SCALABLE SHARED MEMORY INTERCONNECTIONS

Dimitrios Nikolaou Serpanos

(Thesis)

CS-TR-277-90

October 1990

Scalable Shared Memory Interconnections

Dimitrios Nikolaou Serpanos

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

— OCTOBER 1990 —

Abstract

This dissertation presents an architecture and describes an implementation for a high-performance, scalable shared memory interconnection. The architecture is based on a scalable shared memory model called PRAM.

Conventional shared memory multiprocessors provide high performance but they do not scale well to either a large number of processors or over long distances. The PRAM network is scalable and allows heterogeneous processors to be interconnected achieving high effective data transfer rates and low latencies.

An implemented prototype interconnects IBM AT, SUN-3 and MAC-II machines demonstrating performance improvements over conventional high-performance scalable multiprocessors.

The successful prototype implementation proves that high-performance, low-cost, scalable shared memory interconnections can be built and combine high performance with scalability.

‘Έν οἶδα ὅτι οὐδέν οἶδα.

Σωκράτης

The one I know is that I know nothing.

Socrates

Τό μυρμήγκι ἀνακάλυψε ἕνα καινούργιο

κόκκο στόν ἄμμο τῆς Σαχάρας·

καί χαιρόταν ...

Εἶχε ἀνακαλύψει τήν Σελήνη του.

Ἀνώνυμος

The ant discovered a new

grain of sand in the Sahara desert;

and it was very happy ...

It had just conquered its Moon.

Anonymous

to my parents Nikolaos and Georgia
and my sister Fotini

Acknowledgements

Many people affected my life in Princeton in one way or another.

My advisor, Richard J. Lipton, was definitely the one who inpired me the most. His support and ideas made the work I present in this dissertation possible, while his criticism shaped it. I feel honored that he supervised my work.

I was very fortunate to be working with Jonathan S. Sandberg. Jon as a colleague and a friend helped me overcome many problems with his advice and support. His contributions to the project have been substantial.

Andrea LaPaugh's help and support have been invaluable to me. Not only did she give me enough background for my work, but she was also a good critic of my research. Her comments definitely improved this dissertation.

I am happy that Wayne Wolf has been at Princeton University for the last two years. Working with him was a great experience. Wayne's comments have clearly improved the dissertation.

I am indebted to Hector Garcia-Molina for his support and constructive criticism of my work and to Kai Li for numerous enlightening discussions.

I wish to thank David Dobkin for being so friendly and supportive. I will definitely miss his great sense of humour.

Thanks to Rafael Alonso for his support and especially for his guidance during my first year here.

It has been a privilege to work with Ted Altman, Tom Meyer and Lou Pokrocos. Their contributions to the project are invaluable. I am very happy to acknowledge R. Altman, T. Kyi and C. Zimmerman for their contributions

too. Special thanks to my friend Daniel Barbara for his support and his contribution to the verification of the network protocols.

I will always remember the Computer Science graduate students of my year who were very friendly and in many ways helped me through my years here. Especially, I wish to mention Jeffery Westbrook who has been a good, supportive friend of mine.

I wish to thank the Greek students at Princeton University who made life in Princeton more tolerable. Especially, I should mention my friend Linos Frantzeskakis for his support through all these years.

Special thanks to Cindy Lipton and Nancy Porter who always made me feel welcome at their home.

Many, many thanks to Sharon Rodgers who really made my life easy at the department.

Finally, I wish to express my gratitude to my parents, Nikolaos and Georgia, and my sister Fotini for their guidance and support during all these years. I am proud to be a member of the family. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Scalable Shared Memory Interconnections

Dimitrios Nikolaou Serpanos
Computer Science Department
Princeton University

Chapter 1

Introduction

This dissertation presents an architecture for scalable multiprocessors that allows high-bandwidth and low-latency communication between processors.

Multiprocessors are systems with multiple interconnected processors. The interconnection of many processors raises an important problem: *efficient interprocessor communication.* Communication among processors is necessary not only to transfer computational data but to synchronize cooperating processors, too. Although computational data transfers many times involve large amounts of data, process synchronization (and control) typically requires short messages. Efficient interprocessor communication requires high performance not only for transfers of large amounts of data but for short messages as well.

The cost of interprocessor communication is the major bottleneck in conventional multiprocessors and affects their most important parameter: *scalability*, i.e. their ability to scale to more processors and over long distances

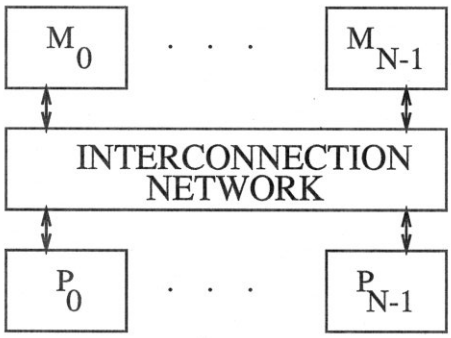

Figure 1.1: A Typical Shared Memory Architecture

with low performance degradation [LS88] [AI83].

In the following sections I present the main characteristics of the major conventional multiprocessor architectures:

- shared memory machines;
- message-passing systems;
- local, wide area and long-haul networks.

1.1 Shared Memory Machines

In most conventional shared memory machines (e.g. [CGBG88], [Got83], [Pfi85]) all processors share a common physical memory. A typical shared memory organization is shown in Figure 1.1, where the interconnection network is typically a bus, a multistage network or a crossbar switch. In some systems the memory modules may be local to processors (e.g., memory mod-

ule M_0 local to processor P_0, M_1 local to P_1, etc.). All interconnected processors P_i, $0 \leq i \leq (N-1)$, access the shared memory (M_0, M_1, ..., M_{N-1}) with regular read/write memory operations and communicate through the shared memory.

One of the main factors that affects the performance of shared memory machines is the access delay of the shared memory [LS88] [AI83]. Busses allow a low delay in memory access but they are not scalable: bus-based systems can accomodate an order of 20 processors. More processors need a different interconnection. A crossbar switch is the desirable solution but its cost and complexity is too high for interconnecting thousands of processors. Interconnection networks provide a compromise between crossbar switches and busses. Their cost and complexity is lower than crossbars but they introduce memory access delays that typically grow logarithmically with the number of interconnected processors. Physical limitations may cause the memory access delays of machines with large multistage networks to be much slower than local memory access delays of uniprocessor systems even when state-of-the-art technology is used [FD86].

So, conventional shared memory systems do not scale well to arbitrarily large numbers of processors and over long distances without serious performance degradation.

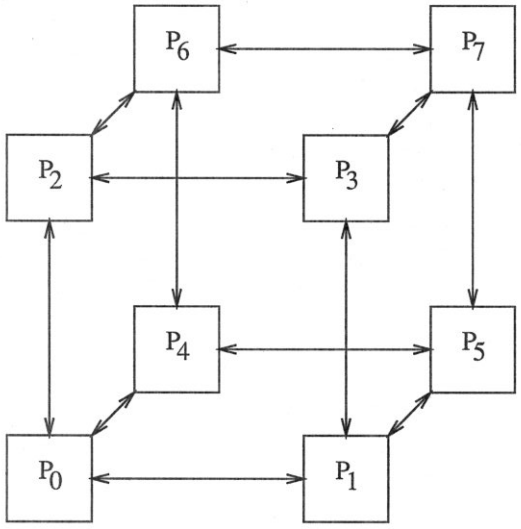

Figure 1.2: A Hypercube Message-Passing System

1.2 Message-Passing Systems

Message-passing machines [AS88] [Sei85] are composed of nodes which are processors equipped with local memory. The nodes are interconnected to form a network such as a hypercube (e.g., [Sei85]) (see Figure 1.2), cube-connected cycles [PV81], etc. The processors can access directly only their local memory and communicate by explicit exchange of messages. The architecture of these systems supports scalability to a higher degree than conventional shared memory systems and allows interconnection of heterogeneous processors [AS88].

The limitations of message-passing machines are their programming paradigm [LS88], and the cost of interprocessor communication which is

higher than the cost in conventional shared memory systems. The programming difficulty of message-passing machines rises from the fact that programmers have to explicitly code the message exchange in their programs. This requires a different, harder programming model than the familiar, easy shared memory paradigm. The performance of interprocessor communication is lower than that of shared memory systems because of software overhead and such delays as DMA setup and path setup.

1.3 Networks

Among conventional networks (local area, wide area and long-haul), the local area networks (LANs) provide the highest performance. This performance is lower than the one provided by message-passing multicomputers and is also unsuitable for many demanding parallel and distributed applications (e.g., real-time audio/video applications, etc.). As Arnould et al. mention [Arn89], the performance of conventional networks is limited by such costs as context switching, data copying, and protocol and interrupt processing.

So, although conventional networks support scalability and heterogeneity, they provide performance unsatisfactory for high-performance parallel and distributed computing.

1.4 The Princeton PRAM Project

The need for high-performance communication as well as the current advances and decreasing cost of fiber optic technology have boosted research

in the area of communication. The progress in optical interfaces allows the development of high-speed interconnections unavailable up to date providing inexpensive, high data transfer rates.

Direct application of optical technology in existing communication models is not always effective, because of architectural limitations of the models; for example, the Ethernet [MB76] requires a minimum latency for all transmitted packets for collision detection [Sch90] ([Sch90] cites [TBF83]). Many projects have started lately focusing on the effective use of the high bandwidth/low latency optical interconnections provide. Many of these projects investigate new processor interconnection architectures in an effort to achieve high process-to-process data transfer rates. These efforts include: the Nectar project at CMU [Arn89], the MERLIN project at SUNY Stony Brook and the Sandia National Labs [WM89], DEC's Autonet [Sch90], FDDI [Ros86] and Princeton's PRAM Project.

The Princeton PRAM Project focuses on the development of memory level multiprocessor interconnections that provide:

- high bandwidth/low latency process-to-process communication;
- scalability to large number of processors;
- geographic separation;
- heterogeneity.

A result of the project's efforts is the PRAM memory model [LS88] (developed by R. J. Lipton and J. S. Sandberg) which supports scalability and geographic separation, while it allows systems to achieve high process-to-process

data transfer rates, equivalent to main memory access rates for many applications. An important property of the model is that it allows *uniform-cost communication*, i.e. messages are inserted in the network at rates (in bytes/sec) independent of the messages' size. Because of this property, PRAM-based systems provide a different model for parallel and distributed computing than conventional systems affecting fundamental problems in the area, e.g. reliable broadcasting, clock synchronization, etc.

The PRAM model was used to build a prototype that connects heterogeneous autonomous systems (IBM ATs, SUN-3s and MAC-IIs) and achieves data transfer rates up to 24 MBits/sec. The network is composed of switches, each with 4 full-duplex ports. The 4 ports of each switch communicate through a high-speed bus. Independent switches are connected with fiber links that are terminated by the switch ports. The prototype is currently used for evaluation and software development.

The development of the prototype led to research in testing, too. The major difficulty identified was the absence of methodologies in testing parallel systems which resulted in great difficulties in testing the prototype. This research led to the development of a simple language, called PLADO, which allows designers to write legible, portable parallel testing programs. Part of the research was devoted to a verifier which identifies deadlocks, timing dependencies and starvation in these programs.

1.5 Thesis Outline

This dissertation presents my contribution to the architecture, implementation and testing of the prototype built for the Princeton PRAM Project. It also describes work of the other members of the PRAM team to provide the necessary context. R. J. Lipton and J. S. Sandberg contributed to the architecture, while T. Altman, T. Meyer and L. Pokrocos worked on the implementation and testing of the prototype. J. S. Sandberg, R. Altman, C. Zimmerman and Ted Kyi contributed to the applications. Daniel Barbara worked on the verification of the hardware protocols of the network.

Chapter 2 introduces the notion of *uniformity* in communication cost and shows how the existence of uniform-cost networks affects the solution of some fundamental problems in parallel and distributed computing.

Chapter 3 describes the PRAM (Parallel Random Access Memory) memory model and identifies its advantages and weaknesses, while Chapter 4 presents an architecture for a PRAM system, the prototype built, the design decisions made, its properties and extensions.

Chapter 5 describes PLADO and the results of the research on the verification of PLADO programs. Chapter 6 presents the performance characteristics of the prototype and discusses some applications.

Finally, Chapter 7 concludes the dissertation with a presentation of open problems related to the PRAM Project.

A short version of material in Chapters 2, 3, 4 and 6 appears in [LS90].

Chapter 2

Uniform-Cost Communication

Uniform-cost communication networks allow processes to effectively insert messages in the interconnection at rates independent of the size of the transmitted message and provide better performance than conventional networks for applications that involve exchange of many short messages. Memory-level interconnections support uniform-cost communication, because processes can effectively transmit messages at rates equivalent to memory access rates independent of the size of the messages.

2.1 Uniform-cost Networks

Conventional scalable multiprocessors use various types of interconnections ranging from local area networks (Ethernet [MB76], Token Ring [IEE85], etc.) to specialized interconnections (hypercubes [AS88], cube-connected cycles [PV81], etc.). The end-to-end delay of a message transmission from a

processor P_i to a processor P_j over one of these networks typically consists of the following delays:

- the *packetization overhead*;
- the *path-setup delay*;
- the *network latency*;
- the *reception delay.*

The *packetization overhead* of a message is due to such delays as dividing the message into the appropriate number of frames (if necessary) and calculating the CRC bytes and headers (in most systems, this delay includes the overhead of a system call). The *path-setup delay* is the delay to setup a path between the communicating processors P_i and P_j: on the Ethernet one has to account for collisions; on Intel's iPSC/2 with the Direct Connect Modules (implementing a variation of wormhole routing) there is a delay to establish a path between P_i and P_j before data starts flowing between the processors [Nug88]; on a token ring a system has to wait until it can use the physical media for transmission. The *network latency* is the delay for a message to propagate through the physical interconnection between P_i and P_j, while the *reception delay* is the delay on the receiving processor to process the incoming message, i.e. perform error checking, remove headers, etc., and extract the useful data from the received bit string. The terms *end-to-end delay* and *transmission delay* used in this chapter are defined as follows:

Definition 2.1.1 *The* **end-to-end delay** *of a message transmission through a network is the delay from the beginning of the message's transmission until*

the last byte of the transmission is received by the receiver.

Definition 2.1.2 *The* **transmission delay** *of a message is the time interval between the initiation of a transmission and the time when the last byte of the transmission is successfully inserted into the network's physical media.*

It is quite important to emphasize here that in most networks the packetization overhead and the path-setup delay are paid every time a processor transmits a message. One can approximate the transmission delay of a message with D data bytes in a conventional network with the formula: $T_D = k_1 + k_2 D$, where k_1 and k_2 are positive real numbers. k_1 accounts for delays independent of the message's size (e.g., the path-setup delay, context switching delays), while $k_2 D$ captures the size-dependent delays (e.g., data copying, CRC calculations). In conventional networks k_1 is much larger than k_2. For example, on our local Ethernet under regular load we measured k_1 in the order of 2 milliseconds, while k_2 is in the order of 2 microseconds per byte. This difference of 3 orders of magnitude between k_1 and k_2 makes k_1 the dominating factor in T_D, when D is relatively small.

The nature of communication in these networks leads to their classification as *non-uniform-cost* communication networks:

Definition 2.1.3 *A* **non-uniform-cost** *network is an interconnection, where the mean transmission delay of D data bytes is: $T_D = k_1 + k_2 D$ with k_1 and k_2 positive real numbers and with $k_1 \gg k_2$.*

The term *non-uniform-cost communication* comes from the observation that the delay of transmitting D data bytes is clearly not equal to D transmissions each of 1 data byte.

Non-uniformity has its merits and has been proven suitable for communicating large amounts of data among systems. There are many problems though which would benefit from a *uniform-cost* communication network.

Definition 2.1.4 *A* **uniform-cost** *communication network is an interconnection, where the mean transmission delay of D data bytes is: $T_D = kD$ with k a positive, real number.*

We study only *uniform-cost* networks that have k the same order of magnitude as k_2; if k is the order of k_1, then one can use a conventional *non-uniform-cost* network to simulate the *uniform-cost* one, thus experiencing poor performance.

Refering to the definition of non-uniform-cost networks, we see that k_2 is quite small. It is expected that k in uniform-cost networks cannot become as small as k_2 although it can be the same order of magnitude. As will become clear with the description of the implementation of the uniform-cost network prototype, if one uses the same technology and design methodologies for building a uniform-cost and a non-uniform-cost network, then one can make the constant k_2 smaller than k. This implies that non-uniform-cost networks provide better performance whenever the quantity of transmitted data is larger than a critical value D_C, while uniform-cost networks are more efficient when the quantity of transmitted data is smaller than D_C. This critical amount of data D_C is: $D_C = \frac{k_1}{k-k2}$. Figure 2.1 shows the mean transmission delay in both types of networks (uniform-cost and non-uniform-cost) as a function of the amount of transmitted data.

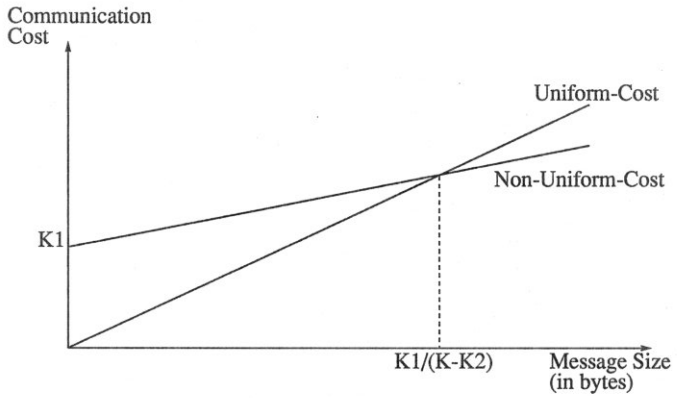

Figure 2.1: Uniform-Cost vs. Non-Uniform-Cost Networks

2.2 Uniform-Cost Networks in Parallel Computing

Uniformity in communication cost of scalable systems provides new "ground rules" and thus affects the solution of some problems in parallel and distributed computing. Two problems affected are drawn from the area of distributed computing: reliable broadcasting and clock synchronization.

2.2.1 Reliable Broadcasting

Reliable broadcasting is a known problem in distributed computing that appears in the studies of various problems in distributed systems, such as reaching asynchronous distributed agreement [MSM89]. The problem can be stated as follows (similarly to the problem descriptions in [CM84], [MSM89]): assume that there is a parallel system connecting N processors through an interconnection. Processors broadcast messages to all other connected sys-

tems. Messages can be lost due to network failures, buffer overflows, etc. Design an efficient protocol which makes certain that all operational processors receive the broadcast messages.

Since in many cases (e.g. [CM84] [WG83]) the problem has been studied for systems interconnected with a broadcast network, such as the Ethernet [MB76], we will use the common Ethernet in our analysis. Although often the problem is studied together with the problem of ordering the broadcast messages in exactly the same order at the receivers (e.g., [CM84] [MSM89]), we do not get into the ordering problem here.

A straightforward solution to the reliable broadcasting problem requires every system which received a broadcast message to acknowledge it to the broadcasting site [Moc83]. This solution requires $(N-1)$ acknowledgments per broadcast message (no faults assumed).

In the following we use the *acknowledgement time* as a metric to compare the networks we consider. Assuming that processor P_B broadcasts a message and then receives the acknowledgements, we have the following definition for the metric (the definition is analogous to the one Tokoro and Tamaru give for the response time [TT77]):

Definition 2.2.1 *The* **acknowledgement time** *is the delay between the beginning of the acknowledgements' transmission and the time when P_B's interface receives the last acknowledgement.*

We also assume that the network latency of the considered networks is negligible.

For the Ethernet, the shortest *acknowledgement time* can be approxi-

mated by: $T_{NU}(D) = k_1 + (N-1)k_2D$, where D is the number of bytes in an acknowledgment (quite small); the delay is such because most of the operations whose costs are described by k_1 can occur in parallel.

The inefficiency of this straightforward solution led to the development of alternative solutions which reduce the number of acknowledgements (e.g., the protocol by Chang and Maxemchuck [CM84] and the Trans protocol [MSM89]). Chang and Maxemchuck [CM84], for example, present a method where all messages pass through a token site, which timestamps each message and acknowledges it with only one acknowledgment per message; the token is rotated among all functioning processors. In case of a token site failure, the token is passed to another processor.

Such complicated solutions have been developed, because of the inefficiency of the simple, straightforward solution when the network is a conventional non-uniform-cost interconnection of the Ethernet type. That solution is quite efficient with a uniform-cost network: the longest possible *acknowledgement time* is $T_U(D) = (N-1)kD$. From the above formulas we deduce that $T_{NU}(D) > T_U(D)$ for a network with $N < D^{-1}k_1/(k-k_2)+1$; using $k_1 = 10^3k_2$, $k = 2k_2$ and $D = 4$ bytes, we see that the uniform-cost network is more efficient than the Ethernet for network sizes up to 250 processors. This solution is not only efficient but also simpler than the protocols described above.

2.2.2 Clock Synchronization

Clock synchronization is another important problem in distributed computing. The *clock synchronization* problem in a system with M processors is to find a fault-tolerant algorithm which resynchronizes the processor clocks periodically in an effort to keep them synchronized (the source of the problem is clock drift) [LMS85].

The problem is quite important and thus has been extensively studied (e.g., see [LMS85], [CAS86], [Cri89]).

A basic step in many clock synchronization methods is the reading of a remote processor's clock. The following discussion is based on a paper by F. Cristian [Cri89] and borrows its notation and method description. When a process P on a processor P_0 wants to read the clock on another processor P_1, then it sends a message ("$time$ =?") to P_1. A process Q on P_1 receives the message and sends back to P the message ("$time$ = ", T), where T is the time Q reads on P_1. Then P estimates the time on P_1 with the formula: $C_Q^P(T, D_{PQ}) = T + D_{PQ}(1+2\rho) - \rho * min$, where $2D_{PQ}$ is the round trip delay between the transmission of the message ("$time$ =?") and the reception of ("$time$ = ", T), min is the delay for an empty message to be prepared on P_0, transmitted towards P_1 and received by P_1 in the absence of any transmission errors and any system load; ρ is the clock drift rate. The maximum error that P can make in its estimation is:

$e_{max} = D_{PQ}(1 + 2\rho) - min.$

Assuming that both messages ("$time$ =?") and ("$time$ = ", T) are each N bytes long, the maximum error with a conventional non-uniform-cost network

is:

$e^{NU}_{max} = (k_1 + k_2 N + \tau^{NU} + R^{NU})(1 + 2\rho) - (min_{NU} + \tau^{NU}_{min} + R^{NU}_{min})$

where τ^{NU} is the network latency, τ^{NU}_{min} is the minimum network latency, R^{NU} is the message reception delay, R^{NU}_{min} is the minimum message reception delay and min_{NU} is the minimum transmission delay of a null message (under no load and transmission errors).

With a uniform-cost network the error becomes:

$e^{U}_{max} = (kN + \tau^{U} + R^{U})(1 + 2\rho) - (min_{U} + \tau^{U}_{min} + R^{U}_{min})$

where τ^{U} is the network latency, τ^{U}_{min} is the minimum network latency, R^{U} is the message reception delay, R^{U}_{min} is the minimum message reception delay and min_{U} is the minimum transmission delay of a 1-byte message instead of an empty message (the delay of sending an empty message with a uniform-cost network is zero, by definition).

To compare the above mentioned errors we need to make some assumptions for the specific architectures of the networks, since the formulas involve the network latency and the reception delay of messages. Since in an experiment such as clock synchronization the messages are short (we use $N \leq 10$ bytes), we assume that $R^{NU} = R^{NU}_{min}$ and $R^{U} = R^{U}_{min}$. Also, for a non-uniform-cost network such as the Ethernet or token rings: $\tau^{NU} = \tau^{NU}_{min}$. For the network latencies in a uniform-cost network, we use the values measured on the PRAM prototype for an interconnection of $P \leq 12$ processors: $\tau^{U} = 155\ microseconds$ and $\tau^{U}_{min} = 5\ microseconds$. With these values, $\rho = 6 * 10^{-6}$ (I assume that ρ has the value used by [Cri89]) and R^{NU}, R^{U} and τ^{NU} less than 1 $msec$, the ratio of the errors in the two networks

is: $\frac{e^{NU}_{max}}{e^{U}_{max}} \approx 10$; this implies that improved results can be achieved, when a uniform-cost network is used. So, the use of a uniform-cost network leads to higher precisions in the estimation of remote clocks (at least for small size networks, as the above measurements show) and thus results to improved synchronization.

Both examples demonstrate that existence of uniform-cost networks allows efficiency and simplicity in problems which require exchange of small sized messages for their solution. Clearly, the important question that deserves study is the feasibility of building such a network with the desired performance characteristics.

2.3 Designing a Uniform-Cost Network

Conventional communication interfaces are expensive: the transmission delay is high. Commonly, a system call is used to transmit a message. The message transmission delay with the system call is quite long: one has to account for context switching, the code executed by the system call and the delay to transfer the data to a communication device. This delay quickly accumulates to hundreds of microseconds in conventional interconnections and is thus prohibitive for efficient uniform-cost communication.

The architecture described in this dissertation uses memory for uniform-cost communication in scalable multiprocessors. The architecture is based on the PRAM shared memory model developed by R. J. Lipton and J. S. Sandberg [LS88]. Memory provides a good means for communication in general, because a one-byte message can be viewed as a byte access in memory: the

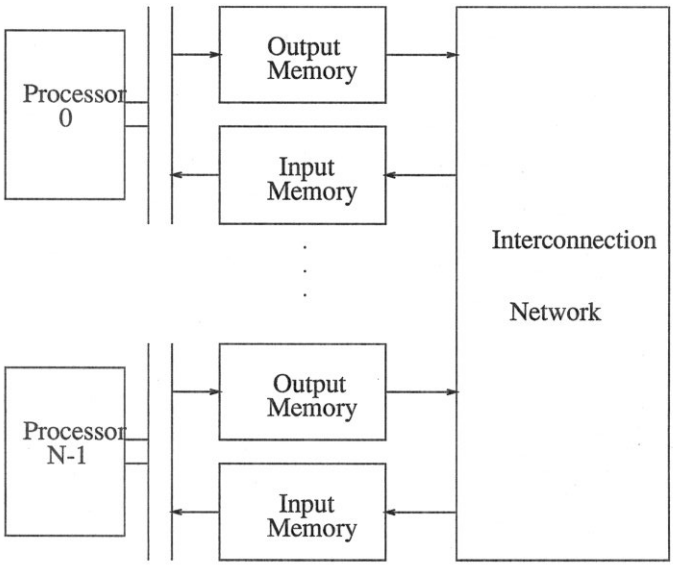

Figure 2.2: A Uniform-Cost Interconnection

destination address of the byte message corresponds to the memory address of the byte and the data of the message corresponds to the data in the memory address. Memory is especially suitable for uniform-cost communication, because:

- a block memory access of N bytes is equivalent in delay to N accesses of 1 byte each (*cost uniformity*);
- a memory access delay is less than a microsecond in conventional systems, which is the desired order for the parameter k in uniform-cost networks (*high performance*).

In a simple configuration, every machine connected to a uniform-cost communication network has some memory specifically used for communicating

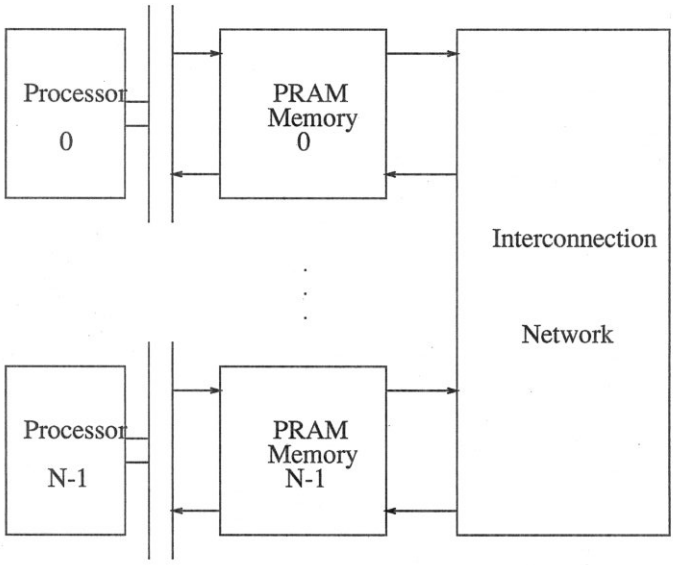

Figure 2.3: The PRAM Memory Interconnection

with the other processors of the network. The design for such a simple system is shown in Figure 2.2. Each processor is equipped with some write-only *output memory* and some read-only *input memory*. When processor P_i transmits a message, it writes it in its *output memory* and the network broadcasts it to all the other systems. The message traverses the network, arrives at the *input memories* of all the other systems and asynchronously updates them. So, the message becomes available to all the other processors P_j, $j \neq i$. In such a system one can pipeline the links of the network so that one data byte (or word) is inserted in the link during every memory access cycle. So, transmission is oblivious to the transmitting system and provided that the transmission/reception and network circuitry is fast enough, the data transfer rates of the interconnection can reach memory access rates.

If one combines the *output* and the *input* memories into one physical memory space (see Fig. 2.3), then one gets a shared memory model characterized by: high degree of scalability and no hardware-enforced coherence. The model, called *PRAM* [LS88], is suitable for fast transmissions with low end-to-end delays. A detailed presentation of the PRAM shared memory model follows in Chapter 3. An important feature of PRAM, aside from its high transmission rates, is that it provides a good interface to the programmer, since it is an easy memory interface.

The advantages of the memory interface become clear when we view interprocessor communication as a producer/consumer synchronization process between the communicating processors. Simple double buffering allows systems to transfer arbitrarily large amounts of data through the interconnect at high transfer rates. If one maps partitions of this memory in a process's address space then one can avoid the large context switching overhead of conventional communication and thus achieve low end-to-end delays, while the operating system can use other partitions of the memory and achieve high data transfers for system applications such as remote procedure calls, etc. Some performance measurements for such applications are presented in Chapter 6.

Chapter 3

The PRAM Memory Model

A distributed shared memory model, called *PRAM* (Parallel RAM) [LS88], allows the design of massively parallel systems with a large number of heterogeneous, geographically separated processors and achieves high data transfer rates and low latencies.

3.1 Introduction

The cost of interprocessor communication is a function of two parameters: the effective data bandwidth of the used network and the end-to-end delay of a message. This cost limits all conventional multiprocessor organizations in one way or another. Shared memory multiprocessors, which require high bandwidth and low latency communication, are conventionally limited to a small number of homogeneous processors that have to be in very short interprocessor distances (e.g., [CGBG88] [Hil86]). Message-passing sys-

tems [AS88], which scale easier and support heterogeneity, have higher communication cost than shared memory systems. Their message transmission delay is non-uniform causing a substantial overhead in problems which require exchange of a large number of small sized messages. Message-passing machines are also harder to program than shared memory systems because they provide a different programming model [LS88]. Finally, networks are slow for many desirable high-performance applications such as real-time CD quality audio data transfers and video data transfers.

The communication cost in all the above systems is high because the computing systems produce messages at higher rates than the effective bandwidth of the used interconnections although the links used for building these networks offer high bandwidth. A common characteristic of the previously mentioned network architectures which increases the cost of communication are the protocols. Protocols are used to meet certain requirements: shared memory MIMD machines require that their local memories or caches remain coherent at all time instances; some networks employ protocols to detect collisions and to recover from message losses, etc.

PRAM is a distributed shared memory model which decreases the communication cost because it dismisses hardware coherence protocols and allows the memory to become incoherent achieving high data transfer rates through the interconnect [LS88]. Since PRAM does not require any path-setup delays or packetization overhead it achieves low end-to-end delays.

3.2 A Communication Model

It is easier to evaluate the effect of PRAM on interprocessor communication, if we use the following simple communication model (this is the model used by Lipton and Sandberg in [LS88] for data motion).

Communication can be viewed as a simple process where one system sends data to another one in a producer/consumer fashion. The transmitting system (producer) sends the data of the message first and then it sends a special message (or signal) indicating that it is "DONE"; when the receiving system (consumer) detects the "DONE" message, then it knows that the whole message has been received and consumes it. If the consumer wants to transmit, then he becomes the producer and sends the message data followed by the "DONE" message and so on.

This model assumes that communication is reliable, but it is easy to extend it to include non-reliable communication channels by allowing error mechanisms, which support recovery in case of a failure; for example, the PRAM architecture presented in Chapter 4 describes one way to include an error mechanism in a PRAM interconnection.

This model provides a simple yet accurate description of communication in all kinds of interconnections: from tightly-coupled multiprocessors to long-haul networks. The PRAM shared memory model is optimal under this communication model and is thus suitable for any communication system [LS88].

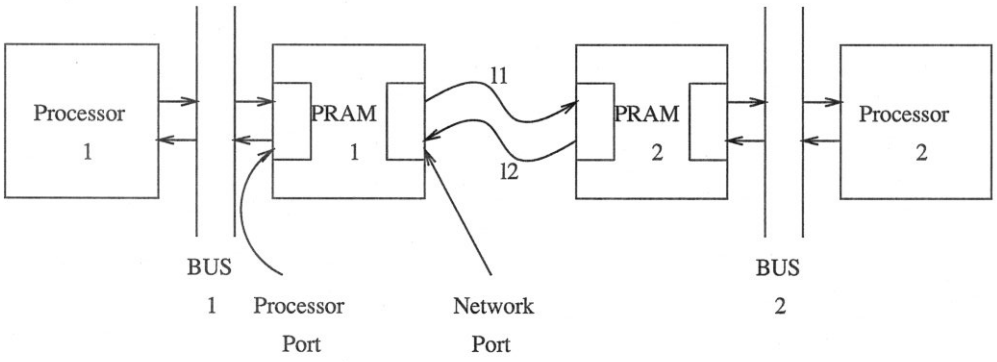

Figure 3.1: A 2-Processor PRAM System

3.3 The PRAM Model

PRAM is a scalable shared memory model developed by Lipton and Sandberg [LS88]. In the following a 2-processor PRAM system is described first to clarify some of PRAM's characteristics. The presentation is expanded from the description in [LS88] and is followed by the definition of the N-processor system from the reference [LS88].

Suppose that two processors P_1 and P_2 use PRAM to share memory. Then each keeps a local copy of the shared memory address space. Each local copy is a dual-ported memory with one port, the processor port, connected to the bus of the local processor and the other port, the network port, connected to the remote system through a communication link as shown in Figure 3.1. In order to read, each processor just reads its own local copy. In order to write, each processor just writes its own local copy and simultaneously sends a message through a link to the other shared memory copy; for example, in Figure 3.1, if processor P_1 writes data d in location k of its local PRAM

memory, it updates its local copy and simultaneously sends a message $\langle k, d \rangle$ through link l_1 to the network port of the local copy of processor P_2. When the message $\langle k, d \rangle$ arrives at the network port of P_2's PRAM memory, it updates through this port the local copy of P_2 with data d in location k asynchronously. There is no synchronization at any level between P_1 and P_2. So, P_1 can continue writing into the memory, being oblivious to the message transfers. In this fashion link l_1 can be pipelined, thus achieving data transfer rates equal to memory access rates and optimal latency. Note that the high performance does not depend on large block sizes: the performance is achieved even for single writes of one word; this is the effect of cost-uniformity. As Lipton and Sandberg [LS88] mention, the disadvantage of PRAM is that the shared memory can become incoherent but coherence can be enforced in software by using locks; furthermore, they show that compiler technology can be used to shield this incoherence from the programmer.

Lipton and Sandberg [LS88] also mention that performance is not affected by incoherence, since in many parallel programs write conflicts represent a small percentage of the memory accesses [EK88].

The PRAM model can be used to share memory among more than two processors. The formal definition of the PRAM model for N processors (from [LS88]) follows:

Definition 3.3.1 *[LS88] Let $P_1, P_2, \ldots, P_N$ be processors that share a memory address space with locations $0, 1, ..., m-1$. Assume that each processor P_i has a local memory M_i with memory locations $0, 1, ..., m-1$; local memory M_i is the i^{th} processor's copy of the shared memory address space. Each lo-*

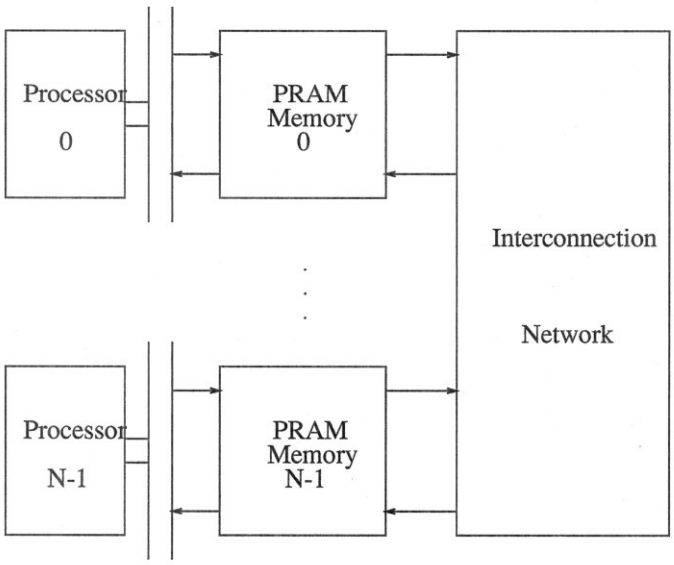

Figure 3.2: An N-Processor PRAM System

cal memory M_i is a dual-ported memory with a processor port and a network port, as described in the 2-processor PRAM case above. The processor port of the local copy is connected to the local processor's bus, while the network port is connected to an interconnection network, as shown in Figure 3.2.

All the local memories are initially in the same state. Each processor executes read and write commands on the shared memory address space:

1. *read(i): processor P_j performs a normal read access on location i in its own local memory M_j. This is a local action;*
2. *write(i, v): processor P_j performs a local action and initializes a global action. Locally, it updates location i of its local copy M_j with data v (this is the local action). Globally, it injects a message $\langle i, v \rangle$ in the interconnection network towards all other processors. This occurs in*

parallel with the local write operation. Processor P_i does not wait for an acknowledgement of successful receipt of the message by the other processors and it never receives one.

As the $\langle i, v \rangle$ messages arrive at the network port of other processors, P_k, they automatically update the local copies M_k asynchronously by writing value v in location i. In both reading and writing, a processor never waits for the completion of a global action, i.e. for successful receipt of the network messages by the rest of the processors.

The decoupling of local and global actions is the key feature of PRAM which allows the memory system to be scalable and with high performance [LS88]. This decoupling is the source of inconsistency in PRAM [LS88], but as the following theorem by Lipton and Sandberg [LS88] shows it is not possible to build a high performance, scalable and consistent shared memory system [LS88]:

Theorem 3.3.1 *[LS88] Let r (respectively w) be the best case (fastest possible) time to read (respectively write) some consistent shared memory. Then, $r + w \geq \tau$, where τ is the latency of the shared memory system (the delay between a request and its fulfillment).*

For a complete discussion of PRAM and its performance the reader is referred to the original report by Lipton and Sandberg [LS88].

3.4 A Classification of the Model

PRAM provides an alternative memory organization in multiprocessing systems, because it combines characteristics of both conventional shared memory and message-passing systems.

Conventional multiprocessor architectures –shared memory and message-passing– present two extremes in memory organization. Both organizations assume that there is a global memory that can be accessed by any processor in the system in some way.

Shared memory systems require this global memory space to be unique and all the processors to access it in the same fashion: with memory accesses that behave functionally as main memory accesses. The performance of these accesses is lower than regular local memory accesses in uniprocessor systems and access conflicts are typically serialized.

Message-passing systems distribute the global memory to the processors. Each memory cluster is assigned as local memory to one of the processors. A processor can directly access only its own local memory; whenever it needs data residing in another processor's local memory, it requests it from that processor with use of messages.

PRAM provides an alternative memory organization, because it replicates the global memory and assigns one copy to each of the system's processors as local memory. In this way, all the memory can be accessed as local memory by any processor achieving the performance of local memory accesses; memory updates by processors are broadcast to all the other processors through an interconnection. So, PRAM provides a third alternative to multiproces-

Table 1: Multiprocessor Memory Organization Characteristics			
Characteristic	*Shared Memory*	*Message Passing*	*PRAM*
Equal access times to all global memory	YES	NO	YES
Memory access delay equal to local access delay	NO	YES	YES
Decoupling of memory accesses and network use	NO	YES	YES
Immediate availability of the whole shared address space	YES	NO	YES
Software level coherence	POSSIBLE	N/A	YES

Table 3.1: Characteristics of Multiprocessor Memory Organizations

sor memory organizations. Since PRAM combines characteristics of both shared memory and message-passing architectures, it provides advantages from both organizations. Table 3.1 presents the main characteristics of the three memory models: shared memory, message-passing and PRAM.

As can be observed, PRAM offers advantages of both conventional multiprocessor memory models, because it allows each processor to have a local copy of the whole shared memory space and to access it with local access delays through the processor port, while remote write operations update the non-local memory copies asynchronously through their network port without the local processor's intervention.

Chapter 4

A PRAM Architecture

This chapter presents the architecture of a uniform-cost network using the PRAM shared memory model [LS88]. The interconnection is a dynamic, asynchronous, packet-switching network with distributed control and allows heterogeneous interconnected systems to communicate through a shared memory space.

4.1 Introduction

An interconnection among N systems, as shown in Figure 3.2, requires an architecture for the PRAM memories and the interconnection network. The PRAM memory designs are simple dual-ported memory architectures which are equipped with the necessary communication circuitry. Section 4.2 describes the organization of the PRAM memories implemented in a prototype and presents their performance characteristics. The described PRAM mem-

ory architecture can be used to implement PRAM memories for any existing computing system. The only difference would be the memory interface which heavily depends on the specifics of the system for which the memory is designed.

The interconnection is a dynamic, asynchronous, packet-switching network composed of switches. Switches have full-duplex ports which communicate through a high-speed bus within the switch. A switch port can be connected to either a PRAM memory or a port of another switch. By interconnecting many switches, one can build arbitrarily large networks.

The architecture of a PRAM system greatly depends on the purpose of the interconnection and the performance requirements: the PRAM memory model can be used for architectures of systems ranging from supercomputers to long-haul networks. Our purpose was to connect autonomous systems which have memory cycles in the order of a few hundred nanoseconds. This goal as well as the technology available at the time, led to the development of the architecture presented in this chapter. Since a prototype was implemented, the presentation is driven by the implementation and includes the characteristics of the prototype as well as its performance, wherever appropriate.

4.2 A Two-Processor PRAM System

The two-processor prototype connects two autonomous systems which can be either IBM ATs, SUN-3s or Mac IIs and is a direct implementation of the

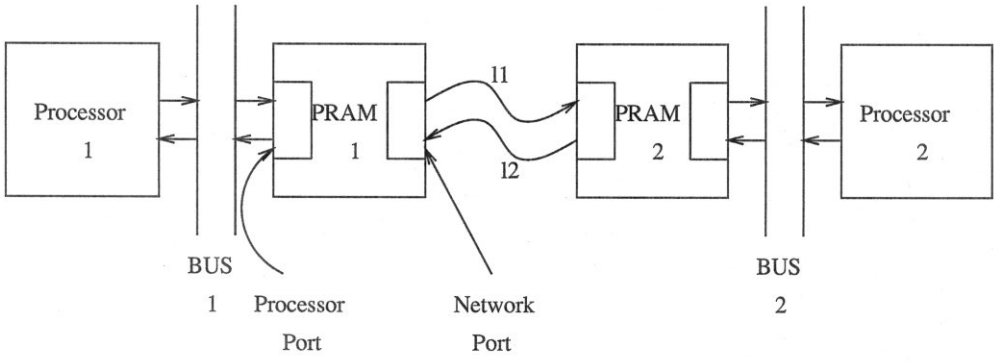

Figure 4.1: The Two-Processor PRAM System

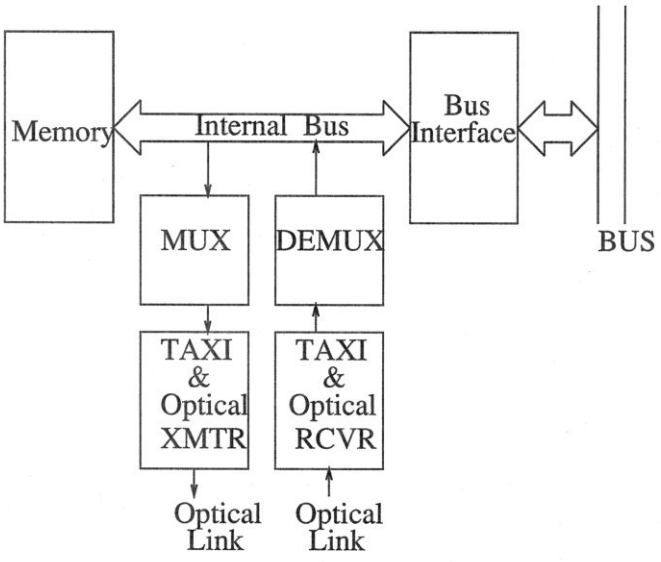

Figure 4.2: Organization of the PRAM Memory Board

2-processor PRAM system as presented in Section 3.3 (see Fig. 4.1). Each of these systems has a custom board on its bus, which contains the PRAM memory. The organization of the board is shown in Figure 4.2.

The two processors share a 32 $KByte$ memory space. Each board contains a 32 KByte memory on it (PRAM memory), which contains the processor's local copy of the shared memory address space. The memory has been implemented with conventional single-ported memory chips. An arbiter time-multiplexes the memory accesses from the local processor and the network (just a link in the two-processor prototype), allowing the memory to operate as dual-ported: each bus cycle of the local processor is divided into two subcycles, the first one of which accesses the memory upon the request of the local processor, while the second one updates the memory upon the network's request. The decision to build a dual-ported memory out of single-ported memory chips was made because of the cost: dual-ported memory chips with large sizes are quite expensive. Since the memory is functionally a dual-ported memory, we model it as a dual-ported memory with each of the ports dedicated to satisfying the accesses of either the local processor or the network. The port that satisfies the local processor accesses is called the *processor port*, while the other one is called the *network port*. The memory can be accessed in a byte (8 bits) or word (16 bits) fashion.

Whenever a processor reads the shared memory, it performs a read access on its local memory which is satisfied through the processor port of its dual-ported PRAM memory. When it updates the shared memory, it writes in the memory through the processor port and simultaneously transmits a 32-bit

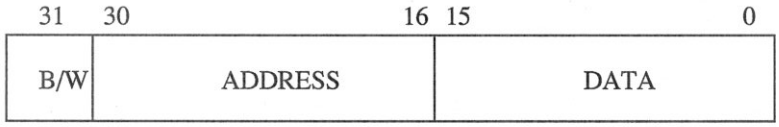

Figure 4.3: The Format of the Messages

message through the transmitter to the other processor. The transmission occurs in the same bus cycle with the update. Further, the processor updating the memory can return to its computations (or continue transmitting) because there is no synchronization of any kind between the two processors. The transmission circuitry serializes the 32-bit message (after encoding it) and broadcasts it through a fiber link.

The format of the 32-bit message is shown in Figure 4.3. The low order 16 bits are data bits and the next 15 are address bits. The MSB (Most Significant Bit) distinguishes byte and word operations.

When the message reaches the remote PRAM board, it updates the board's memory through the network port by writing the data contained in the low 16 bits of the message in the address specified in the message (bits 30 through 16); the access is either a word or a byte access, depending on the value of the Most Significant Bit (bit 31) of the message. This access is identical to the access that occured in the transmitting system through its processor port. If the write access is in one of the lowest 4 bytes, then when the message reaches the network port of the remote PRAM memory, not only does it update the memory but it also raises an interrupt requesting the processor's attention. The interrupts are necessary for efficient use of the

memory and for network management as will be described later.

The boards also have two CRC registers: one for the transmitted messages and one for the received messages; the purpose of the CRC registers is to increase the error detection and correction abilities of the communication system. The CRC registers can be read and reset by the local processors through an I/O port.

This prototype implements PRAM shared memory between the two connected systems, offering uniformity in the communication cost between the two processors: the cost of transmitting one word of information is in the order of 500 nanoseconds (the exact delay depends on the specific system: IBM AT, SUN-3 or Mac II) and the cost of transmitting N words is approximately N times the cost of transmitting one word. The delay of transmitting N words is not exactly $N * T_{memory}$, where T_{memory} is the local memory access delay, because the communication chips available at the time the prototype was built are slower than the memory of the systems interconnected. New, faster circuits are already available which will allow the new generation network to achieve transmission bandwidth exactly equal to the memory bandwidth.

PRAM custom boards for the IBM ATs have been operating since the Fall of 1988 and they have demonstrated highly reliable operation and communication with transfer rates up to 24 $MBits/sec$ and bit error rates less than 10^{-15}. The SUN-3 boards have operated since the early Spring of 1989 achieving the same performance, while printed circuit boards for the Mac II have been operational since late Fall of 1989.

4.3 The Network Prototype

It is clear that one should build a special network to connect more than two systems using PRAM. The network architecture developed is for a dynamic, asynchronous, packet-switching network with distributed control. The network is composed of high-speed switches, called PLANs, each with 4 ports; each switch is organized as shown in Figure 4.4. The prototype interconnection implements the developed architecture, except the memory management scheme which will be presented shortly.

Each processor connected to the network has a custom board on its bus as the one described in Section 4.2 and is connected to a switch port with a full-duplex connection as the one between the two PRAM memories in the 2-processor system: the connection consists of two links, each one dedicated to one-way transmission. The packets traversing the interconnection are the ones shown in Figure 4.3.

The switch architecture is bus-based. The 4 ports of the switch receive incoming messages and they multicast them. When a port receives a message, it demultiplexes it and stores it in a FIFO. The bus is dispatched to all the ports in a dynamic daisy chain scheme (for fairness in bus arbitration) and is eventually granted to the port which received this message. As soon as the port gets control of the bus, it broadcasts over it the message to the other 3 ports. When these ports detect the message on the bus, they latch it and use the message's address bits as a key to access a locally stored table and decide whether they should transmit it or not. Some of the ports will finally transmit it, while the rest will discard it. This scheme is used to reduce the

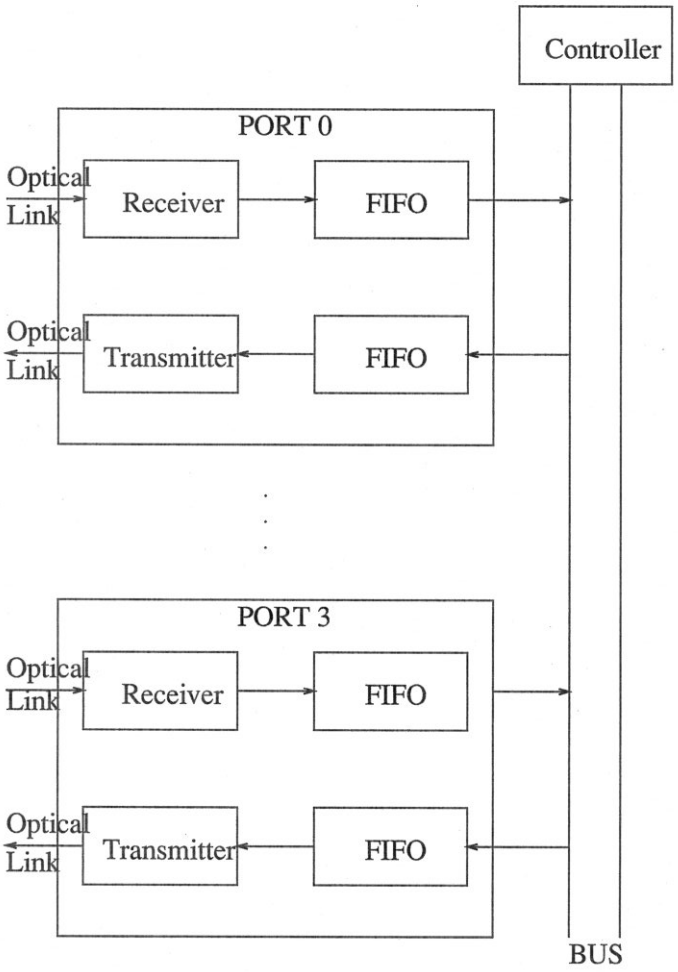

Figure 4.4: The PLAN Switch

network traffic and will be described in more detail in the next subsections.

The design issues of the network that are worthy of further analysis are:

- **data flow control**: because data are lost when buffers overflow;
- **error handling**: since link failures are possible and parts of the network should be functional, when such failures occur;
- **deadlocks**: because network management messages (for flow control and/or error handling) should not bring the network to an unrecoverable deadlock;
- **network traffic control**: because data flood the network if all processors transmit every message to all the other processors.

4.3.1 Flow Control and Error Handling Protocols

PRAM [LS88] is a shared memory model requiring all the local PRAM memories to receive the data inserted by any processor updating its local PRAM memory. This requirement causes broadcasting of all incoming messages to a switch. Since the message transmission rates at the outputs of the switch are equal to the message reception rates at the inputs, the incoming messages to a switch are serviced at a rate equal to approximately 1/4 of the incoming message rate. This implies that if a switch port continuously receives messages, then its input FIFO will eventually overflow and lose data. A solution to this problem is use of a flow control protocol. The data flow protocol implemented in the prototype is a hardware version of a *Stop&Wait* protocol: whenever the input FIFO of a switch becomes `Half-Full`, it sends a `STOP`

message through its output link to the system (switch port or processor) that transmits to it. The system receiving the `STOP` message suspends transmission, until it receives a `START` message. The port that sent the `STOP` message monitors the input FIFO and when it "sees" the FIFO under `Half-Full`, it sends a `START` message to the previously `STOP`ped system, which in turn resumes transmission (specifically, in the prototype the `START` message is transmitted as soon as the FIFO becomes `Empty`). This simple scheme does not allow data to be lost due to buffer overflows; the only condition that has to be satisfied for the scheme to be correctly operating is: the delay to transmit, propagate and recognize a `STOP` message should be less than the delay to transmit and propagate $FIFO/8$ consequtive messages between 2 systems (switch ports or processors). The condition guarantees that when a port sends a `STOP` message, the connected, transmitting system will receive it and suspend transmission, before the `Half-Full` FIFO becomes `Full`.

It should be mentioned here that processors are able to consume messages at the network rate and thus never send `STOP` messages to switch ports connected to them.

This simple hardware *Stop&Wait* protocol is coupled with an error handling mechanism that supports error recovery in case of link failures. The error mechanism operates as follows: whenever a link fails, the switch port (or processor) receiving through it identifies the failure with special circuitry (on the prototype, the TAXI chips used for reception have a special pin that indicates errors). As soon as the failure is detected, the port "shuts" itself down and resets all its storage cells but not before it sends out to all directions a special ERROR message that includes the port address. Shuting

down the port means that the network will be partitioned in two networks, which will be operating independently, until the port becomes operational again. Special care must be taken so that the subnetwork that includes the port transmitting to the failed link does not "wait" for a START message, if it was STOPped, when the failure occured. In the meantime a timer in the failed port monitors the receiving line and as soon as it identifies a time interval T_f (1 $msec$ in the prototype) during which the line does not identify any errors, it makes the port operational again. So, the network is unified as soon as the failure is fixed. A simple extension to the implemented mechanism can have the processors informed of the status change of the port by a special `ALIVE` message the port would send to all directions as soon as it becomes operational.

4.3.2 Deadlocks

One has to pay attention to the coupling of the two protocols and make sure that no combination of START, STOP and ERROR messages brings two communicating ports to a *deadlock*. The *flow control* and the *error* protocols can be modelled with a finite automaton that describes the state of a port at every time instant. The protocols were verified for deadlock and livelock freedom with the protocol verifier SPANNER [ABM88] (see Appendix A).

Although the protocols do not allow deadlocks between two communicating switch ports, there might be a case with STOP messages in the network where a deadlock is unavoidable. Because of the operation of the switch one can prove that

Theorem 4.3.1 *A PRAM network built with the PLAN switch will not have a deadlock* **iff** *there is no cycle in the network topology which contains only switches.*

Proof: Taking into account the operation of the switch, one can prove that a switch stops its operation completely *iff* there are more than two ports which have received a STOP message. If only one port is STOPped then the switch transmits through the rest 3 ports all the messages incoming to the STOPped port.

Clearly, a deadlock in the network would require a circle of switches where all of the ports in the cycle have received a STOP message. In the cycle, all the switches will have simultaneously at least two STOPped ports each and thus a deadlock.

If there is no circle in the network, then the network can be modelled with an unrooted tree, where the leaves are the system's processors. Assume that all the ports of the network have received a STOP message. Since the processors never send STOP messages to the ports that transmit to them, they eventually consume the data of the switches connected to them. Since there are no cycles in the network, the switches on the first level of the tree are connected to 3 processors each. As soon as the processors consume an appropriate amount of the queue of such a switch's STOPped port, the port will send a START message to the switch on the second tree level. Eventually all the second level switches will send START messages to their neighboring switches and recursively, all the network switches will eventually receive START messages and will become operational again. So, there can

be no deadlock, if there is no cycle in the network topology. QED. □

Cycles are allowed in the network only if they include a processor (with 2 PRAM boards). It should be mentioned here that cycles that involve only switches are forbidden for two reasons:

- messages will be locked in a cycle, looping indefinitely and generating new messages;
- cycles can cause deadlock.

4.3.3 Network Traffic Control

Network traffic is a problem in a PRAM network, because every message inserted in the network has to be broadcast to all connected systems. If all processors update their PRAM memories simultaneously, there are $O(N^2)$ messages traversing the network (N is the number of processors in the system) [LS88]. However, it is often not necessary for a processor to send data to all other systems but just to a few of them. In the system described thus far this is not possible: all data written in one PRAM memory are received by all the connected processors. One can reduce network traffic by allowing the switches to multicast incoming messages instead of just broadcast them.

One solution that allows efficient use of the PRAM memory and decreases network traffic is to use a memory management scheme. The memory management scheme described here requires that each switch port stores a small table, which serves the purpose of a *routing table*. The tables implement distributed routing and are programmable by the system processors. The

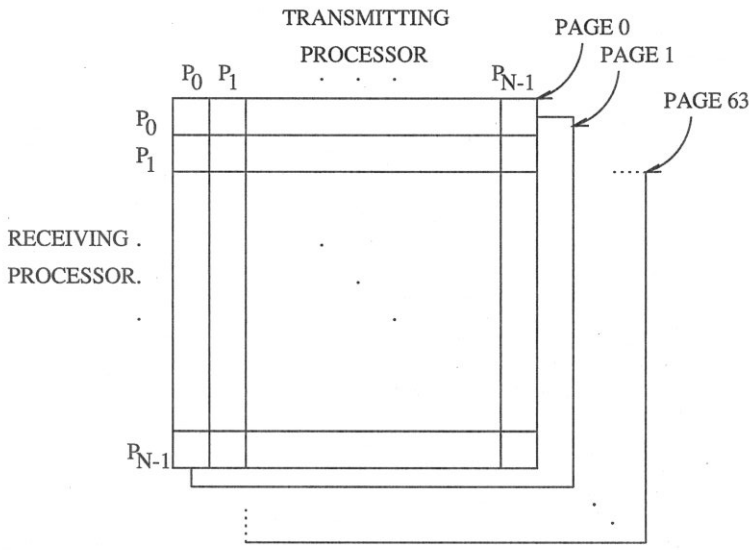

Figure 4.5: The PAGE-SHARE Table

philosophy of the scheme is the following: the PRAM memory space is divided in pages (the current decision is for 512 bytes per page) and each processor defines the pages it shares with a subset of the processors; for example, processor P_1 could share the page with number 2 with processors P_2 and P_3, while processor P_2 wants to share page 5 with processor P_4. When this definition is made, the memories in the switches are programmed so that the appropriate paths are traversed by each message. Referring to the page sharing definition given above, the tables would be used by the ports, so that an update, for example, from processor P_1 in page 2 would be directed only towards processors P_2 and P_3, while processor P_4 would never know that P_1 ever made the update.

PRAM memory is paged in 64 pages of 512 bytes each. Each processor

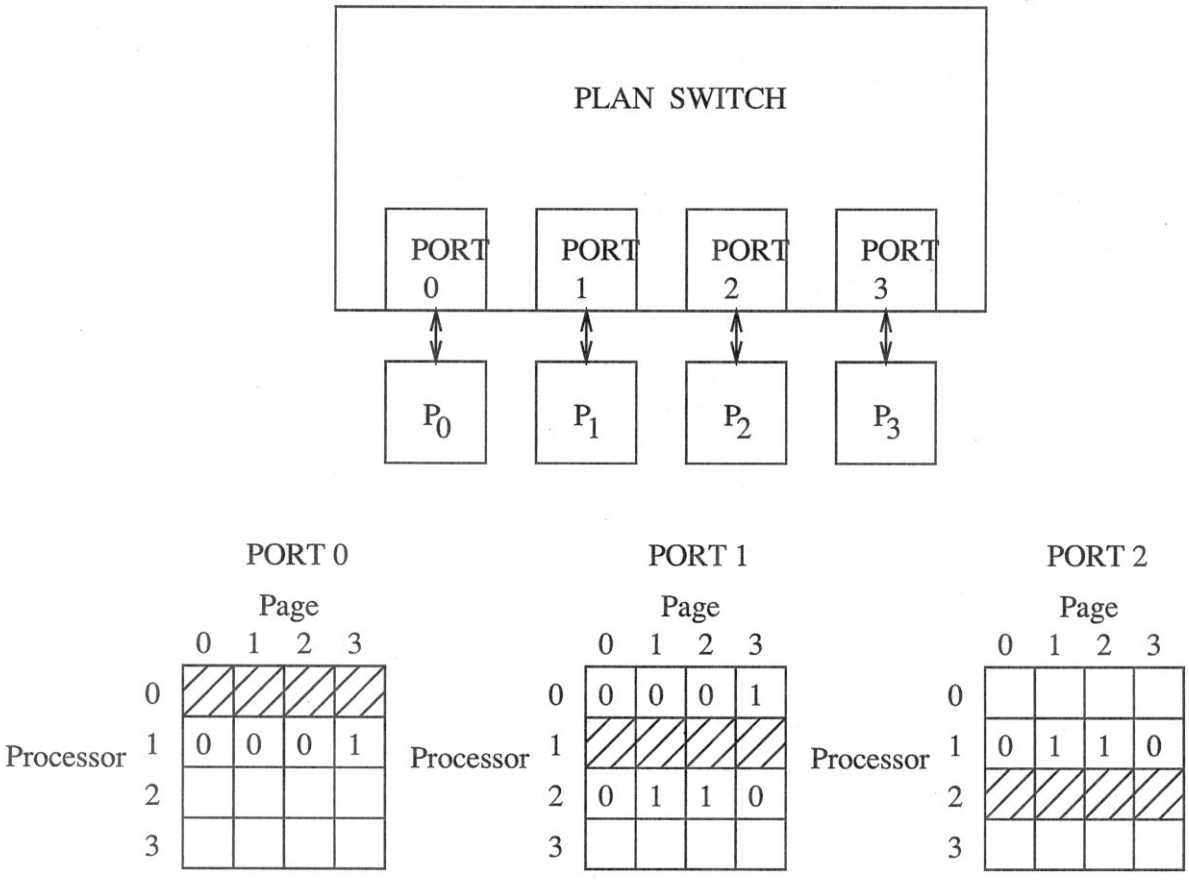

Figure 4.6: Distributed Page Tables

defines the set of pages it wants to share with other processors by updating a *PAGE-SHARE* table (see Figure 4.5). The policy for this definition is: *each processor defines which updates it wants to receive.* This policy makes the switch design simpler and increases security in the system (no processor can forcibly corrupt another processor's memory). The described PAGE-SHARE table is global to the system and is not stored in the network as shown in Figure 4.5 but is distributed to the network switches. The routing table stored in each switch is a 3-dimensional table with the semantics:

$$T[page\_no, in\_port, out\_port] = \begin{cases} 1 & \text{if a message with an address in page} \\ & page\_no \text{ arriving at port } in\_port \\ & \text{should be transmitted by port} \\ & out\_port; \\ 0 & \text{otherwise.} \end{cases}$$

Each port stores a small table which indicates which messages will be transmitted through its transmitter: switch port $Port_i$, $0 \leq i \leq 3$ stores the routing table part with $out\_port = i$. When a message is latched in a port from the internal bus of the switch, the locally stored table (the *routing table*) is accessed with key the incoming port number and the page address of the access (the page address is part of the message). The table returns just one bit, which instructs the port to either transmit the message or discard it.

As an example, assume that the whole memory space has just 4 pages and that we have the network configuration shown in Figure 4.6 with only one switch. The tables stored in the ports allow processors P_0 and P_1 to share page 3, while processors P_1 and P_2 share pages 1 and 2. Port P_3 does

not participate in the shared memory configuration. The switches are programmed with a message that corresponds to one of the interrupt locations of the shared memory address space (memory mapped programming).

This memory management scheme allows distributed control and offers:

- high performance, because of the absence of a central controller;
- low network traffic, since messages are "filtered" in (multicast by) the switches;
- higher effective network bandwidth;
- lower effective network latency;
- more efficient use of the shared memory address space.

Chapter 5

Testing the Prototype

Testing the prototype led to the development of PLADO, a simple language for writing parallel testing programs and motivated research on a verifier, which identifies deadlocks, starvation and timing dependencies in these programs.

5.1 Introduction

Testing and verification of computing systems is a hard process. Hardware testing is a problem that has been extensively studied (e.g., see [Lal85], [BF76]) for system levels ranging from switch and gate level to microprocessor designs. The higher the system level, the harder the problem is. Researchers have proposed some methodologies for testing specific high level designs of microprocessors [SH81] [NR82], and regular designs [VS86], but there is no methodology applying to most systems. The problem seems to be especially

acute for multiprocessing systems and networks, where the issues involved are many (topology, communication method, etc.). Special attention has been drawn to communication protocol testing and verification (e.g., see [Sun81]), while high level system testing is still unexplored; as a result most of the conventional parallel systems and networks are tested using *ad hoc* methods which seem appropriate only for the system at hand.

The purpose of testing a system module is to apply certain binary patterns to the inputs of the tested module and record the results in order to detect (and locate) faults. Testing a parallel system is a complex process that involves testing of many modules and functions: interconnection hardware, memories, protocols, etc. The process can easily fail, because of the many factors involved. For example, a test to exercise a switch of an interconnection using a parallel program can fail due to problems ranging from hardware failures to program errors. Because of the inherent complexity of parallel programs, it is often the case that a failure of a test is not due to the hardware circuit being exercised but due to flaws in the test program itself. This is the main problem we address in this chapter: how to identify problems in test programs of parallel systems and how to prove such programs correct.

In the context of this chapter, a *test* is a set of independent programs which run simultaneously on different processors and communicate through shared memory. A test is considered *correct*, if there is no possibility of deadlock or starvation and if there is no dependency on the processors' relative execution speed (no timing dependencies).

It should be clear that this question of correctness is well defined and is

closely related to the problem of program correctness. Program correctness is a hard problem in computer science that has been extensively studied. Various methodologies have been proposed to prove programs correct, e.g. Hoare's Logic [Hoa69] [Lam77] and temporal logic [Pnu77].

Test programs, as we defined them previously, are a special case of parallel (communicating) programs. Their main characteristic is that they need a small repertoire of instructions. This characteristic led us to the development of PLADO, a special, simple language for writing test programs. PLADO is useful for testing the interconnection parts of shared memory multiprocessors, because in most practical cases the processors used in these systems have been fully tested in a stand-alone mode before they are interconnected. One can write PLADO programs that exercise various parts of a multiprocessor interconnection. The advantages of using a special language are mainly clarity and portability; portability is important, because it allows test programs to run in the same way on heterogeneous machines.

Verification of the correctness of tests requires simulation of the behavior of an appropriate hardware model when the test programs are executed on it. The model's behavior can be simulated with a finite automaton which recognizes all the possible execution sequences of the test. Potential problems can be identified through the structure of the automaton.

5.2 The Shared Memory Model

For every system we distinguish 2 different models:

- the *hardware model*, that includes the necessary details for testing;
- the *programming model*, which is used for programming the system.

A shared memory system, for example, can be implemented in various ways: with a bus, an interconnection network, the PRAM model [LS88], etc., but the programming (or *functional*) model is basically the same: a shared memory accessible by a number of processors providing atomicity of memory operations at the byte level.

When testing a system, one needs the hardware model to find the necessary binary test patterns and write meaningful and useful tests. The programs though which run on the processors of the system and bring the test patterns to the exercised hardware modules of the system, are "correct" under the programming model.

The shared memory programming model we use follows; there are K processors: $P_0, P_1, ..., P_{K-1}$. Each processor has access to a *shared memory* with $(M + N)$ locations: $SM[0], ..., SM[M + N - 1]$. The first m locations of the shared memory SM are *flags*: $SF[0], ..., SF[M - 1]$ with $SF[0] = SM[0], ..., SF[M - 1] = SM[M - 1]$. The flags are binary; each one can be either 0 or 1. The last N locations of the shared memory constitute the *shared buffer* SB, with $SB[0] = SM[M], ..., SB[N - 1] = SM[M + N - 1]$. Each location of the shared buffer SB can contain a character from an alphabet Σ. Every processor executes a program written in PLADO.

When more than one processors access the same shared memory location simultaneously, the result is a random serialization of the accesses.

We do not make any assumptions about the speed at which various processors execute their programs. We just assert that every statement takes a random but finite amount of time to execute on a processor; this accounts for all types of uniprocessor systems that might be connected to the shared memory: time-shared systems, special-purpose systems, etc., but it does not account for complete processor failures.

In the following the terms *machine* and *system* refer to the multiprocessing organization described above, consisting of the K processors $P_0, ..., P_{K-1}$ and the shared memory SM.

5.3 PLADO

A *test* in PLADO is a set of independent programs, each running on a different processor. The programs communicate with each other through the shared memory.

When one wants to test circuits (modules) of a shared memory system's interconnection, one writes a *test* that brings the necessary binary patterns to the inputs of the module under test.

PLADO is a special, simple language for writing the programs of the test. The language was developed for the following reasons:

- Programs should be portable and run in the same way on heterogeneous processors, independent of issues such as byte ordering, memory access

atomicity, hardware synchronization locks, etc.;

- The programs' function should be clear and consise;
- Testing is a process which does not require a large repertoire.

Since we are not interested in testing a system's processors but its interconnection, the test language should just be able to operate on memory and/or I/O ports, depending on the exact system implementation. For example, for the shared memory multiprocessor system we have built, we need to have only a few functions: read memory, write memory and synchronize (to implement interprocessor synchronization) with a busy-wait statement or with interrupts.

The language offers 2 kinds of statements:

- *non-memory statements* which allow one to install the shared memory address space in different locations of each processor's virtual address space, provide statistical or program execution information on the terminal, allow continuous execution (looping) etc;
- *memory statements* which operate on the shared memory.

Our main interest is in the memory statements, which have the syntax and semantics summarized in Table 5.1. The shared memory can be accessed with either byte (8-bit) or word (16-bit) operations.

Statement *write_byte* x fills the shared buffer SB with the 8-bit quantity x with byte accesses. Similarly, *write_word* x fills SB with the 16-bit quantity x in every word boundary.

Table 2: PLADO Memory Statements	
Syntax	**Semantics**
write_flag i x	write value x (0 or 1) in flag $SF[i]$
write_byte "x"	write the 8-bit quantity "x" in all byte locations of SB
write_word "x"	write the 16-bit quantity "x" in all word boundary locations of the shared buffer SB;
wait_flag i x	busy-wait on flag $SF[i] = SM[i]$, until it contains value x (0 or 1)
wait_byte "x"	busy-wait on the shared buffer SB, until every byte location contains "x"
wait_word "x"	busy-wait on the shared buffer SB, until every word location contains "x"

select_flag(i)
{
case 0: statement_0;
case 1: statement_1;
}
	read flag $SF[i]$,
	and select as next statement
	statement_0, iff $SF[i] == 0$
	statement_1, iff $SF[i] == 1$

select_byte(word)(SB)
{
case "x": statement_x;
...
case "z": statement_z;
default: statement;
}
	read buffer SB with byte (word) operations
	select as next statement for execution the statement in *case* "i",
	where i matches the values read in SB.
	If there are more than one different values in SB,
	then execute the case statement *case* "w"
	where w is a string that contains exactly
	the different values read.

Table 5.1: Syntax and Semantics of the PLADO Language Statements

A *wait* statement is a busy-wait, until the shared buffer is filled with a specific value. For example, *wait_byte* x busy-waits, until all bytes in SB contain the 8-bit quantity x. All the accesses involved in the execution of the statement are byte operations. A *wait_word* statement operates analogously but it involves word operations and x is a 16-bit quantity.

Write and *wait* statements exist for the flags, too. Statement *write_flag* i x sets or resets the flag $SF[i]$, depending on the value of x (1 or 0 respectively) and statement *wait_flag* i x busy-waits, until flag $SF[i]$ gets value x.

The last statement of interest is the *select* statement, which resembles the *switch* statement in C. *Select* performs a *read* operation either on a flag $SF[i]$ or on the shared buffer SB, whichever is specified and chooses the statement that should be executed next. There are 3 types of *select*:

- *select_flag(i)*: reads flag $SF[i]$;
- *select_byte(SB)*: reads the shared buffer SB with byte operations;
- *select_word(SB)*: reads the shared buffer SB with word operations.

If a flag $SF[i]$ is read, then the next statement to be executed is chosen depending on the value of $SF[i]$ (0 or 1). If the buffer SB is read, then the value returned by the *read* on the buffer SB is not the exact pattern SB exhibits, but a string of bytes or words (depending on the type of select specified) containing the *different* values read in SB. Then the next statement to be executed is the one in case: *case* x, where string x contains the same values (bytes or words) returned by the read operation.

The memory statements on SB are *composite* statements: they perform a

sequence of *atomic* operations. An *atomic* operation involves a single shared memory location (byte or word). The PLADO *composite* statements consist of N consecutive atomic byte operations (N is the size of the buffer SB in bytes) or $N/2$ consecutive atomic word operations proceeding from the low order shared memory locations to the high order ones; e.g.

write_byte "x" is actually executed as:

$for\ (\ i = \ 0;\ i < \ N;\ i++\ )\ SB[i] = \ $"$x$";.

This implies that whenever 2 or more different processors write into the shared buffer simultaneously, their *atomic* accesses (*write* operations) can be interleaved in an arbitrary way and different locations of the buffer could end up with different values. For example, if one processor fills SB with the character "x" and another one fills it with the character "y", then each location of the shared buffer SB will finally get either an "x" or a "y" in each location. The resulting pattern in the shared buffer will be one out of the 2^N possible patterns of interleaved "x" and "y"s. We assume that all these patterns can occur with some positive probability, since we do not make any assumptions about the execution speed of the processors.

An example of a test with two PLADO programs is shown in Figure 5.1. Here there are 2 processors, P_0 and P_1, performing a test. The test is a producer/consumer synchronization process. Both processors become producers and consumers at different time instances. The test is useful in exercising the arbitration circuitry in virtually all possible shared memory designs, although the faults covered could differ, because of the specific differences of the implementations.

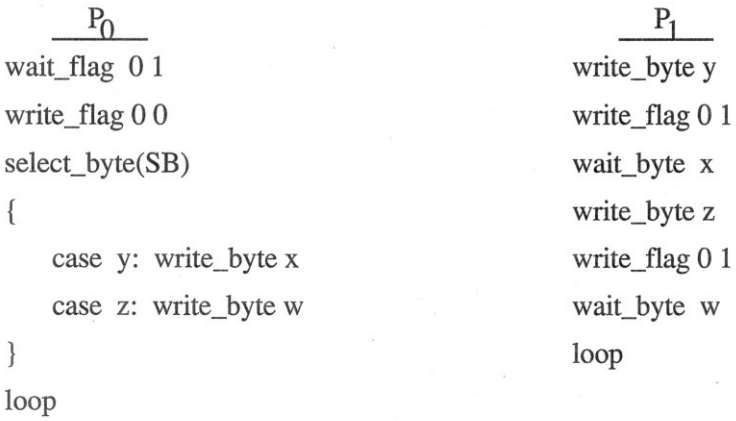

Figure 5.1: A Test

P_0 executes the program on the left of the figure, which waits until flag $SF[0]$ is set (becomes 1); then P_0 resets $SF[0]$ to 0 and reads SB with byte operations (due to the statement *select_byte*(SB)). If all the bytes in SB contain a y, then P_0 proceeds and fills SB with x in every byte. If all bytes contain a z instead of a y, then P_0 writes a w in every byte location of SB. *Loop* causes the program to be continuously executed.

In the meantime, P_1 fills SB with y in every byte and then it sets flag $SF[0]$ and waits until SB contains an x in every byte. When SB is filled with x (by processor P_0), P_1 fills SB with byte z and sets the flag $SF[0]$ again (P_0 has reset it, as explained above) and waits until all bytes in SB contain a w. The *loop* statement at the end causes P_1 to continuously execute the program; in PLADO there is only one kind of *loop* which causes the program

to loop back to the first script statement.

5.4 Simulation

The problem we discuss in the rest of this chapter is that of verifying that test programs are free of timing dependencies, deadlocks and starvation through simulation of their execution. The focus of this section is the simulation of the execution of the programs, while the next one discusses verification.

For the analysis here we use a model simpler than the one presented for PLADO above. This model captures all the essential characteristics of the parallel execution of test programs that affect the complexity of simulation, while it is free of the PLADO specific details that complicate the analysis.

The architectural model we use is the following:

- there are K processors: $P_0, \ldots, P_{K-1}$;
- there is a multiported (K-ported) shared buffer SB with N locations accessible by all K processors of the system (we do not use the flags mentioned in the previous section; the extension to include them is simple);
- whenever more than one processors simultaneously access the same location of SB, the accesses are serialized in an arbitrary way.

Each processor P_i, $0 \leq i \leq (K-1)$, executes script L_i.

Definition 5.4.1 *A* **script** L_i *is a test program executed on processor* P_i*; if* L_i *is a script,* $l_i = \mid L_i \mid$ *is the length (size in statements) of* L_i*.*

The main simplification over PLADO is in the statements that are allowed in a *script*; there are only two types of statements: *read* and *write*. The main characteristics of these statements are:

- each statement of a script is a composite statement which either reads or writes in SB with atomic operations and has the *buffer property*, i.e. the atomic operations of a composite statement access all locations of SB consequtively proceeding from $SB[0]$ to $SB[N-1]$.

- a *write* statement has the syntax: *write* x; the semantics of the statement is that it fills SB with value x in every single location;

- a *read* statement on processor P_i has the syntax of a *case* statement: *read*{"A_1": S_1; "A_2": S_2; ..., "A_M": S_M; *default:* S_{M+1};}; where A_j is a set of values and S_j is a script composite statement; the processor reads SB and accumulates the set of different values, SB_i, in the buffer (not its exact pattern). P_i then finds the set A_j, such that $A_j = SB_j$ and executes the corresponding statement S_j next. This model for the *read* resembles the *select* statements in PLADO. A simple extention to include *jumps* would allow us to use this model for *wait* statements too.

The problem we want to solve is to simulate the execution of the K scripts L_i, $0 \leq i \leq (K-1)$ (L_i is executing on processor P_i). If we can simulate the execution of the scripts in a test, then we can build an automaton that recognizes the *valid* executions of the scripts. Given this automaton we can identify flaws in the test programs, i.e. deadlocks, starvation and timing

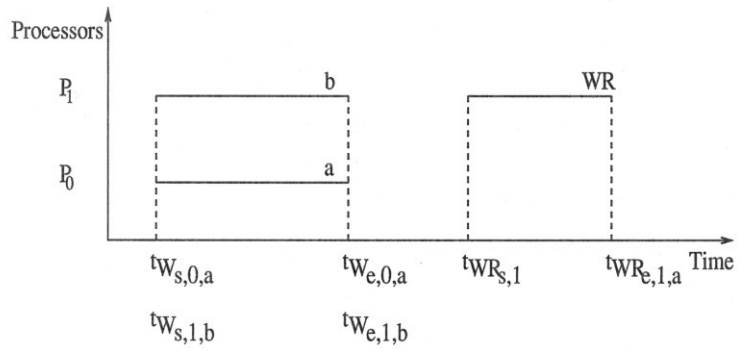

Figure 5.2: Execution of 2 Scripts

dependencies as we will discuss later.

To simulate the execution of the scripts of a test, we must be able to identify at every time instance the sets of values that reside in SB. In the following we focus on this problem, which is the main problem in the simulation.

5.4.1 Problem Description

An execution of a set of scripts in time can be represented with a diagram such as the one shown in Figure 5.2. We call such a diagram an *execution diagram*. The figure shows an execution of 2 scripts on processors P_0 and P_1:
on P_0: *write a;*
on P_1: *write b; read{"a": S_1; default: S_2;};*

In this *execution diagram* we use the following notation: $t_{W_{s,i,x}}$ is the time when processor P_i starts filing SB with x, while $t_{W_{e,i,x}}$ is the time when processor P_i ends filing SB with x. $t_{WR_{s,i}}$ is the time when processor P_i

starts reading SB due to a *read* statement, and $t_{WR_{e,i,(A_1,\ldots,A_M)}}$ is the time when processor P_i finishes reading SB and A_j, $1 \leq j \leq M$, are the sets with which the read value set is compared to determine the statement which will be executed next.

The two processors in the example, P_0 and P_1, execute their *write* statements starting at times $t_{W_{s,0,a}}$ and $t_{W_{s,1,b}}$ and finishing at times $t_{W_{e,0,a}}$ and $t_{W_{e,1,b}}$ respectively, while the *read* statement starts executing at time $t_{WR_{s,1}}$ and finishes at time $t_{WR_{e,1,\{a\}}}$. The figure shows the case when $t_{W_{s,0,a}} = t_{W_{s,1,b}}$ and $t_{W_{e,0,a}} = t_{W_{e,1,b}}$. To verify this execution of the scripts we have to identify the various sets SB_1 the *read* statement may read from the shared buffer SB.

In this simple example it is easy to identify that the *read* statement will return a subset of the set $\{a, b\}$. The nondeterminism of the exact contents of SB_1 is due to the arbitrary interleaving of the atomic statements of the simultaneously executing *write* statements. The following interleaving of atomic statements, for example, shows how the *read* statement may return $SB_1 = \{a\}$:

P_1 writes $SB[0]$; P_0 writes $SB[0]$;

...

P_1 writes $SB[N-1]$; P_0 writes $SB[N-1]$;

P_1 reads $SB[0]$, ..., $SB[N-1]$.

Clearly, this interleaving will result in P_1 reading $SB_1 = \{a\}$. If the order of P_0 and P_1 is interchanged when accessing each cell of SB in the interleaving, then the result would be: $SB_1 = \{b\}$, while in case we interchange the access order of the processors in a few (not all) cells the result would be:

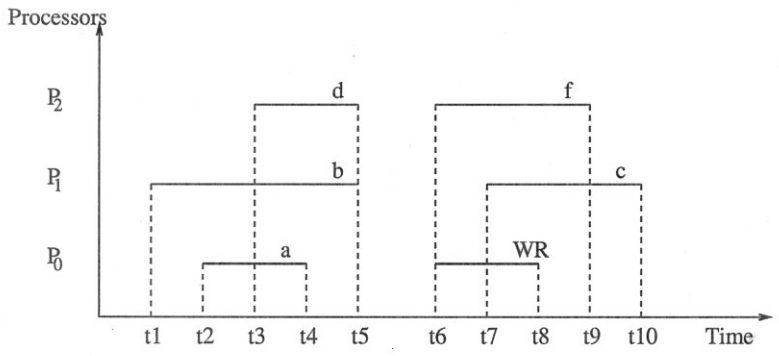

Figure 5.3: A 3 Script Example

$SB_1 = \{a, b\}$.

Although in this simple example it is easy to identify that the *read* statement will return a subset of $\{a, b\}$, it is not easy to identify the possible sets read when many *write* statements are executed on many processors. Figure 5.3 shows a more complicated example with only three processors. The processors P_0, P_1, P_2 execute the following statements:

P_0: *write a; read{"d": S_1; default: S_2;}*

P_1: *write b; write c;*

P_2: *write d; write f;*

The problem here is to identify the possible sets SB_0 returned by the *read* statement; for example, can the *read* statement return $\{d\}$?

Actually, the *read* statement here can return $\{d\}$, and the following interleaving of atomic operations of the scripts proves it (note that the interleaving satisfies the time order given in the figure):

at t_1: P_1 writes $SB[0]$;

at t_2: P_0 writes $SB[0]$;

at t_3: P_2 writes $SB[0]$;

between t_3 and t_4: P_0 writes $SB[1]$; P_1 writes $SB[1]$; P_2 writes $SB[1]$;

...

P_0 writes $SB[N-2]$; P_1 writes $SB[N-2]$; P_2 writes $SB[N-2]$;

at t_4: P_0 writes $SB[N-1]$;

at t_5: P_1 writes $SB[N-1]$; P_2 writes $SB[N-1]$;

At this time, SB is filled with the value d in every location. Then the interleaving continues as follows:

at t_6: P_0 reads $SB[0]$; P_2 writes $SB[0]$;

at t_7: P_1 writes $SB[0]$;

between t_7 and t_8: P_0 reads $SB[1]$; P_2 writes $SB[1]$; P_1 writes $SB[1]$;

...

at t_8: P_0 reads $SB[N-1]$;

at t_9: P_2 writes $SB[N-1]$;

at t_{10}: P_1 writes $SB[N-1]$;

With this interleaving $SB_0 = \{d\}$, but there are other possibilities for SB_0 too: $\{f\}$, $\{c,\ f\}$, etc. It is impossible though for the *read* statement to read only $\{a\}$, or $\{c\}$. For example, P_0 cannot read only $\{c\}$, because P_0 reads $SB[0]$ at time t_6, for which we know that $t_6 < t_7$ and t_7 is the time when P_1 writes c in $SB[0]$; so, P_0 reads a value different than c (actually, it has to be either a d or an f) in $SB[0]$ and thus SB_0 will definitely be different than just $\{c\}$. So, it is not clear what sets of values a statement can read.

Actually, the following general problem is still open:

Problem: Given a set of K scripts with M read statements, WR_1, WR_2, ...,

WR_M, an execution diagram for the scripts, and M sets RB_1, RB_2, ..., RB_M, can we decide if there is an interleaving of atomic statements that satisfies the timing of the execution diagram and has WR_i return to the reading processor the set RB_i, $1 \leq i \leq M$, in time polynomial in K and N? □

This problem is the main one in the simulation of test scripts, because the simulator has to accept valid executions of the scripts, represented by such execution diagrams. Although the above general problem is still open, joint research with R. J. Lipton and A. LaPaugh has given some results which are presented in the following subsection.

5.4.2 Simulation Results

Given a set of K scripts and an execution diagram for them, we can compute the sets of values returned by the *read* statements by expanding each composite statement to its atomic operations, creating all the possible interleavings of these operations and storing the pattern of the whole buffer SB at every time instance. This proves that the problem mentioned above is decidable, but the solution is inefficient because its complexity is clearly exponential in N, the size of the shared buffer SB.

Given an execution diagram of the scripts (which shows the start time and the finish time of each composite statement), we can compute at every time instant the values that may exist in SB as the following lemmas show.

Definition 5.4.2 *At time t, a composite statement with starting time t_{S_1} and finishing time t_{E_1} is* **dominated**, *if there is another composite statement*

which has starting time t_{S_2} and finishing time t_{E_2} and $t_{E_1} < t_{S_2} < t_{E_2} < t$.

Lemma 5.4.1 *At any time instant t, the values written by dominated write operations do not exist in SB.*

Proof: If a *write* statement ST_1 is dominated by a statement ST_2, then every cell written by ST_1 has been overwritten by ST_2 (by definition ST_2 is finished at time t, so it has definitely written in all N cells of SB).

So, the values in SB at time t are not due to dominated write statements. □

Definition 5.4.3 *At time t, a composite statement with starting time t_S and finishing time t_E is* **undominated**, *iff it is not dominated and $t_S < t_E < t$.*

Lemma 5.4.2 *At any time instant t there are at most K undominated write operations.*

Proof: Assume that there are more than K undominated *write* statements. Then there is at least one processor that has more than one undominated *write* statements. This is a contradiction because the later *write* statement definitely dominates the earlier one. □

Definition 5.4.4 *At time t, a composite statement with starting time t_S and finishing time t_E is* **progressing**, *iff $t_S < t < t_E$.*

Given an execution diagram for K scripts and a time t_R in it, we denote with A_i the *progressing write* statement at t_R on P_i, if any, and with B_i the *undominated write* at time t_R on P_i, if any. We denote with C_j the *write* statement that starts j-th in order after t_R.

Lemma 5.4.3 *Given an execution diagram for K scripts with only one read statement starting at t_R and any subset S of the values written by A_i's, B_i's and C_j's (as long as its cardinality is less or equal to N), we can construct an interleaving that makes the read return exactly the values in S. The only requirement for this subset is that:*

- *it includes the value written by the A_i or B_i that started last before t_R (this value definitely resides in $SB[0]$ at time t_R);*
- *it includes the value of the A_i, B_i, or C_j that finished last before the read finished (this will definitely reside in $SB[N-1]$, when the read finishes, i.e. reads $SB[N-1]$).*

Proof: Assume that $S = S_1 \bigcup S_2 \bigcup S_3$, where $S_1 = \{x \mid x \text{ is written by some } A_i\} = \{a_0, a_1, \ldots, a_{|S_1|-1}\}$, $S_2 = \{x \mid x \text{ is written by some } B_i\} = \{b_0, b_1, \ldots, b_{|S_2|-1}\}$, $S_3 = \{x \mid x \text{ is written by some } C_j\} = \{c_0, c_1, \ldots, c_{|S_3|-1}\}$.

We define as t_f the earliest time at which a B_i finishes execution. We know that all B_i start execution before t_f, because if a statement B_j starts execution after t_f, then the statement that finishes execution at t_f is dominated by B_j.

So, we construct the interleaving as follows:
before t_f:

- All B_i's write $SB[0]$ in time order;
- All B_i's write $SB[1] - SB[|\ S_1\ | - 1]$ in any order;
- All A_i's starting before t_f write only $SB[0]$;

- All B_i's write $SB[|\ S_1\ |]$ in any order such that the processor writing b_0 goes last;
- ...
- All B_i's write $SB[|\ S_1\ | + |\ S_2\ | - 1]$ in any order such that the processor writing $b_{|S_2|-1}$ goes last.

At t_f:

- the earliest finishing B_i finishes.

After t_f, before t_R:

- All remaining B_i's finish; new A_i's start at appropriate relative times to the finishing B_i's and write only in $SB[0]$;

After all A_i's are started, we sort all A_i's which write values that are members of S_1 by increasing starting time; assume that the final order is $A_{m_1}, \ldots, A_{m_{|S_1|}}$. We know at this time that $A_{m_{|S_1|}}$ has written last in $SB[0]$, so its value is in $SB[0]$ ($A_{m_{|S_1|}}$ is a member of S by the definition of S).

- A_{m_1} writes $SB[1] - SB[|\ S_1\ | - 1]$;
- A_{m_2} writes $SB[1] - SB[|\ S_1\ | - 2]$;
- ...
- $A_{m_{(|S_1|-1)}}$ writes $SB[1]$.

At time t_R:

We know that all cells $SB[0] - SB[|\ S_1\ | - 1]$ contain the values in S_1, while $SB[|\ S_1\ |] - SB[|\ S_1\ | + |\ S_2\ | - 1]$ contains the values in S_2;

- Statement R reads $SB[0] - SB[\mid S_1 \mid + \mid S_2 \mid -1]$.

Assume that the C_j's which write values in S_3 are in increasing starting time: $C_{m_1}, C_{m_2}, \ldots, C_{m_{|S_3|}}$. From then on:

- loop while a new *write* statement C_j starts or finishes {
  - if C_{m_l}, $1 \leq l \leq \mid S_3 \mid$, starts, then C_{m_l} writes up to $SB[\mid S_1 \mid + \mid S_2 \mid + l - 1]$ and statement R reads $SB[\mid S_1 \mid + \mid S_2 \mid + l - 1]$;
  - if C_j, $j \neq m_l$ with $1 \leq l \leq \mid S_3 \mid$, starts, then C_j writes $SB[0]$;
  - if a C_j or A_i finishes, then C_j or A_i writes up to $SB[N-1]$;

  }

- Finish reading up to $SB[N-1]$ (statement R will read the symbol wrtitten by the last finished *write* statement in $SB[\mid S_1 \mid + \mid S_2 \mid + \mid S_3 \mid] - SB[N-1]$.

So, the *read* statement can read any values due to A_i's, any values due to B_i's and any values written by C_j's (as long as there are at most N values). □

Theorem 5.4.1 *Given an execution diagram for K scripts with only one read statement that starts operating at time t_R, the read can read the values of all undominated write statements at time t_R, all the values of the progressing write statements at time t_R and the values of all the write statements that start execution after t_R, provided that the number of such values is at most N.*

Although a reading statement can read any subset of the values mentioned in the above lemma, there is a coordination among multiple reading statements in many cases. Refering to Figure 5.2, for example, if another *read* is executed after time $t_{W_{e,1,b}}$, then it is will read from SB the exact same values read by the statement starting at time $t_{WR_{s,1}}$. We have not yet identified how reading statements coordinate. This is still an open problem, whose solution will lead to the solution of the general problem stated at the end of the previous subsection.

A solution to that problem will show us how to efficiently build a small sized automaton that simulates the execution of the scripts in a test.

Clearly, we can simulate the execution of K scripts by constructing an automaton where every state is:
$Q_m = \langle (S_0,\ I_0),\ (S_1,\ I_1),\ \ldots,\ (S_{K-1}, I_{K-1}),\ RB_0,\ \ldots,\ RB_{K-1},\ SB_m \rangle$, where S_i, $0 \leq\ i \leq\ (K-1)$, is the statement of script L_i currently under execution on processor P_i, I_i indicates that S_i operated last on $SB[I_i]$ ($SB[I_i + 1]$ should be next), RB_i is the set of values read by statement S_i up to now (if the statement is a *read*; $RB_i = \emptyset$ otherwise), and SB_m is the pattern SB exhibits.

A transition in this automaton occurs whenever an atomic operation is executed.

The automaton simulates the execution of the K scripts by keeping track of the changes each atomic operation causes in SB and the sets each reading processor accumulates and returns. The size of this automaton though is exponential in N, the size of SB. So, with such an automaton it is expensive to simulate the execution of the scripts and verify the test's freedom of

- the statement under execution in each script;
- the values returned by any finishing *wait* (or *select*) statement of each script,

we see that the necessary conditions for deadlock, starvation and timing dependencies are:

1. *Deadlock:* in a state Q in the simulating automaton all the statements in execution by the scripts are *wait* statements and the values they read from SB do not satisfy them;
2. *Starvation:* either
   (a) there is a cycle in the automaton, where all the states Q_i, $i_1 \leq i \leq i_2$ have at least one script executing a *wait* statement that is never satisfied, or
   (b) a state Q_i with no outgoing edges of the simulating automaton has some (not all) the scripts execute *wait* statements that are never satisfied by the contents of SB;
3. *Timing Dependency:* there is at least one state Q_i in the automaton with more than one statements simultaneously writing in SB different values.

Assume that the automaton $SIM(L_1, \ldots, L_K)$ simulates the execution of the scripts $L_1, L_2, \ldots, L_K$.

If the scripts deadlock, then there is an execution sequence which will bring all the scripts to an indefinite blocking. A script blocks only if it

deadlocks, starvation and timing dependencies.

The problem of reducing the size of the automaton to obtain one with a number of states not exponential in N is still open. A solution to the problem stated at the end of the previous subsection will allow us to build such a small automaton.

5.5 Correctness

In the following discussion we use the PLADO model again, where we have the *wait* and *select* statements. The use of PLADO allows a better understanding of the verification process.

The following characteristics of test programs are undesirable due to the deterministic nature of the testing process:

1. *Deadlock:* the case where all processors that execute a script block indefinitely;

2. *Starvation:* the case where at least one processor executing a script blocks indefinitely, while at least one terminates or continues execution;

3. *Timing Dependency:* where execution of the scripts is affected by the relative speeds of the processors, or the contents of the buffer SB depend on the relative speeds of the processors.

Using the simulating automaton described above (or a smaller one if it is proven to exist and can be efficiently constructed), which provides us with the following information at every state:

executes a *wait* statement that is not satisfied. So, for a deadlock all scripts have to execute a *wait* statement that is not satisfied. This occurs if there are more than one values in SB (each *wait* statement waits on only one value) or if the value in SB is not the one that will eventually unblock any of the *waiting* processors. Thus, there exists a state in the simulating automaton that has all scripts executing these *wait* statements and either SB has more than one values or it has a value that does not satisfy any *wait* statement.

So, for a deadlock to appear it is necessary that the mentioned condition is satisfied. The condition is not sufficient, because there exist cases where timing dependencies allow the programs to proceed to a state different than the one which presents the deadlock. For example, a deadlock will appear in the two processor case:

P_0 executes: *write_byte x; wait_byte y;*

P_1 executes: *write_byte y; wait_byte x;*

if both processors execute their *write* statements simultaneously in such a fashion that SB results in an arbitrary interleaving of x's and y's. It is possible though, due to timing dependencies, for SB to result in having only x's in it, thus avoiding the deadlock.

In a similar fashion, we see that starvation occurs if one (or more) scripts block indefinitely, while others finish execution (or continue execution indefinitely due to a loop statement). As mentioned above, a script blocks only due to a *wait* statement and this implies that if starvation occurs then the starving scripts are waiting for some value in SB that either never fills SB or never appears in SB. So, the simulating automaton will have states that satisfy one of the mentioned conditions.

By definition, a timing dependency is the case where the scripts execute the same sequence of PLADO statements and result in different contents in SB. This is due to different possible interleavings of the atomic operations of *write* statements.

Detecting possible deadlocks, starvation and timing dependencies involves identifying states in the simulating automaton that satisfy the above mentioned conditions. Simple extensions of depth first search algorithms allow the detection of such states. If we succeed in obtaining a simulating automaton with small size (i.e. not exponential in N), then we can efficiently identify the above mentioned flaws in the scripts of a test.

If we include the flags in the above analysis, we have to multiply all the complexities by 2^M, where M is the number of flags, because we need to know at every state the exact pattern of the binary flags.

Chapter 6

Performance and Applications

The prototype offers high performance when compared to conventional scalable interconnections. It can be easily programmed and improves the running time of parallel and distributed applications.

6.1 System Performance

The performance characteristics of the prototype switch are:

1. *Aggregate Bandwidth*: 192 MBits/sec;
2. *Effective Bandwidth*: 64 MBits/sec;
3. *Effective Data Bandwidth*: 32 MBits/sec.

The *aggregate bandwidth* shows the available bandwidth of the communication circuitry on a single switch but it is not possible to achieve effective use of the whole available bandwidth, because of the service rates the incoming

Figure 6.1: A K-Switch Network Diameter

messages experience within the switch due to broadcasting. The total number of different messages serviced by the switch in a time interval decreases the bandwidth to 64 MBits/sec. These bits serviced in a second are message bits, i.e. address and data bits of the PRAM memory updates. Since half of the message contains useful data, the effective data bandwidth is even less: 32 MBits/sec. With the memory management scheme described in Section 4.3, the switch can have an increased performance up to 66.6 MBits/sec of effective data bandwidth (or 133.2 MBits/sec effective bandwidth); this performance is obtained by saturating the internal bus of the switch.

In a network such as the one presented in Section 4.3 the important parameter in the system's delays is the size of the network's diameter. If a PRAM network has k switches in its diameter, the delay of transmitting a byte over the network is between: $(k+1)T_{tr}+\sum_{i=0}^{k+1} d_i$ over an idle network (d_i is the delay of the link between switches $(i-1)$ and i in the diameter shown in Figure 6.1) and $3^k T_{tr} + \sum_{i=0}^{k+1} d_i$ over a network where all systems transmit simultaneously. T_{tr} above is the rate at which switch ports transmit (receive) whole messages (4 bytes). When all systems transmit simultaneously and we measure the delay of a message M, the i-th switch in the diameter has to transmit 3^i messages in the worst case, to include the message M. It

should be made clear to the reader that although the delay in a heavily loaded network is exponential to the size of the network's diameter, it is not exponential to the number of processors in a system, since a configuration with diameter k accommodates $4 * 3^{(k-1)/2}$ processors, if k is odd, or $2 * 3^{k/2}$ processors if k is even.

Another important parameter of a network is its bit error rate. The error rate provided by the optical circuitry used on the prototype is 10^{-15} per link. All efforts to measure the bit error rate in the prototype network have been unsuccessful. The reason is that the fiber interface drivers operate under their potential: e.g., they support link lengths up to 1 Km, but the prototype links are much shorter. Experiments transferring Terabytes of data have been successful without a single error. So, there is no reason to doubt the claimed BER (Bit Error Rate), which is 10^{-15} per link. The error rate of a larger network is a function of the number of used links.

6.2 Experiments and Applications

The following subsections present the performance measures of a few experiments with the prototype switch. All the experiments performed are with a network of 1 switch and 4 IBM ATs, as shown in Figure 6.2. For such a network there are only 4 possible loads on the switch, since only 4, 3, 2 or 1 processors can be simultaneously broadcasting. So, the performance of the network in the experiments can have only four different values. Plots are provided wherever reasonable, while the lowest performance measures are

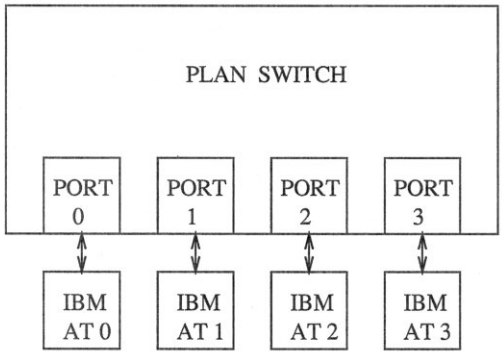

Figure 6.2: Experiment Configuration

presented when the performance measures are comparable under all 4 possible loads. The experiments include applications such as the ones analyzed in Chapter 2.

6.2.1 PRAM vs. Intel's iPSC/2

The PRAM prototype provides better performance than conventional local area networks, since its delays are comparable with memory access latencies: the delays of the prototype are in the order of *microseconds*, while for conventional LANs they are in the order of *milliseconds*. The prototype also provides higher performance than some available multiprocessors; in this subsection I compare the prototype with Intel's iPSC/2 hypercube [Arl88].

Figure 6.3 shows the end-to-end delay of variable size messages on the PRAM system configuration shown in Figure 6.2 and a 4-node Intel iPSC/2 hypercube. The delays measured on the PRAM interconnection are for local memory to local memory transfers, i.e. the data are copied from local

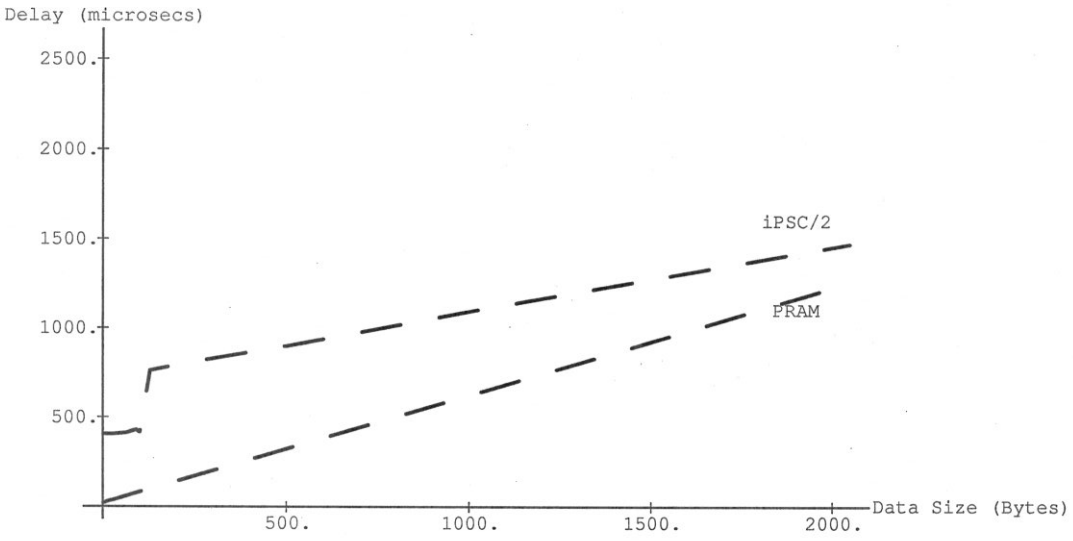

Figure 6.3: PRAM vs. iPSC/2

memory to PRAM at the transmitter and from PRAM to local memory at the receiver. As the figure shows, the PRAM prototype provides better performance for all message sizes up to 2 KBytes. All experiments up to 2 Megabyte messages have proven PRAM faster than the iPSC/2. Experiments with longer messages were not possible due to memory limitations of the used systems.

The iPSC/2 is slower than PRAM mainly because of its *setup delay*; such a delay does not exist on the PRAM system. Figure 6.3 agrees with the analysis of *uniform-cost* and *non-uniform-cost networks* presented in Chapter 2. As Figure 6.3 indicates, there must exist a message size where the iPSC/2 will provide better performance than the PRAM network.

The PRAM performance is measured when only one of the 4 intercon-

nected systems broadcasts. When more systems broadcast the performance of the network degrades but the network with the memory management scheme will provide performance comparable to the one shown in Figure 6.3 for all processor-to-processor message exchanges independent of the number of simultaneously communicating processors (i.e. the load on the network switch) as long as there is no multicasting to more than 1 processors.

6.2.2 Reliable Broadcasting

The system configuration in this experiment is the one shown in Figure 6.2. The processors connected are IBM ATs.

The application is reliable broadcasting augmented with ordering: every system that receives the broadcast messages should order them in exactly the same order as the rest of the systems [CM84] [MSM89]. The solution to the ordering problem is in the spirit of the solution by Chang and Maxemchuck [CM84], but instead of having a token site, where all the broadcast messages are transmitted to, I use the shared memory interface to request a unique timestamp from a controller (similar to the token cite). The timestamp is appended to the message which is then broadcast. Every processor receiving a broadcast message acknowledges it.

So, the solution to the problem can be divided in 3 stages:

- Get unique timestamp from the controller;
- Broadcast message;
- Wait for acknowledgments from message receivers.

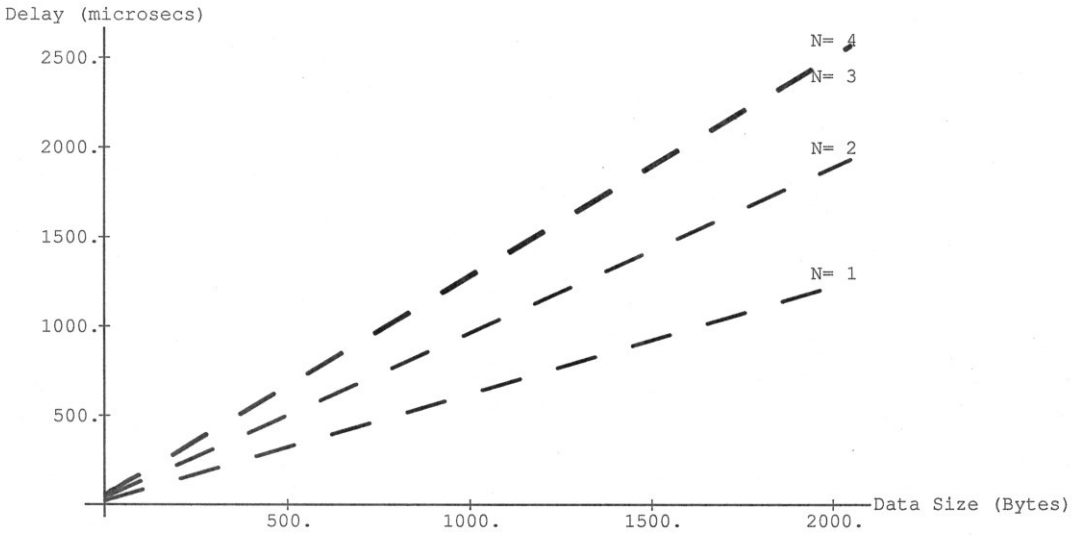

Figure 6.4: Latencies of Broadcast Messages

In the system configuration shown in Figure 6.2 getting a unique timestamp costs less than 200 $\mu secs$, under all possible loads assuming no failures.

Figure 6.4 shows a plot of the delays to transmit K data bytes, $1 \leq K \leq 2\ KBytes$, when N processors, $1 \leq N \leq 4$, broadcast simultaneously. Since the acknowledgements are very short messages, usually they do not coincide on the switch because of marginal differences in the relative speeds of the interconnected processors and the exact transmission timing. All the measured acknowledgments are less than 50 $\mu secs$ (since all acknowledgements progress in parallel, the transmitter receives **all** acknowledgements within 50 $\mu secs$). This agrees with the calculations which show that the longest acknowledgment costs less than $(20 * 3^{k-1} - 8) * (2/3) + 40\ \mu secs$ in a network with k switches in its diameter.

6.2.3 Remote Clock Reading

In the same system configuration as above, when a processor reads the clock of a remote processor (as described in [Cri89]), the round trip delay exhibited is less than 90 $\mu secs$ under all possible loads. So, using the formula: $e_{max} = D(1+2\rho) - min$ with $\rho = 6 * 10^{-6}$ (I assume that ρ has the value Cristian mentions in [Cri89]) and $min = 25$ $\mu secs$, we see that the maximum error of the worst case (assuming no network failures) is approximately 20 $\mu secs$; this compares well to the maximum error achieved with other systems (using the data provided by Cristian in [Cri89] my calculations show that the median round trip delay in the 2-processor configuration described there has maximum error more than 100 $\mu secs$).

6.2.4 Real-Time Audio Data Transfers

The configuration shown in Figure 6.2 was used for an experiment for real-time CD quality audio data transfers. The experiment was mainly performed by Ted Kyi, Ted Altman and Lou Pokrocos.

CD quality audio data were stored on the hard disk of an IBM AT and transfered through the interconnection to a remote AT, which in turn moved the data into a Digital-to-Analog converter that played it in real-time on a speaker. The effective data rate to play CD quality audio data in real-time is approximately 1.41 $MBits/sec$.

The PRAM network can sustain this rate between the two communicating systems even when the other two connected IBM ATs produce heavy load for the interconnection. This compares favorably to many conventional networks

of IBM ATs, where the effective data bandwidth is less than 2.5 $MBits/sec$ (e.g., see [Nov86]). This implies that such conventional interconnections are able to play CD quality audio data in real-time if there is no other load on the network; as soon as another server heavily uses the network though, the audio data cannot be played in real-time anymore, because there is not enough bandwidth available for the transfers.

6.2.5 Remote Procedure Calls

Remote Procedure Calls (RPCs) provide the basis for many distributed applications. Elaborate PRAMBIOS, developed by researcher J. S. Sandberg, provide the basis for many network applications, including RPCs, using the PRAM prototype presented in this dissertation.

The idea behind the RPCs for the PRAM network is to setup a mechanism where:

- procedure parameters are passed to a remote server through PRAM memory;
- an "opcode" is passed, which specifies the procedure to be executed.

The "opcode" can be efficiently passed by writing into one of the lowest 4 PRAM memory bytes which are memory mapped interrupts as described before. The server receiving the request for the RPC looks-up a table containing all the "services" it provides, it finds which procedure has to be executed and executes it, returning the data through the PRAM memory.

For the configuration shown in Figure 6.2, a null RPC requires approximately 50 $\mu secs$ from initiation to the return of null results. With Jon Sandberg's PRAMBIOS, the configuration achieves almost 5000 null RPCs/sec. This compares favorably with the $500 - 600$ null RPCs most available systems achieve. One should take into account here that the measurements were made on systems without multiprogramming (our IBM ATs) and that the RPCs are intended for an autonomous PRAM network (so, issues such as Internet routing, etc., do not rise).

Chapter 7

Conclusions

This dissertation presented an architecture for building scalable, heterogeneous, high-speed interconnections at the memory level. The architecture is based on the PRAM shared memory model [LS88]. The prototype built proves the feasibility of PRAM-based systems which enjoy high-performance interprocessor communication. Applications developed for the prototype show that the architecture not only provides efficient communication but a simple, easy programming model.

7.1 Research Results

The PRAM shared memory model [LS88] provides a new organization for multiprocessor architectures. It supports high-bandwidth/low latency communication among heterogeneous, geographically separated processors and can be used as the basis to build any kind of interconnection, from tightly-

coupled MIMDs to long-haul networks [LS88]. The implemented architecture interconnects heterogeneous, autonomous machines such as IBM ATs, SUN-3s and MAC-IIs. The prototype achieves higher communication performance than many conventional interconnections proving that the PRAM model is a good candidate for scalable, high-performance multiprocessor architectures.

A major advantage of the architecture is its programming paradigm, which makes software development easy. The easy programming model as well as the high-performance communication allows one to quickly and easily develop demanding applications such as real-time CD quality audio/video data transfers, high-speed remote procedure calls, etc.

A by-product of the memory interface and PRAM's operation is the very low latency short messages experience in the system. This characteristic is new to distributed systems which are commonly characterized by long delays for small messages. So, PRAM allows simple approaches to the solution of problems that require exchange of short messages and/or provides higher performance. Such problems include the presented problems of reliable broadcasting and reading remote clocks for clock synchronization.

PRAM's disadvantage is incoherence, but consistency can be easily enforced in software [LS88].

Although the development of the architecture and the applications presented in this dissertation have answered many questions about the feasibility and the performance of PRAM-based systems, there are still many open and challenging questions regarding PRAM, because it is a model which has not received attention until recently. Some of the open questions which will definitely trigger much research in the near future are presented in the next,

final section of this dissertation.

7.2 Open Problems and Future Research

Open problems for PRAM systems range from theoretical questions, such as development of suitable error detection/correction codes, to architectural problems, such as the design of large scale, supercomputer class machines.

It has become clear that within the next few years fiber communication technology will provide chips that achieve transfer rates in the order of gigabits per second at a relatively low cost. Suitable network architectures are necessary to effectively use all the available bandwidth. In a PRAM based network one has to pay special attention to the design of the switches of the interconnection as well as to the protocols used for the network management. The bus-based architecture of the prototyped switches coupled with the hardware *Stop&Wait* protocol and the error handling mechanism solve efficiently the interconnection problems of the autonomous systems of the prototype, but it is necessary to use higher performance designs to take advantage of the new high-speed links.

Since PRAM supports scalability, it is a good candidate for the memory organization of supercomputer class machines. An architecture for such a machine has been developed by Wittie and Maples [WM89]. There are many issues which have to be studied in such an architecture. One of the most important questions is the solution to the problem of coherence. Since consistency is enforced in software, it is necessary to compare various locking schemes as well as to evaluate the size of the memory to lock (locking

small pieces of memory arbitrarily is probably inefficient), so that maximal efficiency is achieved for important applications (these issues have been addressed by Wittie and Maples [WM89]). Also, one needs to develop and evaluate protocols which enhance the error detection/correction abilities of the interconnection; the CRC registers of the prototype do not provide enough support for efficient error protocols, when multiple processors are connected.

Developing PRAM-based MIMDs presents yet another challenge: design of suitable caches, because PRAM organizations require special caches, consistent with PRAM's operation [LS88].

Fault-tolerance is another open question in PRAM. As shown in Chapter 4, cycles are not allowed in the prototype broadcasting network. When multicasting is employed, cycles can exist in this network's topology providing a degree of fault-tolerance. It is clear that for highly reliable systems one needs to develop special architectures. Such architectures may differ depending on the fault-tolerance requirements of the target system.

The list of problems just presented is by no means complete. PRAM is a memory model which has not been used in the architectures of conventional systems. Development of such systems will definitely bring up many more problems which deserve research. Clearly, PRAM brings a new, exciting alternative to conventional memory organizations and networking approaches and thus merits further study.

Appendix A

Verification of the Hardware Protocols

```
%************************************************************************
constants
        N= 5,   /* number of clicks to timeout */
        M= 10,  /* capacity of the input queue */
        K= 4    /* number of messages that have to be consumed once
                   overflow is reached to go back to normal state */

typedef process SR (SR partner; QUEUE q; TIMER tx)

states     0..3 valnm [ready: 0, run: 1, stop: 2, error: 3]
selections 0..3 valnm [sstart: 0, reg: 1, sstop: 2, no: 3]

init run

trans
```

```
        ready               {sstart}
                >run        :~(q:errtrigger);
                >error      :otherwise;

        run                 {reg}
                >stop       :(partner:sstop)&~(q:errtrigger)&~(q:full);
                >error      :(q:errtrigger)|(q:full);
                >run        :otherwise;

        stop                {no}
                >run        :(partner:sstart)&~(q:errtrigger)&~(q:full);
                >error      :(q:errtrigger)|(q:full);
                >stop       :otherwise;

        error               {sstop}
                >ready      :(tx:up)|(q:nfull);
                >error      :otherwise;
end

typedef process TIMER (QUEUE q)

states     0..N valnm [idle: 0]
selections 0..1 valnm [go: 0, up: 1]

init idle

trans
        idle                {go}
                >1          :(q:errtrigger);
                >idle       :otherwise;
```

```
        N                   {up}
                >idle       :(TIMER:up);

        $                   {go}
                >$+1        :(TIMER:go);
end

typedef process QUEUE (SR partner; SR own)

/* states 0 to M indicate the number of messages in the input queue */
/* state  M+1 is used to indicate overflow                          */
/* states M+2 to M+K+1 represent M-1 to M-K messages in the queue   */
/* when the queue gets to M+K+1, the reception is reasummed         */
/* state M+K+2 means the queue is closed, an error has been found   */
/* and the messages there are to be discarded                       */

states     0..M+K+2 valnm [normal: M+K+1, stopped: M+K+2]
selections 0..4     valnm [consume: 1, errtrigger: 2, full: 3, nfull: 4]

init 0

trans
        0                   {0}
                >1          :(partner:reg);
                >0          :otherwise;

        M                   {full}
                >M+1        :(QUEUE:full);
```

```
        normal              {nfull}
                >M-K        :(QUEUE:nfull);

        stopped             {0}
                >0          :(own:sstart);
                >stopped    :otherwise;

        $                   {0..2}
                >$+1        :((partner:reg)&(QUEUE:0)&($ < M+1))|
                            (($ > M)&(QUEUE:consume));
                >$-1        :~(partner:reg)&(QUEUE:consume)&($ < M+1);
                >stopped    :(QUEUE:errtrigger);
                >$          :otherwise;
end

process (i= 0..1; port[i]: SR (port[(i+1)%2], queue[i], timer[i]))

process (i= 0..1; queue[i]: QUEUE (port[(i+1)%2], port[i]))

process (i= 0..1; timer[i]: TIMER (queue[i]))
%*****************************************************************************
```

Bibliography

[ABM88] S. Aggarwal, D. Barbara, and K. Z. Meth. A Software Environment for the Specification and Analysis of Problems of Coordination and Concurrency. *IEEE Transactions on Software Engineering*, TSE-14(3):280–290, March 1988.

[AI83] Arvind and R. A. Ianucci. A Critique of Multiprocessing von Neumann Style. In *10th Annual International Symposium on Computer Architecture Conference Proceedings*, pages 426–436, 1983.

[Arl88] Ramune Arlauskas. iPSC/2 System: A Second Generation Hypercube. In *Concurrent Supercomputing, the Second Generation*, pages 9–13. Intel Corporation, 1988.

[Arn89] Arnould E. A., et al. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1989.

[AS88] W. C. Athas and C. L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, 21(9):9–24, August 1988.

[BF76] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press (Potomac, Md.), 1976.

[CAS86] F. Cristian, H. Aghili, and R. Strong. Clock Synchronization in the Presence of Omission and Performance Faults, and Processor Joins. In *Digest of Papers, 16th Annual International Symposium on Fault Tolerant Computing*, pages 218–223, 1986.

[CGBG88] D. R. Cheriton, A. Gupta, P. Boyle, and H. A. Goosen. The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation. In *15th Annual International Symposium on Computer Architecture Conference Proceedings*, pages 410–421, 1988.

[CM84] J. Chang and N. F. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Cri89] F. Cristian. A Probabilistic Approach to Distributed Clock Synchronization. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 288–296, 1989.

[EK88] S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluations. In *15th Annual International Symposium on Computer Architecture Conference Proceedings*, pages 373–383, 1988.

[FD86] M. A. Franklin and S. Dhar. Interconnection Networks: Physical Design and Performance Analysis. *Journal of Parallel and Distributed Computing*, 3(3):352–372, September 1986.

[Got83] Gottlieb A., et al. The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[Hil86] Hill M. D., et al. Design Decisions in SPUR. *IEEE Computer*, 19(11):8–22, November 1986.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications ACM*, 12(10):576–583, October 1969.

[IEE85] IEEE, editor. *802.5: Token Ring Access Method.* IEEE, New York, 1985.

[Lal85] P. K. Lala. *Fault Tolerant and Fault Testable Hardware Design.* Prentice Hall, 1985.

[Lam77] L. Lamport. Proving the Correctness of Multiprocessor Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[LS88] R. J. Lipton and J. S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[LS90] R. J. Lipton and D. N. Serpanos. Uniform-Cost Communication in Scalable Multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[MB76] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the Association for Computing Machinery*, 20(7):395–404, July 1976.

[Moc83] P. V. Mockapetris. Analysis of Reliable Multicast Algorithms for Local Networks. In *Proceedings of the Eighth Data Communications Symposium*, pages 150–157, 1983.

[MSM89] P. M. Melliar-Smith and L. E. Moser. Fault-Tolerant Distributed Systems Based on Broadcast Communication. In *9th Int. Conference on Distributed Computing Systems*, pages 129–134, 1989.

[Nov86] Novell Inc. *LAN Evaluation Report.* IEEE, New York, 1986.

[NR82] S. Nanda and S. M. Reddy. Design of Easily Testable Microprocessors – A Case Study. In *Proceedings of the 1982 IEEE Test Conference*, pages 480–483, 1982.

[Nug88] Steven F. Nugent. The iPSC/2 Direct-Connect Communications Technology. In *Concurrent Supercomputing, the Second Generation*, pages 59–68. Intel Corporation, 1988.

[Pfi85] Pfister G. F., et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[PV81] F. P. Preparata and J. Vuillemin. The Cube Connected Cycles: A Versatile Network for Parallel Computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[Ros86] F. E. Ross. FDDI-A Tutorial. *IEEE Communications Magazine*, 24(5):10–15, May 1986.

[Sch90] Schroeder et al. Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. Technical Report Report-59, DEC-SRC, April 1990.

[Sei85] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[SH81] T. Sridhar and J. P. Hayes. A Functional Approach to Testing Bit-Sliced Microprocessors. *IEEE Transactions on Computers*, C-30(8):563–571, August 1981.

[Sun81] Carl A. Sunshine, editor. *Communication Protocol Modeling.* Artech House, Inc., 1981.

[TBF83] F. A. Tobagi, F. Borgonovo, and L. Fratta. Expressnet: A High-Performance Integrated-Services Local Area Network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5):898–913, November 1983.

[TT77] M. Tokoro and K. Tamaru. Acknowledging Ethernet. In *Digest of Papers, COMPCON 77 Fall*, pages 320–325, 1977.

[VS86] A. Vergis and K. Steiglitz. Testability Conditions for Bilateral Arrays of Combinational Cells. *IEEE Transactions on Computers*, C-35(1):13–22, January 1986.

[WG83] J. W. Wong and G. Gopal. Analysis of Reliable Broadcast in Local-Area Networks. In *Proceedings of the Eighth Data Communications Symposium*, pages 158–163, 1983.

[WM89] L. Wittie and C. Maples. MERLIN: Massively Parallel Heterogeneous Computing. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I.142–I.150, 1989.