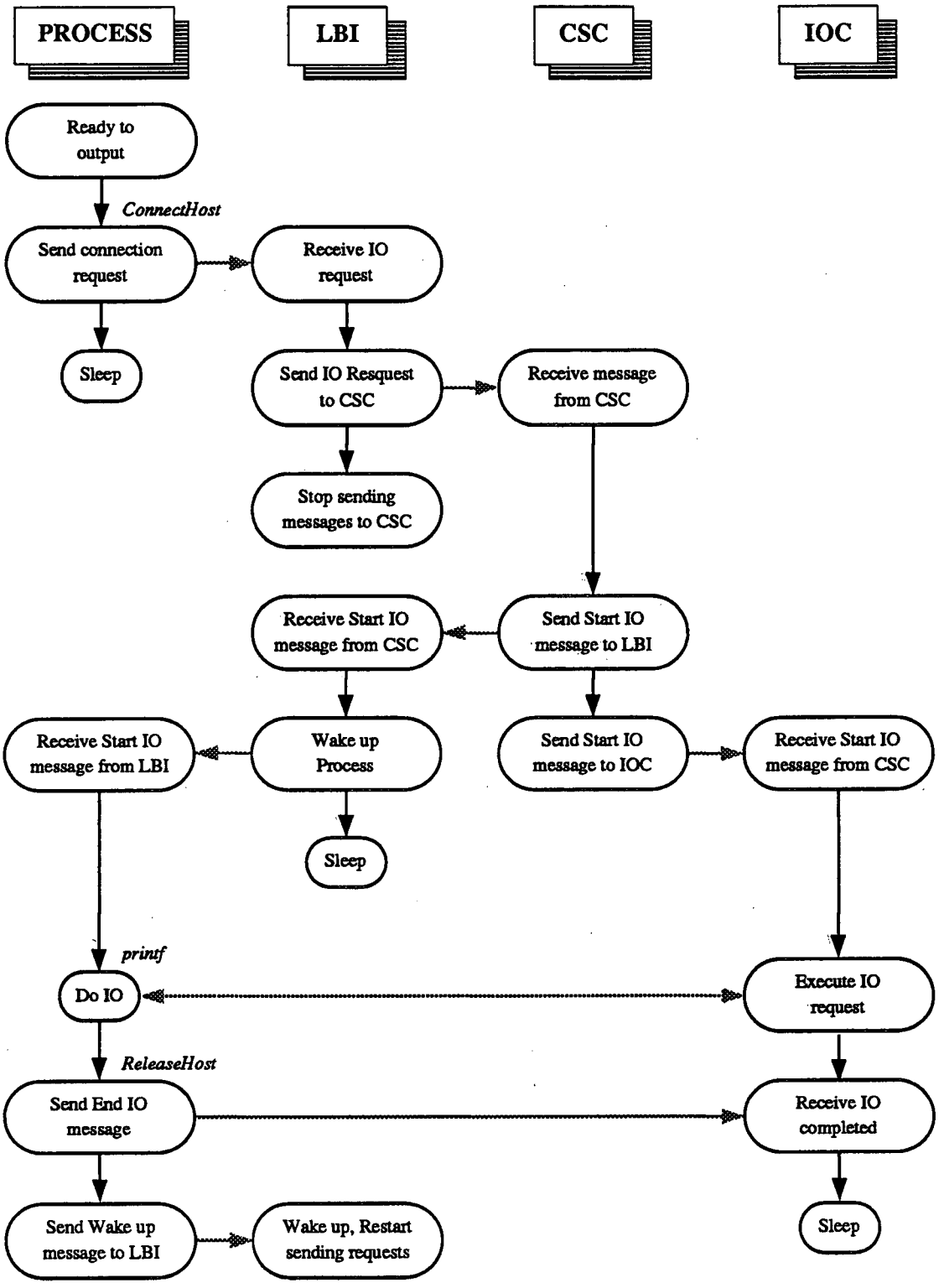


Figure 3.17: Input/output scenario

3.2.1 Allocation of Process Identification Numbers

A unique Process IDentification (PID) number must be allocated to each user process. Even processes running on different transputers cannot have the same PID number. The PIDs are explicitly allocated by the programmer and defined in a Network Information File (NIF) before program execution. The configuration file describes the transputer hardware addresses related to each PID number which is read by the CSC at initialization. For example, the configuration file would include information like: transputer 0 of VMTM board 0 runs processes with PID 0,1,2. Note that, transputer 0 of VMTM 0 maps to a unique VMEbus address.

Since synchronous communication involves a direct connection between two processes, PIDs must be defined at compilation time. The CSC must know where the processes are located, if the processes exists and if both processes are ready to communicate when receiving a connection request. Only the CSC needs to know the hardware address of the processes. User processes only are only required to know the PID of the processes that they need to communicate with.

3.2.2 Operating System Primitives

The operating system supports the following operations:

StartDynamic()

This function must be called at the start of a program running on a transputer. *StartDynamic* creates the Local Bus Interface process and starts it concurrently with the user program.

EndDynamic()

EndDynamic kills the LBI process previously created by *StartDynamic*. This function advises the CSC and the host that all communication channels with this transputer can be closed.

ConnectHost(my_pid)

ConnectHost creates a communication channel between the calling process *my_pid* and the host computer. After this call, the user process has access to the host computer for input/output operations. The user process signals the completion of I/O by executing the

procedure *ReleaseHost*. Multiple processes running on a transputer may concurrently call *ConnectHost* but the host channel will be allocated to only one process at a time.

ReleaseHost(my_pid)

ReleaseHost destroys the communication channel that was created by a previous call to *ConnectHost*.

virtual_channel = OpenChannel(my_pid, source_pid, destination_pid)

This function creates a communication channel between two processes *source_pid* and *destination_pid*, and returns a *virtual_channel* number to the calling process *my_pid*. This function must be called by two process and the returned virtual channel to both callers is identical[3].

CloseChannel(my_pid, channel)

CloseChannel destroys the *virtual channel* created by *OpenChannel*.

hard_channel = FixLink(my_pid, virtual_channel)

FixLink will initiate the creation of a circuit switch between the two processes sharing the *virtual_channel*. This function returns only when the two processes sharing the *virtual_channel* have executed *FixLink* and the path between the two processes have been created. The return value, a pointer to a hard channel, is sent to the calling process. Both processes may then exclusively communicate over *hard_channel* by using the basic communication primitives supported by the transputer instruction set.

ReleaseLink(my_pid, virtual_channel)

ReleaseLink frees the circuit switch that was previously created by the function *FixLink*. This function is called by only one of the two processes that are sharing the *virtual_channel*.

---

3 This operating system call is similar to ChanAlloc() of the Logical Parallel C compiler

SendMail(my_pid, mail, pid1, pid2, pid3, ..., -1)

SendMail distributes a *mail* message to all the PIDs *pid1*, *pid2*, *pid3* .... This routine takes a -1 terminated code as a parameter. The mail message is an integer and is broadcasted to the destination pid mailboxes via the bus. The function *SendMailList* has also been implemented where the address of an array containing PID numbers may be sent as a parameter.

mail_status = GetMail(my_pid, mail)

GetMail reads the *mail* message stored in the mailbox of the process *my_pid.* A *mail_status* is returned indicating if the mailbox was empty or full. The mailbox is located on the same transputer as the calling process.

hard_channel = ChanAlt(my_pid, virtual_channel, chan1, chan2, chan3, ..., -1)

ChanAlt cause the calling process to block until one of the channels in the argument list is ready to communicate, and the circuit switch is created. The channel may be either an input or an output. This function returns a pointer to a *hard_channel* where the message may be sent or received and the *virtual_channel* number. This routine takes a -1 terminated code as a parameter. The function *ChanAltList* has also been implemented where the address of an array containing channel numbers may be sent as a parameter.

FixLink and *ReleaseLink* system calls are referred to as **static** connection calls. In static calls, the circuit switch is created and released under the control of user processes. **Automatic** system calls have also been designed to provide communication primitives where *FixLink* and *ReleaseLink* are called by the LBI process rather than a user process. Automatic calls are:

LinkOut(my_pid, virtual_channel, buffer_address, len, release_flag)

LinkOut initiates the transmission of a message between the two processes which share the *virtual_channel.* Execution of this routine causes the calling process to be blocked until the message has been received by the destination process. Parameters to this procedure include the process identification number of the originator,*my_pid*, the *virtual_channel*, the address of the buffer, the length of the message in byte, and a *release_flag* indicating if the circuit

switched is released by the caller. If the *release_flag* is set, this routine is equivalent to the following three system calls :

```
channel = FixLink(my_pid, virtual_channel)
ChanOut(channel, buffer_address, len)
ReleaseLink(my_pid, virtual_channel)
```

LinkIn(my_pid, virtual_channel, buffer_address, len, release_flag)

LinkIn initiates the reception of a message between the two processes which share the *virtual_channel.* Execution of this routine causes the calling process to be blocked until the message has been received. Parameters to this procedure include the process identification number of the originator, the virtual channel, the buffer address for the incoming message, the length of the message in bytes, and the release flag. This routine is equivalent to the following three system calls :

```
channel = FixLink(my_pid,virtual_channel)
ChanIn(channel, buffer_address, len)
ReleaseLink(my_pid,virtual_channel)
```

Other similar automatic operating system calls have been implemented for inputting and outputting integers and bytes. These functions are : LinkOutInt, LinkOutChar, LinkInInt, and LinkInChar.

To obtain maximum utilization of communication resources, static function calls should only be used when the communication pattern between two processes manifests high **temporal locality.** If the probability that all communications in an interval of time t will be isolated between two processes is high, the communication pattern exhibits a high temporal locality. These processes are not necessarily executing on transputers near one another in the network. On the other hand, if the behavior of communicating processes has low temporal locality, the probability that a process receives two consecutive messages from the same process in the same interval t is low.

Misuse of static calls may prevent processes from communicating with other nodes by tying up external links and cables. A transputer may communicate in parallel with no more than three different neighbours (one link is statically connected to the bus). If three concurrent processes running on a transputer have established communication using static calls, no other processes on that transputer can initiate communication until a communicating process releases the resources by calling *ReleaseLink*.

Automatic calls, however, reserve communication resources for one message only. The link and cables resources are released by the operating system after the message is transmitted.

3.2.3 The Local Bus Interface

An application program is parallelized into multiple transputer programs each of which is executed on a transputer. A transputer program that uses the operating system primitives defined earlier is composed of at least two concurrent processes: the user process and the LBI process. The LBI process is started by the function call *StartDynamic*, and is scheduled as a high priority process. The LBI is inactive unless a user process makes an operating system call. One link on each transputer is under the control of the LBI process. This communication link is connected to the bus and is used to send to or receive messages from the CSC. The role of the LBI is :

1. To receive user communication requests and forward them to the CSC process
2. To ensure that all user requests are handled fairly
3. To receive CSC commands and forward them to the appropriate user process

The LBI is composed of three parallel processes: one buffer process and two link controller processes. As shown in Figure 3.18 the link controller processes communicate with the user processes via a pair of soft channels.

The *link_controller_out* process gathers requests from user processes, encodes the message, and sends them to the output buffer process, *buffer_out*. The *link_controller_in* process reads incoming data from the VMEbus, decodes the messages and distributes them to user processes. The buffer process ensures that the link controllers never become blocked while waiting to communicate with the CSC. Consequently, the *link_controller_in* process is still available to execute user requests while the

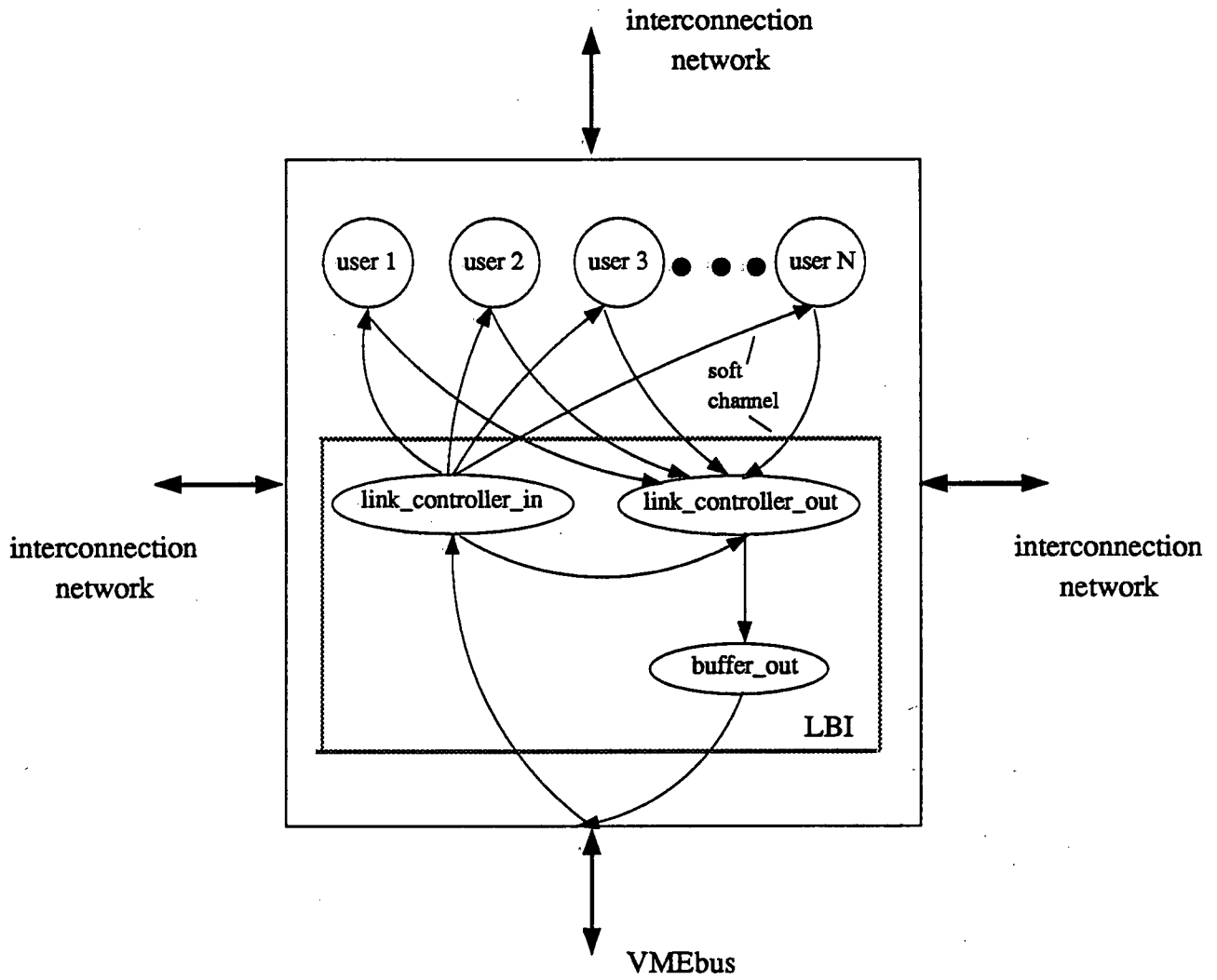


Figure 3.18: LBI process

buffer process handles the bus communication. When the CSC reads the LBI messages, all messages contained in the buffer are read.

The message format is illustrated in Figure 3.19. The first message field specifies the PID number. The *REQUEST* tag specifies the action to be taken. If the *REQUEST* tag indicates that parameters must be passed, the number of parameters is specified by *MSG_LEN*. The parameters are given in the *OS_DATA* field. Most messages are short and contain an average of two parameters.

The *link_controller_out* process must execute the user requests fairly and without deadlocks. *Alternative* primitives are available with languages such as Occam [37], CSP [38], or Parallel C [39],

PID	REQUEST	MSG_LEN	OS_DATA
2	1	1	0 - 255 BYTES

Figure 3.19: Message format

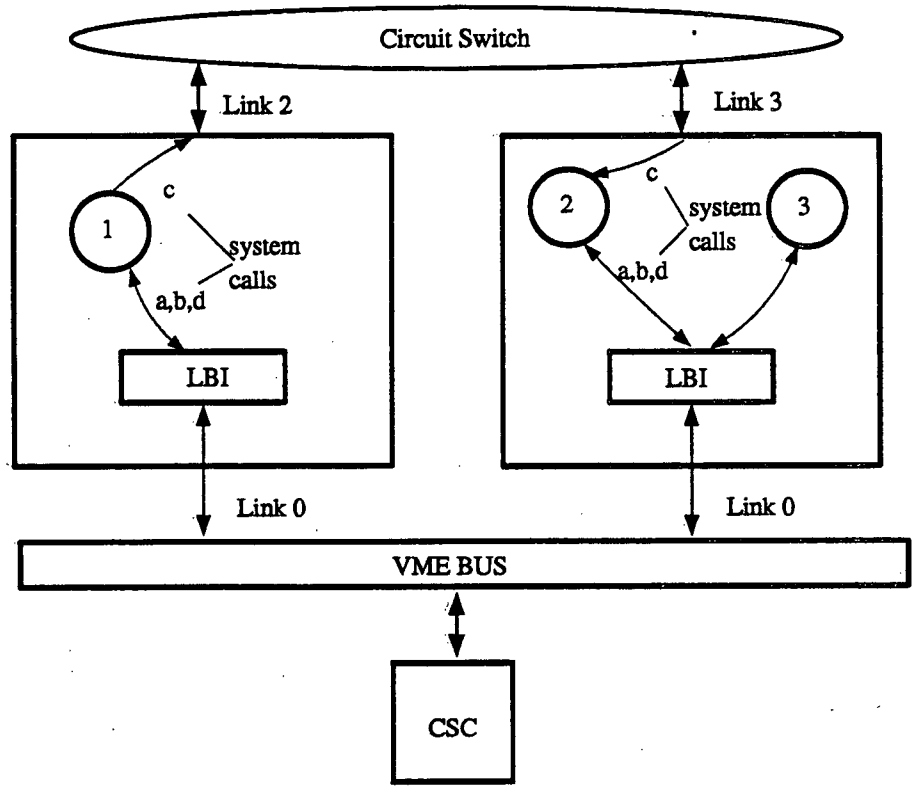
which cause the calling process to block until one of the channels in its argument list is ready. Each user process tries to communicate with the LBI via a soft channel.

The LBI executes an *Alternative* call to connect with a user process that made an operating system call. The *Alternative*, however, is not fair. The function call scans the channel list starting from the top to bottom and the first ready channel found gets connected. If the top channel, used by user process 0, is always ready, only this user process will be capable of communicating with the LBI. The *Alternative* primitives have been modified to provide fairness. The library routines are listed in Appendix B.

The interaction between the LBI, user processes, and the CSC is presented in Figure 3.20. In this Figure, the application program consists of three user processes with PIDs 1,2,3, and distributed between two transputers. Process 1 needs to send an integer to process 2. Before any messages may be sent, a virtual channel must be created between process 1 and 2. The system *OpenChannel* call is executed by both processes. The user processes are interfaced with the LBI via this system call. The LBI encodes the request and sends it to the CSC through link 0 (in this example, link 0 is reserved for the VMEbus interface). Process 1 and 2 are blocked while waiting for a virtual channel number (v) returned from the CSC via the LBI.

Once a virtual channel has been allocated, both processes may then request the allocation of a circuit switch by calling *FixLink*. *FixLink* requests must also be forwarded to the CSC via the LBI. The CSC creates a circuit switch only when the *FixLink* request have been received from both LBIs. Once the circuit switch has been established between the two transputers, the LBI receives a hardware link address from the CSC and forwards it to the user process. In this example, the hard channel addresses were link 2 and link 3. At the reception of the hardware addresses both processes may then communicate synchronously using the communication primitives supported by the transputer instruction set. During communication, the LBI is inactive (Only system calls a,b,

and d required action from the LBI). The circuit switch will stay allocated until process 1 requests a disconnection (ReleaseLink).



v = OpenChannel(1,2)	(a)	v = OpenChannel(1,2)
L2 = FixLink(1,v)	(b)	L3 = FixLink(2,v)
ChanOutInt(L2,integer)	(c)	integer = ChanInInt(L3)
ReleaseLink(1,v)	(d)	

Figure 3.20: Interaction between user processes, the LBI, and the CSC

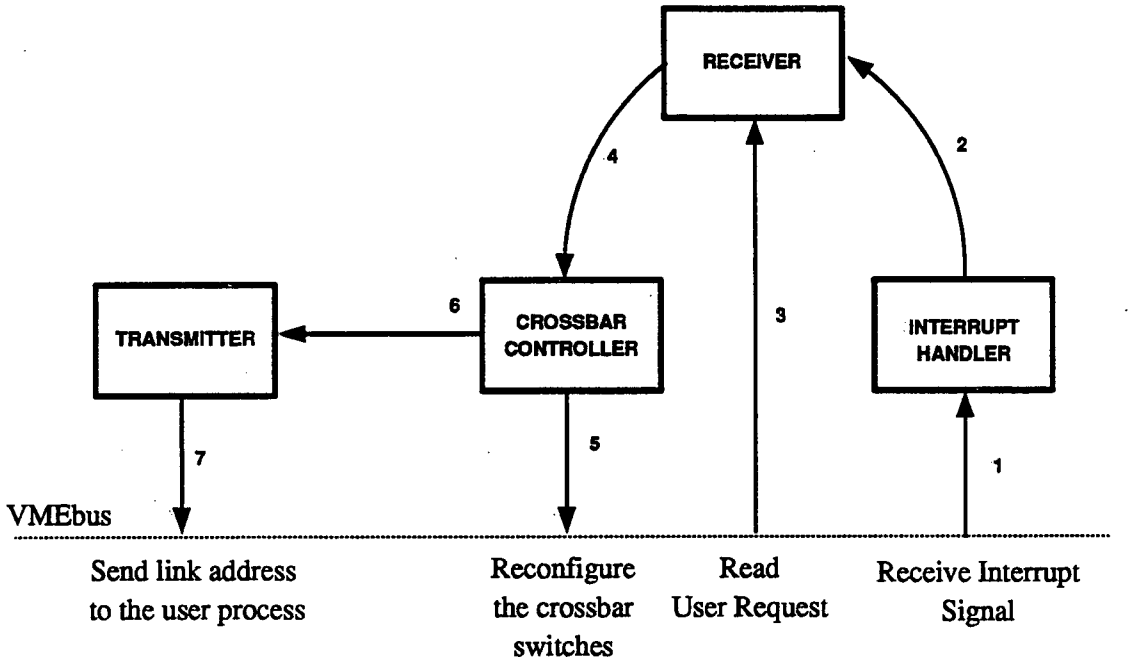
3.2.4 The Central Switch Controller

The Central Switch Controller is composed of seven parallel processes: the Crossbar Controller, the Interrupt Handler, the Receiver, the Transmitter, the Host Interface, and the Monitor. All processes are scheduled with low priority except for the Interrupt Handler. Figure 3.21 (a) shows a block

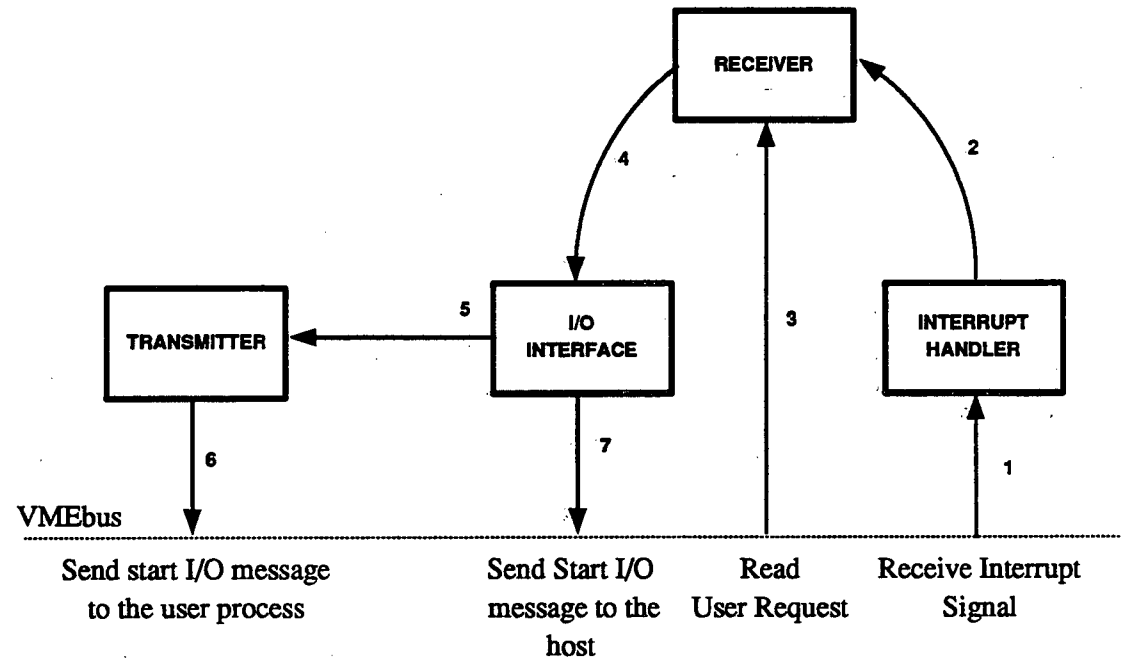
diagram of the interaction between the CSC processes when a circuit switch is created. The numbers printed besides the arrows indicate the flow of actions taken by the CSC as soon as a VMEbus interrupt is created:

1. The Interrupt Handler reads the interrupt source register of the interrupter board and decodes the addresses of the transputers that are waiting to send a message to the CSC.
2. The Interrupt Handler sends the decoded addresses to the Receiver.
3. The Receiver retrieves the user's requests at each node.
4. The Receiver transmits the messages to the Crossbar Controller for processing.
5. The Crossbar Controller creates a circuit switch and reconfigures the network topology, if the user requests require creation of a new communication path.
6. The Crossbar Controller sends reply messages to the Transmitter.
7. The transmitter sends the Crossbar Controller reply messages to the LBIs.

Figure 3.21 (b) illustrates the flow of actions created when a user process requests a connection with the host computer for input/output. The first four steps are similar to the previous example. Once the I/O Interface process receives the message from the receiver, the I/O Interface process sends a start I/O message to the user process via the Transmitter. It then wakes up a process running on the host computer, which will communicate directly with the user and provides input/output support.



(a)



(b)

Figure 3.21: Central Switch Controller

3.2.4.1 The Interrupt Handler

As shown in Figure 3.22, the Interrupt Handler process is composed of two main modules: the Event Handler and the Buffer. The Event Handler is scheduled in response to an interrupt on the VMEbus. Its task is to read the interrupt source register of the interrupter board, to decode all the node addresses that have pending reconfiguration messages, to disable the interrupt for all decoded node addresses, and to send the adapter address from where the interrupt was created to the Receiver process. The 32–bit interrupt source register is accessible from the VMEbus. Each bit of the register is directly associated with a link adapter of a cluster module; each link adapter is connected to a transputer link.

Welch [51] has shown that if a high priority process tries to communicate with a lower priority process, the transputer micro-coded scheduler schedules the high priority process only when the lower priority process executes. The transputer does not support dynamic priority allocation in which a low priority process dynamically acquires the priority of its caller. In order to prevent the event handler from becoming blocked while trying to communicate with the Receiver, high priority Buffer-Prompter processes have been created. There is one Buffer-Prompter processor for each transputer. The Buffer-Prompter processes, each having only two lines of code, receive the adapter address from the Event Handler and send it to the Receiver. With this design, the Event Handler is never blocked. Only the Buffer-Prompter process may become blocked waiting for the Receiver to be scheduled.

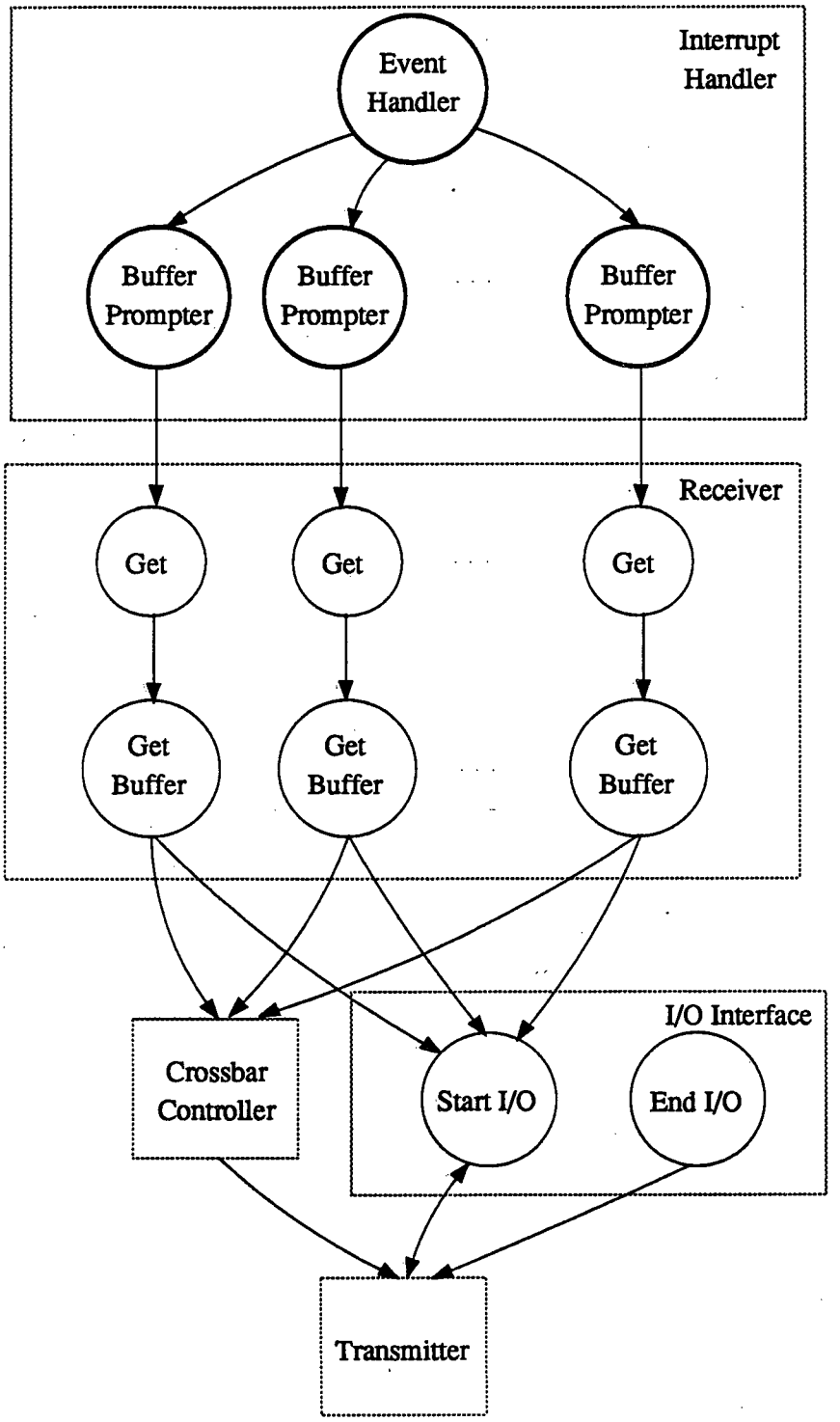


Figure 3.22: Breakdown of CSC processes

3.2.4.2 The Receiver

As presented in Figure 3.22, the Receiver process is composed of multiple parallel Get processes. The Get processes are awakened by a Buffer-Prompter message which contains the address of a node requesting communication. Once scheduled, a Get process will read all messages that have accumulated at the LBI interface, buffer the data to make it accessible to the Crossbar Controller, and re-enable the interrupt at this particular node. Using multiple Get processes provides a fair environment.

A fair environment will guarantee that any communication requests will be serviced equally. If there are n nodes in the network with pending communication messages to be sent to the CSC, 1/n of the bandwidth will be allocated to each node. For example, consider a network composed of 8 transputers where 4 nodes initiate communication requests simultaneously. The interrupt handler will decode 4 node addresses and select 4 Get processes from the ready pool. If one node has a greater amount of communication requests, starvation will not occur. All Get processes run at low priority and are scheduled in round-robin where an equal processing time is allocated to each process. If a new interrupt occurs while some Get processes are still active, another Get process is scheduled and runs concurrently with other running processes. A fair *Alternative* function is used at the Crossbar Controller and the I/O Interface to ensure that each Get process is provided with an equal amount of communication bandwidth.

3.2.4.3 The I/O Interface

The I/O Interface is composed of two processes: the Start I/O and End I/O processes. The I/O Interface communicates with the IOC process running on the host computer. The communication between the I/O Interface and the host computer is memory mapped, since the bus master transputer module is equipped with dual-ported memory.

When the Receiver process receives a *ConnectHost* message, the Receiver sets a word in memory indicating the address of the transputer requesting I/O. The Start I/O process is scheduled periodically and reads this shared memory location. If I/O requests are pending, the Start I/O process sends a message to the IOC which takes appropriate actions.

The End I/O process periodically polls a memory location which is memory mapped with the IOC. When a user process terminates I/O, the IOC sends a message to the End I/O process. The End I/O process advises the Transmitter that the bus link of the transputer that just completed I/O is now free.

3.2.4.4 The Transmitter

As shown in Figure 3.22, the Transmitter process sends Crossbar Controller and Start I/O messages to the LBIs. The Transmitter gives priority to the message coming from the Crossbar Controller since a user process is waiting for the Crossbar Controller reply before it can initiate communication. Inter-processes communication are prioritized over communication with the host.

If a message arrives from the Crossbar Controller for a transputer which is communicating with the host computer, the message is queued until the input/output calls terminate. The transputer link connected to the bus is not free. When the I/O calls are completed at the node, a message indicating I/O completion is sent from the End I/O process to the Transmitter. The Transmitter will then send all queued messages for this particular transputer to the LBI.

3.2.4.5 The Monitor

The Monitor communicates with the Receiver, the Crossbar Controller, the Interrupt Handler, the I/O Interface, and the Transmitter processes via an array of soft channels. If a system error is detected by one of these processes, an error message is sent to the Monitor. The Monitor will then output an error message to the user. The monitor may also abort the CSC process in the case of a fatal system error.

3.2.4.6 The Crossbar Controller

The Crossbar Controller performs operations on user requests read by the Receiver process. When a *OpenChannel* message is received at the Crossbar Controller, memory is allocated (approximately 30 bytes) where the following channel data is stored:

1. PIDs of the two processes that are sharing the channel;

2. the hardware address of the processes sharing the channel;
3. communication status ie. processes are communicating, one process reached the communication point, or both processes are not ready to communicate; and
4. the latest configuration message that was sent to the crossbar(s) if a circuit switch was created.

The *FixLink* operating system call initiates the creation of a circuit switch which may involve on-line reconfiguration of the interconnection network. At the reception of *FixLink* requests from sender/receiver processes, the Crossbar Controller will check the following conditions. If one of the conditions (C) is met, the Controller executes the operation (O) and exits the list, otherwise it continues to the next item.

(C) If a circuit switch already exists between the two processes
(O) The circuit switch is reserved and the network does not need to be reconfigured

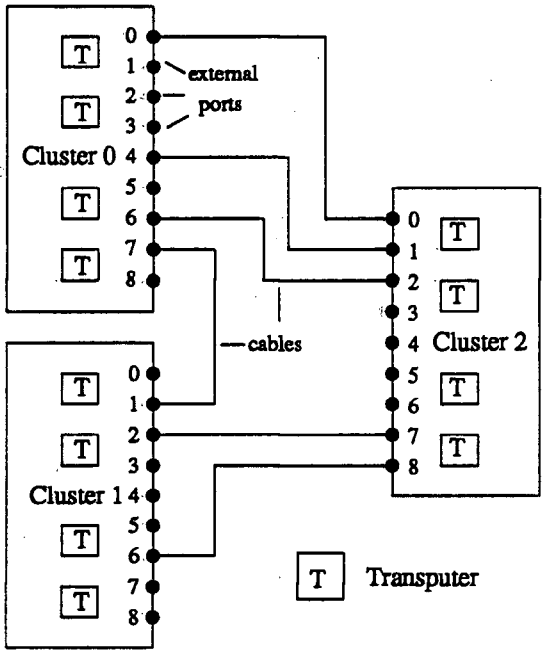
(C) If a circuit switch was created for other processes running on the same nodes as the calling processes, and the circuit switch is presently unused
(O) The circuit switch is reserved, and the network does not need to be reconfigured

(C) If the resources required to create a circuit switch are unavailable
(O) The connection request is queued

(O) A circuit switch is created and the network is reconfigured

The Crossbar Controller requires link and cable resources to create a circuit switch. The resources are stored in link and cable stacks. A link stack is created for each network node and contains the link addresses which are available for dynamic reconfiguration. A link stack holds a maximum of three links, as one link is permanently connected to the bus. A cable stack contains a list of cables used to interconnect two cluster modules. The maximum number of resources in a cable stack equates the number of external ports on a cluster module.

Figure 3.23 represents a network of three cluster modules with the corresponding resource stacks. For instance, the cable stack (cluster 0 / cluster 1) contains one cable entry indicating the connection between cluster 0, port 7 (P7) and cluster 1, port 1 (P1). The link stack of each node, however, consists of three link entries L1, L2, and L3 (link 0 is connected to the bus).



Cable stack (Cluster 0-1)

P7	P1

Cable stack (Cluster 0-2)

P0	P0
P6	P2
P4	P1

Cable stack (Cluster 1-2)

P6	P8
P2	P7

Link Stack (same for each node)

Link 1
Link 2
Link 3

Route Table

Boards	Route 1	Route 2	Route 3
Cluster 0/1	0 / 1	0 / 2 / 1	none
Cluster 0/2	0 / 2	0 / 1 / 2	none
Cluster 1/2	1 / 2	1 / 0 / 2	none

Figure 3.23: Resource stacks and route table

The number of resources required to create a circuit switch is determined by the direction in

which the messages are forwarded. To route messages through each cluster module, a channel entering a board (input port) must be linked to a channel leaving the board (output port). The Crossbar Controller maintains a lookup table called **routing table** to perform this function. The routing table is **static** where the table is created off-line, read by the Controller before the transputer network is loaded, and not changed thereafter.

An entry in the routing table gives a maximum of three possible routes in order of priority. The priority of a route decreases as the number of hops increases ie. shorter routes have high priority. In Figure 3.23 for example, a three hop circuit switch that interconnects a transputer on cluster 0 to a transputer on cluster 1 will route messages via the switches on clusters 0,2, and 1. One route, however, may contain many paths if clusters are interconnected with multiple cables. A message may be forwarded directly from cluster 0 to cluster 2 via three paths since the cable stack for Cluster 0–2 contains three cables.

To create a circuit switch, the Crossbar Controller extracts in a round-robin fashion the required resources from their respective stack, and creates a message to be sent to the crossbar switches. If one of the stacks is empty, the request is queued. The Controller, however, will try to connect a path via three different routes prior to queueing a request. The number of routes has been limited to three to save memory and to avoid lengthy searches. The request will stay in a queue until the needed resources become available as a result of a *ReleaseLink* system call.

Figure 3.24 illustrates the creation of a circuit switch between a transmitter (T1) to a receiver (T12). In this example, we assume that the selected route from cluster 0 to cluster 2 is via cluster 1. The inter-board connections are the same as Figure 3.23. Two links and two cables are extracted form the stacks and are used to create a connection message to be sent to the respective switches (XB 0, 1, and 2).

The routing algorithm is **adaptive**. An adaptive algorithm readjusts with the traffic load of the network. An increased of network traffic is detected when a cable stack is found empty. An empty stack will force the creation of a circuit switch which consists of multiple hops that circumvents the traffic load. The cost of routing messages, however, increases as the number of hops increases.

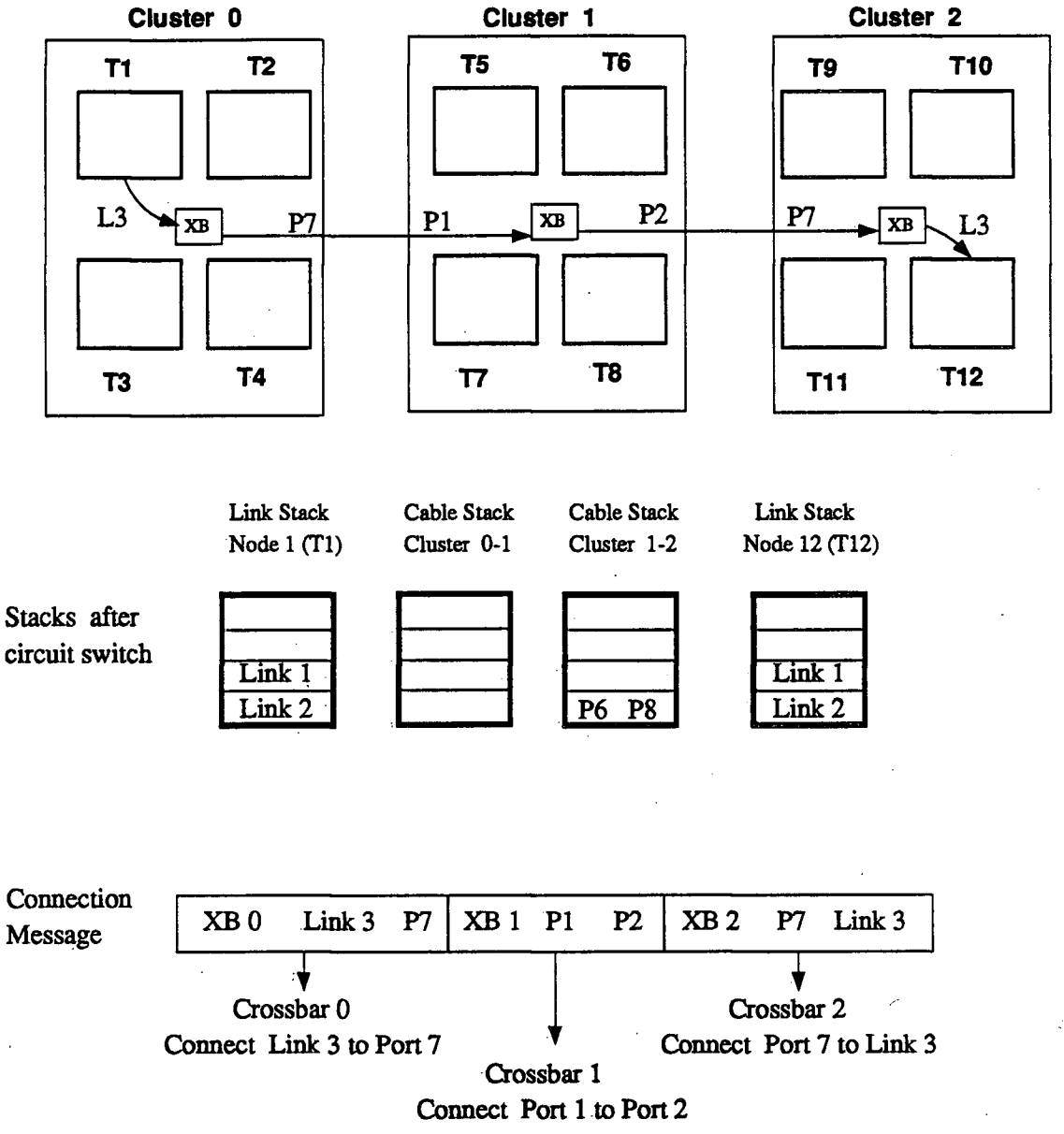


Figure 3.24: Resource allocation

For the standpoint of fault-tolerance, the routing algorithm is categorized as robust and reconfigurable. The robustness is achieved when the algorithm is capable of selecting a redundant path to bypass a failure. A faulty component may be bypassed by deleting the resources related to the faulty component from the stacks. In Figure 3.24 for example, if link 1 of node T1 becomes faulty, the crossbar controller deletes link 1 from the link stack which will imply that no more connections will be established with the faulty link.

The software overhead attributed in routing message around a faulty component is equivalent to bypassing a communication hot spot. Once the faulty resources are removed from the stacks, the routing algorithm processes the connection requests exactly as if the system was fault-free. The communication scheme is reconfigurable.

3.2.5 The I/O Controller

The I/O Controller process (IOC) runs on the host computer and executes input/output commands received be the user processes. As shown in Figure 3.25, the IOC is composed of UNIX processes which are: a Master process, Slave processes and I/O Servers.

The Master process communicates with the I/O interface process of the CSC. Four registers have been created for communications which are input status, input data, output status, and output data. A user process on node 1 for example requests a connection with the host. The Start I/O process of the I/O Interface process will set the first bit of the input data word and set the input status register to valid. The master process is polling the input status register until the value becomes valid. When the status register becomes valid, the Master sends a message to a Slave process which forwards it to a Server process. The Server process starts polling the transputer bus adapter which requested I/O, hence communicates directly with the user process that requested I/O.

The original version of the Server process was written by Logical Systems [39], where the server processes were continuously polling all the transputer adapters at the fastest speed for the duration of the transputer program execution. This version was modified to ensure that the server processes are only polling the VMEbus when an I/O request is pending. This new version, decreases the VMEbus traffic and permits multiplexed accessing of the transputer bus links between the CSC and the IOC processes.

When the Server process receives a *ReleaseHost* request, a message is sent for the Server to the Master. The master then indicates the I/O completion to the End I/O process of the I/O Interface by setting the status and data output registers. Figure 3.25 shows that the Server process 8 has completed I/O.

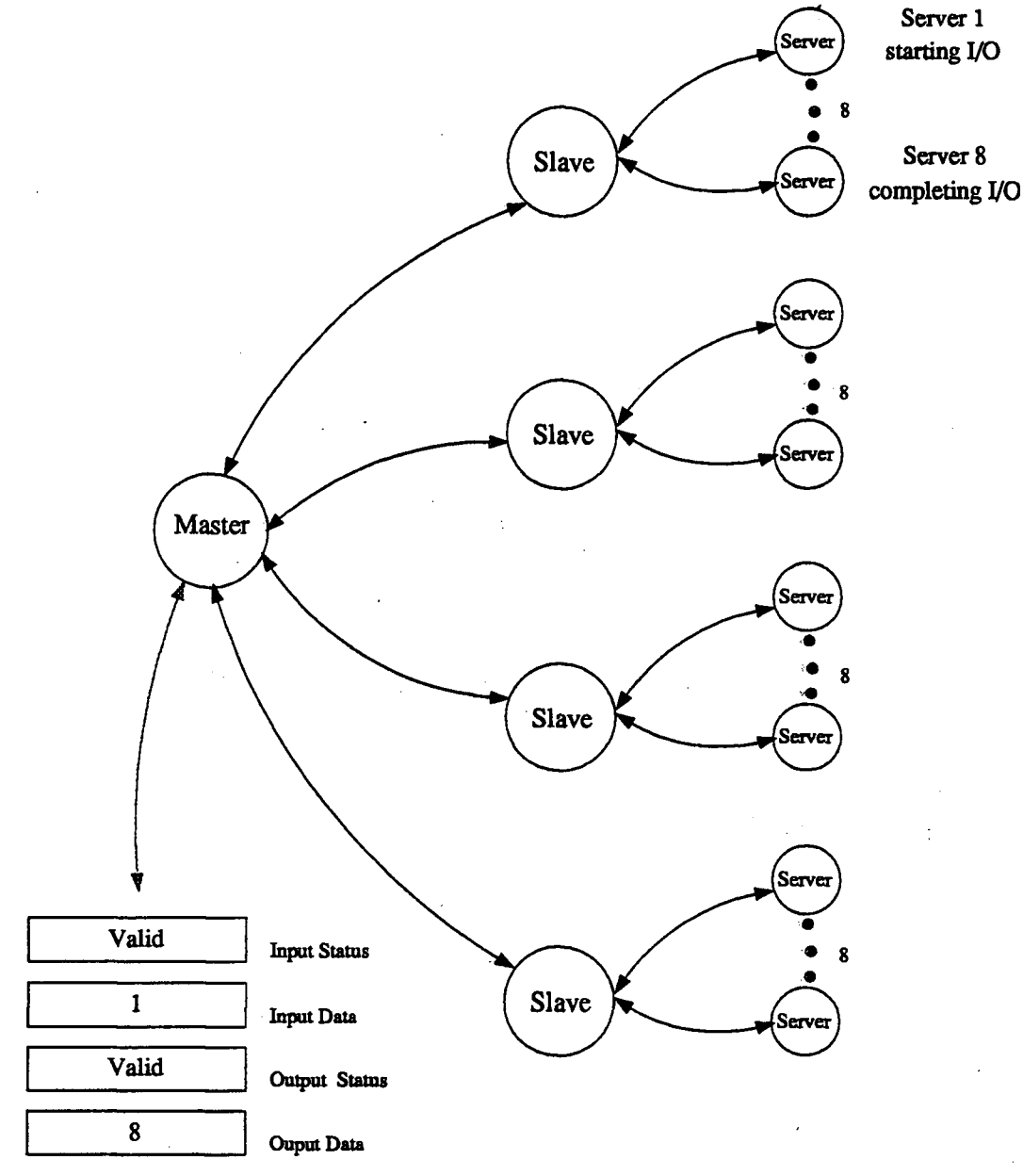


Figure 3.25: I/O Controller process

Chapter 4

Performance Analysis

Communication protocol performance is a critical measurement for a multiprocessing architecture. It is important that the time needed to establish a communication be small relative to the total communication time. In this chapter, test programs were developed to evaluate the efficiency of the communication hardware and the reconfiguration overhead using communication patterns ranging from one to several hops. The performance of the operating system design is also tested by running processes that exhibit temporal and spatial communication patterns.

In addition, the synchronous communication protocol is compared with the FPS-T Series hypercube, which is running a Store-And-Forward model developed by Helminen [1].

4.1 Communication Time

The communication time may be subdivided in two parts: initialization overhead and transport overhead. In a circuit switching strategy, the initialization overhead represents the time to create a circuit switch. The transport overhead represents the time to transmit a message once a circuit is established. Tests programs were design to calculate the total communication time of the system.

4.1.1 Initialization Time Overhead

In order to derive the upper and lower bounds of the reconfiguration overhead when a circuit switched is created, the following communication pattern was modeled. The processors formed a ring topology. Each node must send two consecutive messages to every other node in the network where the distances vary from one to eight hops.

The topology reconfiguration to send the first message creates an upper bound delay. When a sender and a receiver process initially call *FixLink*, all crossbar switches along the path are configured. The second message to be sent will not change the topology configuration, as the first communication topology will still be valid. The overhead created by the second transmission will then represent the lower bound. The length of both messages was set to 256 bytes but it is irrelevant for this test. Statistics were gathered from the LBI and the CSC.

The total reconfiguration time is a function of the network diameter (D) and is given by the following expression:

$$T_s = T_{lbi} + T_{csc}$$

where:

T_s represents the Total Software reconfiguration time;

T_{lbi} is the Time delay created at the Local Bus Interface;

T_{csc} is the overhead of the Central Switch Controller.

In the present case, the network diameter (D) is equivalent to the number of hops traversed by a message.

The communication delay, T_{csc}, will depend on whether the connection request causes the creation of a completely new path (delay upper bound) or if a path from a previous request had already been created and is still valid (lower bound). The creation of a new path causes the switch controller to do the following actions:

1. Interrupt handling;
2. Creation of the reconfiguration path message to be sent to the crossbars;
3. Crossbar reconnections;
4. Transmission of a reply to the user process.

Steps two and three are expensive as the number of hops increases. As shown in Figure 4.26, up to 40% of the total reconfiguration time may be spent sending configuration messages to the various switches. The lower bound delay , Figure 4.26, was found to be independent of the path. If the path is already valid, step 2 computation is very short and step 3 is completely eliminated.

For a circuit switching protocol, a routing mechanism which minimizes the number of hops will, in addition to minimizing the communication delays, also minimize the reconfiguration delays.

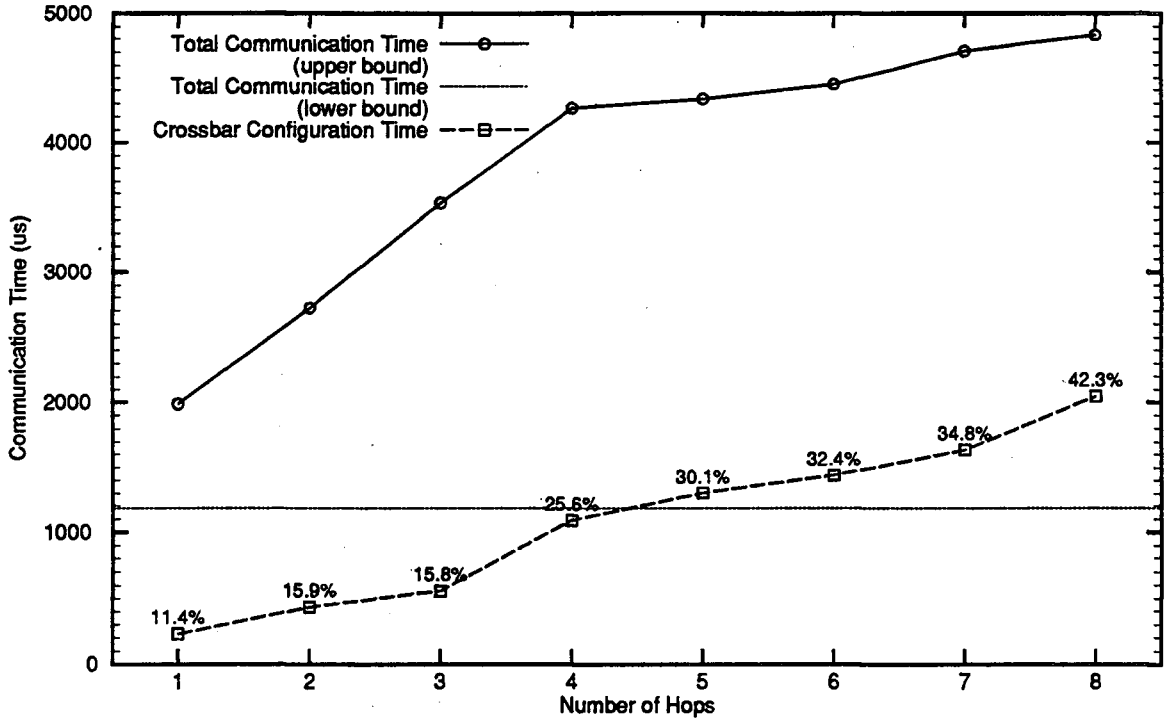


Figure 4.26: Initialization time overhead

4.1.2 Transport Time Overhead

A test program was designed to investigate the transmission speed of the communication hardware. A single link unidirectional communication pattern was chosen in order to achieve the optimum link speed. The link capacity of each transputer was set to 20MBits/sec. Messages ranging from 4 bytes to 8K bytes were sent across the link. The transfer was repeated 1024 times for each message length and for paths varying from zero to eight hops. An input/output process was executed at high priority in order to obtain a better clock granularity (1 tick = 1 microsecond). The results are presented in Figure 4.27.

The communication throughput (mean link speed in Kbytes/sec) is degraded significantly as the number of hops increases. The throughput degradation is due to a signal delay occurring at each crossbar switch for clock resynchronization in order to avoid signal skews. This effect is magnified by the fact that the communication protocol of the transputer expects an acknowledgement for each byte sent and the transmitter is blocked until an acknowledgement is received.

Consequently, for a circuit switching protocol, the bandwidth allocated to a communication channel will depend on the number of switches that the signal must traverse. The crossbar network

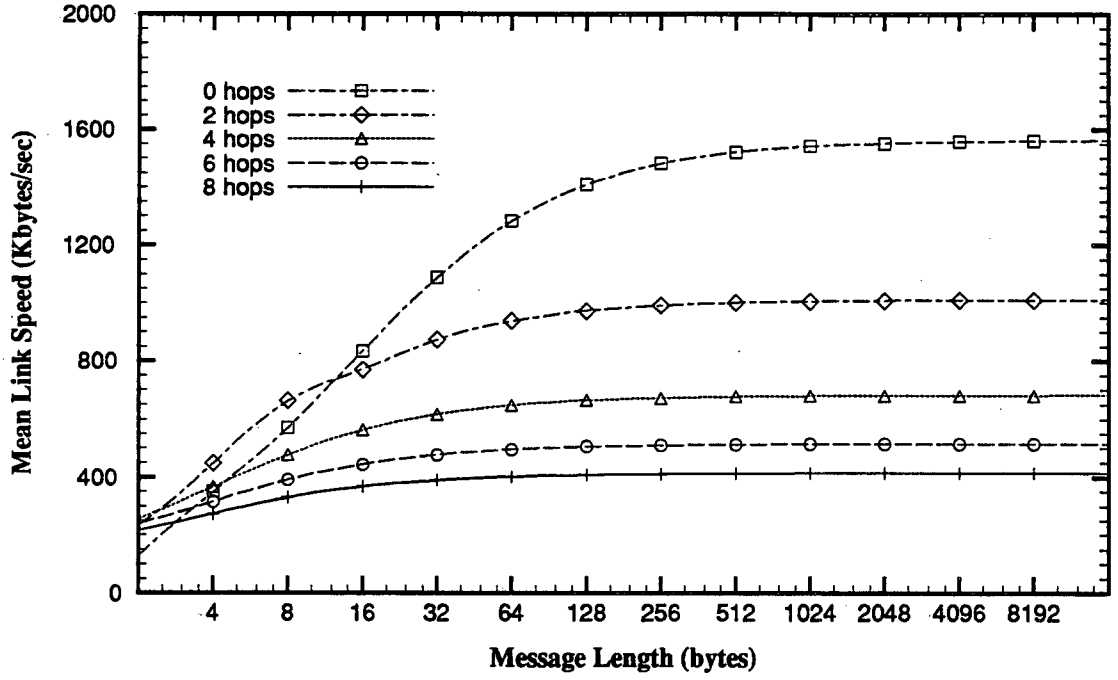


Figure 4.27 Transport Time Overhead

should be connected to minimize the number of hops and the routing algorithm should be designed such that the selected path between the transmitter and the receiver nodes also crosses a minimum number of hops.

4.2 Synthetic Communication Benchmark

4.2.1 Temporal versus Spatial Locality

In order to study the communication protocol under various network loads and communication patterns, a synthetic benchmark program based on the work done by Grunwald and Reed [52] has been developed. This model attempts to replicate a variety of global communication patterns that are representative of actual applications. Each node of the network executes a copy of the model and generates network traffic that reflects both spatial and temporal locality.

Temporal locality defines the pattern of internode communication in time. If an application exhibits a high temporal locality, the probability that all communications will be isolated between

a subset of nodes in an interval of time t is high. These nodes are not necessarily physically near one another in the network. On the other hand, if the behavior of the application has low temporal locality, the probability that a node receives two consecutive messages in the same interval t is low.

The implementation of temporal locality is based on the Least Recently Used Stack Model (LRUSM) developed by Denning [17]. This model was developed to study virtual memory paging algorithms. In the adaptation of LRUSM made by Grunwald and Reed, a destination node in a network is analogous to pages in address space of a process in a memory management context. Each node has its own stack containing the n nodes that were most recently sent messages. For any time t, the stack distance d(t) associated with communication c(t) is the position of c(t) in the stack defined just after communication c(t-1) occurs. These distances are assumed to be independent random variables with the probability [d(t) = i] = b_i for all t. In this model, $\sum_{i=0}^{n} b_i < n$, which implies that a stack miss can occur (d(t) > n). For example, if b_1 = 0.5, there is a 50 percent probability that the next destination node will be the same as the previous transmission. When a stack miss occurs, a new node is added to the top of the stack. The selection of the new node is based on the spatial locality model.

Spatial locality defines the communication pattern based on physical proximity. Two types of spatial models have been derived: uniform and sphere of locality. In the uniform model, any node can be chosen to be the destination with equal probability. This distribution makes no assumption about the communication pattern. It will presumably give the upper bound of the mean internode distance since most application programs manifest some measure of locality.

In the second spatial model, sphere of locality, each node is considered to be the center of a sphere of of radius L. There is a probability ϕ of transmitting a message to a node inside the sphere, and a probability 1 - ϕ of transmitting a message to a node outside the sphere.

4.2.2 Test environment

The benchmark program is divided into two parts. The first part is executed by the host computer and generates a sequence of messages, depending on the benchmark model parameters, that will be sent by each node.

The second part is executed by a simulation program running on each transputer. This program consists of a Controller process, a Receiver process, and a Transmitter process. The Controller process reads the message file created by the host and waits for a start command from a master transputer node. At the reception of the start command, the Controller distributes messages to each Sender and Transmitter process. When the test is completed, the Controller process sends the time to the host computer for analysis.

4.2.3 Results and discussion

The first test evaluated the effects of varying temporal locality parameters of the model. The interconnection network was connected in a three dimensional hypercube to minimize the number of hops. Each node sent 300 messages with a message length taken from a negative exponential distribution with a mean message length of 8192 bytes. The temporal stack size was set to one entry and the probability b_0 was varied between 0 and 1. The results are shown in Figure 4.28.

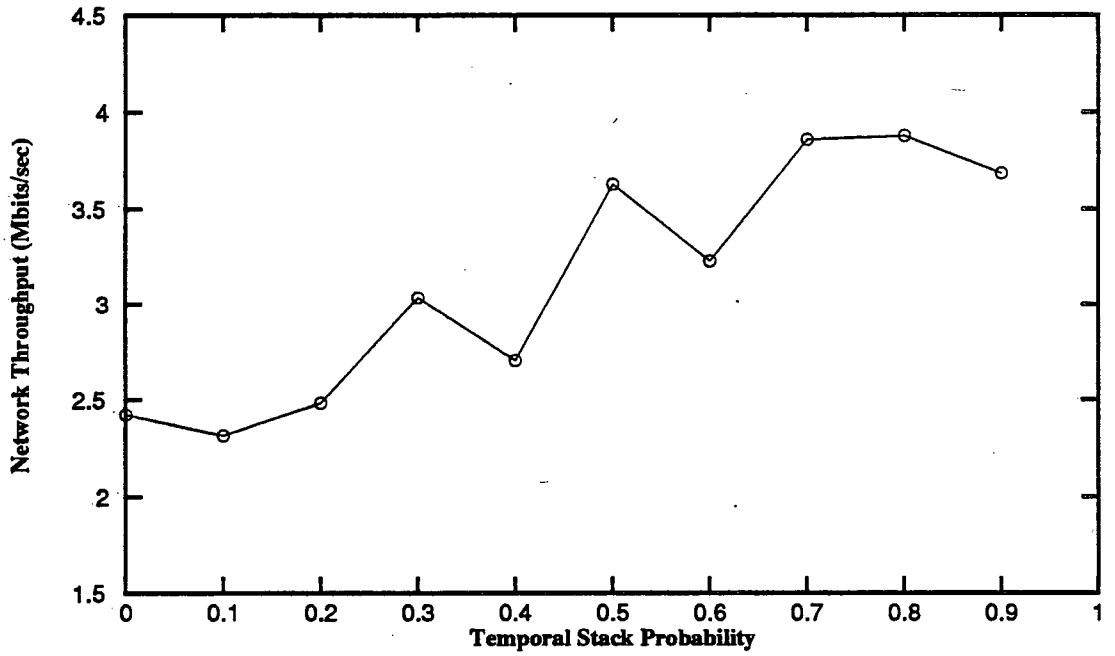


Figure 4.28: Temporal Locality Model

The throughput increases by 40% as the temporal locality parameter increases from zero, indicating a uniform communication pattern, to one, indicating a communication pattern with a high

temporal locality. This throughput increase is due to the circuit switching algorithm which allows multiple messages to be sent with only one network reconfiguration overhead being incurred at the start of the communication.

The second test program was designed to evaluate the influence of spatial locality. Each node sent 300 messages for each run. The message length for each run varied from 8 bytes to 8 Kbytes. The spatial locality parameter was set to 0.5 ($\phi = 0.5$). The curves obtained are presented in Figure 4.29. For comparison, the results obtained using a communication pattern exhibiting temporal locality ($b_0 = 0.5$) are also shown in Figure 4.29.

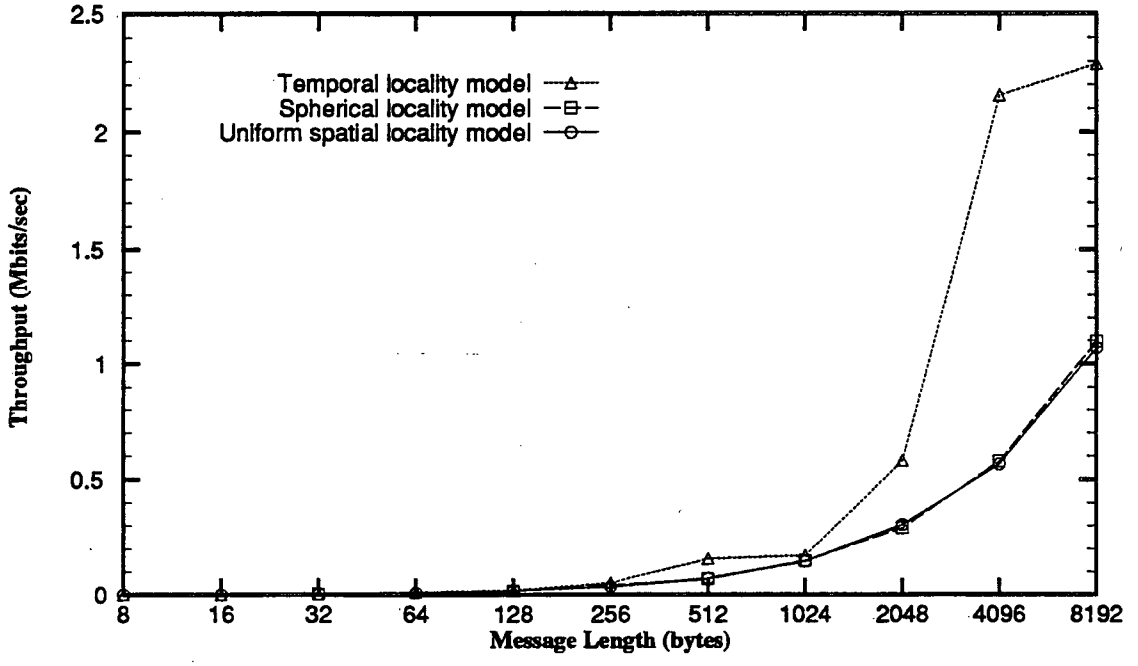


Figure 4.29: Spatial Locality Models

From Figure 4.29, it is shown that the spatial locality of a communication pattern does not improve the throughput of this network beyond that obtained using a uniform communication pattern. Temporal locality, on the other hand, does improve the throughput. Therefore, a circuit switching communication protocol is well suited to applications that exhibit temporal locality.

4.3 Comparison with FPS-T Series

The tests described in Section 4.1, 4.1, and 4.2 are functionally similar to the tests run by

Helminen [1] to evaluate the performance of the FPS T-Series hypercube. The FPS T-Series, as described in Section 2.2.4, is a transputer-based architecture. The communication protocol package used on the FPS T-Series is an asynchronous store-and-forward model.

Helminen [1] approximated the transport time overhead (T_h) of the FPS T-Series with the following linear model:

$$T_h = T_{start} + NT_t$$

where T_{start} is the start up time or latency of the communication, T_t is the transmission time per byte, and N is the number of bytes sent. The linear model obtained by Helminen was calculated with a zero hop path (no switch between the sender and receiver) and with the transmission link capacity set to 10MBits/sec. The linear model, expressed in microseconds, is given by:$T_h = 8.3 + 1.44N$

Table 4.1 was derived using regression analysis of the results obtained from the hardware tests described in Section 4.1 where the link capacity was set to 20 MBits/sec. The last column represents the maximum throughput obtained during the test. Since the link capacity of our network is two times greater than the FPS T-Series hypercube, the coefficient of the linear term in our regression results is approximately double.

hops	Tstart (usec)	Tt (usec)	MBits/sec
0	8.07	0.62	12.2
1	5.07	0.76	10.0
2	4.75	0.97	7.90
3	4.84	1.20	6.34
4	4.90	1.43	5.33
5	4.84	1.67	4.58
6	4.85	1.90	4.02
7	4.93	2.12	3.60
8	4.94	2.36	3.24

Table 4.1 Transport time overhead regression analysis results of the synchronous communication protocol

An equation which represents the total communication overhead (T_{total}) was also derived by Helminen which represents the time (in microseconds) for a message of length N to be sent between two nodes a distance D apart on the FPS T-Series hypercube:

$$T_{total} = 788.23 + 268.6D + 1.45ND$$

The first term is attributed to initialization overhead, the second term to routing overhead and the last term to the asymptotic communication rate.

For our circuit switching protocol, results of Section 4.1 and 4.1 were combined, and the following equation, which gives the total communication time, was derived:

$$T_{total} = (T_{start} + T_{lbi}) + (T_t N + Tcsc)$$

Figure 4.30 represents the total communication time that a fixed message length (512 bytes) takes to cross a variable network diameter (1 to 8 hops). The lower bound corresponds to the situation where all circuit switches are already established and the upper bound represents the situation where each message transmission requires reconfiguration of the network in order to establish a circuit switch.

For a route traversing four or less hops, the upper bound of the communication time using a synchronous communication protocol is comparable to the communication time using the asynchronous communication protocol of the FPS-T series. For routes traversing more hops, synchronous communication using circuit switching provides better communication time. Figure 4.31 represents the comparison between the two models if the network diameter is kept constant (D = 4) and the message size varies from 0 to 8K bytes. It is shown that for messages greater then 512 bytes, the circuit switching mechanism provides better performance.

For circuit switching, the initialization overhead is high compared to the store-and-forward model. The transport time, however, is relatively small for circuit switching. Because of this, circuit switching performs better for longer messages and longer routes where the transport time dominates the communication time.

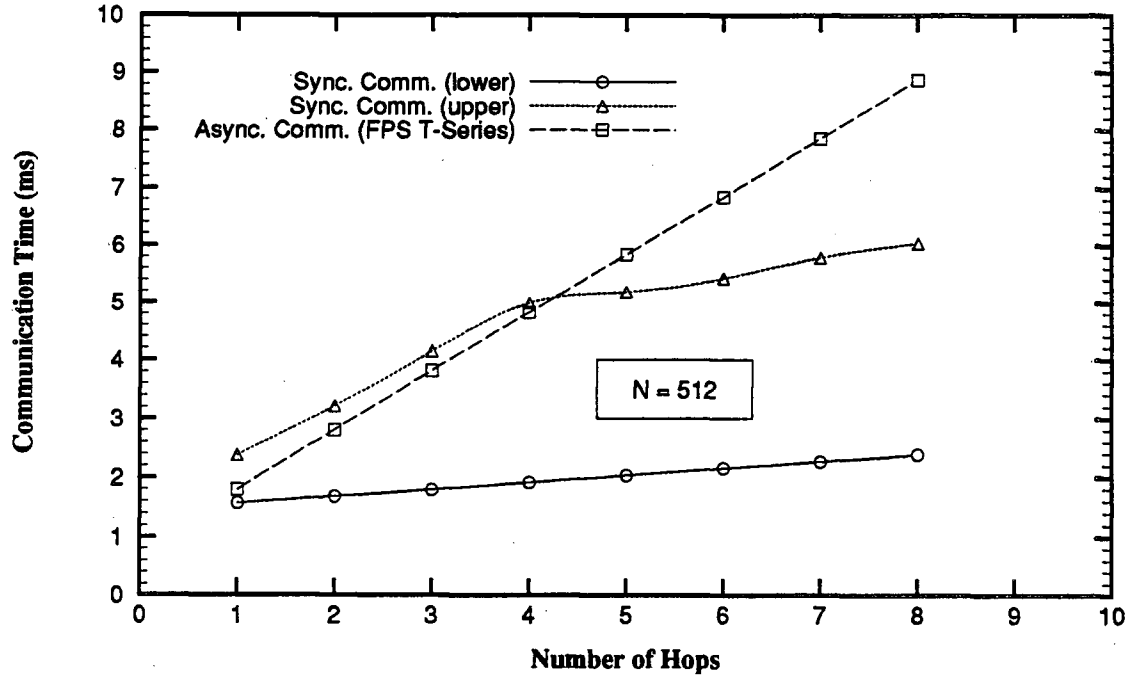


Figure 4.30 Communication Time as a function of number of hops for synchronous and asynchronous (FPS) communication protocols.

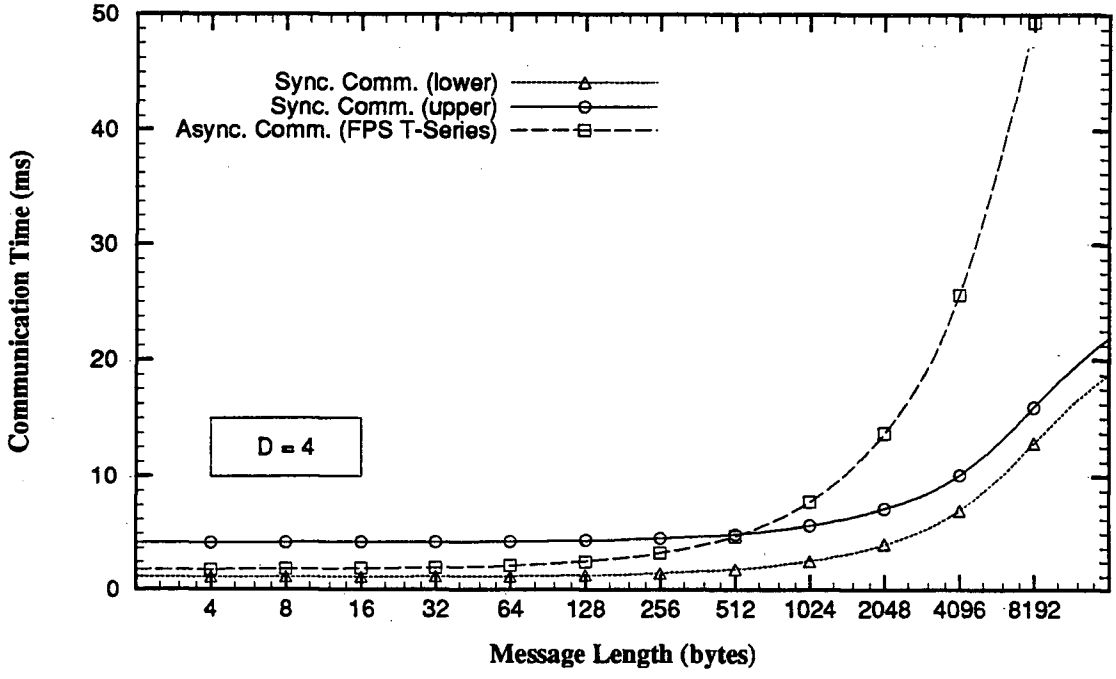


Figure 4.31 Communication Time as a function of message length for synchronous and asynchronous (FPS) communication protocols.

Chapter 5

Conclusions

A distributed system for real-time robotic control has been investigated. The hardware architecture is composed of Inmos T800 transputer nodes connected to a hybrid reconfigurable interconnection network which consists of point-to-point link connections and a global bus. An operating system has been designed to provide inter-node synchronous communication and input/output support. In this chapter, a summary of the results is presented and suggestions for further research are outlined.

5.1 Summary

A transputer-based distributed architecture, consisting of a novel hybrid interconnection network, was evaluated for applications in robotics and its related areas. The hardware used to implement a distributed system resides in two VMEbus card cages and consists of a host computer, two transputer bus masters and cluster modules. The transputers of the cluster modules communicate via point-to-point connection links. The point-to-point connection links are connected to switches and form a distributed crossbar interconnection network. The reconfigurability of the interconnection network provides important advantages ranging from increased performance, through design flexibility to greater fault-tolerance.

Each node in the network is directly accessible from the shared bus. This bus may be used for error checking and broadcasting of information. In addition, the bus is also an efficient communication medium with the host computer for software development and input/output support.

On-line reconfiguration has been implemented for fault-tolerance and flexibility in circuit-switched applications.

An operating system has been designed to support inter-node communication and node-to-host communication. A synchronous communication protocol has been implemented with a circuit switching strategy. The synchronous communication protocol is deadlock free where processors do not need to store messages and no buffer management is required. A circuit switching strategy is

attractive to real-time systems since a maximum communication latency and a minimum bandwidth are guaranteed when the circuit is established.

The circuit switch is created by a central switch controller which ensures that each node has fair access to the network communication resources. The routing algorithm is adaptive where communication hot spots may be detected and bypassed. From the standpoint of fault-tolerance, the routing algorithm is robust and reconfigurable where the algorithm can use redundant paths in case of failure and can keep the communication overhead low with or without the presence of a fault.

The input/output functions supported by the operating system provide arbitration between concurrent processes requesting I/O operation on the same node. Most parallel systems, such as those using Occam or Parallel C, support input and output for only one process per node.

The communication time to send a message from a source to a destination may be divided into two parts: the initialization time and the transport time. In the present implementation of a circuit switching protocol, the initialization time is the software overhead time required to create a circuit switch. The transport time is the time required by the hardware to transmit a message once a circuit switch has been established. The initialization overhead and the transport time have both been calculated and are found to increase with the number of hops over which the message must travel.

A simulation program has been developed to measure the performance of the communication protocol with various temporal and spatial communication patterns. The results of the simulations show that for longer message lengths or greater temporal locality, an increase in throughput is observed.

The circuit switching protocol was compared to the store-and-forward model developed by Helminen's [1] for the FPS T-Series hypercube. At its worst case, the communication time of the synchronous communication protocol is comparable to the asynchronous communication protocol of the FPS T-Series and that as the network diameter or as the message length increases, a 2–5 fold increase in speed is observed.

5.2 Contributions

The following is an outline of contributions accomplished in this thesis:

- A complete library of driver routines was written in Occam and Parallel C for the BBK-V2 bus master transputers.
- An operating system shell supporting synchronous communication was designed. The modularity of the design will ease future software enhancements. It is probable that the majority of the software enhancement will be directed to the CSC Crossbar Controller process. The Receiver, Transmitter, Interrupt Handler, Monitor and input/output support may be integrated with a new crossbar controller with minimal modifications which makes the operating system an efficient development environment.
- Experience using both Occam and Parallel C have been acquired. The Occam language with its guarded communication primitives, its communication protocol definition and its compact syntax for representing processes provides an attractive parallel processing environment. Occam, however, does not support dynamic memory allocation. In contrast, Parallel C provides all the C language dynamic memory allocation features. Dynamic memory allocation is essential for the implementation of task migration which is an important feature for reliable systems.
- A Fair Alternative routine, as listed in Appendix B, has been added to the Parallel C Library. These routines provide each process a fair access to communication resources.
- The I/O server process received from Logical Systems polls the VMEbus at the fastest possible rate. If multiple I/O servers are executing concurrently, the VMEbus traffic is greatly increased. The I/O server was modified to provide an adaptive polling rate which is dependent on the frequency of I/O requests.

5.3 Future Areas of Research

This following is a list of possible extensions to the work that has been done in this thesis:

- The operating system should support task migration for fault-tolerance purposes.

- The number of VMEbus link adapters available on each cluster board is presently four and one is reserved to connect the crossbar configuration link. Therefore, three link adapters must be shared by four processors. The design of the operating system currently limits the number of dynamically reconfigurable transputers to three. A hardware board should be designed with multiple IMS C012 link adapters where the configuration link of each switch could be connected.
- A distributed switch controller should be implemented in order to avoid a switch controller bottleneck as the number of processors in the system grows.
- The communication protocol should prioritize the communication requests and support resource preemption. Prioritized communication would increase the system predictability which is an important feature for real-time systems.

References

[1] B.K. Helminen. *An Analysis of the Floating Point and Communication Performance of the FPS T-Series Hypercube*. Master's thesis Michigan Technological University, 1988.

[2] T.N. Mudge and T.S. Abdel-Rahman. *Architectures for Robot Vision*. Gordon and Breach science publisher, New York, United States, 1987.

[3] Y.Wang and S.E.Butner. A new architecture for robot control. *IEEE Int. Conf. Robotics Automation*, pages 664–670, 1987.

[4] S. Leung and M. Shanblatt. Computer architecture design for robotics. *IEEE Int. Conf. Robotics Automation*, pages 453–456, 1988.

[5] P.K. Khosta and S. Ramos. A comparative analysis of the hardware requirements for the Lagrange-Euler and Newton-Euler dynamics formulations. *IEEE Int. Conf. Robotics Automation*, pages 291–296, 1988.

[6] O. Lubeck and J. Moore. A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S-810/20, and Cray X-MP/2. *IEEE Computer*, 18(12):10–23, Dec 1985.

[7] Daniel A. Reed and Richard M. Fujimoto. *Multicomputer Networks Message-Based Parallel Processing*. The MIT Press, London, England, 1987.

[8] D. Hillis. The connection machine. *Scientific American, Trends in Computing*, 1988.

[9] K. Hwang. New supercomputing of architectures and technology. Jun 1988.

[10] B.K. Helminen and D.Poplawski. A performance characterization of the FPS T-series hypercube. *Proceedings of the 1987 Array Conf., Mtl ; 26-29 April*, pages 71–83, 1987.

[11] R. Arlauskas. iPSC/2 system: A second generation hypercube. *Conf. on Hypercube Concurrent Comp. and Appl.*, 1988.

[12] S. Narasimhan and al. Condor: A revised architecture for controlling the Utah-MIT hand. *IEEE Int. Conf. Robotics Automation*, pages 446–449, 1988.

[13] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proc. AFIPS Thompson Washington D.C*, 30:483–485, 1967.

[14] R. Nigam and C. Lee. A multiprocessor-based controller for the control of mechanical manipulators. *IEEE Int. Conf. Robotics Automation*, 1:173–182, 1985.

[15] J. Barhen and E.C. Halbert. Advances in concurrent computation for autonomous robots. *Proc. Conference on Robotics Research, Scottsdale*, Aug 1986.

[16] B.Furht S.Greffin and A.Katbab. A massively parallel architecture for robot arm control. *Miami Technicon 87 (IEEE) Miami, FL; 28-30 October 1987*, pages 417–421, 1987.

[17] P.J. Denning. Working sets past and present. *IEEE Trans. on Software engineering*, 6(1):64–84, January 1980.

[18] G. Lee and R. Chang. Efficient parallel algorithm for robot inverse dynamics computation. *IEEE Int. Conf. Robotics Automation*, pages 851–857, 1986.

[19] G. Lee and R. Chang. Efficient parallel algorithms for robot forward dynamics computation. *IEEE Int. Conf. Robotics Automation*, pages 664–659, 1987.

[20] H.T. Kung. Why systolic architectures ? *IEEE computer*, pages 37–46, January 1982.

[21] J.P. Jones. A concurrent on-board vision system for a mobile robot. *IEEE Int. Conf. Robotics Automation*, pages 1022–1032, 1988.

[22] Y. Eva Ma and al. High preformance special-purpose computer architectures for robotics and sensing applications. *Computer Architecture for Robotics and Automation*, pages 151–171, 1987.

[23] J. Poon. *Multiprocessor-compatible inverse kinematics and path planning for robots*. PhD. Thesis University of British Columbia, 1988.

[24] J.A. Stankavic. A serious problem for next-generation systems. *IEEE Computer*, pages 10–19, October 1988.

[25] S. Narasimhan and al. Implementation of control methodologies on the computational architecture for the Utah-MIT hand. *IEEE Int. Conf. Robotics Automation*, pages 1884–1889, 1986.

[26] T. Feng. A survey of interconnection networks. *IEEE Computer*, pages 12–27, Dec 1981.

[27] J. Tuazon. Mark IIIfp hypercube concurrent processor architecture. *Conf. on Hypercube Concurrent Comp. and Appl.*, 1988.

[28] Inmos. Ims t800 transputer : Preliminary data. 1987.

[29] J. Harp. Esprit project P1085 - reconfigurable transputer project. *Conf. on Hypercube Concurrent Comp. and Appl.*, pages 122–127, 1988.

[30] A.J. West. *Monitoring Distributed Occam Programs*. Master's thesis University of Manchester, 1987.

[31] S. Yalamanchili and J. Aggarwal. Reconfiguration startegies for parallel architecture. *IEEE Computer*, 18:43–61, Dec 1985.

[32] P. Milligan. Network topology a critical factor in the implementation of algorithms intended for efficient execution on a transputer network. *Microprocessing and Microprogramming*, 23:253–257, Mar 1988.

[33] J.G. Kuhl and S. Reddy. Fault-tolerance considerations in large, multiple-processor systems. *IEEE Computer*, pages 56–67, Mar 1986.

[34] D. Wong. *Reliable Multiprocessing Systems for Real-Time Manipulator Control and Simulation.* PhD. Thesis Proposal University of British Columbia, Nov 1987.

[35] B. Harry. A fault-tolerant communication system for the B-Hive generalized hypercube multiprocessor. *Conf. on Hypercube Concurrent Comp. and Appl.*, 1988.

[36] R.M. Stein. T800 and counting. *BYTE*, pages 287–296, Nov 1988.

[37] A. Burn. *Programming in OCCAM 2*. Addison-Wesley Publishing Company, Workingham, England, 1988.

[38] C.A.J. Hoare. Communicating sequential processes. *Comm. of ACM*, 21(8):666–677, Aug 1978.

[39] J. Mock. Processes, channels and semaphores (version 2). *Parallel C, Logical System documentation*, 1988.

[40] W.J. Dally. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Computer*, 36(5):547–553, May 1987.

[41] A.W. Roscoe. Routing messages through networks: An exercise in deadlock avoidance. *Proceedings of the 7th occam User Group Technical Meeting*, pages 55–80, Sep 1988.

[42] J.R. Einstein and J.Barhen. Virtual-time operating-system functions for robotics applications on a hypercube. *Conf. on Hypercube Concurrent Comp. and Appl.*, pages 100–107, 1988.

[43] L. Snyder. Introduction to the configurable highly parallel computer. *IEEE Computer*, pages 49–55, Jan 1982.

[44] S. Nugent. The iPSC/2 direct-connect communication technology. *Conf. on Hypercube Concurrent Comp. and Appl.*, 1988.

[45] D.C. Grunwald and D.A. Reed. Networks for parallel processors: Measurements and prognostications. *Conf. on Hypercube Concurrent Comp. and Appl.*, pages 610–619, 1988.

[46] P. Kermani and L. Klenrock. Virtual cutthrough: a new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[47] H. J. Denuell. BBK-V2 hardware documentation, version 2.3. *Megaframe Series*, Apr 1988.

[48] Micrology pbt. The VMEbus specification rev c.1. 1985.

[49] J. Lowenhag. VMTM hardware documentation, version 1.1. *Megaframe Series*, Oct 1987.

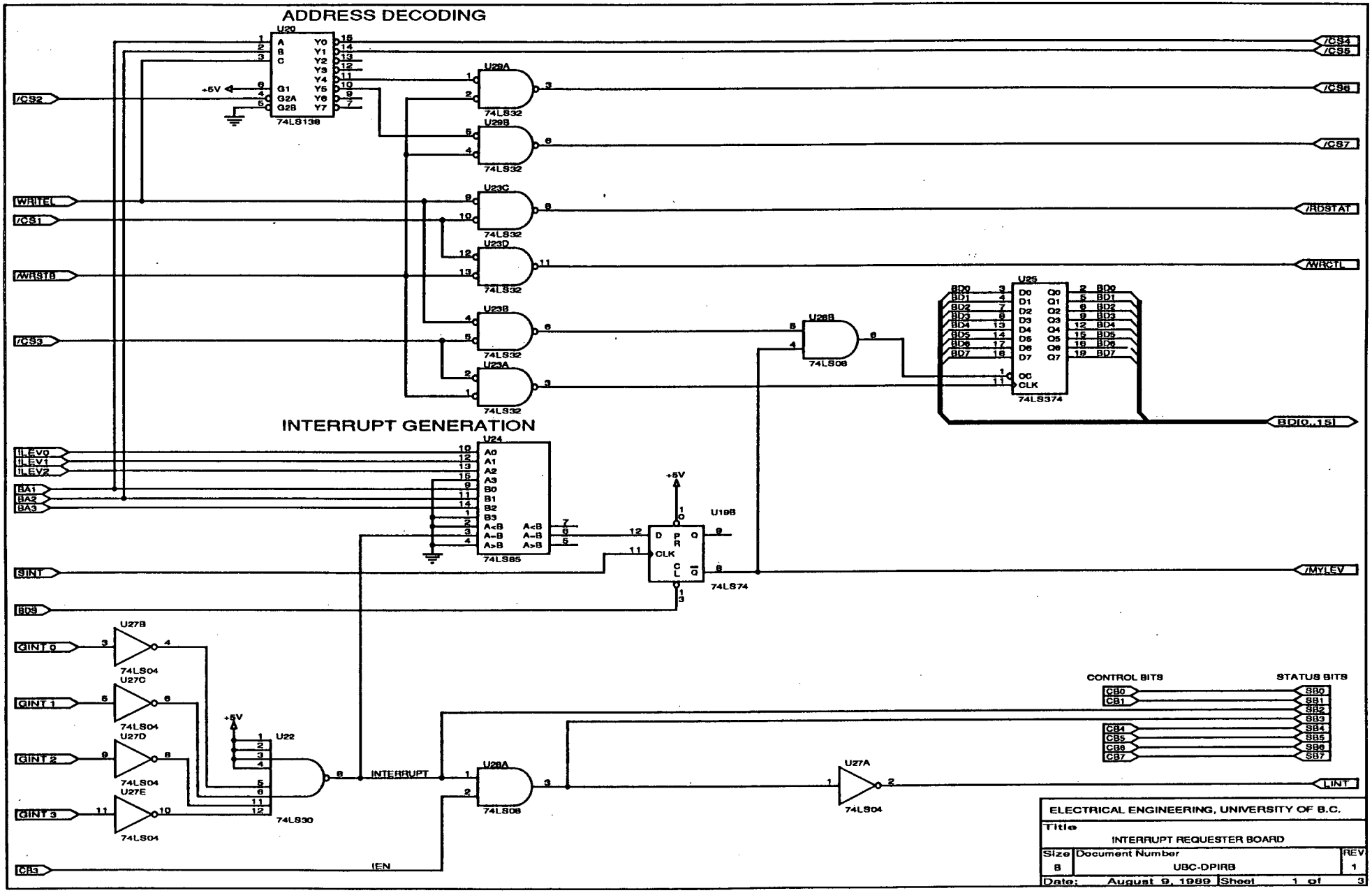
[50] Incorporated Performance Technology. 32-bit VMEbus repeater, model PT-VME902A. *User's Manual*, Jul 1987.

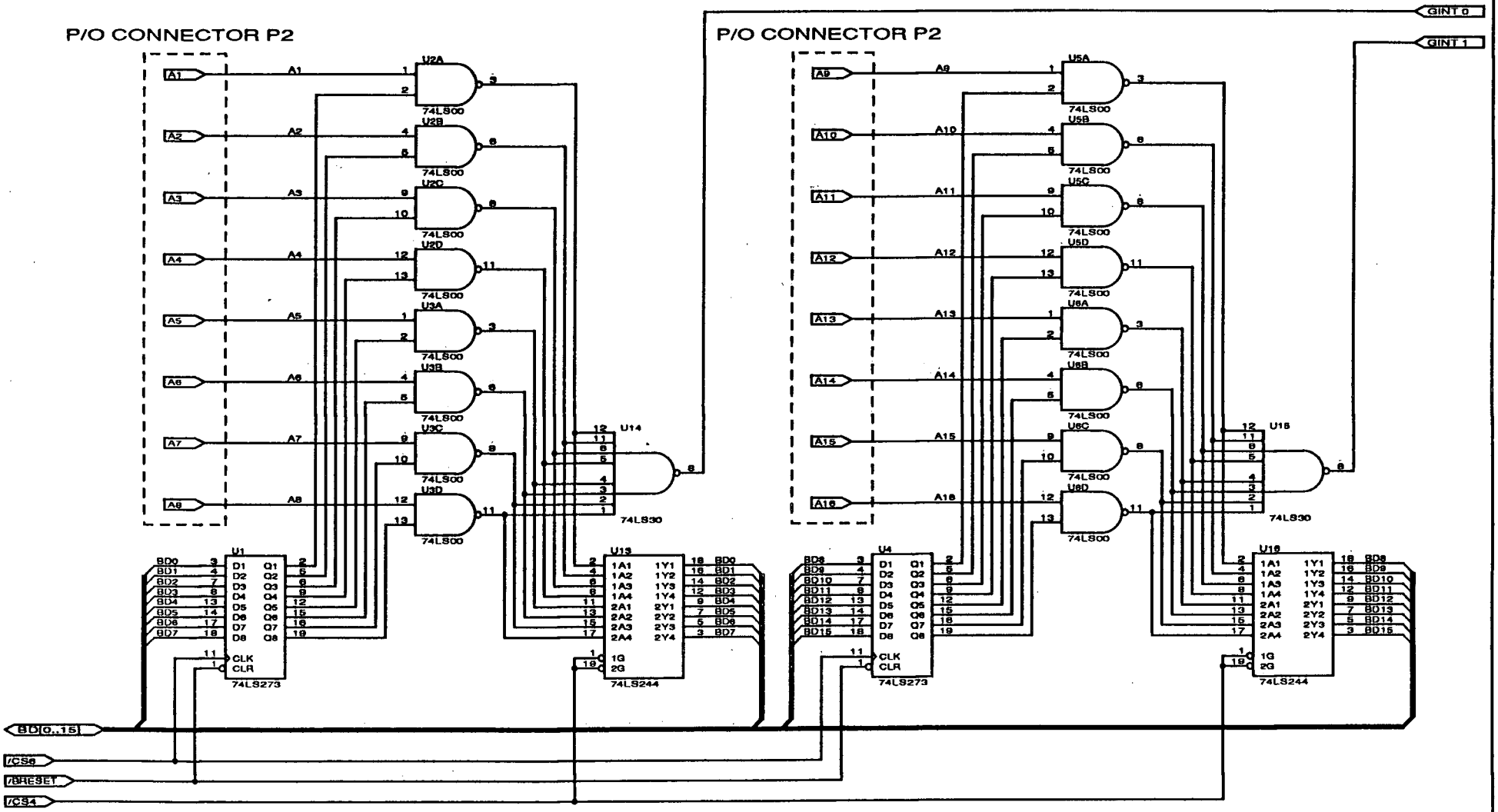
[51] P.H. Welch. Managing hard real-time demands on transputers. *Proceedings of the 7th occam User Group Technical Meeting*, pages 135–146, September 1988.

[52] D.C. Grunwald and D.A. Reed. The performance of multicomputer interconnection networks. *IEEE Computer*, pages 63–73, June 1987.

Appendix A

Interrupt Board Schematics


ADDRESS DECODING
U20
74LS138
+5V
/CS2
/CS4
/CS5
/CS6
/CS7
U29A
U29B
74LS32
WRITEL
/CS1
/WRSTB
/CS3
U23C
U23D
U23B
U23A
/RDSTAT
/WRCTL
U28B
74LS08
U25
74LS374
BD0
BD1
BD2
BD3
BD4
BD5
BD6
BD7
BD[0..15]
INTERRUPT GENERATION
U24
74LS85
ILEV0
ILEV1
ILEV2
BA1
BA2
BA3
SINT
U19B
74LS74
/MYLEV
BD9
U27B
U27C
U27D
U27E
74LS04
GINT 0
GINT 1
GINT 2
GINT 3
U22
74LS30
INTERRUPT
U28A
U27A
CONTROL BITS
STATUS BITS
CB0
CB1
CB4
CB5
CB6
CB7
SB0
SB1
SB2
SB3
SB4
SB5
SB6
SB7
LINT
CB3
IEN
ELECTRICAL ENGINEERING, UNIVERSITY OF B.C.
Title
INTERRUPT REQUESTER BOARD
Size
B
Document Number
UBC-DPIRB
REV
1
Date: August 9, 1989
Sheet 1 of 3

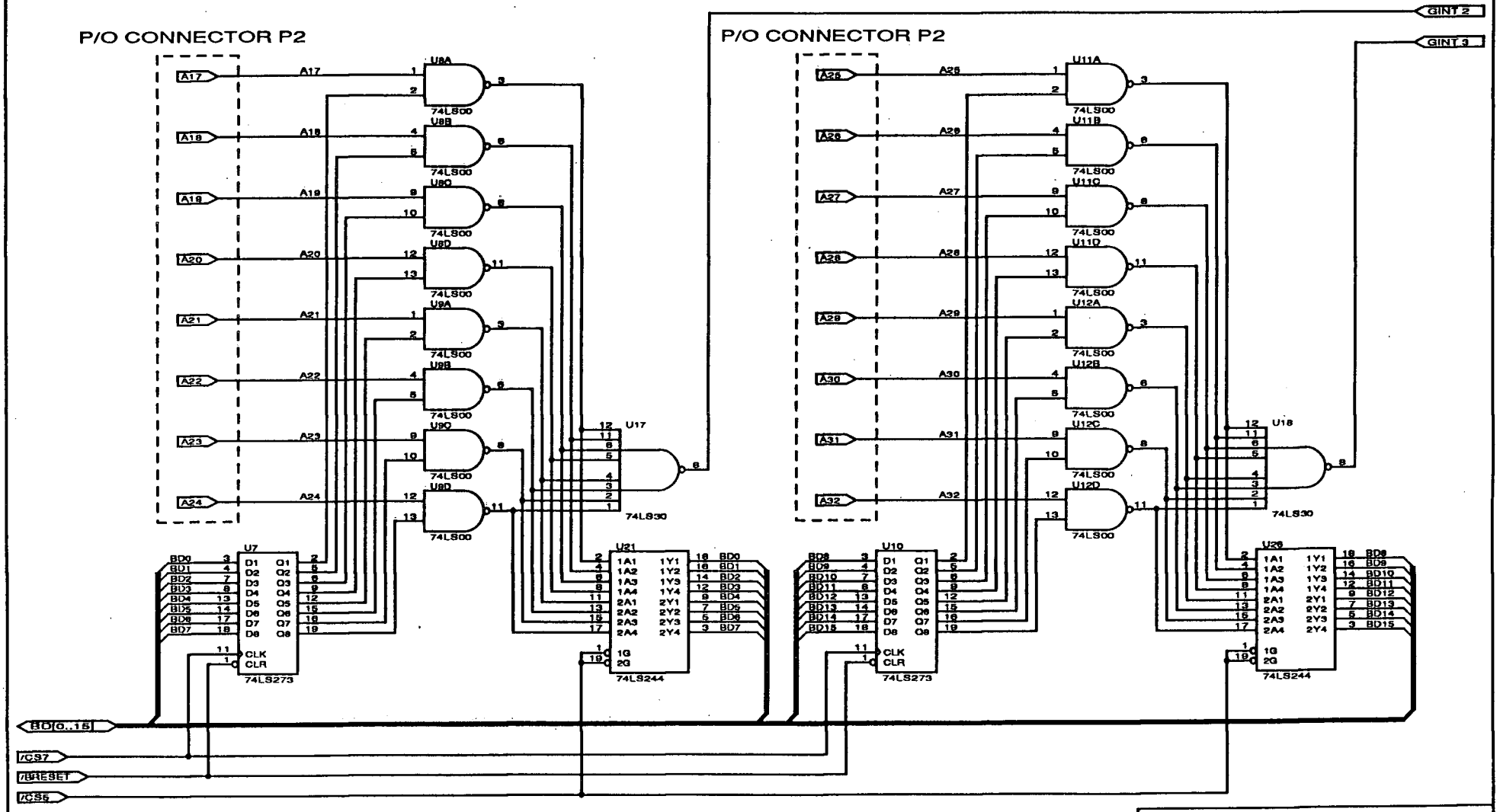


ELECTRICAL ENGINEERING, UNIVERSITY OF B.C.

Title: INTERRUPT REQUESTER BOARD

Size	Document Number	REV
B	UBC-DPIRB	1

Date: August 9, 1989 Sheet 2 of 3


P/O CONNECTOR P2
P/O CONNECTOR P2
GINT 2
GINT 3
A17
A18
A19
A20
A21
A22
A23
A24
A25
A26
A27
A28
A29
A30
A31
A32
U8A
U8B
U8C
U8D
U9A
U9B
U9C
U9D
U11A
U11B
U11C
U11D
U12A
U12B
U12C
U12D
74LS00
U17
U18
74LS30
U7
U10
74LS273
U21
U20
74LS244
BD0
BD1
BD2
BD3
BD4
BD5
BD6
BD7
BD8
BD9
BD10
BD11
BD12
BD13
BD14
BD15
BD[0..15]
/CS7
/BRESET
/CS5
ELECTRICAL ENGINEERING, UNIVERSITY OF B.C.
Title
INTERRUPT REQUESTER BOARD
Size
B
Document Number
UBC-DPIRB
REV
1
Date: August 9, 1989
Sheet 3 of 3

Appendix B

Fair Alternative Library

Fair Alternative Libraries

```
/* Author: B. Vachon
 *   Date: July 14th, 1989
 */

#define false 0
#define true  1

#include <stdio.h>
#include "conc.h"

#define FAIRALTSLOT             10
#define MAXCHAN                 100
#define SLOTUNUSED              -1
#define ALT_NOT_DYNAMIC         -2
#define ALT_NO_MORE_ROOM        -3
#define ALT_MEMORY_ALLOC_ERROR  -4
#define ALT_UNKNOWN_ERROR       -5

typedef int AltHandle;

AltHandle       AllocFairAlt(Channel **clist);
int             DeallocFairAlt(AltHandle Alt);
int             FairProcAltList(AltHandle Alt);
int             FairProcSkipList(AltHandle Alt);
int             FairProcTimerAltList(int time, AltHandle Alt);

typedef Channel *ChanP;

/*  global variables */
static ChanP   AltSlot[FAIRALTSLOT][MAXCHAN];
static int     SlotStatus[FAIRALTSLOT];
static int     SlotNumChan[FAIRALTSLOT];
static int     InitFairAlt = false;

/* local functions */
static int InitFair(void);
static int FindSlot(void);
static int CountChannel(Channel **clist);
static int NumChan;
```

AllocFairAlt

```
int  AllocFairAlt(Channel **clist)
{
  int i, slot;

  if (!InitFairAlt)
  {
    if (InitFair() == -1)        /* can't initialize Fair Alt */
      return(-1);
  }

  if ( (slot=FindSlot()) == -1)  /* no more slot left */
    return(-1);

  NumChan = CountChannel( clist );
  SlotNumChan[slot] = NumChan;

  /* Copy the Channel Over */
  for(i=0; i<NumChan; i++)
    AltSlot[slot][i] = AltSlot[slot][i+NumChan] = clist[i];
  AltSlot[slot][2*NumChan] = NULL;

  /* successful allocation */
  return(slot);
}


static int InitFair(void)
{
  register int i;

  /* double initialize */
  if (InitFairAlt == true) return(-1);

  for (i=0; i<FAIRALTSLOT; i++)
  {
    SlotStatus[i] = SLOTUNUSED;
    SlotNumChan[i] = 0;
  }
  InitFairAlt = true;
  return(1);
}


static int FindSlot(void)
{
  register int i = 0;

  /* look for endsignal */
  for(i=0; (i<FAIRALTSLOT) && (SlotStatus[i] != SLOTUNUSED); i++);

  if ( i != FAIRALTSLOT)
  {
    SlotStatus[i] = 0;
    return(i);
  }

  return(-1);
}
```

DeallocFairAlt

```
int DeallocFairAlt(int slot)
{
  if (SlotStatus[slot] == SLOTUNUSED)
    return(-1);

  /* mark slot as unused and reset slot status */
  SlotStatus[slot] = SLOTUNUSED;
  SlotNumChan[slot] = 0;
```

```
  return(slot);
}

static int ChangeList(int slot)
{
  if (SlotStatus[slot] == SLOTUNUSED)
  AltSlot[slot][SlotStatus[slot]+SlotNumChan[slot]] = NULL;
  return(0);
}

static void FixList(int slot)
{
  AltSlot[slot][SlotStatus[slot]+SlotNumChan[slot]] =
    AltSlot[slot][SlotStatus[slot]];

  /* readjust pointer */
  if (++SlotStatus[slot] == SlotNumChan[slot])
     SlotStatus[slot] = 0;
}

static int CountChannel(Channel **clist)
{
  register int i=0;

  for(i=0; (clist[i] != (Channel *)0) && (i<MAXCHAN); i++)
  if (i>=MAXCHAN/2)
    return(-1);  /* too many channels */

  return(i);
}
```

FairProcAltList

```
int FairProcAltList(int AltHandle)
{
  int result;

  if (ChangeList(AltHandle) == -1)
    return(-1);

  result = ProcAltList(&AltSlot[AltHandle][SlotStatus[AltHandle]]) +
           SlotStatus[AltHandle];
  if (result >= NumChan) result -=  NumChan;
  FixList(AltHandle);
  return(result);
}
```

FairProcSkipList

```
int  FairProcSkipList(int AltHandle)
{
  int result;

  if (ChangeList(AltHandle) == -1)
    return(-1);
  if (result >= NumChan) result -=  NumChan;
  FixList(AltHandle);
  return(result);
}
```

FairProcTimerAltList

```
int  FairProcTimerAltList(int time, int AltHandle)
{
  int result;

  if (ChangeList(AltHandle) == -1)
    return(-1);
  result = ProcTimerAltList(time,&AltSlot[AltHandle]
            [SlotStatus[AltHandle]]+ SlotStatus[AltHandle] );
  if (result >= NumChan) result -=  NumChan;
  FixList(AltHandle);
  return(result);
}
```