HARDWARE BARRIER SYNCHRONIZATION FOR A CLUSTER OF PERSONAL COMPUTERS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Tariq Muhammad

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical Engineering

May 1995

To my parents.

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Hank Dietz, whose support and guidance enabled me to progress through the graduate studies. In him, I not only found a mentor, but also a friend who was there to motivate and help me in my research. He provided me with the opportunities to be active in various technical forums which were invaluable towards my understanding of the field of parallel processing. This acknowledgment would not be complete without the mention of Professor Joseph D. Eigel. He not only relieved me from the financial burden of going to a graduate school, but was also very understanding of my erratic work schedule.

I am also grateful to Tai Chung for his help and encouragement throughout my graduate studies. I would like to thank my room-mate Omer Farooq and my colleagues Zafar Ali, Farooq Hameed, Abbas Kazmi, Taimur Aslam, and Shahab Baqai for being there through my ups and downs.

DISCARD THIS PAGE

TABLE OF CONTENTS

	Page
LIST OF TABLES	vii
LIST OF FIGURES	viii
ABSTRACT	ix
1. INTRODUCTION	1
2. BACKGROUND	4
2.1 Barrier Synchronization	4
2.2 Classes of Barrier Synchronization	5
2.3 Partitionable Barrier Mechanism	6
2.4 Survey of Hardware Barrier Synchronization Schemes	7
2.4.1 FEM	7
2.4.2 FMP	7
2.4.3 Fuzzy Barrier	8
2.4.4 CM-5	8
2.4.5 Triton/1	9
2.4.6 OPSILA	10
2.4.7 OSCAR	10
2.4.8 Cray T3D	11
2.5 Parallel Computing Using Clusters of Workstations	11
3. BARRIER SYNCHRONIZATION RESEARCH AT PURDUE	14
3.1 PASM Parallel Machine	14
3.2 Barrier MIMD Architectures Introduced	16
3.3 Barrier MIMD Architectures Formalized	17
3.4 The CARP Machine	18
3.5 The CARD Project	20
3.5.1 CARD System's Overview	20
3.5.2 The CARD System's Host Computer	21
3.5.3 CARDBoard	23

Page

3.5.4 CARD's Data Communication/Synchronization Card 24
3.5.5 Evolution of Barrier Synchronization Mechanism in CARD System 25
3.6 Partitionable Dynamic Barrier Mechanism 30
3.6.1 Barrier Architecture 30
3.6.2 Dynamic Barrier Synchronization 32
3.6.3 Partitioning A Barrier Group 34
3.6.4 Enlarging A Barrier Group 36
3.6.5 Scalability 36

4. PURDUE'S ADAPTER FOR PARALLEL EXECUTION AND RAPID SYNCHRONIZATION (PAPERS) 39

4.1 Computer's Interface to PAPERS 40
4.1.1 Centronics Printer Port as an Interface to PAPERS 42
4.1.2 Software Control of Printer Port 43
4.1.3 Data Communication Network 44
4.2 Dynamic Barrier Mechanism in PAPERS 44
4.2.1 Background 45
4.2.2 Why a Memory Element in Barrier Logic of PAPERS? 46
4.2.3 Barrier/Anti-Barrier Sequence 48
4.3 Interrupt Mechanism in PAPERS 51
4.4 Block Diagram of PAPERS 55

5. PAPERS0 IMPLEMENTATION 57

5.1 Defining Input/Output Signals for PAPERS0 57
5.2 PE Interface to PAPERS0 58
5.2.1 Physical Connectors 58
5.2.2 PE Port Bit Assignment 62
5.3 PAPERS0 Hardware Overview 65
5.4 PAL for Barrier/Interrupt logic 65
5.4.1 Pinout 66
5.4.2 PALASM Code 66
5.4.3 Description 68
5.5 Fabrication 70
5.6 Schematics 72

6. RESULTS FROM PAPERS0 IMPLEMENTATION 74

6.1 Performance of PAPERS0 74

Appendix

Page

6.2 SIMD/MIMD/VLIW Code Execution on Cluster of Workstations using PAPERS 76
6.3 PAPERS as a Data Network 76
6.4 Scheduling Data Network Accesses using PAPERS 77
6.5 Minimizing Port Operations per Barrier 78
6.6 Need for Data Buffers in PAPERS 78
6.7 Fabrication Blues 79

7. CONCLUSION 81

LIST OF REFERENCES 83

DISCARD THIS PAGE

LIST OF TABLES

Table Page

5.1 DB25 Parallel Port Pin Assignments 60

5.2 Centronics Connector Contact Assignments 61

5.3 PortBase + 0, Bit Assignments 62

5.4 PortBase + 1, Bit Assignments 63

5.5 PortBase + 2, Bit Assignments 64

5.6 Meaning of U1 and U0 Signals 64

5.7 Meaning Of PEaGO and PEaINT Signals 70

5.8 Front Panel LEDs 71

6.1 Timing of Key Operations of PAPERS0 75

DISCARD THIS PAGE

LIST OF FIGURES

Figure		Page
2.1	Difference Between Static & Dynamic Barriers	5
3.1	Hardware Parallelism Constraints for SIMD -> MIMD	16
3.2	Basic CARP Node	19
3.3	A Typical CARD System	21
3.4	A CARD Host	22
3.5	Dynamic Barrier Mechanism for PEk	27
3.6	Problem in Recombining Subgroups	29
3.7	Partitionable Dynamic Barrier Mechanism for PEk	31
4.1	PAPERS	39
4.2	Barrier Logic of PAPERS	49
4.3	Interrupt Logic (Method 1) for PEk of PAPERS	54
4.4	Interrupt Logic (Method 2) for PEk of PAPERS	54
4.5	Block Diagram of PAPERS	55
5.1	PAL Pin Layout	66
5.2	PAPERS0 Schematic 1	72
5.3	PAPERS0 Schematic 2	73

ABSTRACT

Muhammad, Tariq. M.S.E.E., Purdue University, May 1995. Hardware Barrier Synchronization for a Cluster of Personal Computers.
Major Professor: Dietz, Henry J.

Personal computers offer execellent performance per unit cost, and a cluster of such machines might make a useful parallel computer. However, conventional networking methods cannot provide the low-latency barrier synchronization needed to coordinate interactions between fine-grain parallel processes distributed across a cluster. This thesis describes how simple custom hardware can be interfaced to a group of unmodified personal computers to provide an appropriate barrier mechanism. Both the interface and the hardware implementation are discussed, with an emphasis on how the notion of barrier synchronization has evolved from the first theoretical work to the lessons learned in experiments with the first prototype PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization).

1. INTRODUCTION

If clusters of personal computers and workstations are to function effectively as reasonably fine-grain parallel computers, conventional networking methods and hardware are not sufficient. Low-latency barrier synchronization and communication is needed to coordinate interactions between fine-grain parallel processes distributed across a cluster. The mechanism described in this thesis uses simple custom hardware, generically called PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization), to effectively integrate a group of unmodified personal computers as a single parallel machine.

Although the initial goal of the project was simply to test a new architecture for dynamic barrier synchronization, much was learned in the implementation and testing of the new mechanism, and the resulting system performance was unexpectedly good. Thus, this thesis focuses on what we initially wanted to accomplish, how we went from theoretical work on barrier mechanisms to the actual implementation of the new architecture, precisely what was implemented, and what we learned from this implementation. In summary, this thesis traces the jump from theory to practice in the creation of the first PAPERS prototype: PAPERS0.

Chapter 2 explains the basic principles of barrier synchronization, and provides a survey of previous work on barrier synchronization done outside Purdue. A discussion of methods currently used to execute parallel code on a cluster of workstations is also presented. In contrast, Chapter 3 presents a history of barrier synchronization research within Purdue, with emphasis on the CARP Machine and the CARD system. It also details the evolution of the dynamic barrier mechanism which forms the basis for PAPERS.

How the interface between the PAPERS unit and the personal computers was defined is described in Chapter 4. Part of this involves the choice of computer interface and part involves modifying the dynamic barrier synchronization mechanism to better fit that interface. For example, the concept of parallel interrupts arises out of the need to obtain an initial synchronization. The resulting design of PAPERS0, the first implementation of PAPERS concept, is given in Chapter 5.

The measured performance and lessons learned from experiments with PAPERS0 are reviewed in Chapter 6. Although it may seem strange to be discussing lessons learned from a prototype that is less than a year old, at this writing, our experiences with PAPERS0 have already led to the creation of five more types of PAPERS units. In addition to the theoretical insights, there have been a wide range of practical contributions, including the demonstration that fine-grain cluster computing really can work. Chapter 7 briefly summarizes the contributions and describes the new directions that we are currently pursuing.

Although this thesis uses few terms unfamiliar to those involved in parallel computing, the following definitions may aid other readers:

Compile-time : This refers to anything involving compiler recognition, analysis, and transformation of a program to generate executable code for some target machine.

Run-time: This refers to anything involving information generated as a program is executing.

MIMD: Multiple Instruction Stream, Multiple Data stream. A parallel computer architecture with multiple program counters, each following a separate instruction stream.

SIMD: Single Instruction stream. Multiple Data stream. A parallel computer architecture with a single program counter, but for which each instruction applies across multiple data items.

VLIW: Very Long Instruction Word. A computer architecture with a very long instruction word that can control multiple, heterogeneous operations in parallel during each clock cycle. Like a SIMD machine, a VLIW behaves as though it has a single program counter.

Barrier Stream: The sequence of barrier group specifications (e.g., barrier masks) that will be used by a parallel program as it executes.

2. BACKGROUND

2.1 Barrier Synchronization

A barrier is a point in program instructions where a processing element must wait until all other processing elements associated with the barrier have finished their part of the program instructions.

Barrier synchronization can be accomplished by either software or hardware. A *software barrier synchronization* usually involves message broadcast from each processor to all the other processors. On the other hand, a *hardware barrier synchronization* requires some kind of dedicated hardware which is interfaced to all the processing elements of a system.

A processor typically performs the following three steps upon reaching a barrier:

1. Marks itself as present at the barrier.

2. Waits for all other *participating* processors to arrive at the barrier.

3. After all participating processors have arrived at the barrier, it proceeds past the barrier.

In contrast, our barrier mechanism changes step [3] into:

3. After all participating processors have arrived at the barrier, and after small (bounded) delay to detect this condition, all participating processors *simultaneously* resume execution past the barrier.

This subtle difference is actually the enabling condition for the use of static timing analysis and compile-time code scheduling.

2.2 Classes of Barrier Synchronization

There are actually two separate classes of barrier mechanisms that can provide precise timing constraints: *static* [1] and *dynamic* [2].

The difference between these techniques involves how the hardware determines which barrier synchronization should be the next to fire. The static version assumes that there is a complete order for all barrier synchronizations, whereas the dynamic version allows barrier synchronizations to be specified as a partial order. Thus, a dynamic barrier mechanism allows barrier synchronizations involving disjoint portions of the machine to fire in any order.

To illustrate the difference between static and dynamic barriers, consider the simultaneous execution of two different programs on a four processor machine such that program A is executed by processors 0 and 1, and program B is executed by processors 2 and 3. Since these two programs are independent, each may contain any number of internal barrier synchronizations, as shown in Figure 2.1.

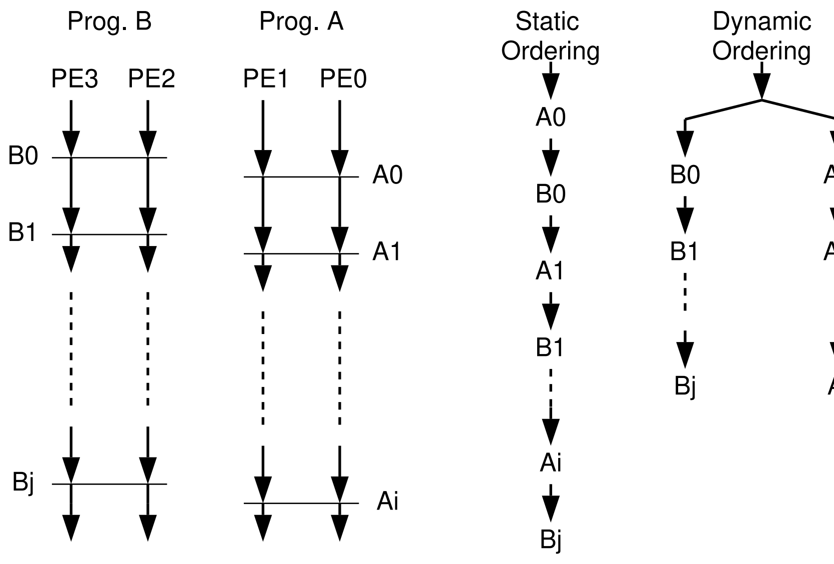

Figure 2.1 Difference Between Static & Dynamic Barriers

While either the static or dynamic barrier mechanism can be used, the static mechanism requires a complete ordering of the barriers, while the dynamic mechanism allows a partial ordering.

Thus, for the static mechanism, because barrier $A0$ is first in the static order, if processors 2 and 3 reach barrier $B0$ before processors 0 and 1 reach $A0$, barrier $B0$ will be 'blocked' from firing until after barrier $A0$ has fired. In contrast, the dynamic barrier mechanism allows the barriers within independent groups of processors to proceed without interfering with each other; no delays can be introduced by blocking.

In other words, a dynamic barrier mechanism allows simultaneous firing of multiple barrier streams, whereas a static barrier mechanism allows execution of only one barrier stream. Therefore, the static barrier mechanism requires merging of multiple barrier streams if multiple programs are executing on disjoint sets of processors. Thus, the dynamic mechanism is clearly superior to the static mechanism.

2.3 Partitionable Barrier Mechanism

If a barrier group (PEs involved in a barrier stream) can be broken in subgroups such that each subgroup can execute an independent barrier stream of its own, then that barrier mechanism is *partitionable.*

The dynamic barrier mechanism requires partitioning capability from the barrier synchronization logic. A barrier mechanism that uses *barrier masks* (bit vectors with one bit for each PE in the machine) to specify the PEs in a barrier group is partitionable. Partitioning can be classified as either *compile-time (static)* or *run-time (dynamic).*

Compile-time partitioning requires all the barrier masks to be computed by the compiler before the start of program execution and is therefore restricted by the amount of information that can be extracted by the compiler. Run-time partitioning allows the formation of sub-groups of PEs on the basis of the result of conditional constructs like `if-then-else`, `case statement`, and `do-while`. Therefore, new barrier masks are computed and enqueued during the program execution. A run-time partitionable dynamic barrier mechanism provides the maximum functionality to parallel code execution.

2.4 Survey of Hardware Barrier Synchronization Schemes

This section briefly describes some of the barrier mechanisms and the machines that implement barrier synchronization. Most of the following subsections are taken or modified from [3].

2.4.1 FEM

The term 'barrier synchronization' was first used in a paper by Harry Jordon [4]. This paper described the FEM (Finite Element Machine), a MIMD machine designed to efficiently manage problems with an SPMD structure such that all processors must complete one phase of the program before any can enter the next.

In contrast to the direct wire connections and logic tree used in the proposed barrier architecture, the FEM used serial 'priority chain' connections to transmit synchronization status information to and from all processors. This yields a simple implementation, but causes the propagation delay for synchronization to be proportional to the number of processors. Further, there was no method for partitioning the machine into multiple barrier groups.

2.4.2 FMP

The Burroughs FMP [5] was designed to be the Flow Model Processor in a system for performing aerodynamic simulations. Although it was never built, it is the first machine design to incorporate hardware barrier synchronization with timing properties and hardware structure similar to the barrier mechanism discussed in this paper.

The FMP's barriers are implemented using an AND tree that spans all 512 processing elements. When a PE executes a WAIT instruction, that instruction does not terminate until a GO signal is received. The GO signal is received by all PEs within

160ns after the last PE has begun to execute a WAIT instruction. Given that each PE has a peak performance of 3 MFLOPS, this synchronization cost is only about half the time taken to perform a floating point operation, hence, very fine grain.

The FMP's barrier tree can be partitioned by configuring AND gates at lower levels in the tree as root nodes for independent barrier groups. This partitioning supports multiple user programs sharing a single machine, but is insufficient to support the dynamic partitioning of the machine suggested in the descriptions of the SPMD worker and structured SIMD models.

2.4.3 Fuzzy Barrier

The hardware barrier synchronization scheme described by Gupta [6] is known as the *fuzzy barrier*. The 'fuzzy' part of the fuzzy barrier is basically a delayed barrier firing mechanism where the actual wait, if necessary, may occur several instructions after a processor indicates it has encountered a barrier. The concept is similar to the delayed branches in pipelined machines. The fuzzy barrier scheme uses *m-bit barrier tags* with each barrier along with the barrier signal ('I am at the barrier') to distinguish it from other barriers.

Several problems exist with the fuzzy barrier, hardware complexity is the most significant. Each processor has its own separate barrier processor, and has expensive matching hardware at each node to match the barrier tags. There are N^2 connections between the N processors, and each connection has m lines. The hardware connections required for a system restricts the fuzzy barrier implementation to systems with only small number of processors.

2.4.4 CM-5

The Thinking Machines CM-5 [7] is a commercial supercomputer that combines conventional processors in an architecture supporting both MIMD and SIMD execution. It has the distinction of being the first machine to implement SIMD execution without a traditional SIMD control unit broadcasting instructions.

In the simplest sense, a CM-5 consists of a hypertree network linking up to 16,384 computational nodes, host interfaces, or I/O connections. Each computational node in a CM-5 contains a standard Sparc processor, a custom network interface unit, and four custom vector arithmetic units. The Sparc peak floating point speed is only 5 MFLOPS, thus, the Sparc's primary purpose is to control the network interface and four custom vector arithmetic units yielding 32 MFLOPS each, or 128 MFLOPS per computational node. The Sparc also runs the node operating system.

A barrier synchronization is initiated by sending a control message noting arrival at a barrier. The synchronization is terminated by receiving a barrier-completed message. This decoupling is sometimes called a 'fuzzy barrier' mechanism [6]. Although the control network has the ability to be partitioned at configuration time, there is no support for partitioning barrier groups under program control. The result is a static barrier hardware structure resembling that of the FMP.

In that the Sparc is only a small portion of the computational node design, it can be argued that the CM-5 is not really based on a standard processor. However, the more significant issue is that the vector units within each node are very fast compared to the control network that is used to implement barriers. Thus, although the CM-5 implements both MIMD and SIMD execution, it does so with a grain size of 100s of instructions, quite different from the few instructions overhead implied by our mechanism.

2.4.5 Triton/1

The Triton/1 [8] is a 260 PE SIMD/MIMD machine closely resembling the PASM prototype. There are many differences, but the similarities are striking: PEs are based on MC68010 microprocessors, the interconnection network is very fast, and the mechanism for SIMD instruction broadcast is much like that in PASM. However, barrier synchronization is implemented using a 'global wired-OR' across the processors. Thus, Triton/1 supports only static barrier synchronization in which all processors participate.

2.4.6 OPSILA

Unlike the other machines described in this paper, OPSILA [9] employs PEs constructed using bit-slice processors (AMD 29116). Despite this difference, the methods used to support SIMD and SPMD are again very similar to those used in the PASM prototype. The most serious difference between OPSILA and PASM is that OPSILA's interconnection network is only operable in SIMD mode; in SPMD mode, PEs cannot communicate. Although SIMD execution is directly implemented in hardware, a barrier synchronization (referred to as a 'join' operation) is used to regain synchronization when switching from SPMD into SIMD execution mode.

2.4.7 OSCAR

OSCAR [10], the Optimally SCheduled Advanced multiprocessoR, differs from the other machines in that it is explicitly oriented toward compile-time static scheduling of MIMD code rather than toward implementing a simpler fine-grain execution model.

OSCAR is a shared-memory MIMD machine using 16 custom processing elements. Each of these PEs completes one operation per clock cycle, yielding 5 MFLOPS per PE. Because each operation takes a known amount of time, synchronization can only be lost by some PEs executing conditionals or loop iterations while other PEs take different paths. To support this type of asynchrony, there is a hardware barrier synchronization mechanism implemented using a control line on a bus. However, the machine is capable of being arbitrarily partitioned into two or three independent PE clusters, each with its own bus, bank of shared memory, and barrier synchronization hardware.

Although OSCAR's barrier mechanism appears to be dynamic, it is actually just three static mechanisms within a single machine. Further, making changes to a barrier group is not addressed in [10], and it appears that making any change would require multiple communication operations among the processors.

2.4.8 Cray T3D

The Cray T3D [11] is a shared-memory MIMD machine incorporating up to 2,048 nodes. Each node contains an Alpha processor and a large amount of custom support circuitry, including an interface to a barrier mechanism most closely resembling eight separate copies of the FMP's mechanism.

2.5 Parallel Computing Using Clusters of Workstations

In the last couple of years, parallel computing industry has seen a shift from Massively Parallel Computing(MPP) to loosely coupled parallel computing. This trend has blurred the distinction between the parallel and distributed computing.

A distributed memory MIMD parallel computer can be constructed by interconnecting standard single/multiple processor workstations through a conventional or high speed data communication network; such systems are referred to as *workstation cluster*. Workstation clusters have gained popularity with the availability of workstations that are capable of above 100 MFLOPS performance, and the falling cost of high speed data communication networks like Fast Ethernet, FDDI, HIPPI and ATM. Most of the current implementations of the workstation clusters use PVM (Parallel Virtual Machine) library routines under the UNIX operating system for parallel codes execution.

A number of studies have been undertaken on the clusters of workstations connected through ATM (Asynchronous Transfer Mode) network at various universities and research centers.

Results from a study of communication efficiency using various protocols over an ATM workstation cluster are presented by Megjou Lin et al. in [12]. The test set-up used 4 Sun workstations with a Fore Systems ASX100 ATM switch for ATM network, and also had a standard Ethernet network. Specifically this study addresses point-to-point communication latency atop ATM using protocols such as

1. Fore Systems ATM API over ATM AAL3/4

2. Fore Systems ATM API over ATM AAL5
3. BSD stream socket over TCP/IP over ATM AAL5
4. PVM Advise mode using Stream sockets over ATM AAL5 and
5. Sun RPC/XDR using Stream sockets over ATM AAL5.

It was shown that all the protocols have a high startup latency with a minimum of 869μs for ATM API AAL5 and the maximum of 2957μs for Sun RPC/XDR protocol. An interesting observation is that the network utilization of ATM for BSD stream is only 16% and the effective bandwidth (2.09 Mbyte/s) is only twice that of a 10 Mbits/s Ethernet.

A paper by Chengchang Huang et al. [13] deals with the software and hardware multicast operations on an ATM cluster. The cluster testbed used in experiments comprised 11 Sun SPARCstation-10 workstations running SunOS, three Fore Systems ASX100 switches, and a conventional Ethernet network. Software multicasting was implemented using a modified PVM, while AAL5 protocol was used to implement the multicasting in hardware. Again, the latencies for data communication are significantly high (1000μs for 1Kbyte block of data).

C. A. Thekkath has presented a model of network communication based on *remote memory access* to support multicomputing on ATM networks [14]. The testbed for this experiment used 4 DECstations 5000s with Fore systems ASX100 ATM switch. This report presents a network access model along with OS interface for implementing distributed shared memory system on an ATM workstation cluster. The report quotes a figure of 30μs for a 40 byte remote memory write operation and 45μs for a 40 byte remote read operation. The report claims to achieve these extremely good results using the Fore ATM host-network interface, but these numbers have not been duplicated elsewhere. In any case, the PAPERS mechanism yields markedly lower latency.

NAS (NASA AMES Research Center) has experimented with a number of clusters of workstation with both standard (Ethernet, FDDI) and proprietary (IBM Allnode switch) data communication networks [15] [16]. DCF (Distributed Computing Facility) Condor and LACE are the few of such clusters. All the implementations use some variation of the PVM parallel programming package. A comparative study of data communication latencies and cluster performance is presented in [15] [16].

In summary, there has been significant efforts towards parallel processing using clusters of workstation. With the availability of high performance workstations and high speed data communication networks, clusters seems to be the dominant mode of parallel processing in future.

3. BARRIER SYNCHRONIZATION RESEARCH AT PURDUE

Since 1987, hardware barrier synchronization has been a key element of parallel computer architecture research at Purdue. Starting with the PASM (PArtitionable Simd Mimd machine) prototype, which was actually the first parallel system built at Purdue to implement hardware barrier synchronization, all the subsequent theoretical designs including the CARP Machine and the CARD System have depended on a hardware barrier mechanism for fine-grain parallel code execution. The concept of Barrier MIMD architecture was proposed in 1987 by H. J. Dietz and T. Schwedersky. This idea led to the development of whole new class of MIMD architectures that was based upon hardware barrier synchronization. The Purdue notion of hardware barrier synchronization was formalized by Matthew O'keefe in his Ph.D. dissertation [17]. This classification, and the implementation architectures it suggested, remained the standard view of barrier hardware until October 1993. At that time, while working on the design of the CARD system, a new implementation architecture for the dynamic barrier mechanism was discovered. It was this new structure that triggered the creation of PAPERS, and inspired this thesis.

3.1 PASM Parallel Machine

The PASM prototype [18], a PArtitionable Simd Mimd machine designed and built at Purdue, has the distinction of being the first machine to implement both instruction-level SIMD and MIMD execution using conventional processors and special barrier synchronization hardware [3].

Although PASM's design is said to scale to 1,024 processors, the PASM prototype implements just 16 PEs, each of which is a standard Motorola MC68010 microprocessor. These 16 PEs are divided into four groups; each group has a separate control unit incorporating a Motorola MC68010 and custom hardware implementing a SIMD fetch unit, enable masking, and barrier synchronization. Thus, PASM can be partitioned into at most four barrier groups (or 32 groups for a 1,024 PE machine), with partitioning restrictions.

A PE invokes a barrier synchronization by making a read access to an address that is decoded as a barrier synchronization request; memory wait states are inserted to stall until synchronization has completed. Basically, the barrier hardware in the control units contains queues of both mask patterns and values to return. PASM's enable mask patterns are used to determine which processors participate in each barrier. The return values are typically ignored in conventional barrier synchronizations, but are the instruction sequence to broadcast in SIMD execution. If the barrier read is implemented by a LOAD operation, a barrier synchronization is performed; if the implementation is an instruction fetch, a barrier synchronization is performed and the next SIMD instruction is returned. Thus, changing between modes is simply a matter of MIMD mode executing a JUMP into barrier address space or of SIMD mode broadcasting a JUMP out of that space. Therefore, PASM prototype supports only static barrier synchronization.

The above implementation provides limited functionality in that only the static barrier ordering is permitted. However, the more severe limitation is that only the control units can enqueue barrier patterns and return values. Thus, PASM is very inefficient if mask patterns must be derived from the result of runtime evaluation of parallel expressions empirically, the most common case. Stated differently, PASM's barrier mechanism is a very powerful and efficient implementation of static barrier synchronization hardware, but PASM's barrier enqueue hardware is too centralized.

3.2 Barrier MIMD Architectures Introduced

Although PASM was the first prototype to implement hardware barrier synchronization at Purdue, the idea of new MIMD architectures based upon hardware barrier synchronization was presented for the first time by H. G. Dietz and T. Schwederski in 1987 [19] [20].

This research was the result of an effort to extend the PASM hardware to support a VLIW mode besides SIMD and MIMD modes. Although VLIW mode could not be supported on the PASM prototype, the PASM architecture was trivially modified to provide the support for barrier synchronization in MIMD mode. This formed the basis for defining new MIMD architectures, namely; Lock-Step MIMD (LSM), Static Barrier MIMD (SBM) and Dynamic Barrier MIMD (DBM). The Table 1 from [19], reproduced here in Figure 3.1 provides a comparison of different modes.

	SIMD	VLIW	Lock-Step MIMD (LSM)	Static Barrier MIMD (SBM)	Dynamic Barrier MIMD (DBM)	MIMD
Simultaneous Operations	1	$1 < k \leq N$	N	N	N	N
Control Flow Threads	1	1	N	N	N	N
Relative Time Sync. Error	0	0	0	$\leq k$	$\leq k$	$\geq \log N$
Sync. Control Flow Threads	0	0	0	1	$N/2$	N
Directed Sync. Primitives?	—	—	—	no	no	yes

Figure 3.1 Hardware Parallelism Constraints for SIMD -> MIMD

Static code scheduling for Static Barrier MIMD (SBM) mode was also described and a comparative study between VLIW code scheduling and SBM scheduling was also presented in [19].

3.3 Barrier MIMD Architectures Formalized

Matthew T. O'Keefe's Ph.D. dissertation in 1990 [17] was a major step toward formalizing the definition of and implementation methods for barrier MIMD architectures. It expanded the original notion of barriers by formally defining and studying the properties of various types of barrier MIMD, as well as proposing generic architectural implementations for:

- Static Barrier
- Hybrid Barrier
- Lookahead Barrier
- Dynamic Barrier

The static barrier mechanism presented in the thesis is the simplest of the barrier mechanisms, while the dynamic barrier mechanism is the most complex. The common element between the various barrier schemes is that they all use centralized barrier control logic, termed the *barrier processor*. The synchronization is not PE to PE, but rather PE to barrier processor. The barrier processor is responsible for tracking which processing elements participate in each barrier synchronization.

One of the distinguishing features of the barrier MIMD synchronization mechanisms is that the set of processing elements that will participate in each barrier is represented by a bit mask. In all the architectures presented by O'Keefe, a queue or associative memory within the barrier processor is used to manage these masks. Masks for a program can be derived at compile time and enqueued by one or more processing elements at run time, however, the actual mechanism to be used is essentially unspecified. Thus, it is not clear how barrier masks would be managed for a program construct that partitions the current mask based on run time conditions (e.g., a parallel `if` statement in which both the `then` and `else` clauses contain independent barrier synchronizations). This omission is partly due to the focus on enhanced VLIW-style execution (in which all masks are known at compile time). The other reason is that the barrier processor by itself provided no way for PEs to agree upon a new mask; whatever communication mechanism the machine possessed would have to be involved, but the communication mechanism was not specified.

Beside the barrier architectures, this thesis also presents compiler techniques for using the barrier MIMD architecture.

3.4 The CARP Machine

CARP[1] (Compiler oriented Architecture Research group at Purdue) machine was an effort to design a parallel supercomputer which can achieve higher efficiency by using information extracted from the program at compile time. The CARP machine used a barrier MIMD architecture to support fine-grain parallelism.

Superscalar processors use dynamically extracted (run time) information to optimize the code execution. Due to the small size of instruction window used by these systems for extracting information, the optimizations which can be performed to speedup code execution are limited. Besides, some aspects of architecture like cache efficiency (utilization of data brought in the cache) cannot be increased at all by dynamic mechanisms. The design of CARP machine incorporates a variety of low level features to help utilize statically derived (compile time) information for better code efficiency.

Normally, a processor fetches a block of data from main memory on every cache miss and places it into the cache memory regardless of the future references to other cache elements in the block. If only one word is used by the processor and the rest of the cache block is not referenced, then fetching a data block on each miss increases the fetch time and results in inefficient cache utilization. A processor working upon a small instruction window cannot predict future references to a memory block. Cache efficiency can be significantly increased by using the information about memory references extracted by the compiler. The CARP Machine's design supported

[1]Information about CARP Machine is from unpublished internal documents

the bypassing of a cache block fetch on the basis of additional information tagged by the compiler on all memory references. This mechanism reduces unnecessary memory fetch cycles and hence increases the performance of a memory system.

Hardware barrier synchronization is a key element for fine-grain MIMD parallelism. Additionally, low latency barrier synchronization provides the system support necessary for meeting static (compile time) timing constraints of SIMD and VLIW code execution on a MIMD system like the CARP Machine. The design of the CARP machine provided not only the hardware barrier synchronization mechanism on each CARP node, but also built access to the barrier mechanism directly into every instruction.

A typical CARP Machine would have consisted of 64 CARP nodes with each CARP node having one floating-point processor, four 32-bit integer units, a fine-grain hardware synchronization mechanism and the interconnection network interface. The block diagram of a CARP node is given in Figure 3.2

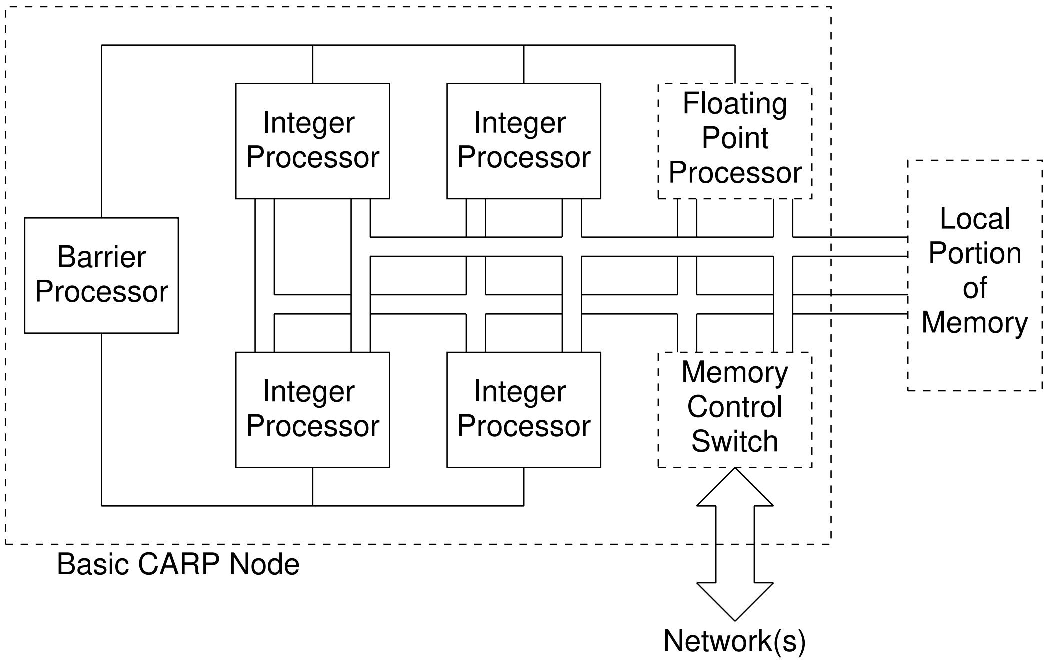

Figure 3.2 Basic CARP Node

A single VLSI chip was proposed for implementing the CARP node. As the CARP Machine was intended to be a stand-alone system, it required not only the design of computation nodes but also the development of I/O and data storage sub-systems.

3.5 The CARD Project

Although theoretically attractive, building the CARP Machine requires design and fabrication of a custom VLSI processor, as well as a wide range of other hardware and software components. It simply is not something that Purdue's current resources can support.

Thus, the CARP research drifted toward using as much commercially available technology as possible, yet preserving at least some of the interesting and novel features of the system design. The result was a focus on designing the CARD system to make use of a custom board design incorporating standard parts and hosted by standard personal computers. This board was dubbed CARDBoard [2] (the Compiler-oriented Architecture Research Demonstration Board).

With the goal of constructing a 32 GFLOPS supercomputer with minimal custom hardware and relatively low-cost parts, the CARD project focussed on combining commercially available micro-processors in a barrier MIMD structure based on the CARP Machine design. Thus, the CARD system also is designed to support fine grain parallelism and to utilize static (compile-time) timing constraints to enhance execution efficiency.

The following sections provide the design methodology and block design of various components of the CARD system. However, design details may change in future when fabricating a CARD system.

3.5.1 CARD System's Overview

CARD system was designed on the same principals as that of CARP Machine, therefore the basic element (computation node or CARDBoard) of CARD system

[2] Information about CARDBoard Project is from unpublished internal documents

was designed to provide much of the functionality of a CARP node using off-the-shelf microprocessors. To eliminate the design of I/O and data storage systems, it was decided to use a standard computer to house CARDBoards (Computation Nodes). In this way, CARD system uses the resources of host computer to simplify program development and debugging. The complexity of custom components is also reduced. In order to provide scalability, each host computer can house a communication interface which is used to connect compute nodes housed in different hosts. As more compute nodes means a larger number of host computers in a CARD system, data storage and I/O speed also scales with the system's computation power.

Figure 3.3 gives the overall picture of typical CARD system.

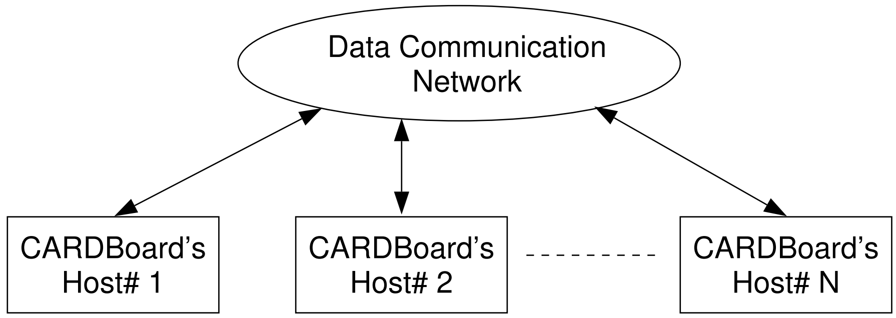

Figure 3.3 A Typical CARD System

3.5.2 The CARD System's Host Computer

The criteria in selecting the type of host computer for the CARD system were:

1. Type of expansion bus available for plug-in compute nodes and host-to-host communication interface.

2. Availability of free-ware UNIX and support software on that computer.

Due to the simplicity of the interface logic required for the ISA bus available on the IBM PC and compatible computers, and the availability of Linux operating system with numerous application softwares on these systems, it was decided to use IBM PC and compatibles as the host computer for CARD system.

A 16-bit ISA bus interface can support a peak bandwidth of 16 Mbytes/sec on a 16 MHz PC AT system [21]. This may create a I/O bottleneck on a large CARD system. The PCI bus interface is a better alternative for a higher performance CARD System, because it can support data transfer rates up to 120 Mbytes/sec for a 32-bit interface. However, PCI is relatively new, and it requires a complex interface circuit. With the availability of PCI chip-sets from different vendors, this problem could be solved and we expect to use PCI bus when CARDBoard is finally built. New PCs and compatibles also provide a PCI expansion bus, therefore the choice of platform for the host computer remains the same even when PCI bus is chosen.

A typical CARD host will have 4 CARDBoards (plug-in computation cards) with each node having four floating-point microprocessors and a communication/barrier interface card. This communication/barrier card will provide inter-host communication and inter-host barrier synchronization. This card may also provide inter-node (CARDBoard to CARDBoard) communication and inter-node barrier synchronizations for the CARDBoards mounted on the same host computer, but this issue is still unresolved.

Figure 3.4 provide the basic block diagram of the proposed CARDBoard host.

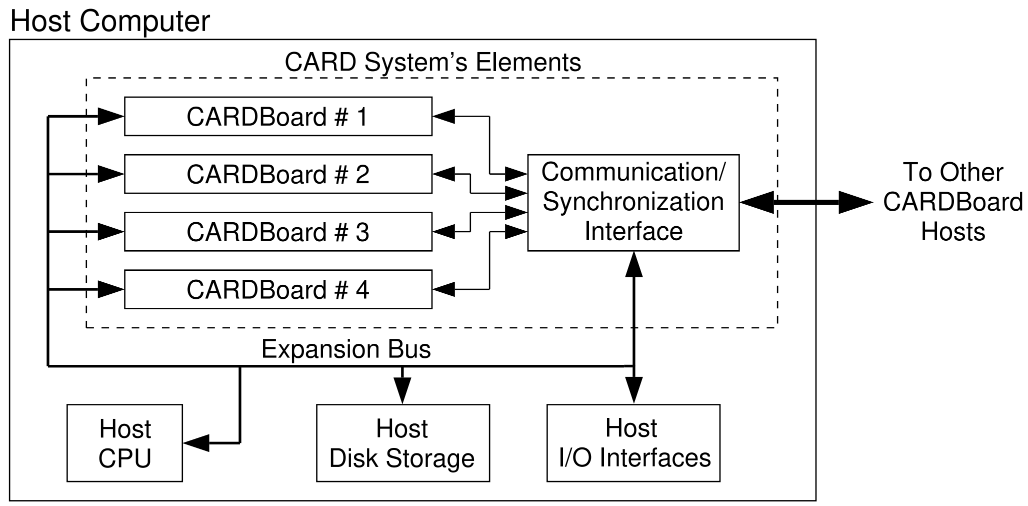

Figure 3.4 A CARD Host

3.5.3 CARDBoard

The CARD system is designed to be modular and scalable in nature, therefore a system can be based on either a single node (only one CARDBoard), or can have up to 4 CARDBoards in a single host, or it can be a large system with multiple hosts each containing 4 CARDBoards.

The most detailed CARDBoard design is for an ISA plug-in card with:

1. Four microprocessors with floating point units, each with a local memory.

2. A small shared memory which is connected to all the PEs and also to the ISA bus. This could be a dual port memory with one side connected to the ISA bus and the other side shared between the 4 PEs.

3. ISA bus interface logic.

4. Barrier Synchronization Logic.

5. Communication mechanism for data transfer with other computation nodes on the same host as well as to with the computation nodes on other host computers.

Evaluation of timing constraints during compilation of code requires predictable timing for each instruction of the target processor. Therefore, it was the basic criteria in selecting a microprocessor for the CARDBoard. Beside this, we wanted to get a peak performance of at least 100+ MFLOPS from each CARDBoard (Computation Node) which translated into the requirement for a microprocessor with high floating point performance.

Initially, the TMS320C30, a digital signal processor (DSP) was selected as the microprocessor for CARDBoard. TMS320C30 can execute 33 MFLOPS with 17 MHz clock [22]. Except for a few instruction that take 2 cycles each, the instructions of the TMS320C30 execute in 1 clock cycle. This simplifies the compiler's timing analysis. Another reason for selecting TMS320C30 was the availability of two separate memory buses on the TMS320C30, which could have simplified the interface to the shared memory. Texas Instruments promised to donate TMS320C30 chips but none have arrived to date, so we had to look for other processors.

In the meantime, while working on the barrier mechanisms for the CARD system (described in section 3.5.5 and 3.6) we discovered new efficient and simple hardware implementations for dynamic barrier synchronization. Originally we assumed that the CARD System would use a static barrier mechanism that can support only one barrier stream. However, we would be able to use the new dynamic barrier implementations that support multiple independent barrier streams. Thus, space sharing would be possible.

The current choice of CARDBoard microprocessor, AMD's Am29050, is partially based upon the donations we received from AMD. The Am29050 is a RISC processor that can execute 80 MFLOPS at 40 MHz [23]. The instruction execution times for Am29050 are not as tightly bound as in TMS320C30. However, an instruction timing is still relatively predictable.

A major advantage of the Am29050 is the availability of 3 user defined signals which are controlled by bits 16-18 of a load or store instruction. These user defined signals can be used to construct new instructions for a barrier hardware interface, thus bringing more of the functionality of a CARP node to the CARDBoard. A slight hitch in using the Am29050 arises from the separate data and instruction buses that fetch data and instruction from separate memories, but share a single address bus. This requires additional buffers for memory interfacing.

3.5.4 CARD's Data Communication/Synchronization Card

An additional plug-in card is used by each host computer for data communication between the computation nodes of different hosts. This network can also be used to transfer barrier mask and barrier synchronization information between the PEs of the CARD system. It can also be used for communication between PEs belonging to separate CARDBoards, but housed in the same host computer.

Due to long delays involved in transmission of data through copper cables, it was proposed that a fiber-optic link should be used for data communication. Use of a fiber link also reduces the latency of barrier hardware when barrier synchronization is performed across multiple host computers. However, fiber optic link results in new design problems. The conceptual design of this card is not yet final.

3.5.5 Evolution of Barrier Synchronization Mechanism in CARD System

As mentioned earlier, the CARD system is based on the design concepts of the CARP machine, therefore hardware barrier synchronization is a key feature of CARD system. The hardware barrier mechanism is used by the CARD system for execution of fine-grain MIMD codes and for efficient emulation of SIMD and VLIW codes.

Since the inception of the CARD project, great progress has been made in efficient and cost-effective implementations of hardware barrier synchronization. The CARD project has been the driving force behind all these efforts. Starting with a machine-wide static barrier implementation, we have progressed to a dynamically partitionable dynamic barrier mechanism, which in turn has led to the design and implementation of the PAPERS system.

At the start of the CARD Project, the only feasible implementations of barrier mechanism were the ones proposed in [2] by Matthew O'Keefe. The centralized barrier processor and use of a central queue or associative memory was not compatible with the distributed and scalable design of a CARD system. Besides, a centralized control unit requires different hardware for different numbers of processing elements in a parallel system. Additionally, the masks for a program are computed at compile-time and enqueued before executing the program. Execution of multiple programs on a group of PEs requires merging of barrier streams from different program before staring the execution of any program. Once the barrier masks are enqueued and a program starts executing, *no new* program can be started on free PEs without stopping the whole machine. This is highly undesirable in a CARD system with large number of PEs. Therefore, much of the effort was directed towards an implementation which can be distributed across multiple host computers and yet could be connected together to scale up the system without any modification to hardware design.

The barrier synchronization mechanism for CARD started with a *machine-wide* implementation of a *static barrier*. The barrier hardware in this case was simply an AND tree connected to barrier requests from all the PEs in the machine similar to Burrough's FMP [4]. A barrier synchronization is achieved by performing a LOAD operation from a specific address space to indicate a barrier synchronization request. This barrier request signal is routed to the AND tree, and the output from the AND tree is used to force a processor to wait for the completion of barrier. This was termed the *Extended-LOAD Barrier Mechanism*. The sequence of operations involved in a barrier are:

1. A PE marks itself present at a barrier by simply referencing a specific memory region. The address decode logic of the PE decodes this as a barrier request and sends a 1 to the barrier AND gate.

2. These bits from all the PEs are ANDed together. The return bit from the gate is used to insert wait states in the memory cycle. A return value of 0 from the AND gate indicates that all the PEs have not reached the barrier; a return value of 1 indicates completion of the barrier.

3. As soon as the value of the AND becomes 1, all the processors are released from their memory waits. Therefore, all the PEs terminate the barrier request simultaneously, and execute their next instruction in unison.

Next, we realized that a *dynamic barrier mechanism* could be implemented by replicating the barrier tree described in [1] at each PE and by keeping the barrier masks computed at compile time in the local memory of all the PEs. The purpose of this tree is to determine whether the processor it is associated with needs to wait for the barrier synchronization to complete. The processor needs to wait if and only if at least one processor it should synchronize with has not notified this processor's tree that it is waiting. This technique eliminates the barrier processor and the central barrier queue or associative memory required by dynamic barrier implementation described in [17].

In this implementation, each PE in a group of PEs (PEi through PEj) executing a parallel program keeps the barrier masks associated with that program within their own local memory. The PEs which are not part of the parallel program are masked off in the barrier masks of PEi through PEj. Therefore, new programs can be started on the idle processors without interrupting the programs already executing on the system.

The hardware which is replicated at each PE is simply an OR-AND tree as shown in Figure 3.5.

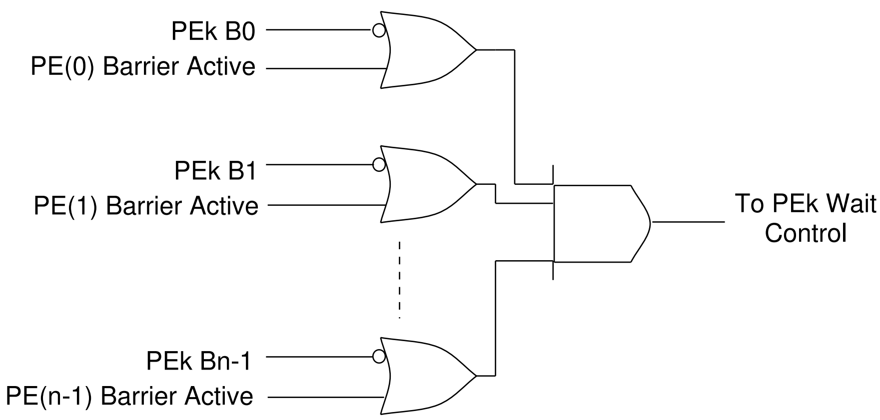

Figure 3.5 Dynamic Barrier Mechanism for PEk

This new barrier synchronization method uses a modified extended-LOAD barrier mechanism. In a general implementation, a 1 at the most significant address bit is used to indicate a barrier request and the rest of the address bits are used as *barrier mask bits* to specify the PEs with which a PE wants to synchronize.

The sequence of operations for a barrier is:

1. An array of bit masks is kept inside the program memory of a processor for the series of barriers required in a program. The bit position corresponding to

the processors which are enabled for a barrier is a 1 and for others it is a 0. A barrier counter is also maintained by the processor.

2. A processor indexes into barrier mask array and reads the barrier mask, forms a barrier address with a 1 at most significant address bit, and performs a LOAD operation.

3. The address decode circuitry of the processor decodes the address as the presence of this processor at the barrier, and sends this one bit of information to all the other processors in the system.

4. The barrier logic of the processor also gets the presence-at-barrier bits from all the other processors. The return bit from the barrier tree is used to insert wait states. A return value of 0 from the logic tree indicates that all the processors included in the barrier (processors with corresponding barrier mask bits set to 1) have not reached the barrier, while a return value of 1 from the barrier tree indicates the completion of a barrier.

5. The output from the logic tree of processors with identical mask bits becomes 1 after the arrival of last processor at the barrier. Therefor, all the processors in a group resume their execution simultaneously.

6. After returning from barrier, the PE increments its barrier counter.

For the CARDBoard that is based upon Am29050, we can define a new barrier synchronization instruction (a modified LOAD) which utilizes the user defined signals to generate a barrier request, however, the address lines are still used as barrier mask bits.

This implementation of dynamic barrier hardware eliminated the barrier processor and associative memory at the cost of higher wiring complexity. There was only one output from PE to barrier processor and one input from barrier processor to each PE in the dynamic barrier implementation presented by Matthew O'Keefe [17]. But now, the barrier tree is replicated at each PE, and there are N wires coming to the barrier logic. However, each PE sends *only one signal* to all the other PEs, therefore the wiring complexity of this implementation is $O(N)$.

The issue of efficiently partitioning a group of PEs dynamically on the basis of evaluation of some parallel conditional statement remained unresolved. As mentioned earlier, a possible solution is to use the data communication network to gather information about conditional evaluation from all the PEs in the barrier group and use this information to derive a new barrier mask. This operation has $O(N)$ latency as compared to the constant latency of barrier synchronization. However, we do not have the problem of enqueuing the new masks calculated. All the PEs keep the barrier masks in their own memory, and therefore can update their barrier masks without any problem.

Although partitioning can be performed using the data communication network, this mechanism does not support the enlarging of partitioned subgroups (recombination of subgroups). Only one signal is sent to other PEs by a PE to indicate the presence of the PE at the barrier. A PE does not indicate which PEs it actually wants to synchronize with. Figure 3.6 is used to explain the problem in recombining subgroups with this implementation.

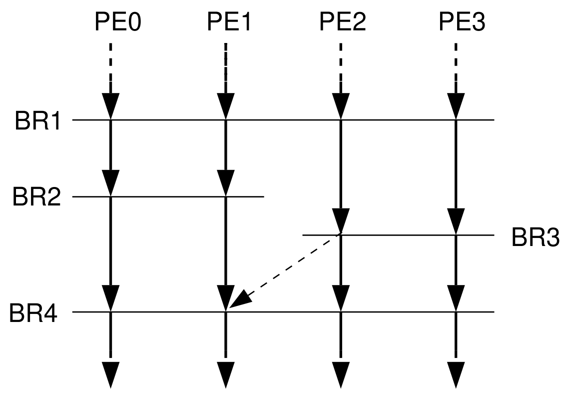

Figure 3.6 Problem in Recombining Subgroups

PE0-PE3 execute a full barrier BR1 and then repartitions to form two subgroups, PE0 and PE1 form one group while PE2 and PE3 forms other group. The masks are therefore appropriately set for the PEs in the two subgroups. PE0 and PE1 execute the barrier BR2 and then restores there mask for recombining barrier BR4. PE2 and PE3 arrive at BR3, a subgroup barrier. As PE0 and PE1 are waiting for PE2 and PE3 at the barrier point, they will see that the PE2 and PE3 have reached barrier and have no mechanism to tell that PE2 and PE3 do not want to synchronize with them at this point. PE0 and PE1 will complete the barrier BR4 and will resume their execution, thus recombining barrier BR4 will not be executed properly and PEs will get out-of-sync.

As this implementation uses address lines to specify barrier masks during barrier synchronization, the scalability issue also arises from the limitation of available address pins (32 on a Am29050).

3.6 Partitionable Dynamic Barrier Mechanism

The quest for a more efficient barrier mechanism for CARDBoard led to a a novel approach to implement *run-time partitionable dynamic barrier mechanism* in October 1993 [24] [3] [25]. This implementation forms the basis for the PAPERS design presented in the following chapters. Therefore, this section presents some *excerpts* from [24] to convey the key concepts of this new mechanism.

3.6.1 Barrier Architecture

This partitionable dynamic barrier mechanism also uses an extended-LOAD operation to perform barrier synchronization. The address referenced by the LOAD must be decoded as a barrier synchronization and all relevant signals must be appropriately latched by the processor for the barrier logic. The new barrier architecture, which is replicated for each processor, is depicted in Figure 3.7

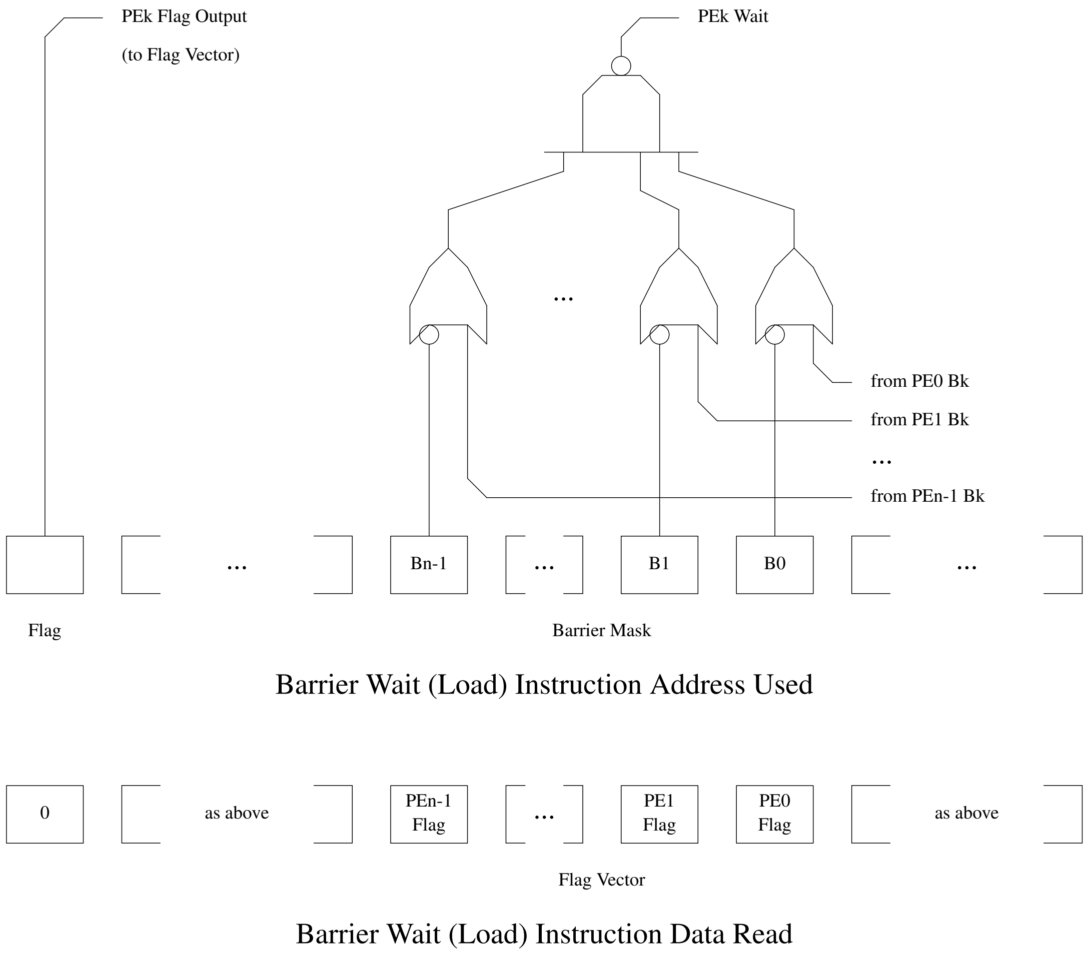

Figure 3.7 Partitionable Dynamic Barrier Mechanism for PEk

The logic tree in Figure 3.7 is virtually identical to Figure 3.5. The new architecture also eliminates the barrier processor and central barrier mask queue or associative memory by replicating the barrier tree for each processor.

The interesting twist is that each of the processors is responsible not only for determining when it may proceed past the barrier, but also for informing all the other processors in that barrier that it is waiting. Both these functions are accomplished by use of the barrier mask, which is extracted from the LOAD address. Thus, instead of having just one output line from each processor, there is *one output line for each* of the other processors in the machine (i.e., for N processors, $O(N^2)$ wiring complexity), and it is the responsibility of each processor to set these lines appropriately for each barrier. This feature is the key for enlarging current barrier groups (recombination of subgroups), which was not possible with the barrier mechanism described in the previous section.

If the set of processors participating in each barrier was known at compile time, or did not change during execution, the above portion of the hardware would be sufficient. However, to support enqueuing of different barrier masks, the barrier architecture also constructs a return value for the LOAD. The value LOADed is identical to the barrier address referenced in the LOAD, except in that the *flag* field is forced to 0 and the barrier mask is replaced by the *flag* bits gathered from all the processors. This gathering is implemented by direct wiring. Thus, data is gathered from all the PEs on each barrier synchronization which gives the capability of low latency dynamic partitioning to this implementation of dynamic barrier without any external communication network.

To better understand how this barrier architecture functions, it is useful to describe in detail how barrier masks are manipulated to perform each of the fundamental types of barrier manipulations. The most basic barrier operation is to perform a dynamic barrier synchronization. In addition, we describe the two most fundamental ways of *enqueuing* new barrier patterns: partitioning the current barrier group and enlarging the current barrier group, which was not supported by the dynamic barrier mechanism in previous section. Because the hardware does not literally use a queue, these two cases degenerate into simply determining the proper barrier mask within each processor on the basis of data gathered on each barrier synchronization.

3.6.2 Dynamic Barrier Synchronization

The basic operation of a barrier synchronization is accomplished by each participating processor asynchronously performing the following sequence of operations. Notice that the sequence is not strict; some *steps* can be overlapped.

1. Internal to processor, a bit mask is created or maintained such that the bit position corresponding to each processor that will participate in the barrier is a 1, and the positions corresponding to the other processors are all 0.

2. Internal to the processor, the bit mask is aligned to the 'Barrier Mask' positions B0 through B$n-1$, and is inserted into the base address that will be decoded as a barrier synchronization request.

3. The processor initiates an external reference, apparently to fetch the contents of the address computed in [2].

4. The barrier hardware recognizes that the address reference is actually a barrier synchronization request, and thus gates each of the barrier mask bits both to this processor's tree and to the corresponding processor's tree. For example, the bit Bi from processor j would be an input to both the tree from processor j and the tree from processor i. Notice that there is no network involved in routing each bit; each connection is literally a dedicated wire. The bits coming from processors that are not performing barrier synchronization are forced to be zero.

5. After a small propagation delay, this processor's tree yields a single bit answering the question: 'Is there a processor that the local barrier mask indicates should participate in the barrier, that has not sent this tree confirmation that it has reached this barrier?'. Notice it is up to each processor to ensure that it knows which processor it should synchronize with (as suggested in step [1]; if these masks are inconsistent, strange behavior can result.

6. If the signal generated by the tree is a 1 (true), then this signal is used to stall the processor until other processors cause the tree's signal to change to a 0 (false). Typically, this staling is implemented by inserting *memory wait states*.

7. Upon the tree's signal becoming a 0, the barrier hardware is reset (the barrier mask latches are cleared) and the processor is allowed to complete the access that initiated the barrier.

It is also useful to note that, since all processors determine their firing conditions independently, there is no conflict in firing multiple non-overlapping barriers simultaneously. This constitutes yet another improvement over the original dynamic design [2].

3.6.3 Partitioning A Barrier Group

Because it is the responsibility of each processor to know the set of processors with which it will synchronize, a method is needed to notify all processors that will participate in a barrier as to the complete set of processors in its barrier group. If the partitioning into groups is known at compile time, as it might be in ELP [26], then this notification is accomplished by simply placing appropriate lists of barrier addresses within the code generated for each processor. However, it is more common that this partitioning is not statically known. Instead, new barrier groups are most often created by partitioning an existing barrier group into two groups based on the runtime evaluation of a conditional expression. Those processors for which the condition evaluates as true form one barrier group and those for which it evaluates as false form the other.

The dynamic partitioning of a barrier group is accomplished by using a barrier synchronization to ensure that all processors in the original barrier group have evaluated their conditional expressions. This same barrier is also used to gather and broadcast the results of the conditions for all the processors. The barrier operation sequence is as described in Section 3.6.2, with the following changes:

Insert step [1a] before step [2]:

1a. Internal to the processor, the conditional expression is evaluated. Depending on the truth of the expression, the barrier base address is selected as an address that will be decoded as a barrier synchronization with the flag bit equal to either 0 (false) or 1 (true).

Insert step [4a] before step [5]:

4a. In addition to gating and routing the barrier mask bits, the flag bit from each processor is sent to a global register such that the flag from processor k is placed in the bit position that aligns with bit Bk. The bits of this global register that do not align with barrier mask bits are hardwired to match the aligning bits in the base barrier address with a flag value of 0.

Change step [7] to:

7. Upon the tree's signal becoming a 0, the processor is allowed to complete the access that initiated the barrier. The value read from the data bus is the value sampled from the global register. This value is essentially a barrier address, including the barrier mask bits.

Add steps [8], [9], and [10]:

8. The barrier hardware is reset (latches cleared) and the bit position in the global register that corresponds to this processor is reset.

9. This step is performed only if the conditional expression is false on this processor. The new barrier address for this processor should contain barrier mask bits for only those processors that had flag values of 0, thus, the barrier mask field within the value read should be inverted. Typically, this is done using an XOR with a value that has 1s in the barrier mask positions and 0 in all other bits.

10. If all processors participated in the original barrier, the result value is directly usable as the new barrier address, including the barrier mask bits, for the processors in this processor's barrier group. However, if some processors do not participate, the barrier mask bits corresponding to processors that did not participate in the barrier may have undefined values. In this case, ANDing the value with the original barrier address will force the undefined bits to be 0, thereby excluding the corresponding processors from the new barrier group.

If a partitioning must decompose a barrier group into more than two subset barrier groups, a sequence of binary partitionings can be used to create the subset barrier groups.

3.6.4 Enlarging A Barrier Group

Just as each processor is responsible for determining which processors it should synchronize with in the case of partitioning a barrier group, each processor is also responsible for determining which processors it should synchronize with when the current barrier group is to be enlarged. There are two ways in which the current barrier group can be enlarged - and neither one requires execution of a barrier synchronization.

The first case involves enlarging the current barrier to encompass a statically-known set of processors. This is accomplished by simply embedding the (compile-time constant) barrier address for the new barrier group in the code for each processor that will participate.

The second case, which is the most common case for structured programs, involves restoring the barrier group that existed prior to a partitioning operation. This can be accomplished by having each processor save its current barrier address just before each partitioning operation. Thus, any partitioning can be *undone* without a barrier synchronization.

Notice that there is nothing to prevent processors from partitioning or enlarging barrier groups using whatever communication hardware mechanisms are available, because barrier masks/addresses can be transmitted by any mechanism capable of sending integer/address values. However, this barrier implementation provide low latency flag gathering that supports the execution of fine-grain parallel programs.

3.6.5 Scalability

The scalability issue arises as the partitionable dynamic barrier mechanism uses the extended-LOAD mechanism and has an $O(N^2)$ wiring complexity.

There are limits to the number of processors that can be synchronized with a single LOAD instruction. For most modern processors, the direct implementation of the above architecture is limited to systems with fewer than about 32 processors. However, in a machine using many more than 32 high-performance processors, signal propagation delays alone are likely to extend the cost of synchronization well beyond the cost of a single LOAD. Thus, the most reasonable scaling method is to use this barrier architecture within a cluster and another method across clusters.

Acknowledging that the proposed barrier architecture does not scale well to massively parallel systems, it is useful to understand that the processor interface can scale to massively parallel systems. For example, the barrier mask field could be used to represent the number of the barrier group that this processor wants to synchronize with, and external barrier hardware could maintain information about which processors participate in which barrier. The *flag* bit could still be used to generate new partitions of the barrier group, but the external barrier hardware would have to assign a new group number and arrange for the processors to be notified of their new group number through the return value LOAD. Although such a scheme implements a weaker form of barrier synchronization, and probably executes somewhat more slowly due to the complexity of the barrier synchronization unit, it would yield the same functionality (provided the maximum number of active barrier groups was not exceeded).

Another possible variation would be to maintain the barrier architecture as described here, but to use multiple operations to load barrier masks and to retrieve flag vectors. This can be thought of as simply 'time multiplexing' the operation of the barrier hardware's inputs and outputs to meet limitations on address and data bits available and to dramatically reduce wiring complexity. This achieves the complete functionality, but with a significant performance penalty. Notice that the performance penalty in detecting that a barrier has fired is proportional to the multiplexing factor, but the other overheads might not increase significantly. For example, if the same barrier mask will be used in multiple consecutive barrier synchronizations, there is no need to enqueue the barrier each time nor to compute and examine the return value. It is also possible to specify only the portion of a barrier mask which is different from the previous barrier mask, or even to reference barrier masks from a 'cache' maintained within the barrier synchronization hardware.

4. PURDUE'S ADAPTER FOR PARALLEL EXECUTION AND RAPID SYNCHRONIZATION (PAPERS)

As mentioned earlier, CARDBoard is one of the proposed target systems which will use the partitionable dynamic barrier mechanism [24] described in section 3.6, but the given implementation of dynamic barrier can be used by any parallel computer for hardware barrier synchronization. This meant that building of a CARDBoard prototype was not the only way to test the partitionable dynamic barrier synchronization logic.

We realized that we could use commercially available computers as Processing Elements (PEs) to test the barrier synchronization mechanism. Use of the barrier synchronization mechanism with stand-alone computers proved to be an effective method for executing parallel codes on a cluster of computers. Thus, the *Purdue's Adapter for Parallel Execution and Rapid Synchronization(PAPERS)* has become much more than a convenient testbed for dynamic barrier synchronization.

Figure 4.1 gives the overall picture of N PEs (PCs/Workstations) connected with each other through PAPERS.

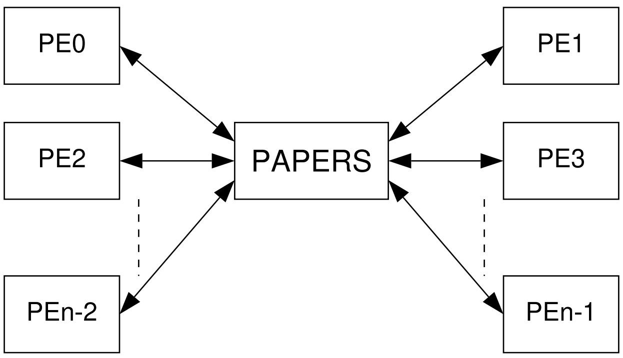

Figure 4.1 PAPERS

The PAPERS design is greatly influenced by our choice of the printer port as the computer's interface to the barrier synchronization logic. Printer port input/output operations are significantly different from the LOAD instruction interface used by the barrier implementation described in 3.6. This required modifications to the original implementation of dynamic barrier synchronization. Besides, connecting computers with each executing an independent and non-parallel OS (Linux) required additional functionality from the PAPERS, which resulted in addition of an *Interrupt mechanism*.

4.1 Computer's Interface to PAPERS

When we began to design a system to test the implementation of dynamic barrier synchronization using standard computers, we had two options for interfacing a computer to the barrier synchronization logic:

1. Design a custom interface for the computer.
2. Use one of the standard I/O interfaces: the serial port, SCSI bus, or printer port.

Design of a custom interface would have required building a plug-in I/O card for a specific peripheral expansion bus. At that time, there was no single bus standard across different platforms. Until recently, IBM PCs and compatibles have used and ISA bus or an enhanced version of ISA like EISA or VESA. Pentium based computers and workstations from various vendors now use PCI as the primary bus interface, but also support an enhanced variant of the ISA bus standard. All Sun workstations use the SBus expansion bus. PCI seems to be the bus standard that will be supported by the most workstations and high-end PCs, but this was far from obvious in 1993.

Since all these buses are used for memory and I/O expansion, appropriate (address, data and control) signals from the computer's processor are available on these buses. Therefore, the dynamic barrier mechanism based on an extended LOAD operation, as described in previous chapter, can be directly implemented by a custom interface card.

However, depending upon the structure of the refresh circuit for dynamic memory on a computer, the number of memory wait states that can be inserted in an extended LOAD operation may be limited. The logic tree of the dynamic barrier mechanism is replicated on each node, and requires only the appropriate signals from other PEs. Therefore, the barrier logic can be built on the interface card itself with only the signal connections between interface cards. Hence, an *external* circuit module (like current PAPERS units) is *not* required. Besides, a custom interface card can perform the barrier synchronization operations at the computer's interface bus speed.

The worst drawback of a custom interface design lies in its non-portability across different platforms. None of the interface buses mentioned earlier are compatible with each other. Thus we would have to design multiple interface cards to support dynamic barrier synchronization on different computers. Besides, design of an expansion I/O card generally requires expensive interface chip-sets for a specific expansion bus. In addition to the hardware design, a custom interface card also requires writing of Unix *device drivers* for software interface to the hardware. Device drivers are also machine dependent, therefore each interface card would have required different device driver software.

The advantages of a custom interface card, outlined in previous paragraphs may seem good enough to implement dynamic barrier synchronization on a cluster of computers using custom interface card. In fact, after experimenting with PAPERS units, we now realize that the hardware dynamic barrier synchronization is a very effective way to provide the capability of fine-grain code execution and low-latency communication to a cluster of workstations. Therefore, design of a custom interface card to build a tightly coupled parallel computer using a cluster of workstations is quite viable in the future.

However, the disadvantages of a custom interface card outweigh its advantages for the design of a system that was intended only to be a testbed for the dynamic barrier mechanism. Therefore, we postponed the design of a custom interface card, instead focusing on using a standard interface. Use of a standard interface requires an external hardware module (PAPERS) to implement the dynamic barrier synchronization. Computers in a cluster communicate with this module using a standard interface, therefore clusters based on heterogenous collection of computers can use the same synchronization hardware. Thus use of a standard interface makes the barrier synchronization hardware portable.

The implementation of dynamic barrier synchronization requires parallel input and output signals. Therefore, the selection of an interface was based on the availability of input and output pins on that interface. Among the standard I/O interfaces available on computers, the serial port was not selected due to the low bandwidth serial nature of its data I/O and limited number of I/O pins. Although, the SCSI bus supports parallel input and output, it was not selected due to its complex interface and difficult software control. Thus, the only reasonable option was to use a *printer port* as an interface to the barrier synchronization hardware. A printer port has enough input and output signals and provides a simple, direct, software control of I/O pins.

4.1.1 Centronics Printer Port as an Interface to PAPERS

Almost all PCs and workstations provide a centronics parallel printer port interface. A centronics printer port is basically a parallel port having an 8 bit output bus with some control outputs and status inputs. Counting these status lines, there are actually 12 output signals and 5 input signals on a centronics printer port. Some computers provide extended functionality parallel ports that allow 8-bit bidirectional data connections, but the specifications for the bidirectional printer port from different vendors are not standard, and the performance improvement is not as great as one might imagine. However, all of these parallel ports can work as centronics compatible unidirectional ports.

To ensure that PAPERS can be used with any PC/Workstation, PAPERS makes use of only those functions that are supported by a standard *Centronics Parallel Printer Port.*

Physically, a centronics compatible printer port interface is available as a DB25 connector on almost all PCs and workstations. A notable exception to this is older SUN workstation hardware, which does not have a printer port interface. SPARCstation 5 and SPARCstation 20 workstations provide a centronics compatible printer port interface through a non-standard 26 pin miniature SCSI like connector.

4.1.2 Software Control of Printer Port

PCs and Workstations executing a UNIX operating system provide control of the printer port through device-specific control functions (using `ioctl()`) and system calls. Although these functions provide a secure access to the printer port by a user process, these functions incur system call overhead which significantly limits the number of reads/writes to the printer port that can be performed in a given time period. Besides, depending upon the system's internal hardware interface to the printer port registers, these operations are geared towards block data transfer.

A direct read/write access to the printer port's control registers provides the fastest mechanism for controlling signals on the printer port. These control registers can either be mapped into the computer's memory address space or I/O address space. PC XT/AT/386/486 systems and some of the Pentium-based computers have an I/O mapped printer port with provision for three printer ports with base addresses 378H, 3bcH, and 278H corresponding to MS-DOS printer names LPT1, LPT2 and LPT3 [27]. Typical workstations, including IBM Power PCs, DEC Alphas, HP Apollos and SUN SPARCstations, have the printer port mapped in the memory address space. Workstations having a PCI local bus generally provide an ISA bus interface via PCI/ISA bridge circuitry. Although I/O devices on an ISA bus are intended to be I/O mapped, workstations having PCI buses map the ISA interface (address space) to a portion of memory [27]. Therefore, the printer port on such systems is in fact memory mapped.

Some version of UNIX, including Linux, allow user processes to have direct access to I/O devices. However, for those versions of UNIX that do not allow direct access to the I/O device, the printer port register can be mapped into a user process's memory for direct control of the printer port.

4.1.3 Data Communication Network

The fact that PAPERS itself can be used for data communication between PEs was not realized until the completion of the first prototype, PAPERS0. Therefore, during the initial phase of the PAPERS design it was assumed that PEs (PC/Workstations) would have a separate data communication network. In the simplest form, the data communication network can be a SLIP connection (null modem connection through a serial port), but generally it is an Ethernet network. Although PAPERS provides low latency data communication between PEs, the data bandwidth is low (20-150 Kbytes/s). Therefore, a data communication network is desirable for block data transfers between the PEs.

For the current implementations of PAPERS connecting 4 or 8 PCs, high bandwidth data networks are not required. However, high bandwidth communication networks like HIPPI, FDDI, Fast Ethernet (100Mbits/s) and ATM are attractive choices for block data communication within a large (16 processors or more) cluster of high-end workstations.

4.2 Dynamic Barrier Mechanism in PAPERS

As mentioned earlier, use of a printer port as an interface to the dynamic barrier synchronization hardware (PAPERS) for a cluster of workstation resulted in new problems. An extended LOAD dynamic barrier synchronization operation as described in section 3.6.2 cannot be implemented using the printer port interface. The solution to this problem, although similar to the extended LOAD implementation, required not only a new operation sequence for barrier synchronization, but also modification of the barrier logic itself. The key differences are the use of a memory element in the barrier logic and a two-phase barrier synchronization operation, both of which are described in the following sections.

4.2.1 Background

The sequence of operations for a dynamic barrier synchronization, as described in section 3.6, requires only *one memory reference* per barrier synchronization and can be implemented using a combinatorial circuit for barrier synchronization logic as:

- The address and data buses of the PE (processors) are directly connected to the barrier tree logic. The address decode circuit of the processor is responsible to decode the memory references to specific memory space as barrier synchronization request. Therefore, no additional bit is required to indicate barrier synchronization request. Besides, the address decode logic is responsible for latching the address (Data flag + Mask) for barrier logic and for the insertion of memory wait states on the basis of the output from the barrier tree. Hence, barrier logic tree is implemented exactly as shown in Figure 3.7.

- The PEs that execute barrier synchronization request (a LOAD) are forced to wait (if barrier not satisfied) by the address decode logic. This is accomplished by extending the LOAD operation (by inserting memory wait cycles), therefore, a PE that is waiting cannot execute any other operation until the barrier is satisfied (until PE returns from LOAD). This eliminates the possibility of a PE arriving at the barrier and then performing other operations which can result in the PE missing the output of the combinatorial barrier logic tree.

- All the PEs participating in a barrier, terminate their LOAD cycles simultaneously after the arrival of the last PE in the group at the barrier. Thus, all the PEs read (load) the same bit vector from the data bus as a result of completion of LOAD cycle. No PE can change its output fast enough to cause the problem of other PEs seeing a wrong (different) bit vector on its data bus.

These observations are presented to clarify the modifications required for implementing dynamic barrier synchronization on cluster of workstation using a standard printer port interface.

4.2.2 Why a Memory Element in Barrier Logic of PAPERS?

Why is there a memory element within each PAPERS unit?. The short answer is, because it is needed to ensure that the barrier go signal remains valid until it has been obtained by all the processors. The long answer follows.

PCs and Workstations are connected to the barrier synchronization logic of PAPERS through a parallel printer port interface. A printer port is an I/O device in the address space of a PC or workstation and the signals on the printer port output are controlled by internal registers. The processor of a PC or workstation controls the status of the printer port's output signals by writing to the output registers (latches), therefore a printer port maintains the previous value until a new value is written to the register. To get the status of the input signals a processor performs a read operation on input register (buffer). Thus, *none* of the processor's signals (address, data and controls) are available on the printer port.

The use of printer port poses new issues:

- As read and write operations on registers are internal operations, these operations (in fact LOAD operations) cannot be extended by an external device connected to printer port. Therefore a single cycle extended LOAD dynamic barrier mechanism cannot be implemented. The minimum implementation will require a PC/Workstation to generate a barrier synchronization request by executing one port write operation and then performing one read operation to check the answer (go/wait) from the barrier logic.

- Depending upon the hardware of the PC or Workstation, the minimum time between two consecutive port operations ranges from 1μs to 5μs.

- The PAPERS unit is designed to connect PCs or Workstations executing UNIX operating system. UNIX provides a multi-tasking environment, therefore the possibility of a context switch or an interrupt occurring between the two successive port operations cannot be ruled out (Although it is possible to temporarily disable interrupts, this is a dangerous practice and will be discussed no further). A context switch or interrupt can introduce a delay in the order of milli-seconds between consecutive port operations.

A printer port interface cannot support one cycle extended LOAD barrier synchronization and there are significant delays in-between consecutive printer port accesses, therefore, a combinatorial circuit cannot be used for the barrier synchronization logic in PAPERS. An additional signal is also required to explicitly indicate a barrier synchronization request and is called *Strobe*.

The following scenario, assuming the use of barrier logic tree of Figure 3.7 (a combinatorial circuit) clarifies this.

1. PEi outputs a barrier synchronization request with bits Bi and Bj high (included in barrier).

2. Context switch occurs for PEi.

3. PEj reaches the barrier instruction and outputs a barrier synchronization request with bits Bj and Bi high (included in barrier).

4. PEj reads a go signal (barrier done) on its input.

5. PEj enqueues a new barrier synchronization request with a different barrier mask.

6. PEi returns from context switch and reads its port to check the status of its barrier tree output. PEi will not see the barrier synchronization (although it has occurred). Thus, PEi has *missed* a synchronization point.

A new mechanism is required to ensure that all the PEs in a barrier group have seen the completion of a barrier and that no PE which was involved in the barrier changes its output (barrier mask) during this interval. There has to be a *memory element* in the barrier logic to hold the status of the output of the barrier tree until all the PEs have seen the barrier. Thus, this memory element must be reset when all the PEs have seen the barrier. The two logic levels of the 'Strobe' signal could be used to differentiate between the barrier synchronization request from a PE and the information that a PE has seen the completion of the current barrier.

Therefore a *sequential circuit* is required for implementing the dynamic barrier synchronization for a cluster of computers using this parallel port interface. This sequential circuit can be a clocked synchronous circuit or it can simply be an event driven asynchronous circuit.

On the basis of these results, the implementation of dynamic barrier synchronization was modified as shown in Figure 4.2.

This modification basically takes into account the use of an additional 'Strobe' signal. Barrier requests (Strobe = 1) with barrier mask bits *sets* the memory element to signal the completion of a barrier whereas barrier-seen (Strobe = 0) with barrier mask bits *resets* the memory element to signal that all the PEs in the group have seen the completion of the barrier. For PEk's barrier tree, the Strobe signal from PEi is ANDed together with the kth barrier mask bit of PEi before it is ORed to the ith mask bit of PEk. The barrier-seen tree or anti-barrier tree is similar to the barrier tree except for the inversion of Strobe signal.

4.2.3 Barrier/Anti-Barrier Sequence

The modifications in the implementation of the dynamic barrier synchronization mechanism also resulted in modifications to the sequence of operations required to perform a barrier synchronization.

The sequence of operations for a barrier synchronization in PAPERS is:

1. Same as step [1] of 3.6.2. Internal to each PC/Workstation a bit mask is created or maintained such that the bit position corresponding to each processor that will participate in the barrier is a 1, and the positions corresponding to the other processors are all 0.

2. The Strobe Signal is appended to the barrier mask.

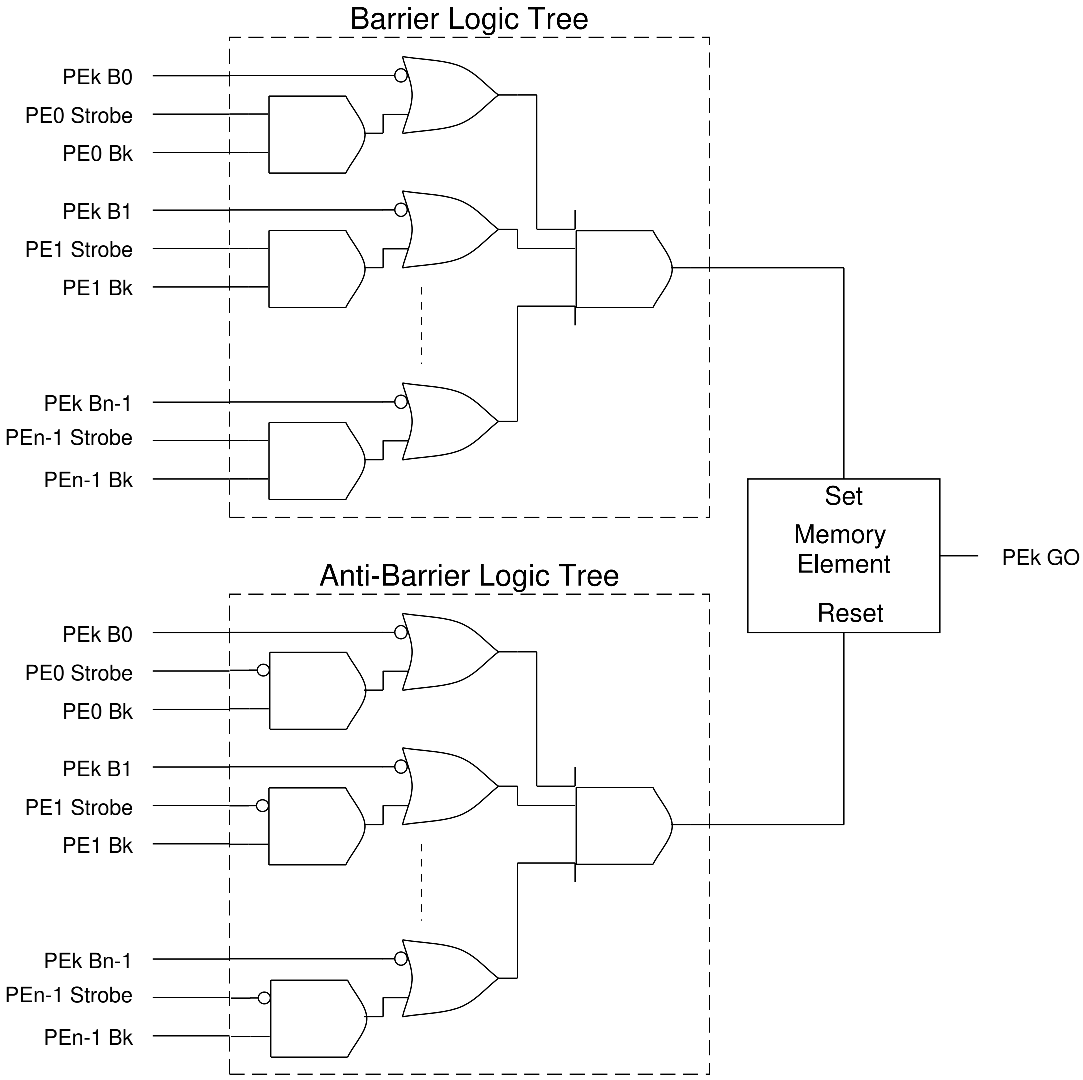

Figure 4.2 Barrier Logic of PAPERS

3. The PC/Workstation executes a write to the register controlling the output data bits of the printer port. The barrier mask and Strobe output signals are latched by the printer port. The barrier hardware recognizes the barrier synchronization request by the logic 1 on Strobe signal. Strobe bit and barrier mask bits are routed to this processor's trees and to the corresponding processor's trees.

4. After a small delay the barrier tree yields a single bit result indicating whether all the PEs in the group specified by the barrier mask have also sent a barrier synchronization request (have performed steps [1] through [4]). It is up to each

processor to know which processors it should synchronize with (as suggested in step [1]); if these masks are inconsistent, strange behavior can result.

5. If the output generated by the barrier tree is 1 (true), the memory element is set to 1 and indicates that the barrier is completed. This memory element in turn drives a signal which is connected to one of the input pins (hereafter referenced as RDY) of the printer port.

6. The PE (PC/Workstation) performs a read from the register connected to the input signals of the printer port. If the RDY signal is 0 (i.e the barrier not completed) then the PE repeats this step (reads and test the register) until the RDY signal is 1. In fact, a PE spin-locks waiting for the RDY signal.

7. The PE reads the flag vector from the input port by executing another read operation (after seeing the completion of the barrier) on input register.

8. The PE notifies the barrier logic that it has seen the barrier by outputting the same barrier mask bits with an inverted Strobe (a logic 0).

9. After a small delay, the anti-barrier tree yields a single bit result indicating whether all the PEs in the group specified have also indicated that they have seen the completed barrier.

10. If the signal generated by the anti-barrier tree is 1 (true), the memory element is reset to 0 to indicate that all the PEs which were part of the previous barrier have seen the completion of that barrier. This is an indication to the PE that it can initiate a new barrier synchronization request.

11. Before initiating a new barrier synchronization request, a PE performs a read on its input port. If the RDY signal is 0, then it can send a barrier request by repeating steps [1] onwards. If RDY is a 1, then a PE must keep reading the input port until the RDY becomes 0; only then it can send a new barrier synchronization request.

From the barrier/anti-barrier sequence, it is clear that a single partitionable dynamic synchronization will require 5 port references. Step [7] can be eliminated as the PE can extract the flag vector from the input data read during step[6]. However, step[7] has more to do with the noise problems on the signal, and is dealt with in chapter 6. With simple modifications to the barrier circuit, the complete barrier synchronization can be accomplished using just 2 port references. The insight about these modifications was gained only after implementing PAPERS0, therefore this issue is dealt with in chapter 6.

Due to the limitations on output and input signals available using the printer port, the above barrier/anti-barrier sequence is valid only for 4 or fewer PEs. PAPERS can be implemented for a higher number of PEs by having latches within the barrier hardware to hold the barrier masks and input flags. Multiple write operations will be required to update the barrier mask and, similarly, multiple read operations will be needed to read the input flags. The barrier/anti-barrier sequence remains the same, except for step [3] and [7]. These steps will require multiple writes and multiple reads on the printer port.

4.3 Interrupt Mechanism in PAPERS

Execution of parallel programs on a cluster of workstation requires some mechanism to handle system level operations, exceptions and errors. This is in addition to the synchronization and data communication capability provided by the network and dynamic barrier synchronization hardware.

Most of the applications in MIMD execution mode utilize the data-parallel model where each PE locally maintains its data. SIMD execution on a cluster of workstations require the PEs have same program code, while the VLIW execution model yields different program code on each PE with either a complete or partial data set. To execute a parallel program from one PE's console (or, equivalently from a `rlogin` to one PE), the PE has to initiate execution of the code on each of the PEs involved in the program. As mentioned earlier, PAPERS was originally expected to use an

Ethernet network in a UNIX environment for communication. UNIX's `'rsh'` command provides a simple way for starting the parallel program on all the PEs from a single PE. The initial PE uses `'rsh'` with appropriate parameters to sequentially start the parallel program on each of the other PEs. This mechanism restricts the cluster to execute only one parallel program at a given time. This limitation was considered to be insignificant for the first implementation of PAPERS. In fact, PAPERS can support multiple parallel programs to execute simultaneously on one cluster with modifications in the scheduler of standard Unix.

A correct startup requires that a PE involved in the parallel code must have all the other PEs of the same group included in the barrier mask. Starting the program execution sequentially using the `'rsh'` command does not guarantee a correct *initial synchronization* between the PEs as the mask bits are in a unspecified state at the start of the code. Therefore, in order to achieve initial synchronization among the participating PEs, a mechanism is required in addition to the normal barrier synchronization logic.

As explained in section 4.2, an error in a barrier mask pattern of one PE can result in an improper program execution. The possibility of this happening cannot be ruled out as a single error in reading the flag vector is enough to generate an incorrect barrier mask. This can even halt the parallel code, as one or more PE may go into an infinite wait for barrier synchronization. Therefore, a mechanism for recovering from such errors must be provided by PAPERS. This error recovery mechanism must have priority over the barrier synchronization logic.

The operating system on one PE may detect an anomaly in program execution, but then it must inform the other PEs about the error. Irrecoverable errors like divide by zero, disk read error, etc. are some of the errors which may require special handling. This also requires a mechanism of higher priority than the barrier synchronization logic.

Although it was not realized until the completion of PAPERS0, PAPERS can provide data communication among the PCs/Workstation connected in the cluster.

Therefore, an additional data communication network is not essential for executing parallel code. In order to execute parallel programs on clusters which do not have a data communication network, system commands must be sent through PAPERS. System commands must be distinguishable from the data communication performed by the parallel program. This cannot be done by only having barrier synchronization logic in PAPERS.

If system level communication can be distinguished from the communication done by a parallel program, then the code as well as data can be sent through PAPERS by one PE to all other PEs before starting the program execution. Therefore, data and program reside permanently on one PE, which reduces the data storage requirement for other PEs.

From previous paragraphs, it is clear that PAPERS requires some higher priority mechanism in addition to barrier synchronization logic for proper operation. This high priority mechanism is used to interrupt other PEs during their normal execution and hence is called the *Interrupt Mechanism* of PAPERS. (even if it does not generate a true hardware interrupt)

In a simple implementation, the interrupt mechanism can be an OR of interrupt request signals from all the PEs. An interrupt signal from one PE is seen by all other PEs.

As PAPERS supports partitioning of PEs into barrier groups, it is essential that a PE (PEi) gets an interrupt signal only if the interrupt is requested by a PE which is part of the same group as PEi. Interrupts are thus partitioned into the same subgroups as barriers; and barrier mask bits are used to enable or disable the interrupt requests from other PEs. This facilitates the concurrent execution of multiple parallel programs on a PAPERS cluster.

A partitionable interrupt mechanism can be implemented by replicating an AND-OR tree at each node of the PAPERS unit. The AND in the tree is used to enable the PEs from which a PE can receive interrupt. This is done by using the barrier mask bits. Two approaches can be used for an interrupt mechanism.

In the first approach, a PE decides *from* which PEs it will accept interrupts. Therefore, for the PEk interrupt tree, only the mask bits of PEk are used as shown in Figure 4.3.

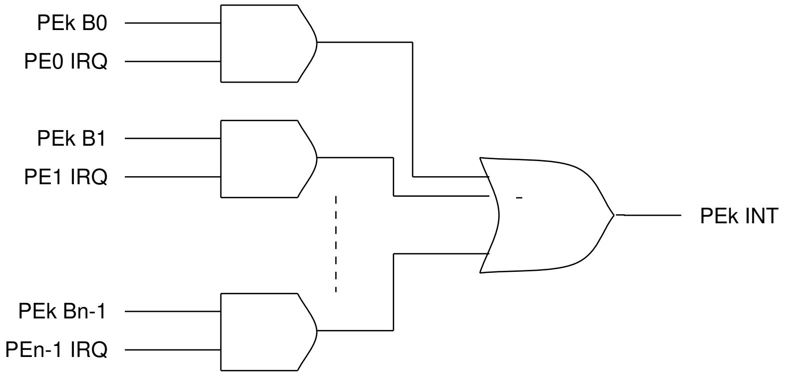

Figure 4.3 Interrupt Logic (Method 1) for PEk of PAPERS

In the second approach, a PE decides about the PEs it will send an interrupt *to*. In this way, a PE receiving the interrupt does not have control over which PEs may interrupt it. For the PEk interrupt tree, the interrupt signal from PEi is ANDed with the kth barrier mask bit of PEi as shown in Figure 4.4.

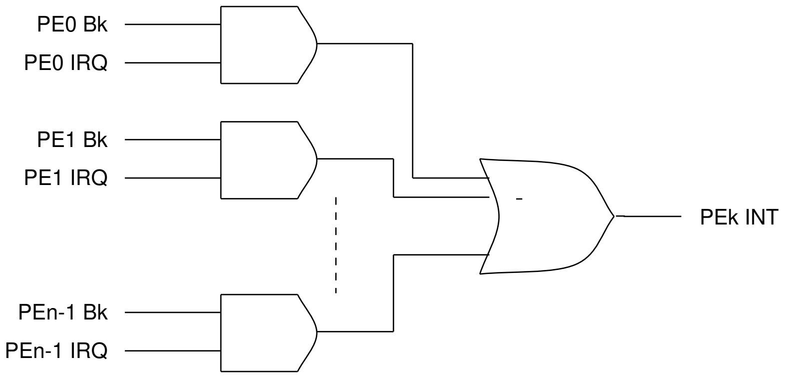

Figure 4.4 Interrupt Logic (Method 2) for PEk of PAPERS

Any of the above mentioned methods can be used to implement the interrupt mechanism for PAPERS. However, the second method is less prone to errors.

4.4 Block Diagram of PAPERS

From our discussion, it is clear that dynamic barrier synchronization logic and interrupt logic are the main components of the PAPERS hardware. The run-time partitionable dynamic barrier scheme is implemented by replicating the logic tree of Figure 4.2 at each PE (Node). Beside this, logic tree of either Figure 4.3 or Figure 4.4 implements the interrupt logic and is also replicated at each node. Therefore, PAPERS can be implemented by having an identical logic module for each PE.

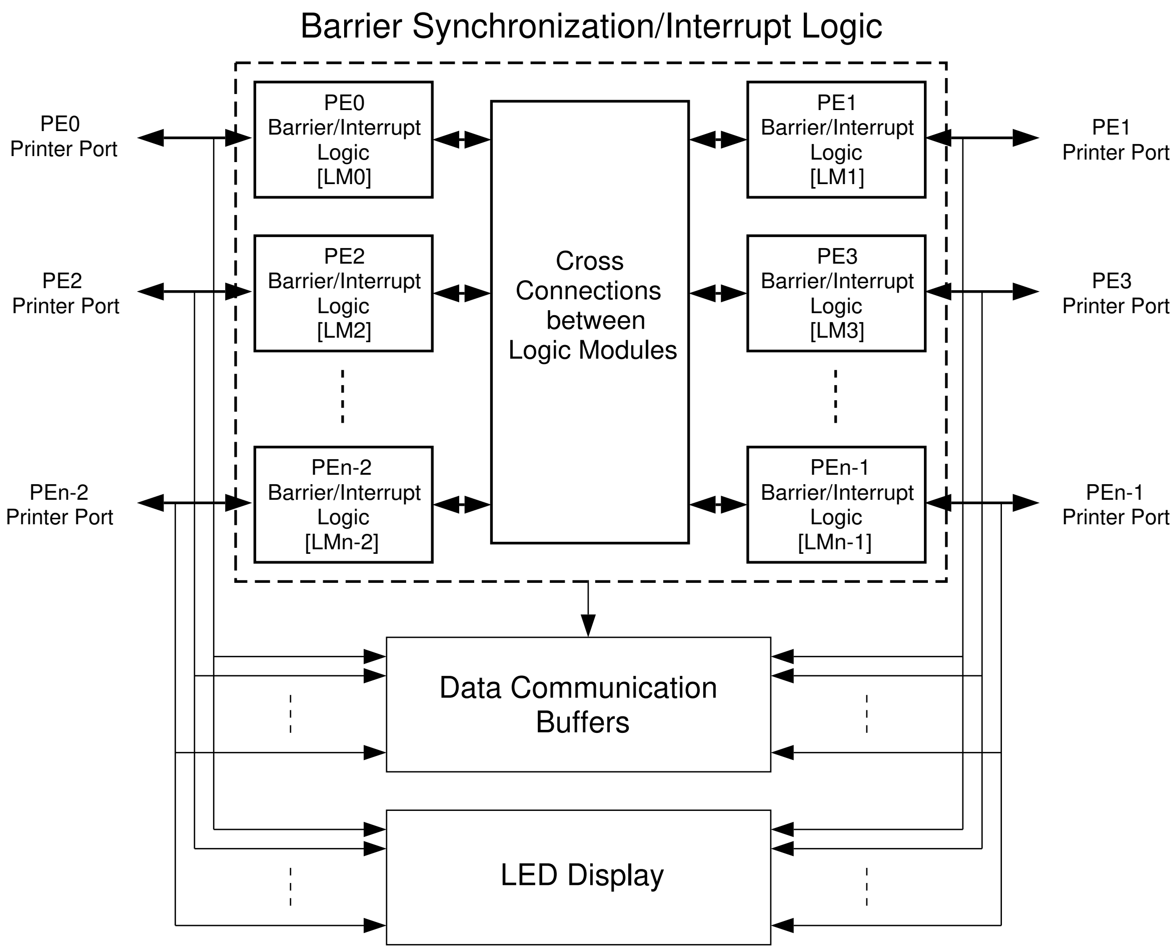

Figure 4.5 Block Diagram of PAPERS

Apart from the barrier synchronization and interrupt logic, PAPERS also provides data buffering for data/flags and has display drivers for a LED display. This display is used to present a visual status of PEs in PAPERS. Initially, there was no data buffering in the PAPERS design but the PAPERS0 implementation proved the requirement for it beyond any doubts and is dealt with in chapter 6. Hence, data buffering is part of all the PAPERS implementation.

Figure 4.5 shows the block diagram for the PAPERS unit with N processors. The basic architecture as shown in this figure is common to all of the PAPERS implementations.

PAPERS units can be fabricated in two ways:

1. Individual logic modules fabricated to be close to the PE (PC/Workstation), which are then connected to each other.

2. A central system where all the logic is fabricated in a single box. In this case, wires from the printer port of all the PEs come directly to the PAPERS unit and interconnections between logic modules are done within that single box.

The second approach is better for implementing PAPERS with 4 to 16 PEs. All the current implementations of PAPERS use the second approach for fabrication. In order to connect a higher number of PEs, a distributed fabrication approach may be suitable. For example, n PEs (where n is a fraction of the total PEs N) can connect to one logic unit and then these units can be connected together to form a N PE PAPERS system.

5. PAPERS0 IMPLEMENTATION

To keep the design simple and to minimize the fabrication and debugging efforts for implementing PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization), *PAPERS0*, the first working system, connects only 4 PCs. Most of the description given in this chapter is borrowed from, or derived from, the Purdue Technical Report describing PAPERS0 [28].

PAPERS0 provides the full functionality of the partitionable dynamic barrier mechanism described in the previous chapter. The following sections detail the design of PAPERS0.

5.1 Defining Input/Output Signals for PAPERS0

The following is the list of notations and description of signals required to implement the partitionable dynamic barrier mechanism and interrupt logic.

Barrier Mask (B0-B3) Outputs from PE. Each bit in the barrier mask of a PE corresponds to each PE connected to PAPERS0.

Barrier Sync. Request (S) Output from PE. A logic high is decoded by the PAPERS0's barrier logic as barrier synchronization request. A logic low is used to indicate that the PE has seen the completion of barrier synchronization.

Barrier Sync. Completed (GO) Input to PE. Driven by PAPER0's barrier logic, a logic high indicates the completion of the pending barrier; a logic low indicates that all the PEs in the barrier group have seen the completion of the barrier.

Data Output (D) Output from PE. The value of this signal is read by all other PEs - except the PE generating the signal. It is used by PEs to calculate the next barrier mask in dynamic partitioning of barrier groups.

Data Inputs (I0-I2) Inputs to PE. These signals are connected to the data outputs (D) from each of the other PEs. Thus, a PE can read a vector containing one bit of data from each of the other PEs in one read operation.

Interrupt Request (IR) Output from PE. A PE puts a logic high on this signal to request an interrupt. Interrupt takes precedence over the the normal barrier sequence.

Interrupt Pending (INT) Input to PE. A logic high indicates that at least one PE in the current partition has requested an interrupt.

Beside these signals, we have used 4 additional output signals. One signal (GI) is used to control interrupt mechanism, one (CE) is used to indicate that the PE is connected to the PAPERS0 unit and two signals (U0,U1) are used for status indication. Only one of these signal, (GI) is connected to the barrier logic in PAPERS0; the others are used for display only.

5.2 PE Interface to PAPERS0

PAPERS0 redefines the use of input and output pins of centronics printer. The following subsections provide the pin definitions for the physical connections and corresponding bits in control registers.

5.2.1 Physical Connectors

The PAPERS0 design uses 11 of the 12 available output lines and all 5 of the input lines on a standard uni-directional centronics printer port. A PE(PC/Workstation) connects to PAPERS0 by a standard PC printer cable. The pin/contact assignment in PAPERS0 for each of these lines is given in Table 5.1 and Table 5.2. Table 5.1 lists the pin numbers as they appear on the PE's DB25 connector. Table 5.2 lists the contact numbers for the signals as they appear on the 36-pin Centronics connector which connects to PAPERS0.

Table 5.1 DB25 Parallel Port Pin Assignments

Pin #	Std. Name	Use In PAPERS0
Pin 1	Strobe	U0 (User bit 0)
Pin 2	D0	D (Data Bit Value)
Pin 3	D1	
Pin 4	D2	IR (Interrupt Request)
Pin 5	D3	S (Barrier Sync. Request)
Pin 6	D4	B0 (Barrier Mask Contains PE0)
Pin 7	D5	B1 (Barrier Mask Contains PE1)
Pin 8	D6	B2 (Barrier Mask Contains PE2)
Pin 9	D7	B3 (Barrier Mask Contains PE3)
Pin 10	Ack	INT (Interrupt)
Pin 11	Busy	GO (Barrier Sync. Completed)
Pin 12	PE	I2 (PExI2 = PEyD such that $y = (x+3)\%4$)
Pin 13	SlctIn	I1 (PExI1 = PEyD such that $y = (x+2)\%4$)
Pin 14	AutoFD	U1 (User bit 1)
Pin 15	Error	I0 (PExI0 = PEyD such that $y = (x+1)\%4$)
Pin 16	Init	GI (GO Causes Interrupt)
Pin 17	Slct	CE (Connection Established)
Pin 18	Gnd	
Pin 19	Gnd	
Pin 20	Gnd	
Pin 21	Gnd	
Pin 22	Gnd	
Pin 23	Gnd	
Pin 24	Gnd	
Pin 25	Gnd	

Table 5.2 Centronics Connector Contact Assignments

Contact #	Std. Name	Use In PAPERS0
Contact 1	Strobe	U0 (User bit 0)
Contact 2	D0	D (Data Bit Value)
Contact 3	D1	
Contact 4	D2	IR (Interrupt Request)
Contact 5	D3	S (Barrier Sync. Request)
Contact 6	D4	B0 (Barrier Mask Contains PE0)
Contact 7	D5	B1 (Barrier Mask Contains PE1)
Contact 8	D6	B2 (Barrier Mask Contains PE2)
Contact 9	D7	B3 (Barrier Mask Contains PE3)
Contact 10	Ack	INT (Interrupt)
Contact 11	Busy	GO (Barrier Sync. Completed)
Contact 12	PE	I2 (PExI2 = PEyD such that $y = (x+3)\%4$)
Contact 13	SlctIn	I1 (PExI1 = PEyD such that $y = (x+2)\%4$)
Contact 14	AutoFD	U1 (User bit 1)
Contact 19	Ground	
Contact 20	Ground	
Contact 21	Ground	
Contact 22	Ground	
Contact 23	Ground	
Contact 24	Ground	
Contact 25	Ground	
Contact 31	Init	GI (GO Causes Interrupt)
Contact 32	Error	I0 (PExI0 = PEyD such that $y = (x+1)\%4$)
Contact 36	Slct	CE (Connection Established)

5.2.2 PE Port Bit Assignment

Although the parallel port hardware is not altered to work with PAPERS0, the parallel port lines are not used as they would be for driving a Centronics-compatible printer. Thus, it is necessary to replace the standard parallel port driver software with a driver designed to interact with PAPERS0.

There are three port registers associated with a PC parallel port. These ports have I/O addresses corresponding to the port base address (henceforth, called *PortBase* plus 0, 1, or 2). As mentioned earlier, typically, PortBase will be one of 0x378, 0x278, or 0x3bc.

The first port register at base address controls the 8 data pins on a printer port. The bit assignments for the first port register, PortBase + 0, are listed in Table 5.3. A logic 1 at each bit location correspond to a high level signal at the corresponding data pin. This register is used to send PAPERS0 the information used in each barrier synchronization. Notice that bit 1 is currently unassigned and is the only signal not used in PAPERS0.

Table 5.3 PortBase + 0, Bit Assignments

Bit	Name	Use In PAPERS0
bit 7	D7	B3 (Barrier Mask Contains PE3)
bit 6	D6	B2 (Barrier Mask Contains PE2)
bit 5	D5	B1 (Barrier Mask Contains PE1)
bit 4	D4	B0 (Barrier Mask Contains PE0)
bit 3	D3	S (Barrier Sync. Request)
bit 2	D2	IR (Interrupt Request)
bit 1	D1	0 (reserved for future use)
bit 0	D0	D (Data Bit Value)

The second port register, PortBase + 1, is connected to the input pins of the printer port. It is used to receive information from PAPERS0. Bit assignments for this register are given in Table 5.4. The arrangement of bits within this register is the result of the fact that PCs usually can generate an interrupt signal when Ack is set; the interrupt line must be the Ack signal. The three remaining contiguous bits of the register are thus designated as the data input from other PEs. This leaves bit 7 as the GO signal - the bit tested to determine if synchronization has been achieved. It happens that the sense of bit 7 is inverted on the port; the PAPERS0 hardware compensates for this so that a port read sees the GO bit as a 1 when the barrier has fired.

Table 5.4 PortBase + 1, Bit Assignments

Bit	Name	Use In PAPERS0
bit 7	Busy	GO (Barrier Sync. Completed)
bit 6	Ack	INT (Interrupt)
bit 5	PE	I2 (PExI2 = PEyD such that $y = (x+3)\%4$)
bit 4	SlctIn	I1 (PExI1 = PEyD such that $y = (x+2)\%4$)
bit 3	Error	I0 (PExI0 = PEyD such that $y = (x+1)\%4$)
bit 2	unused	
bit 1	unused	
bit 0	unused	

The third port register, PortBase + 2, controls the handshake pins on the printer port. It is used by PAPERS0 only for output bits that change value relatively rarely - the software does not access this register in the course of executing a typical barrier synchronization . In other words, this register is used for the *modal* information outlined in Table 5.5. Although this discussion refers to the signals as they are listed in Table 5.5, the port actually inverts the sense of bits 3, 1, and 0; compensation for this inversion is done (by XOR with 0xB) inside the lowest-level PAPERS0 port driver.

Table 5.5 PortBase + 2, Bit Assignments

Bit	Name	Use In PAPERS0
bit 7	unused	
bit 6	unused	
bit 5	unused	
bit 4	IntEn	IE (Interrupt Enable)
bit 3	Slct	CE (Connection Established)
bit 2	Init	GI (GO Causes Interrup)
bit 1	AutoFD	U1 (User bit 1)
bit 0	Strobe	U0 (User bit 0)

Three modal bits in port (CE,U0,U1), PortBase + 2, are not actually used by the logic in PAPERS0, but rather are used to drive an informational status display. The CE bit is used to indicate that the PAPERS0 hardware has been properly connected to the PE. The other two bits are *user-defined status* bits that can be used in any way desired, however, the suggested use is to encode the function that the PAPERS0 hardware is being used to implement. This use is summarized in Table 5.6.

Table 5.6 Meaning of U1 and U0 Signals

U1	U0	Meaning
0	0	PAPERS is not currently in use
0	1	PAPERS is being used to barrier synchronize
1	0	PAPERS is being used to transmit user data
1	1	PAPERS is being used by the operating system

5.3 PAPERS0 Hardware Overview

As described earlier, there are only two events in barrier synchronization sequence of PAPERS, namely, barrier synchronizarion completed and barrier seen by all PEs. These events can be handled easily by an event driven asynchronous sequential circuit. Hence, PAPERS0 hardware implemnts a simple asynchronous circuit.

PAPERS0 follows the barrier architecture defined in section 4.5. The logic modules (LM0-LM3 for PAPERS0) are implemented using AMD 22V10 PALs (Programmable Array Logic) to minimize the chip count as well as to reduce the wiring complexity. A total of 4 PALs are used, one for each PE.

The flag bits (D bits) from all the PEs were initially hardwired to each other, but a buffer, a single 74LS541, was added during the test and debug phase of the PAPERS0 (see Chapter 6).

74LS05, open collector inverters are used to drive LED display. Each PE has 10 LEDs, therefore the total number of buffer chips used is $\lceil 40/6 \rceil = 7$.

5.4 PAL for Barrier/Interrupt logic

The four PALs used in PAPERS0 have same internal logic but are connected differently. However, the differences in connections follow a simple pattern. The following description refers to the PAL for PEa with respect to PEb, PEc, and PEd. Given a PE number for a, the PE numbers for b, c, and d can be derived by: $b = (a+1)$ `Mod` 4, $c = (a+2)$ `Mod` 4, $d = (a+3)$ `Mod` 4. Externally, each PAL appears as shown in Figure 5.1. PALASM software was used to generate the fuse mask for the PAL from the logic equations.

5.4.1 Pinout

Figure 5.1 PAL Pin Layout

PAL22V10

CLK	1	24	VCC
PEaBb	2	23	PEaBa
PEaBc	3	22	BREG
PEaBd	4	21	PEaGI
PEbBa	5	20	PEaGO
PEcBa	6	19	PEdS
PEdBa	7	18	PEcS
PEaIR	8	17	PEbS
PEbIR	9	16	CLKRST
PEcIR	10	15	CLKSET
PEdIR	11	14	PEaINT
GND	12	13	PEaS

5.4.2 PALASM Code

```
;-------------------------- PIN Declarations ----------------------


PIN     1       CLK     COMBINATORIAL
PIN     2       PEaBb   COMBINATORIAL
PIN     3       PEaBc   COMBINATORIAL
PIN     4       PEaBd   COMBINATORIAL
PIN     5       PEbBA   COMBINATORIAL
PIN     6       PEcBa   COMBINATORIAL
PIN     7       PEdBa   COMBINATORIAL
PIN     8       PEaIR   COMBINATORIAL
```

```
PIN     9       PEbIR   COMBINATORIAL
PIN     10      PEcIR   COMBINATORIAL
PIN     11      PEdIR   COMBINATORIAL
PIN     12      GND
PIN     13      PEaS    COMBINATORIAL
PIN     14      PEaINT  COMBINATORIAL
PIN     15      CLKSET  COMBINATORIAL
PIN     16      CLKRST  COMBINATORIAL
PIN     17      PEbS    COMBINATORIAL
PIN     18      PEcS    COMBINATORIAL
PIN     19      PEdS    COMBINATORIAL
PIN     20      PEaGO   COMBINATORIAL
PIN     21      PEaGI   COMBINATORIAL
PIN     22      BREG    REGISTERED
PIN     23      PEaBa   COMBINATORIAL
PIN     24      VCC
NODE    1       GLOBAL

;---------------------- Boolean Equation Segment ------------------
EQUATIONS

PEaINT = ((PEaIR*PEaBa) +
  (PEbIR*PEbBa) +
  (PEcIR*PEcBa) +
  (PEdIR*PEdBa) +
  (BREG*PEaGI))

BREG = VCC
```

```
GLOBAL.RSTF = CLKRST

CLKSET = ((/PEaBa+(PEaBa*PEaS)) *
  (/PEaBb+(PEbBa*PEbS)) *
  (/PEaBc+(PEcBa*PEcS)) *
  (/PEaBd+(PEdBa*PEdS)) *
  /BREG)

CLKRST = ((/PEaBa+(PEaBa*/PEaS)) *
  (/PEaBb+(PEbBa*/PEbS)) *
  (/PEaBc+(PEcBa*/PEcS)) *
  (/PEaBd+(PEdBa*/PEdS)) *
  BREG)

/PEaGO = ((PEaINT*/((PEaIR*PEaBa) +
    (PEbIR*PEbBa) +
    (PEcIR*PEcBa) +
    (PEdIR*PEdBa))) +
  (/PEaINT*BREG))
```

5.4.3 Description

The equations described in the previous sections are in fact implementing the barrier logic tree of Figure 4.2 and the interrupt logic tree of Figure 4.4. However, the PEaGO signal is not directly connected to the memory element (BREG) of the barrier logic tree, and the interrupt tree is slightly modified to achieve better functionality.

The internal BREG signal is the memory element which gets set and reset asynchronously by the events *barrier completed* and *barrier seen by all* respectively. As described earlier, an additional signal, Strobe (S), is used with the barrier mask bits

(B0-B3) by the barrier logic to achieve proper functionality. Strobe high with barrier mask bits indicates the arrival of this PE at a barrier; Strobe low with the same barrier mask indicate to the barrier logic that the PE has seen the barrier completion.

The latching of the internal flip-flops of the 22V10 PAL cannot be controlled by an internaly derived signal, rather the clock inputs of all the flip-flops of the 22V10 are connected to the pin 1 of PAL. PRESET on the internal flip-flop is synchronous. Therfore, the output of the barrier tree (upper part) of Figure 4.2 is brought out of the PAL as CLKSET. The output of the antibarrier tree is connected to the CLKRST output and ineternally to the reset pins of the internal flip-flops. The CLKSET output is externally connected to the CLK input signal, and a logic 1 is hardwired to the input of BREG (the memory element). The completion of a barrier forces a 0 to 1 transition on the CLKSET signal, which causes a 1 to be latched in the BREG flip-flop. When all PEs involved in a synchronization have read their input data, the (asynchronous) reset of the BREG register is internally triggered by the CLKRST signal, which forces the flip-fliop (BREG) state to 0.

The interrupt signal PEaINT is generated by two separate types of events:

1. A PE determines that the other PEs should be informed of some event. It can interrupt any subset or all of the PEs by setting its IR bit (PEaIR).

2. If desired, the PAPERS0 barrier hardware can be set to interrupt the PE (PEa) when PAPERS0 determines that this PE's barrier has been satisfied. This response can be independently enabled/disabled by each PE setting its GI bit (PEaGI).

If any PEx asserts an interrupt request and PEa is contained in the barrier mask of the PEx, then the PEaINT bit will become 1. Notice that the PE requesting an interrupt will only interrupt itself if the corresponding bit is on in its barrier mask. If PEaGI is 1, then completion of a barrier is signaled by setting both the PEaINT and PEaGO bits to 1. If PEaGI is 0, then only the interrupt request can cause PEaINT to be 1. The equations for PEaINT and PEaGO signals implements the functionality described above. The difference between the internal BREG register and the PEaGO output signal is that it can change the meaning of the BREG bit. In essence, the PEaGO and PEaINT signals are really encoding a two bit PAPERS0 hardware state, as outlined in Table 5.7.

Table 5.7 Meaning Of PEaGO and PEaINT Signals

PEaGO	PEaINT	Meaning
0	0	No interrupt, synchronization not achieved
1	0	No interrupt, synchronization achieved
0	1	Interrupt for all PEs
1	1	Interrupt for synchronization achieved

5.5 Fabrication

The prototype PAPERS0 unit is housed in a natural finish red oak box that is 11.75" wide by 6" deep by 6" tall, with a simple 3" steel handle protruding by 1" on the left side (to aid in carrying the system for demonstrations at remote sites). The cover of the case is a simple piece of 0.25" thick oak, attached by velcro and perforated above the power supply to allow convection cooling. The AC cord enters the box from behind the power supply. In front of the circuit card, rear-mounted on the front panel, is an array of LEDs used as a status display for the PEs connected to PAPERS0.

Inside the box, there is one 4" by 6" *wire-wrapped card* containing the PALs and LED driving circuitry and a 5 volt linear power supply (although a maximum of less than 1.5 amps is needed, we used a supply rated at 3 amps). Behind the circuit card on the back of the box are four panel-mounted Centronics connectors - so that the cables used to connect PEs to PAPERS0 are standard PC parallel printer cables. Ribbon cables connect the circuit board to the the centronics connectors. One end of each ribbon cable is crimped to the centronics connector while the other end is soldered to an 18 pin single-in-line (SIL) connecto on the circuit board.

Strictly speaking, there is no need to have any display connected to the PAPERS0 hardware. Indeed, eliminating the display can greatly simplify the hardware because it eliminates the need for LED drivers and perhaps even eliminates the separate power supply (the PALs might be powered by the combination of 7805 5V regulator and a simple 9V-12V AC-adaptor). However, PAPERS0 is a research prototype: the LEDs make it a lot easier to see what is happening... and to debug the system.

Table 5.8 Front Panel LEDs

Display	Position	Label On The PAPERS0 Unit	PAPERS0 Signal
Green	LED 5	PE Connection Established	CE
Green	LED 4	Interrupt Request	IR
Green	LED 3	User-defined Status Bit 1	U1
Green	LED 2	User-defined Status Bit 0	U0
Green	LED 1	Data Bit Value	D
Green	LED 0	Barrier Sync. Request	S
Amber	LED 3	Barrier Mask Contains PE3	B3
Amber	LED 2	Barrier Mask Contains PE2	B2
Amber	LED 1	Barrier Mask Contains PE1	B1
Amber	LED 0	Barrier Mask Contains PE0	B0

The prototype LED display consists of 40 LEDs arranged in 4 columns, each column representing the status of one PE. These columns are numbered in decreasing order from left to right (as the LEDs are normally viewed), i.e., PE3 PE2 PE1 PE0. The signal descriptions are given in Table 5.8. Notice that none of the LEDs displays a derived signal - this is because the two derived signals change value only momentarily, so fast that the state change would not be perceptible.

5.6 Schematics

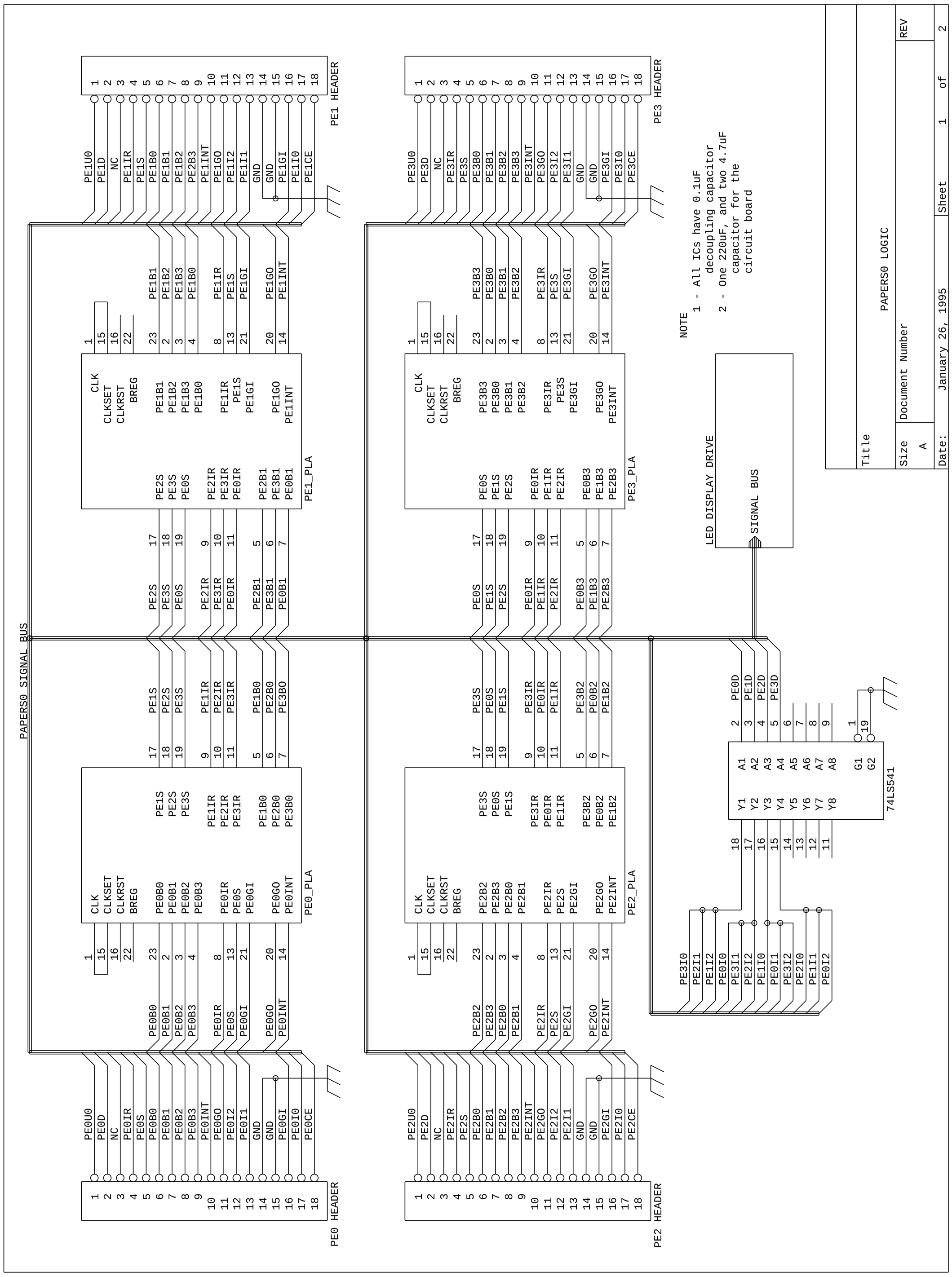

Figure 5.2 PAPERS0 Schematic 1

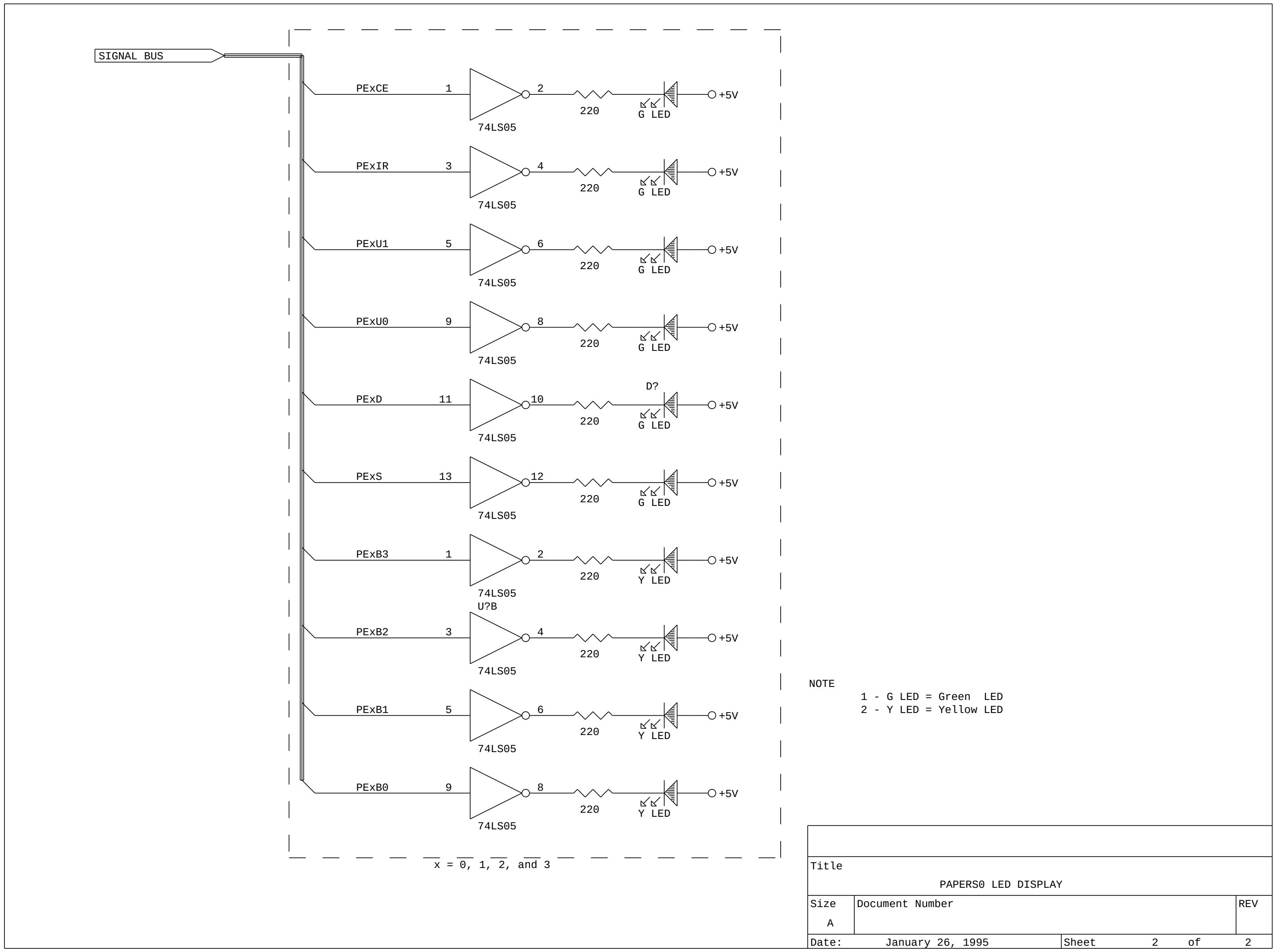

Figure 5.3 PAPERS0 Schematic 2

6. RESULTS FROM PAPERS0 IMPLEMENTATION

Although, PAPERS0 was primarily designed and fabricated to test the partitionable dynamic barrier synchronization mechanism, the low-latency barrier synchronization we achieved through PAPERS, and our discovery that PAPERS can be used for arbitrary low-latency communications, led us to conclude that *PAPERS is an efficient way to execute fine-grain parallel programs on a cluster of computers.*

The debugging and testing of PAPERS0 gave us insights about the mistakes we made in implementing PAPERS, not just restricted to logical design flaws, but also fabrication follies. However, our experience with PAPERS0 was very beneficial in the implementation of the next five PAPERS prototypes, including TTL_PAPERS [29], PAPERS1 and 8-PE PAPERS.

6.1 Performance of PAPERS0

After fabricating PAPERS0, a software library was developed for the C language executing under the Linux OS [28]. These library routines provide the software support required by applications to use PAPERS for barrier synchronization and data communication.

As mentioned in section 4.1.2, direct access to the parallel port control registers is the fastest method for controlling the PAPERS interface. Therefore, the library routines access the printer port registers directly using assembly language constructs, eliminating the overheads of system calls.

Theoretically, this should have allowed us to access the port registers at a speed comparable to the memory fetch time. However, the printer port on the IBM PC and compatibles (the computer used in our experiments) do not perform at the same speed as memory. Most systems apparently inserts wait states for all I/O references to comply with the ISA bus specifications and to ensure that the signal transitions would not outrun a centronics compatible printer. Depending upon the host processor, the CPU clock rate, and the printer port implementation, the minimum time between two successive port operations can vary from 1-5 μs, which translates into 200,000 to 1 million port operations per second.

Although the delay within the PAPERS0 barrier logic is 7.5-25 nanoseconds (depending on the PLA used, 25 nanoseconds in our implementation), long cables used to interface computers to the PAPERS0 introduce a delay of 50-150 nanoseconds. Therefore, the total delay involved in sending a signal to PAPERS0 and receiving the result is roughly 125-325 nanoseconds.

Table 6.1 gives some benchmark figures for the basic PAPERS operations on a 33 MHz 386 cluster supporting an I/O speed of 800,000 port operations per second.

Table 6.1 Timing of Key Operations of PAPERS0

Operation	PAPERS0 Time/Speed
Dynamic Barrier Sync. with arbitrary PEs	11.2μs
Multiple Broadcast communication operation	25.0μs 40 Kbyte/s
`ANY` conditional	11.2μs
Random Communication 0f One Byte (Per PE)	92.0μs 10.86 Kbyte/s

These timings, although much slower then the theoretical limits, are still orders of magnitude faster than any synchronization mechanisms available on a cluster of computers. In fact, these latencies are lower than those provided by most parallel computers.

6.2 SIMD/MIMD/VLIW Code Execution on Cluster of Workstations using PAPERS

Traditional methods of executing parallel programs on a cluster of computers are restricted to the message-passing software constructs using standard data communication networks like Ethernet, HIPPI, FDDI and ATM. PVM (Parallel Virtual Machine) is the most common library for such parallel programming model [30]. However, these communication networks are designed for block data transfers, they suffer from high startup latency, usually in 1000s of micro-seconds. Even, ATM has a startup latency of more than 1000μs [12]. Due to the high latency of data communication, traditional methods of parallel program execution on a cluster of computers are restricted to coarse-grain MIMD and SPMD parallel applications only.

Low latency communication and fast barrier synchronization is the key to fine-grain MIMD code execution, and for emulating SIMD and VLIW models on a MIMD machine. A cluster of computers connected through a PAPERS unit provides low latency communication, as well as fast synchronization (see Table 6.1). Therefore, a PAPERS-based cluster of computers is capable of fine-grain MIMD and SPMD code execution. In fact, a PAPERS cluster (an inherent MIMD system) can also be used for efficient emulation of SIMD and VLIW codes [31].

6.3 PAPERS as a Data Network

A barrier synchronization on a PAPERS0 cluster is accompanied by 1 bit flag output and 3-bit flag vector input. The exchange of flags is accomplished by using the `waitvec()` [28] software routine of the PAPERS library. This flag vector is used by the PEs to dynamically partition the barrier groups on the basis of the result obtained by evaluating conditional statement. Thus, data can be exchanged among the PEs belonging to same barrier group at each synchronization point.

After implementing PAPERS0, we realized that these flags can indeed be *arbitrary data bits* as long as the PEs are not executing a barrier synchronization operation for partitioning of the barrier group. Data transfer at each barrier synchronization point provides a *low latency synchronous multi-broadcast capability* to PAPERS. Hence, PAPERS can be used for low latency data communication. An arbitrary amount of data can be exchanged through PAPERS by executing a series of barriers, one data bit sent by each processor.

The data communication bandwidth of PAPERS is dependent upon the barrier synchronization speed of PAPERS, which in turn depends upon the speed of the printer port. The effective communication bandwidth of PAPERS0 is 4 * *(Barrier Sync. speed) bits/sec* due to its single-bit multi-broadcast capability.

The PAPERS0 data bandwidth of 40 Kbytes/s may seem very low as compared to the 10 Mbits/s (1.25 MByte/s) bandwidth of an Ethernet network. However, PAPERS0 is faster for small data transfers (up to a few hundred bytes), because data communication through PAPERS0 do not suffer from the *startup latency (packetizing latency)* and *data collision* problems of an Ethernet network.

Because, the PAPERS unit provides a data communication capability, parallel programs can be executed on a group of computers that do not have another data communication network but are simply connected through the PAPERS unit.

6.4 Scheduling Data Network Accesses using PAPERS

The effective bandwidth of bus-based networks like Ethernet is greatly reduced by collision. This effect is compounded while executing a parallel code on a cluster connected via Ethernet because multiple PEs tend to access the network at the same time.

PAPERS can be used to get the vote from all the computers that want to access the network and then statically schedule the network accesses [31]. This eliminates the possibility of data collision on the network as only one computer has access to the network at a given time. Therefore, Ethernet, or for that matter any network, can be used at its maximum efficiency by scheduling network accesses through PAPERS.

6.5 Minimizing Port Operations per Barrier

PAPERS0 implements the barrier sequence described in section 4.2.3, which requires a minimum of 5 port operations to perform one barrier synchronization operation. Step [6] and step [7] in section 4.2.3 can be combined together by using a delay element on the RDY line (to let the data settle) to reduce the number of operations per barrier to 4. 4/5 port operations per barrier are the result of reading the input flag vector which is required to perform a partitionable barrier synchronization. Similarly data communication also requires 4/5 port operations for one data bit exchange.

If the barrier group is fixed (no partitioning) and there is no data communication associated with the barrier synchronization then the barrier and anti-barrier can be used to achieve *two* barrier synchronizations by keeping track of the last operation (Strobe bit value). Hence, simple barrier synchronization can be performed with only 2 port operations using the PAPERS0 hardware.

The requirement for a complete barrier/anti-barrier sequence arises from the fact that a PE can change its data value after a barrier and another PE can miss reading the correct data. This problem can be solved by latching the input data at the completion of each barrier. Thus, if each logic module latches the data at the completion of each barrier then a fully partitionable barrier synchronization and arbitrary data communication can be done with only 2 port operations. This fact was realized only after implementing PAPERS0, and is used by PAPERS1 (an enhanced implementation of PAPERS) to achieve high data bandwidth and faster synchronization.

6.6 Need for Data Buffers in PAPERS

During the debugging phase we encountered a lot of problems stemming from the printer port driving long cables at high speeds using TTL logic levels.

10 foot long standard printer cables were used to connect PAPERS0. Initially, the data bit (D) coming to the PAPERS0 was just hard-wired (without buffer) to the input lines (I0-I2). Thus, in effect, a single port pin was driving 40 feet of cables.

This slowed signal transitions on this line and also accounted for the fact that the GO signal could outrun the I0-I2 lines. This caused the PE to get a valid GO signal before the data lines stabilized, resulting in wrong data inputs. This problem was partially remedied by adding the 74LS541 TTL buffer to drive the lines I0-I3. For error free data inputs, PAPERS0 performs an additional (step [7] section 4.2.3) read after getting a valid GO signal, therefore slowing data communication from 4 to 5 port operations.

Since that time, all the implementations of PAPERS have taken into account the driving capability of the printer port, and provide data buffers to drive the cables.

6.7 Fabrication Blues

A number of lessons were learned about fabrication after building PAPERS0.

- 10 foot long cables were a cause of great trouble in PAPERS0, therefore it was decided to use shorter cables whenever possible in subsequent implementations of PAPERS.

- Another cable-related problem was due to the use of connectors on the PAPERS0 box, adding two more contacts between the circuit board and the PE, as well as increasing the cost of the system. Reliability became a problem due to broken solder connections between ribbon cables and the SIL connectors on the circuit board inside the PAPERS0.

  It was decided to do away with the connectors, therefore, in all the other PAPERS implementations the cable is directly soldered to the circuit board. For larger PAPERS configurations, PC-board mounted connectors may be used.

- The large number of LEDs (40) did help in the debugging phase, and provides a good display, but increases the number of components, wiring complexity, and the cost of PAPERS0. Weighing the pros and cons, all the subsequent PAPERS versions use a single LED for each PE.

- PAPERS0 uses a $40 linear power supply. It is a waste of money in the sense that we can get equal or even better performance by using a 7805 5V regulator with a cheap 9-12V AC adaptors. Total cost for 7805 and AC adaptor is less than $10 Besides PAPERS0 and PAPERS1 (another implementation), all the other PAPERS systems use 7805 regulators with AC adaptors.

7. CONCLUSION

In this thesis, we have discussed the evolution, design, construction, and performance of PAPERS0 – the first PAPERS prototype. Although originally intended to be just a testbed for the new dynamic barrier architecture, PAPERS0 has proved far more useful:

- PAPERS0 demonstrates that the new dynamic barrier mechanism works well and provides very low latency – even across a cluster of personal computers
- PAPERS0 showed that low-latency data communication as a side-effect of a barrier synchronization is both efficient and useful
- PAPERS0 exposed a variety of implementation issues that the theory had never noticed; for example, the need for parallel interrupt handling

In summary, PAPERS0 showed that fine-grain parallel computing on a cluster is practical, and gave us good directions to pursue for making fine-grain, mixed-mode, cluster computing a viable alternative to building customized parallel supercomputers.

Current work focuses on these new directions. For example, the one-bit multi-broadcast communication of PAPERS0 has led to a wide range of other aggregate communication mechanisms being developed in the later prototypes – along with the software libraries and compilers to use them. Methods to effectively scale the design to large numbers of processors (e.g., 128 or more) have also become a major research focus.

Finally, we have taken advantage of the surprisingly good performance obtained by creating a still simpler PAPERS design (TTL_PAPERS [29]), which can be easily and cheaply replicated by other researchers. Both the hardware design and support software have been made available as a full public domain release on the WWW at URL `http://garage.ecn.purdue.edu/~papers/`

LIST OF REFERENCES

LIST OF REFERENCES

[1] M. T. O'Keefe and H. G. Dietz, "Hardware Barrier Synchronization: Static Barrier MIMD (SBM)," in *Proceedings of the 1990 International Conference on Parallel Processing*, vol. I, (St. Charles, Illinois), pp. 35–42, August 1990.

[2] M. T. O'Keefe and H. G. Dietz, "Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM)," in *Proceedings of the 1990 International Conference on Parallel Processing*, vol. I, (St. Charles, Illinois), pp. 43–46, August 1990.

[3] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors; part i," Tech. Rep. TR-EE 91-9, Purdue University, Mar. 1994.

[4] H. F. Jordan, "A special purpose architecture for finite element analysis," in *Proceedings of the International Conference on Parallel Processing*, pp. 263–266, 1978.

[5] S. F. Lundstorm and G. H. Barnes, "A controllable mimd architecture," in *Proceedings of the International Conference on Parallel Processing*, pp. 19–27, 1980.

[6] R. Gupta, "The fuzzy barrier: A mechanism for the high speed synchronization of processors," in *Third Int. Conference on Architectural Support for Programming Languages and Operating Systems*, (Bostan, MA), pp. 54–63, April 1989.

[7] T. M. Corporation, *Connection Machine CM-5 Technical Summary*. Thinking Machines Corporation, Cambridge, Massachusetts, November 1992.

[8] M. Philippsen, T. Warschko, W. F. Tichy, and C. Herter, "Project triton: Toward improved programmablity of parallel machines," in *26th Hawaii International Conference on System Sciences*, vol. I, pp. 192–201, January 1993.

[9] M. Auguin and F. Boeri, "The OPSILA Computer," *Parallel Languages and Architectures*, pp. 143–153, 1986.

[10] H. Kasharm, H. Honda, A. Mogi, and et al., "A Mult-Grain Parallelizing Compilation Scheme for OSCAR (Optimally Scheduled Advanced Multiprocessor)," *Languages and Compilers for Parallel Computing*, pp. 283–297, 1991.

[11] C. R. Inc., *Cray T3D System Architecture Overview.* Cray Research Inc. ,Mendota Heights, Minnesota, September 1993.

[12] M. Lin, J. Hsieh, D. Du, J. Thomas, and J. MacDonald, "Distributed network computing over local atm networks," in *Supercomputing 94*, pp. 154–163, Nov 1994.

[13] C. Huang, E. P. Kasten, and P. K. McKinley, "Design and implementation of multicast operation for atm-based high performance computing," in *Supercomputing 94*, pp. 164–173, Nov 1994.

[14] C. A. Thekkath, H. M. Levy, and E. D. Lazowska, "Efficiant support for multicomputing on atm networks," Tech. Rep. TR 93-04-03, University of Washington, Seattle, Washington, April 1993.

[15] R. Fatoohi and S. Weeratunga, "Performance evaluation of three distributed computing enviroment for scientific applications," in *Supercomputing 94*, pp. 400–409, Nov 1994.

[16] K. Chatengmera, D. Cheng, R. Fatoohi, and et al., "Nas experience with a prototype cluster of workstation," in *Supercomputing 94*, pp. 410–419, Nov 1994.

[17] M. T. O'Keefe, *Barrier MIMD Architecture Design and Compilation.* PhD thesis, Purdue University, W. Lafayette, Indiana, Aug 1990.

[18] T. Schwederski, W. G. Nation, H. J. Siegel, and D. J. Meyer, "The implementation of the pasm control hierarchy," in *Proc. of Secong Int'l Conference on Supercomputing*, vol. I, pp. 418–427, 1987.

[19] H. G. Dietz and T. Schwederski, "Extending Static Synchronization Beyond SIMD and VLIW," Tech. Rep. TR-EE 88-25, Purdue University, June 1988.

[20] H. G. Dietz, T. Schwederski, M. T. O'Keefe, and A. Zaafrani, "Extending Static Synchronization Beyond VLIW," in *Supercomputing 89*, (Reno), pp. 416–425, Nov 1989.

[21] L. C. Eggebrecht, *Interfacing to the IBM Personal Computer.* SAMS, 1990.

[22] T. I. Inc., *Third Generation TMS320 User's Guide.* Texas Instruments Inc., 1988.

[23] A. M. Devices, *Am29050 Microprocessor.* Advanced Micro Devices, 1991.

[24] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors," in *Proc. of International Conference of Parallel Processing*, vol. II, (St. Charles, IL), Aug. 1994.

[25] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors; part ii," Tech. Rep. TR-EE 91-10, Purdue University, Mar. 1994.

[26] M. A. Nichols, H. J. Siegel, and H. G. Dietz, "Data management and control-flow aspects of an simd/spmd parallel language compiler," in *Frontiers' 90: The Third Symposium on the Frontiers of Massively Parallel Computation*, pp. 397–406, October 1990.

[27] M. Sargent and R. L. Shoemaker, *The IBM PC From The Inside Out.* Addison-Wesley Publishing Company, 1986.

[28] H. G. Dietz, T. Muhammad, J. B. Sponaugle, and T. Mattox, "PAPERS: Purdue's Adaptor for Parallel Execution and Rapid Synchronization," Tech. Rep. TR-EE 91-11, Purdue University, Mar. 1994.

[29] H. G. Dietz, T. M. Chung, T. I. Mattox, and T. Muhammad, "Ttl_papers: An implementation of purdue's adapter for parallel execution and rapid synchronization," *submitted to 24th IEEE International Conference on Parallel Processing*, Aug. 1995.

[30] V. S. Sunderam, "Pvm: A framework for parallel distributed computing," in *Concurrancy: Parctice and Experience*, vol. 2, pp. 315–339, Dec 1992.

[31] H. J. Dietz, W. E. Cohen, T. Muhammad, and T. I. Mattox, "Compiler techniques for fine-grain execution on workstation cluster using papers," in *7th Annual Workshop on Languages and Compilers for Parallel Computing (also to appear as a book chapter from Springer Verlag)*, (Cornell University), Aug. 1994.