1-1-1990

# Hardware Barrier Synchronization: Static Barrier MIMD (SBM)

Matthew T. O'Keefe
*Purdue University*, okeefem@ecn.purdue.edu

Henry G. Dietz
*Purdue University*

# Hardware Barrier Synchronization: Static Barrier MIMD (SBM)

Matthew T. O'Keefe
Henry G. Dietz

TR-EE 90-8
January, 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

# Hardware Barrier Synchronization: Static Barrier MIMD (SBM)

*Matthew T. O'Keefe and Henry G. Dietz*

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
*January 1990*
okeefem@ecn.purdue.edu

*ABSTRACT*

In this paper, we give the design, and performance analysis, of a new, highly efficient, synchronization mechanism called "Static Barrier MIMD" or "SBM." Unlike traditional barrier synchronization, the proposed barriers are designed to facilitate the use of static (compile-time) code scheduling for eliminating some synchronizations. For this reason, our barrier hardware is more general than most hardware barrier mechanisms, allowing any subset of the processors to participate in each barrier. Since code scheduling typically operates on fine-grain parallelism, it is also vital that barriers be able to execute in a small number of clock ticks.

The SBM is actually only one of two new classes of barrier machines proposed to facilitate static code scheduling; the other architecture is the "Dynamic Barrier MIMD," or "DBM," which is described in a companion paper[1]. The DBM differs from the SBM in that the DBM employs more complex hardware to make the system less dependent on the precision of the static analysis and code scheduling; for example, an SBM cannot efficiently manage simultaneous execution of independent parallel programs, whereas a DBM can.

**Keywords:** Synchronization hardware, barrier synchronization, static barrier MIMD (SBM), VLIW, performance analysis.

1. The companion paper is "Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM)," also submitted to ICPP '90. So that these two papers can be reviewed independently, some overview material appears in both papers; this redundancy will be removed if both papers are accepted.

## 1. Introduction

Barrier synchronization is an important mechanism for coordinating parallel processes. For this reason, many research efforts have focused efficient implementations in both hardware [Lund80], [Poly88], [Gupt89a] and software [ArJo87], [Luba89], [Broo86], [HeFM88]. Other research efforts [Call87] considered minimizing the number of barrier synchronizations required in scheduling nested loop structures on parallel machines. In this paper, several new designs for fast barrier synchronization, exhibiting a range of cost/performance tradeoffs, are described. The new hardware implements a generalized barrier synchronization mechanism, whereby a barrier can be placed across any subset of the processors.

The new barriers execute in a very small number of clock cycles, and the resulting fine-grain mechanism may potentially replace the more common directed (e.g., producer/consumer) synchronization primitives found in most parallel architectures today. Machines that implement these barriers are referred to as *barrier MIMD (Multiple Instruction Stream, Multiple Data Stream) architectures*. Results from analytic and simulation models [OKDi89], as well as from scheduling synthetic benchmarks [ZaDO90], have shown the effectiveness of the new barrier synchronization designs.

A barrier is a synchronization point. In the old definition of barriers, *all* typically meant every physical processor in the machine. In a barrier MIMD, this condition is relaxed to include only those processors participating in the current barrier. A processor typically performs the following three steps at a barrier synchronization point:

[1] Marks itself as present at the barrier.

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

[3] Proceeds past the barrier with the other participating processors.

An additional constraint is added in *barrier MIMD architectures*:

[4] When the last processor has reached the barrier, and after some small delay to detect this condition, *all* processors simultaneously resume execution past the barrier.

Although not discussed in detail in this paper, recent work has shown that adding constraint [4] to the definition of barrier synchronization allows the static instruction scheduling properties of VLIW and SIMD machines to be extended into the MIMD domain [DSOZ89], [DiSc88], [ZaDO90]. This means that many conceptual synchronizations can be resolved at compile-time, without the use of a run-time synchronization mechanism.

The paper is organized as follows. Section 2 reviews previous hardware barrier synchronization mechanisms and points out both their strong and weak points. Section 3 provides models and definitions that are essential in understanding the new barrier mechanism, while section 4 gives an outline of the basic hardware design for barrier MIMDs. Detailed analytic and simulation results on the performance of *Static Barrier MIMD (SBM)* are given in section 5, followed by conclusions and a description of ongoing work in section 6.

## 2. Survey of Current Hardware Barrier Schemes

Hardware barrier synchronization has become more important recently for several reasons, including the unpleasant fact that software implementations of barriers using traditional synchronization primitives result in $O(\log_2 N)$ growth in the synchronization delay $\Phi(N)$ [ArJo87], [Luba89], [Broo86], [HeFM88] on $N$ processors. Fine-grain parallelism cannot be exploited with such large delays.

In addition, the directed synchronization primitives employed in these software barriers contend for shared resources such as network paths and memory ports, and this contention introduces stochastic delays that make it impossible to bound the synchronization delays between processors. As shown in a recent paper, the ability to bound these delays is vital to removing synchronizations through static scheduling [DSOZ89].

Barrier synchronization has become more popular as a synchronization construct, and can be found in parallel programming languages (such as the *Force* and *XPC* [Phil89]) and as a subroutine call in many parallel programming libraries. This makes hardware support for barrier synchronization more important for fast, efficient execution of parallel programs.

In this section we review several previous proposals for hardware barrier synchronization.

### 2.1. Finite Element Machine

The term "barrier synchronization" first appeared in a paper by Jordan [Jord78]. This paper described a MIMD multiprocessor designed to solve finite element analysis problems. This application is typically reduced to forming a stiffness matrix and solving a sparse matrix problem using iterative techniques (which preserve sparsity.) Quoting Jordan [Jord78]:

> Several aspects of the (finite element) algorithm require synchronization of (all) the nodal processors. Consider, for example, the transition from the formation of the stiffness matrix to solution of the linear equations. *No processor should start the latter until all complete the former*. Let us call such a synchronization a *barrier synchronization.*

A hardware scheme for implementing barrier synchronization was proposed for the architecture. This scheme employed global bit-serial busses; each bus had an enable bit and flag associated with each processor. The flag could be set or cleared, and conditions "Any", "All", and "First" for all other processor flags tested. The bus implemented a priority chain, wired-OR, and wired-AND function. To implement a barrier, two flags, a barrier flag and report flag were required. One processor (the controller) tested the "All" condition for the report flag, while all other processors set the report flag once they completed their work. The processors then performed the "Any" test on the barrier flag as long this flag was true. The controller cleared the barrier flag once the report flag satisfied the "All" condition. This simple scheme will work for small numbers of processors, but the global busses preclude scalability. Experiments run on an eight-processor prototype of this machine were discussed in [AdCr84].

## 2.2. Burrough's Flow Model Processor (FMP)

The Burroughs Corp. proposal [Burr79], [Burr81] for the Flow Model Processor (FMP) [Lund80], [BaLu81], [Lund85], [Lund87] included the first detailed description of a hardware implementation for barrier synchronization. This machine was originally designed to support applications in computational aerodynamics, but implements a general MIMD model, and can efficiently solve a variety of modeling and simulation problems. The designers of the FMP began with a definite performance goal: *sustained* computation rates in excess of one billion floating-point operations per second using early 1980s circuit and packaging technology. The FMP was to be built for NASA and was to perform computational wind tunnel studies that require this high level of sustained performance. Although the machine was never built, design specifications were completed down to the chip and board level [Burr81].

The Flow Model Processor architecture design was driven by application considerations [Lund87]. This is partly reflected in the extensions made to standard FORTRAN to support FMP execution. These extensions included a `DOALL` loop construct that is closely related to the standard FORTRAN DO, except that the iterations (instances) of the `DOALL` are completely independent and may execute in parallel. This represents a MIMD execution model, and the construct is useful in expressing parallelism in aerodynamic programs. These programs represent a three-dimensional fluid space with data arrays, and the computations consist of repetitive updates of each grid point in the space using data from adjacent grid points. The updates are similar for most instances of the `DOALL` except boundary grid points; these differences are reflected in different control flow paths in each instance. Another extension to standard FORTRAN was the DOMAIN statement, which provides a technique for specifying rectangular subsets of iteration space.

The hardware barrier mechanism in the FMP arose from a need for an efficient and fast way to synchronize all processors after they complete execution of a `DOALL` [Burr79], [Lund80]. After a processor has executed all its assigned `DOALL` instances, it executes a WAIT instruction. A processor continues to execute WAIT until a "GO" signal is received. The "GO" signal is generated by a synchronization network, known as the Processor Control and Maintenance Network (PCMN), which acts as a massive "AND" gate. When the last processor to finish executes a WAIT, this signal propagates up the "AND" tree in a few gate delays, and then "GO" is reflected back down the tree enabling all processors to continue execution past the `DOALL`. The Flow Model Processor can be partitioned into subsets executing different programs by configuring "AND" gates at lower levels of the synchronization tree as root nodes for each subset [Lund85]. The partitioning was conceived to allow the FMP to run smaller jobs during the day when users would be compiling and debugging prototype programs, and then work as a single unit late at night on very long jobs.

Scheduling is performed without a global control unit; each processor is identified by a processor number and is given the number of instances in the `DOALL` just before execution of the `DOALL` begins. At that point, each processor has enough information to independently determine the remaining instances it will execute, and no global control is necessary. Simulation studies showed that such *static* scheduling worked well on the FMP [Burr79].

Thus, the FMP barrier scheme is fast, executing a barrier synchronization in a few clock ticks, scalable, and within limits, partitionable. Partitions are constrained to certain subgroups related to the "AND" tree structure, and only certain processors may be grouped together. This constraint is not surprising given the nature of the parallel code executed on the FMP, and does not affect its performance; but it does unnecessarily constrict the generality of the machine. A masking capability is provided so that only a subset of the processors in a partition participate in a barrier.

## 2.3. Barrier Modules

Another hardware barrier scheme was developed by Polychronopolous [Poly88] and studied by Beckmann [BePo89] in the context of bus-based multiprocessors. In this scheme, barriers are implemented through a hardware module consisting of bit-addressable registers R($i$), ($i = 1, 2,...,p$), one associated with each of $p$ processors, an enable switch, logic to test for the all zeroes[2] (all processors have reached the barrier), and a barrier register BR. To better understand this scheme, consider executing a `DOALL` loop nested inside a serial outer loop. A barrier is required after the `DOALL` to synchronize all PEs before beginning the next iteration of the serial outer loop. The BR register is set at the beginning of each iteration of the outer loop. Each processor would execute several iterations from the inner `DOALL` loop, setting its associated register R(italici) when it begins an iteration and clearing the register when it completes. This continues until the processor executing the last iteration of the `DOALL` turns on the enable switch, allowing the "all zeroes" logic to determine when all processors have finished, at which point the BR register is cleared. This basic scheme was duplicated to handle multiple barriers [Beck89].

There are several problems with the the hardware barrier module scheme. First, all processors must participate in the barrier because there is no masking capability, i.e., the BR register can only be cleared once ALL $p$ processors have set their associated bit-register R($i$) for the last time. This restriction could easily be removed by incorporating a masking register into the enable switch so that certain processors could be disabled from participating in the barrier. The process that set the BR register, and hence controls the dispatching of iterations for the barrier, must then insure that iterations are sent only to processors participating in the barrier. It must also set the mask to include only participating processors in the barrier. If a self-scheduling algorithm is employed, processors not involved in a barrier must be prevented from taking iterations participating in the barrier. This could be implemented straightforwardly using a tag for each iteration to identify its barrier.

Second, a separate hardware unit is needed for each barrier executing concurrently with other barriers. This means the global connections from each barrier module to all PEs as well as the "all zeroes" logic must be repeated. An alternative to repeated global modules was suggested in [Poly86] in the context of the Cedar multiprocessor system. Cedar consists of clusters of tightly-coupled processors connected to a global shared memory through a multistage network. Barrier hardware modules would be placed in each cluster, and modules could communicate across clusters through some dedicated hardware,

2. This logic would be the similar to the "AND"-tree network in the FMP.

possibly a bus.

Third, no hardware is provided to signal the processors that they may proceed past the barrier. That is, once the BR register is cleared, one or more processors must contend to set it, and then dispatch the next set of iterations. Alternately, one processor could be assigned to set the BR register, and the barrier module could send an interrupt to this processor to inform it that all PEs have finished iterations associated with the barrier. If scheduling was not distributed and dynamic, the barrier register could be hardwired to a global control unit (GCU) [Poly86], and this unit could then dispatch the iterations associated with the next barrier.

Finally, unless the process (iteration) dispatching and switching times are very small, the time saved by the barrier module scheme in detecting barrier completion may be swamped by the time necessary to dispatch the next set of iterations. Hence, the run-time overheads of a dynamic, self-scheduled machine could kill the fine-grain advantages of hardware barrier synchronization.

## 2.4. Fuzzy Barrier

Gupta [Gupt89a], [Gupt89b] also considered hardware support for barrier synchronization. The scheme was described as the "fuzzy" barrier. The "fuzzy" part of the fuzzy barrier is basically a delayed barrier firing mechanism where the actual wait may occur several instructions after a processor indicates it has encountered a barrier. The instructions that the processor may execute while a barrier is pending are known as the *barrier region.* The concept is similar to delayed branches in pipelined machines: the branch does not take place until several clock ticks after the the branch instruction, masking the memory delay for fetching the instructions at the branch target. The barrier remains pending across several instructions; a wait delay occurs at the barrier only if the processor reaches the end of its barrier region before all of the other processors participating in the barrier reach the beginning of their respective barrier regions. The goal is then to make the barrier regions as large as possible, so that even with small variations in execution time among the participating processors, no waits occur. It should be noted that just such a delayed barrier mechanism was described in the Burroughs proposal for the FMP ([Burr79], sec. 5, pg. 31).

In [Gupt89a], several examples of reordering loop code to enlarge barrier regions are given. Although enlarging barrier regions tends to reduce the occurrence of barrier waits in the fuzzy barrier scheme, it is not necessarily a good idea. Some of the code motions proposed to take advantage of the fuzzy barrier [Gupt89a] are the exact opposite of the traditional code motions for loops. That is, loop optimizations typically attempt to remove invariant code from loops to reduce execution time. This is the precisely the purpose of induction variable simplification, constant folding with value propagation, common subexpression elimination, and dead code removal [AhSU86]. In several of the code motion examples given in [Gupt89a] to enlarge barrier regions, much of the code that is placed inside loops to enlarge them would have been removed by these optimizations.

The basic premise behind the fuzzy barrier — the necessity of avoiding barrier waits at all costs — may be the result of current implementations of barrier synchronization, in which a processor waiting at a barrier does an expensive context switch rather than a simple busy-wait. These expensive context switches are cited as the primary reason for improvements in simulation results using the fuzzy barrier on an Encore Multimax [Gupt89a]. The synchronization time relative to execution time is reduced as the barrier region size increases. However, another solution would be to simply turn off the context switch and pay the price for the barrier waits. If the processor load is reasonably well-balanced, this should be an acceptable solution for these bus-based multiprocessors. The results of several studies have supported the idea of static (or pre-) scheduling of loop iterations [KrWe84], [BePo89]. This suggests that it is better to put the code re-ordering efforts into balancing region execution times rather than preventing waits with larger barrier regions.

Several problems exist with the fuzzy barrier hardware implementation [Gupt89b], the most significant being the hardware expense. Each processor has its own separate barrier processor, and all processors must send a barrier signal ("I am at the barrier") to all other processors, along with a tag to distinguish itself from other barriers. Expensive matching hardware is duplicated in each processor to determine if the tags match for processors participating in a barrier. There are $N$ barrier processors in an $N$ processor machine and $N^2$ connections among these processors. Each connection contains at least $m$ lines (given an $m$-bit tag to identify $2^m-1$ different barriers.) The large number of connections and hardware required per processor limits the fuzzy barrier to a small number of processors. Another severe limitation of the fuzzy barrier is that procedure calls, interrupts and traps cannot be executed in barrier regions.

## 2.5. Other Hardware Techniques Implementing Barriers

Various other hardware mechanisms have been used to implement barrier synchronization, including combining networks [Gott83], cache-coherence hardware [GoVW89], and synchronization busses. These mechanisms are typically more general than the previous, specialized hardware barrier schemes, but have lower performance for barrier synchronization. During barrier synchronization, all processors access a single shared synchronization variable. Recent studies have shown that such concentrated access in multistage networks results in a "hot spot" that significantly increases memory access times, even for accesses to locations other than the "hot spot." Combining networks have been proposed as a solution, but the switches required are very complex and the additional hardware in each switch increases the access time for all references. In addition, a recent study [Lee89] found that the size of switches necessary to support effective combining must increase as the machine size increases, calling into question the scalability of combining networks.

An implementation of barrier synchronization using a software combining tree scheme along with cache-coherent hardware is described in [GoVW89]. Once all processors have reached the barrier, a *Notify* operation is used to update all shared copies of the barrier synchronization variable, rather than merely invalidating it. This prevents the processors from spinning on the global copy of this variable after it is invalidated, as would happen in most hardware cache-coherence schemes.

Another approach to supporting barriers in hardware is the synchronization bus and concurrency control units on the Alliant FX/8. This bus is shared by up to eight processing elements, each containing a vector unit. Synchronization primitives executed on the bus support barrier synchronization. This scheme is effective for a small number of processors.

## 2.6. Summary

The FMP and barrier module schemes are not quite general enough to meet the need for a generalized barrier synchronization mechanism, and the fuzzy barrier and other hardware techniques for barriers do not scale well. Also, the concept of *simultaneous* resumption of execution after the barrier is not inherent in any of the previous schemes. The barrier designs proposed in this paper are both scalable and general enough to barrier synchronize any subset of the processors, and simultaneous resumption of execution past the barrier is implicit in the hardware design.

## 3. Models for Barrier Synchronization

We now introduce representations for barrier synchronization in concurrent processes. These representations will help in understanding barrier MIMD execution and design alternatives. In this work, the *barrier embedding* for a set of concurrent processes will be represented as in figure 1. The vertical lines represent concurrently executing processes while the horizontal lines represent barriers across the processes they intersect. The semantics of these barriers are that the participating processes cannot proceed until all have arrived at the barrier, e.g., in figure 1, processes *P0, P1, ..., P4* cannot proceed past barrier 0 until all have arrived there. At that point, they all start execution of the instruction following the barrier *simultaneously*. Process execution proceeds in the downward direction.

Several concepts and results from the theory of partially ordered sets are useful in understanding barrier embeddings within concurrent processes. Recall that a *binary relation R* on a set *P* is a subset of the Cartesian product $X^2$, that is $R \subseteq X \times X$. Let *xRy* correspond to $(x, y) \in R$, and not(*xRy*) represent $(x, y) \notin R$. The binary relation $<_b$ on a set of barriers *B* is a *partial ordering* because $<_b$ is both *irreflexive* and *transitive*[3] [Fish85]. The partially ordered set $(B, <_b)$ may be illustrated by a directed acyclic graph (dag), with the graph nodes representing barriers and edges representing the ordering relations $<_b$ among the barriers. A barrier dag for the barrier embedding in figure 1 is shown in figure 2. Here we see that $b_2$ (barrier 2) must execute before $b_3$ (barrier 3), hence $b_2 <_b b_3$, and similarly $b_3 <_b b_4$. Transitivity implies $b_2 <_b b_4$. These properties are derived from the barrier semantics: barrier $b_3$ must be executed after the process *P3* has encountered barrier $b_2$. Similarly, $b_4$ must be executed after the process *P2* has encountered $b_3$.

A *synchronization stream* is defined to be an ordered sequence of barriers *S*. As mentioned previously, the ordering is implied by the embedding of the barrier synchronization instructions in the separate instruction streams. More formally, a synchronization stream corresponds to a *chain* in the partially ordered set $(B, <_b)$. A *chain* in a poset *$(B, <_b)$ is a set* $S \subseteq X$ such that $(S, B \cap (S \times S))$ is a *linear ordered set*[4]. Conversely, *A* is an *antichain* if *x~y* for all $x, y \in A$, where

3. A binary relation *R* on *X* is *irreflexive* if not *xRx* for every *x* in *X*. It is *transitive* if *(xRy, yRz)* => *xRz* for all *x, y, z* in *X*.

4. A binary relation *R* on *X* is a *linear order* if *R* is *asymmetric* and *complete*. *R* is asymmetric if *xRy* =>

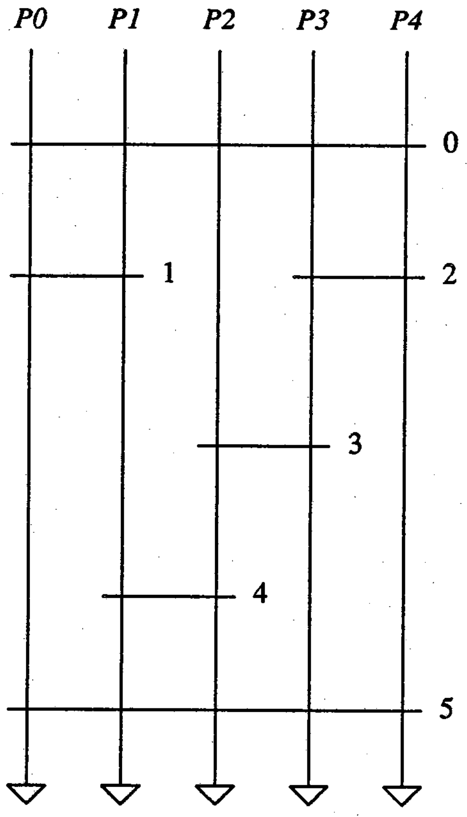

Figure 1

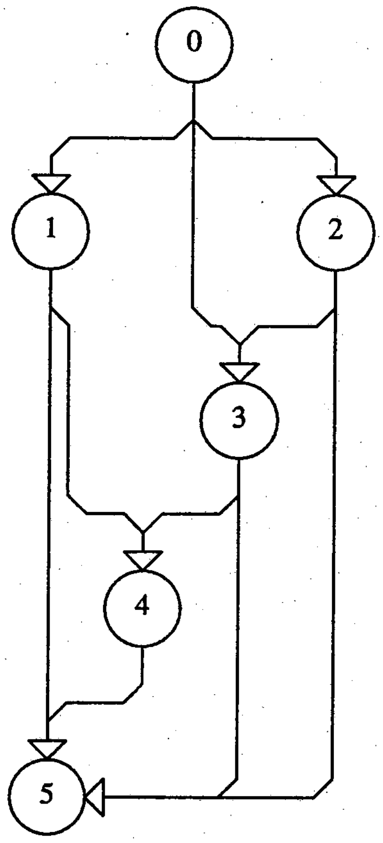

Figure 2

$$x \sim y \quad \text{if not}(xRy) \text{ and not}(yRx) \ .$$

Barriers $x$ and $y$ that satisfy $x \sim y$ are said to be *unordered.* The *width* $W(X, R)$ of a poset is $sup\{|A|: A$ is an antichain in $(X, R)\}$[5]. Clearly, the barriers contained in an antichain may be executed in any order; indeed, they may even be executed in parallel. The maximum number of synchronization streams for a particular barrier embedding corresponds to the *width* $W$ of the poset $(B, <_b)$.

Examples of partial, *weak*[6], and linear orders are given in figure 3. Antichains are specified in the weak order example. The largest antichain in the weak order shown in figure 3 contains three barriers: hence, the width of the weak order is three. The partial order shown in the figure also has width three. Clearly, the linear order in the figure contains a single synchronization stream, whereas multiple synchronization streams (chains) are evident in the weak order.

---

not$(yRx)$ for all $x$ and $y$ in $X$. Relation $R$ is complete if $x \neq y \Rightarrow (xRy$ or $yRx)$ for all $x$ and $y$ in $X$.

5. The function *sup* is the *suprenum,* or least upper bound. $|A|$ represents the cardinality of the set $A$.
6. A *weak order* is a partial ordering in which the symmetric complement $\sim$ is *transitive.*

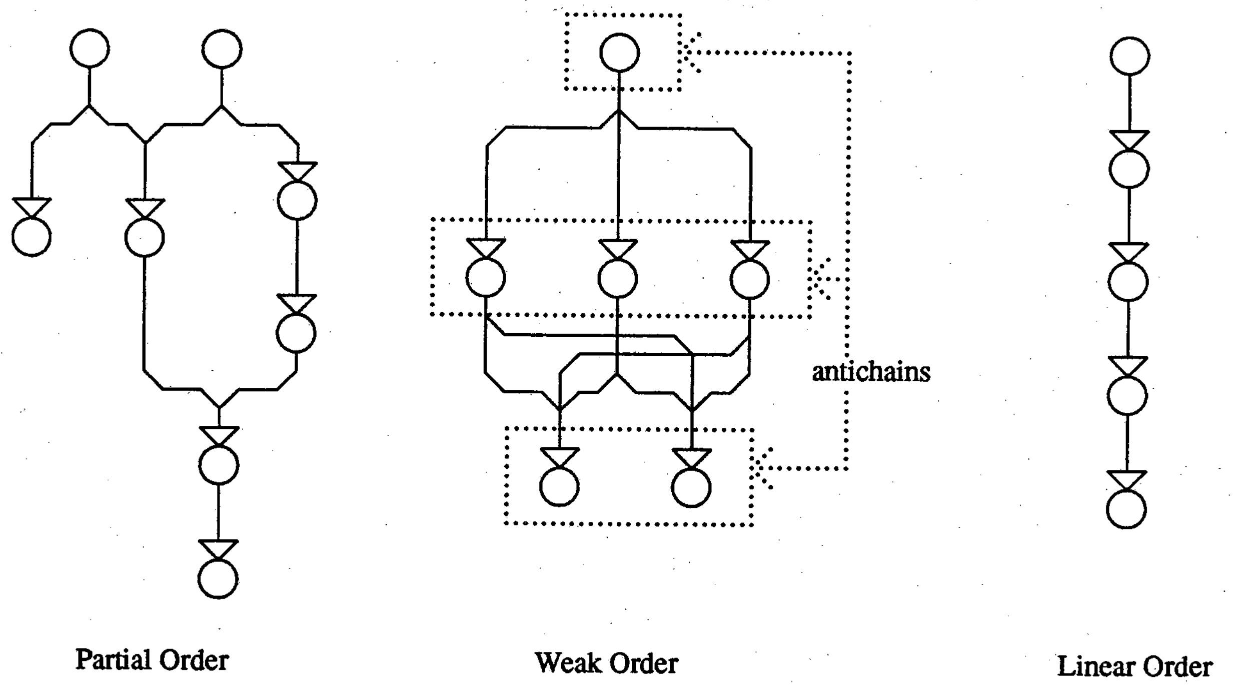

**Figure 3**

Intuitively, the number of synchronization streams that a given architecture supports corresponds to the number of different synchronization operations that are candidates for the next synchronization operation to occur. If this number is not zero (no synchronization), then larger values imply fewer delays when performing multiple synchronizations. For example, suppose a four processor machine requires processors 0 and 1 (barrier a) to synchronize as well as processors 2 and 3 (barrier b), as shown in figure 4. This yields two synchronization streams, traced out by dotted lines in figure 4.

If the order in which the synchronization operations occurs cannot be predicted at compile time, a machine which permits multiple synchronization streams will insure that the synchronizations execute in the correct order or even in parallel. A machine which permits only one stream will sometimes suffer a delay due to, for example, processors 0 and 1 waiting for 2 and 3 because the compiler incorrectly guessed that the synchronization of 2 and 3 would occur first. Another approach is to combine both synchronizations into a single barrier across processors 0, 1, 2, and 3 (as shown in figure 4) if the machine supports only a single synchronization stream. This yields a slightly longer average delay to execute the barriers.

In general, a barrier dag, $(B, <_b)$ corresponding to a barrier embedding in $P$ concurrent processes has a maximum width of $P/2$. This follows from the fact that the smallest number of processes participating in a single barrier is two, yielding a maximum of $P/2$ barriers in a single antichain. Note that there are $2^P - P - 1$ possible subsets of the $P$ processes with cardinality greater than or equal to two and therefore this same number of possible barrier patterns.

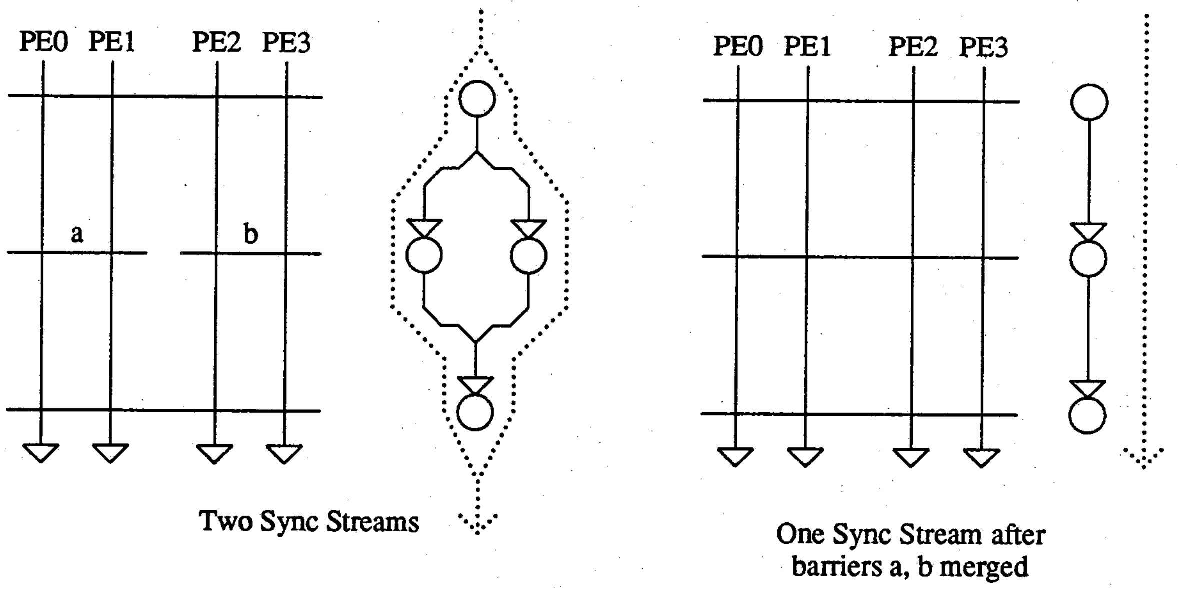

**Figure 4:** Merging Barriers Reduced Number of Sync Streams

The two basic forms of barrier MIMD, *static* and *dynamic*, differ in that *static barrier MIMD (SBM)* supports only a single synchronization stream whereas *dynamic barrier MIMD (DBM)* supports multiple synchronization streams. In essence, the SBM imposes a linear order on the partial ordering inherent in the barrier dag; the DBM imposes no constraints on the partial order. A *hybrid barrier MIMD (HBM)* architecture that imposes a *weak order* on the barrier dag is introduced in the next section. The implementation and performance of the static and hybrid barrier MIMD architectures are discussed in more detail in the following sections.

## 4. Barrier Synchronization Hardware Design

The original catalyst for the hardware barrier synchronization designs proposed in this work was *PASM*, the Partitionable SIMD/MIMD machine designed by H. J. Siegel at Purdue University [SiSi81]. PASM is a reconfigurable parallel computer that can be dynamically partitioned to form independent virtual SIMD and/or MIMD machines of various sizes. A 16 processing element PASM prototype has been constructed at Purdue University [ScNa87].

The barrier MIMD idea arose in an attempt to implement a VLIW execution model [Elli85] on the PASM prototype. During this attempt, it quickly became clear that PASM could not easily support VLIW execution. However, a new mode of execution was discovered that was not SIMD, MIMD, nor alternately or simultaneously SIMD and MIMD. Instead, processors would run and communicate in MIMD mode, but would also employ the logic that normally enables and disables processors in SIMD mode to implement a barrier synchronization mechanism. In addition, it was realized that code generation and scheduling for PASM in this new barrier execution mode could be accomplished using techniques similar to *Trace Scheduling*[7] for VLIW machines. Several benchmarks have been run on the

7. *Trace Scheduling*$^{TM}$ is a trademark of Multiflow Computer, Inc.

PASM prototype in this mode [FCSS88], [BrCJ89] and preliminary results have been very promising. In [BrCJ89], several versions of the fast fourier transform algorithm were executed on PASM, and the barrier execution mode outperformed both SIMD and MIMD execution mode in all cases.

The barrier synchronization mechanism in PASM can be applied across all processors or selected subsets. The "barrier instruction" is actually a read from the SIMD data address space: PASM processors switch between SIMD and MIMD modes by reading instructions and data and writing data in the separate MIMD and SIMD address spaces. A barrier mask of participating processors corresponds to the SIMD mask word: these masks are enqueued in a FIFO along with a SIMD instruction (which is ignored in barrier mode.) An "AND" tree detects when all processors in the mask pattern have executed the SIMD data read, and the participating processors are then released from the barrier and continue execution. Thus, the PASM SIMD enable logic provides a fast, flexible barrier synchronization mechanism.

The design of the PASM prototype made it clear that the problem of generating a barrier synchronization across any subset of the processors is identical in nature to the problem of generating enable/disable masks for a SIMD processor. Hence, just as a SIMD processor has a *control unit* to generate enable/disable masks, a barrier MIMD has a *barrier processor* that generates barrier masks to identify the processor subsets participating in a particular barrier synchronization. The barrier processor generates barrier masks into the *barrier synchronization buffer* where each mask is held until it has been executed. A single WAIT line from each processor to the barrier synchronization buffer is used to indicate that a particular processor is participating in a barrier synchronization.

Each mask consists of a vector of bits, referred to as MASK, one bit for each processor. The value of bit MASK($i$) indicates whether the corresponding processor $i$ will participate in that particular barrier synchronization[8]. In the SBM execution model, the barrier synchronization buffer corresponds to a simple queue. This queue imposes a linear order on the execution of the barrier masks that will not, in general, correspond to the execution ordering that occurs at runtime. In figure 5, a set of five barriers across four processors must be executed; the first two barriers, across processors 0 and 1 and processors 2 and 3 can be executed in any order. The barriers masks are ordered as shown on the right side of the figure, where a one in the mask corresponds to a participating processor. Here the first barrier across processors 0 and 1 is assumed to execute first: the other four barriers are then placed in the SBM queue in the order that they must execute at run-time.

In the DBM model, barriers are executed and removed from the barrier synchronization buffer in the order that they occur at runtime. This implies the need for an associative match capability in the DBM synchronization buffer, and it is this buffer which supports up to $P/2$ synchronization streams.

8. Note that unlike the fuzzy barrier and barrier module schemes, no tags are necessary to identify particular barriers, as this is implicit in the manner in which they are stored. This reduces the number of connections between the barrier processor and the computational processors and the complexity of the matching hardware significantly.

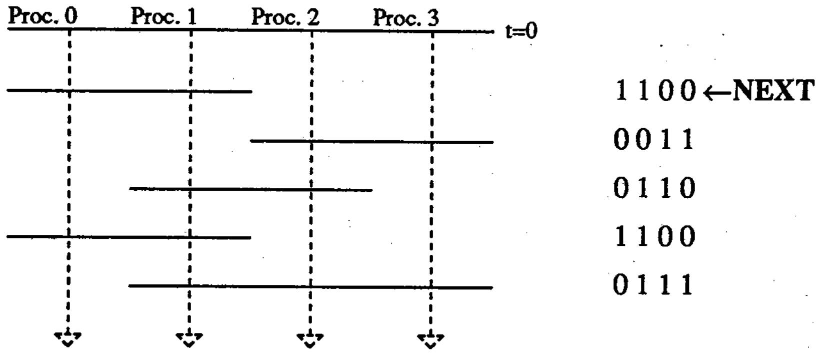

**Figure 5**

In both the SBM and DBM model, processors execute a `wait` instruction (or an instruction tagged with a `wait` bit) but do not continue past the `wait` until the current processor `wait` pattern WAIT causes the next barrier to complete. Processors participating in this barrier (processor $i$ participates if MASK($i$) is one) then continue execution past the `wait` instruction. A processor that is not involved in the current SBM barrier need not execute a `wait` for that barrier — if a `wait` is issued by a processor not involved in the current barrier, the SBM simply ignores that signal until a barrier including that processor becomes the current barrier. Since barrier patterns can be created asynchronously by the barrier processor and buffered awaiting their execution, the computational processors see no overhead in the specification of barrier patterns. Hence, both SBM and DBM machines can achieve essentially perfect synchronization of any subset of the processors with only a very small, roughly constant overhead.

The logic equation representing the condition that all processors participating (MASK($i$) = 1) in the current barrier have arrived at the barrier and executed a `wait` instruction is

$$\text{GO} = \prod_i \overline{\text{MASK}(i)} + \text{WAIT}(i) \ .$$

Of course, in addition to generating code for the computational processors, for either the SBM or DBM machines the compiler must precompute the order and patterns of all barriers required for the computation and must generate code that the barrier processor will execute to produce these barriers. The code for the main processors also must contain the appropriate `wait` instructions or instruction tags. Separate `wait` instructions are probably easier to implement than tags, but tags would permit more frequent use of barriers.

## 5. Static Barrier MIMD (SBM)

Figure 6 shows the basic SBM architecture for a four-processor machine. The barrier masks and ordering are the same as those used in figure 5. The NEXT barrier mask that is being matched is "OR"ed with the WAIT bits from the processors. The "OR" output bits then propagate through the "AND" tree to produce the GO signal. This signal indicates that the all processors participating in the

NEXT barrier have encountered the barrier and generated a WAIT signal. The active GO signal causes the NEXT barrier mask to be sent out on the processor GO lines, indicating to participating processors that they may proceed past the barrier. The barrier masks remaining in the queue then advance to the next available position, and the first barrier in the queue becomes the NEXT barrier. The barrier and computation processors are not shown in the figure, nor is the barrier queue load logic.

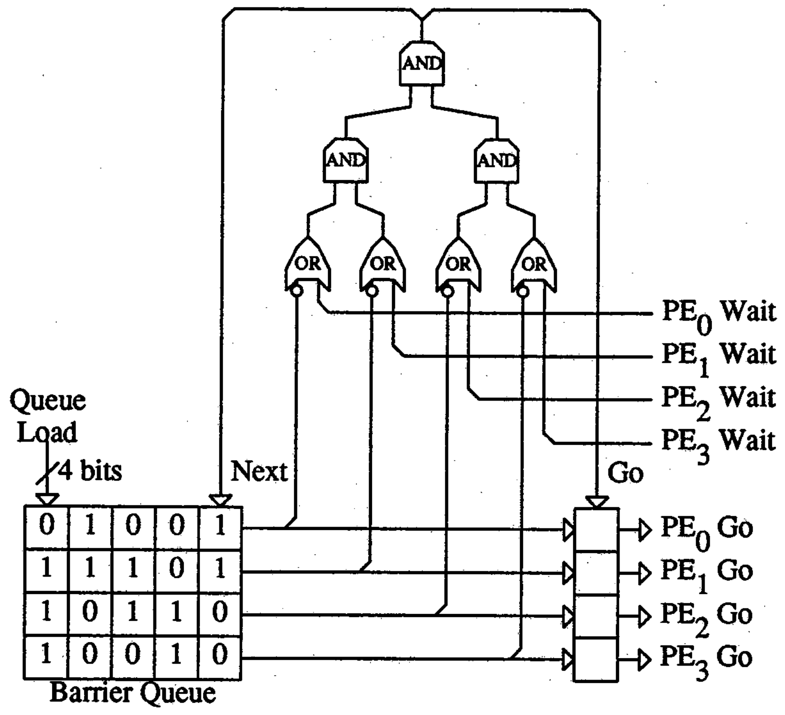

**Figure 6**

As stated previously, the SBM execution model imposes a linear order on the execution of barriers; that is, unordered barriers have an ordering relation imposed on them when the are loaded into the SBM barrier queue. The SBM barrier ordering will correspond to the *expected* runtime ordering of the barriers, and may not, in general, correspond to the *actual* runtime ordering. In order to measure the performance of the static barrier MIMD architecture, it is important to understand the impact of delays resulting from the difference between the expected and actual runtime barrier ordering. In the following section, a model is proposed for the problem and the percentage of barriers that experience blocking due to the SBM barrier ordering is derived. Results from a simulation study of the SBM are also described.

## 5.1. Analytic Modeling

To understand the potential impact of delays imposed by the linear order in the SBM queue we will consider a barrier embedding containing an $n$ barrier antichain. Note that there are $n!$ possible runtime orderings of these barriers. If the $n$ barriers have the same expected execution times, or if no information is available about the expected execution times, then no assumptions concerning the runtime ordering of the barriers can be made, and the placement of the barriers in the antichain in the SBM queue is essentially a random selection.

The percentage of the barriers in an antichain that are blocked by a particular SBM queue ordering will be characterized, and it will be shown that blocking is equivalent to "combining" the barriers into several larger barriers or even a single barrier. After characterizing the percentage of barriers blocked for a given schedule, it is possible to estimate the delay caused by this blocking phenomena.

Consider a barrier embedding with a three barrier antichain. There are $3! = 6$ possible execution time orderings of the three barriers (labeled 1,2, and 3) in the antichain[9]. Consider the execution ordering of barrier 3, followed by barrier 2, then barrier 1: barriers 3 and 2 are blocked by barrier 1, and the effect is equivalent to the three barriers being combined into a single barrier. A barrier embedding with a three barrier antichain ( barriers 1,2, and 3) and this execution time ordering is shown in figure 7.

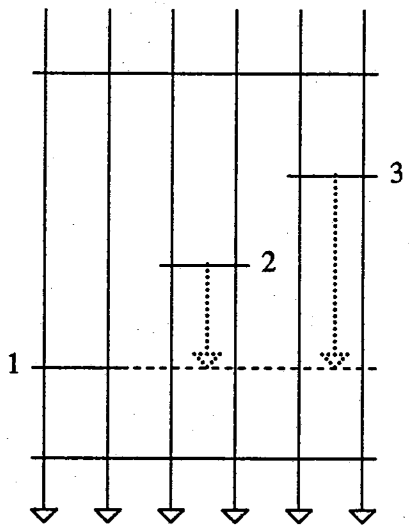

**Figure 7:** Effect of "Bad" Static Barrier Order

If the execution ordering is barrier 2 first, followed by 1 and then 3, barrier 2 is blocked by barrier 1, and these two barriers are, in effect, combined. The different execution time orderings for $n = 3$ can be represented as a tree, shown in figure 8.

Each level of the tree corresponds to the firing of a particular barrier. The leaves of the tree have been annotated with the number of barriers that are blocked given the particular execution ordering. Let the *blocking quotient*, $\beta(n)$, be the expected value for the percentage of $n$ barriers in an antichain that are

9. Note that in this discussion, the numbering scheme for the barriers corresponds directly to their ordering in the SBM queue. Hence, barrier 1 is first in the queue, barrier 2 is second, etc.

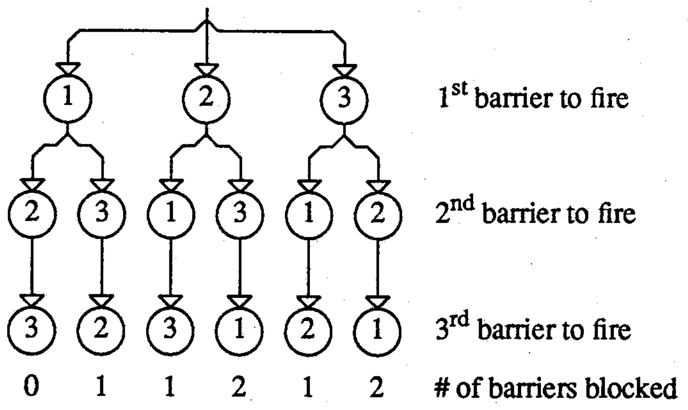

**Figure 8:** Tree Representing All Possible Execution Orders

blocked. It can be determined by weighting the number of barriers blocked by the appropriate probability, and summing these weighted values over all possible blockings.

Under our assumptions, all execution orderings are equiprobable. Hence, the probability that $p$ barriers are blocked is given by $\frac{\kappa_n(p)}{n!}$, where $\kappa_n(p)$ corresponds to the number of execution orderings with $p$ blocked barriers given $n$ barriers in the queue. This yields

$$\beta(n) = \sum_{p=0}^{n-1} p \frac{\kappa_n(p)}{n!}$$

It can be shown that

$$\kappa_n(p) = \begin{cases} 0 & \text{if } p < 0 \text{ or } p \geq n \\ 1 & \text{if } p = 1 \\ \kappa_{n-1}(p) + n\kappa_{n-1}(p-1) & \text{if } p > 1 \end{cases}$$

Figure 9 shows that as $n$ increases, the blocking quotient, $\beta(n)$, increases asymptotically as expected. It can be seen from figure 9 that over 80% of the barriers are blocked when there are more than 11 barriers in an antichain. The percentage is less for smaller numbers of barriers. When $n$ is from two to five, less than 70% of the barriers are blocked.

This result, though interesting, merely confirms what our intuition would expect: that most barriers in an antichain will experience some blocking effects due to the linear order imposed by the queue in the SBM synchronization buffer.

One way to reduce the blocking quotient would be to add a small associative memory at the front of the SBM queue, as shown in figure 10. In essence, a window of barriers at the front of the queue would be candidates for the next barrier to execute instead of a single barrier. This idea was originally proposed in [OKDi89] where it was described as a "hybrid" scheme between static and dynamic barrier MIMD

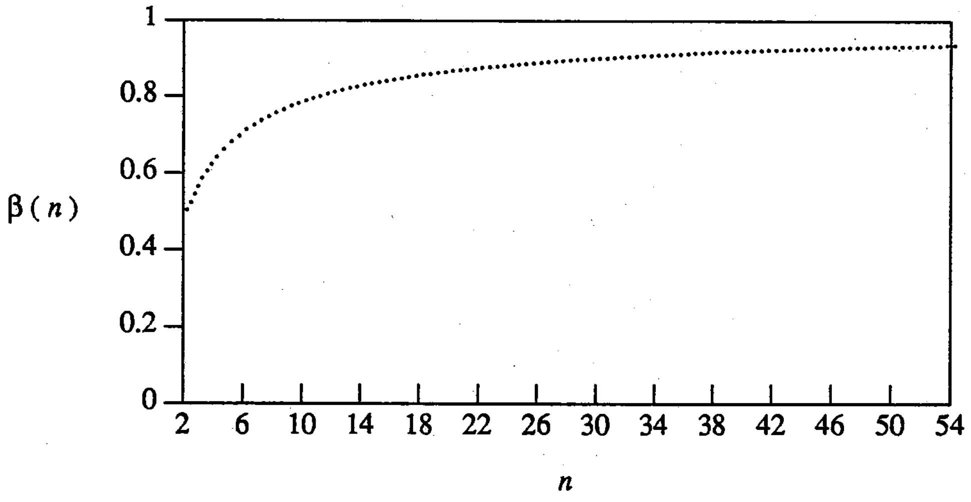

**Figure 9:**Blocking Quotient Vs. $n$

models. It will be referred to as *hybrid barrier MIMD (HBM)* in this work. Note that a linear order is still imposed on the barriers as they are loaded into the queue. Any barriers $x$ and $y$ occupying the associative memory simultaneously must satisfy $x \sim y$, since the associative memory cannot distinguish between such barriers. Hence, in the HBM the proper ordering remains implicit in the run-time queue ordering.

The previous equation for $\kappa_n(p)$, the number of execution orderings with $p$ blockings in a SBM, can be generalized to $\kappa_n^b(p)$, where $b$ represents the size of the associative buffer in a hybrid barrier MIMD. The correct expression is

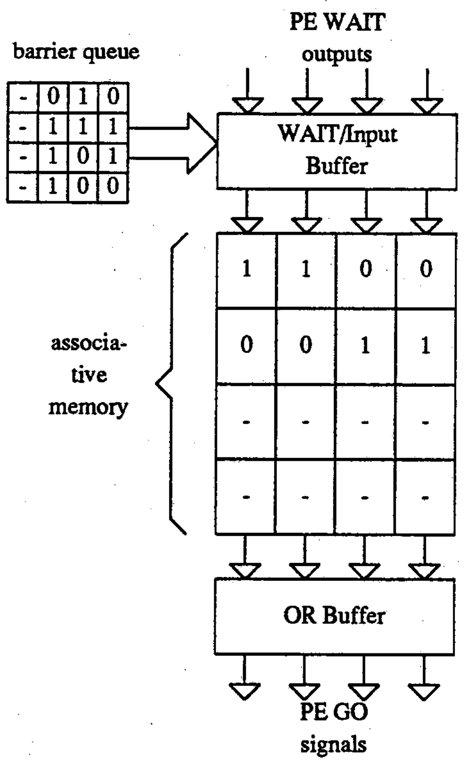

**Figure 10:** Hybrid Barrier MIMD

$$\kappa_n^b(p) = \begin{cases} 0 & \text{if } p < 0 \text{ or } p \geq n \\ 0 & \text{if } p \geq 1,\ n \leq b \\ n! & \text{if } p = 0,\ n \leq b \\ b\kappa_{n-1}^b(p) + (n-b)\kappa_{n-1}^b(p-1) & \text{if } p \geq 1,\ n > b \end{cases} .$$

The proof of the validity of this equation can be found in [OKee90]. When $b = 1$ this equation reduces to the equation given for $\kappa_n(p)$. Using the equation for $\kappa_n^b(p)$, curves for the blocking quotient of a hybrid barrier MIMD with various associative buffer sizes $b$ were computed. The results are displayed in figure 11. It can be seen that each increase in the size of the associative buffer yielded roughly a 10% decrease in the blocking quotient.

These analytic models suggest that a large percentage of unordered barriers experience some sort of blocking due to the linear ordering imposed at run-time in an SBM. In the next section, simulation is used to measure the delays resulting from blocking effects and scheduling techniques are proposed to reduce these delays.

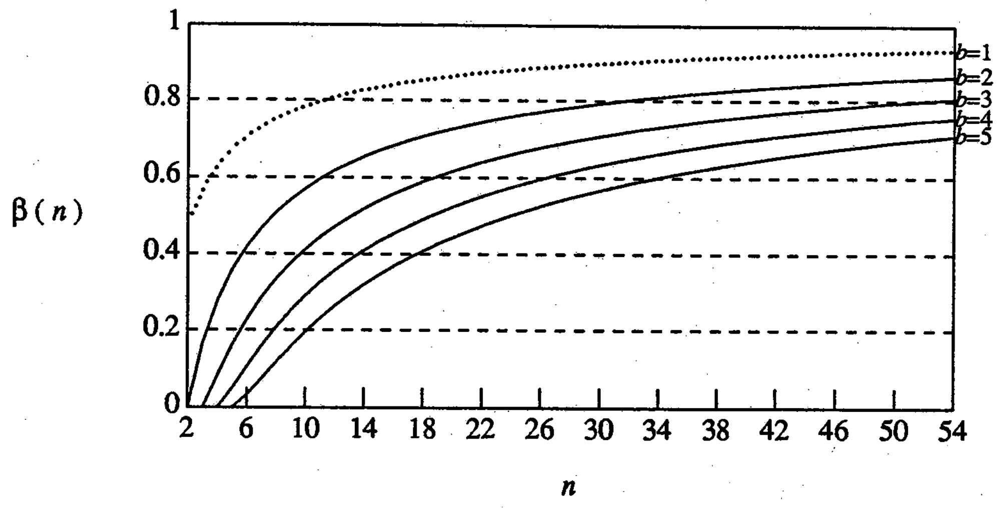

**Figure 11:**Blocking Quotient Vs. *n*

## 5.2. Simulation Study

The analysis given in the previous section made the worst case assumption that the unordered barriers where scheduled such that they all had the same expected execution time. In this situation, the compiler has no useful information concerning the ordering of the barriers in the SBM queue. Any random ordering of the barriers would be expected to perform just as well as any other ordering. We now introduce the concept of *staggered* barrier scheduling. This refers to scheduling barriers so that the expected execution time of a set of unordered barriers $\{b_1, b_2, \cdots, b_i, \cdots, b_n\}$ is a monotone nondecreasing function. Let $E(b_i)$ be the expected execution time of barrier $b_i$. Then the following equation

$$E(b_{i+\phi}) - E(b_i) = \delta E(b_i)$$

defines the *stagger coefficient* $\delta$ and the integral *stagger distance* $\phi$. We say that two barriers $b_j$ and $b_k$ are *adjacent* if $|j-k| = \phi$. The stagger coefficient $\delta$ refers to the percentage difference between the expected execution times of adjacent barriers. Figure 12 shows a schedule of four barriers with a stagger coefficient $\delta = 0.10$ and stagger distance $\phi = 1$.

Figure 13 shows a similar schedule of four barriers, except the stagger distance $\phi = 2$.

The advantage of staggered scheduling is that the barriers can now be expected to execute in a particular order with a higher probability than without staggering. This "expected" execution ordering can then be used as the ordering of the barriers in the SBM queue. Let us consider an example. Let $X_i$ represent the random variable for the execution time of barrier $b_i$. We wish determine $P[\, X_{i+m\phi} > X_i \,]$, the probability that barrier $b_{i+m\phi}$ executes after $b_i$. The former barrer is staggered $m\delta$ percent from the latter. We have

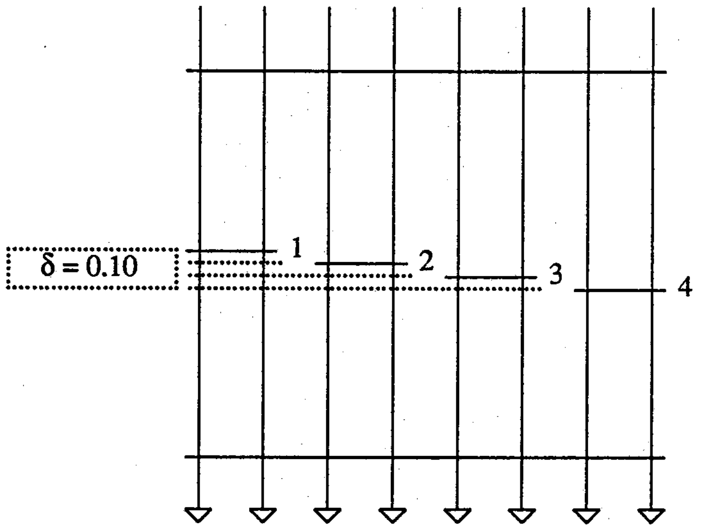

**Figure 12:** Staggered Barrier Schedule (ϕ=1, δ=0.10)

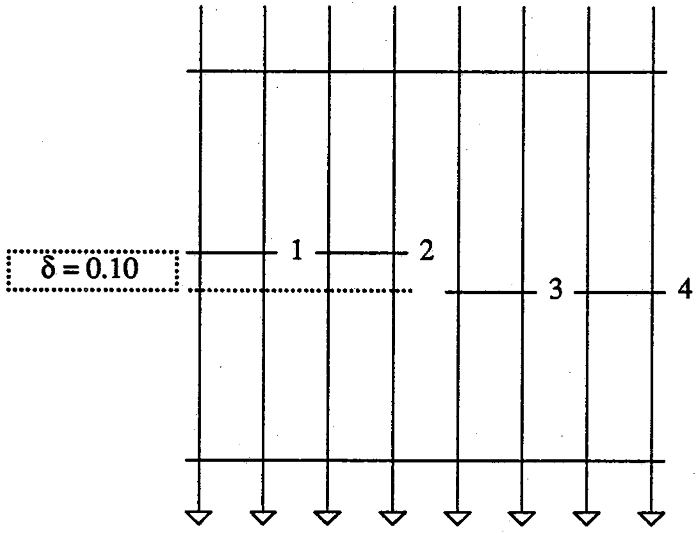

**Figure 13:** Staggered Barrier Schedule (ϕ=2, δ=0.10)

$$P[\,X_{i+m\phi} > X_i\,] = P[\,X_{i+m\phi} - X_i > 0\,] = 1 - P[\,X_{i+m\phi} - X_i < 0\,] = 1 - F_{X_{i+m\phi}-X_i}(0)$$

and if exponential distributions are assumed

$$P[\,X_{i+m\phi} > X_i\,] = \frac{(1.0+m\delta)\lambda}{\lambda + (1.0+m\delta)\lambda} \quad .$$

Simulations results show that staggered scheduling reduces the delay caused by *queue waits*, i.e. waits caused solely by the SBM queue ordering. Figure 14 shows the simulation results assuming that region execution times have a normal distribution with $\mu$=100 and $s$=20, $\phi$=1 and $\delta$ set to 0.0, 0.05, and 0.10.

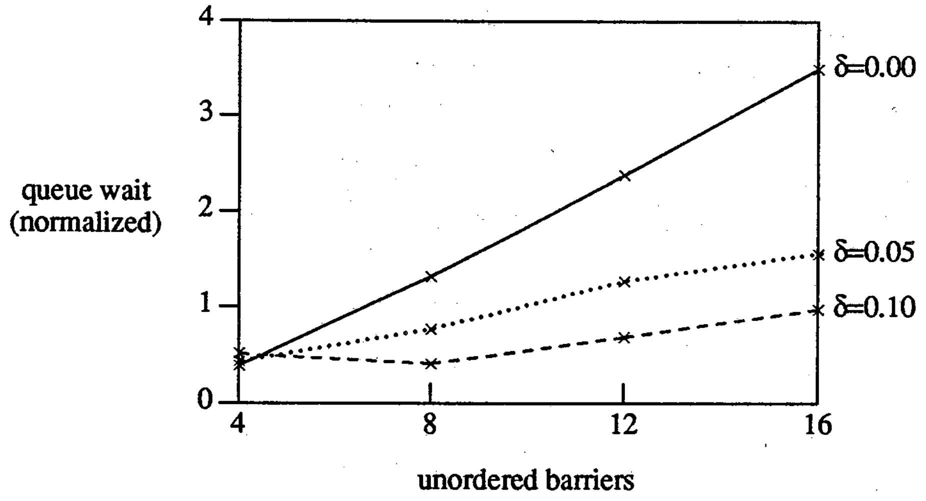

**Figure 14**

It is evident from figure 14 that staggering the barriers can significantly reduce the accumulated delays caused by queue waits.

The effects of the hybrid barrier MIMD mechanism have also been simulated. Preliminary simulation results have shown that the associative memory in the hybrid barrier architecture need be no larger than four to five cells to effectively remove delays caused by the blocking between unordered barriers.

Preliminary simulation results are displayed in Figures 15 and 16. The horizontal axis indicates the number of unordered barriers that are to be executed, while the vertical axis represents the total barrier delay, normalized to μ. The region execution times are taken from a normal distribution with $\mu = 100$ and $s = 20$ before staggering is applied.

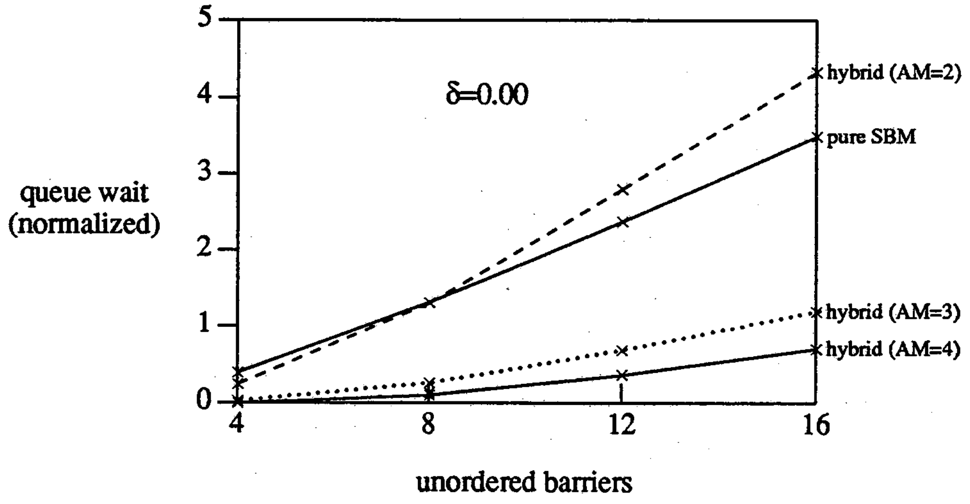

**Figure 15**

From Figure 15, it is evident that the hybrid barrier scheme reduces barrier delays almost to zero for small associative buffer sizes. There is an anomaly here for an associative buffer size of two: in this case, the barrier delays are greater that those of the pure static barrier scheme when the number of barriers is greater than about eight. The reasons for this anomaly are currently under investigation, but no clear answer is currently available. This anomaly is of more theoretical than practical significance.

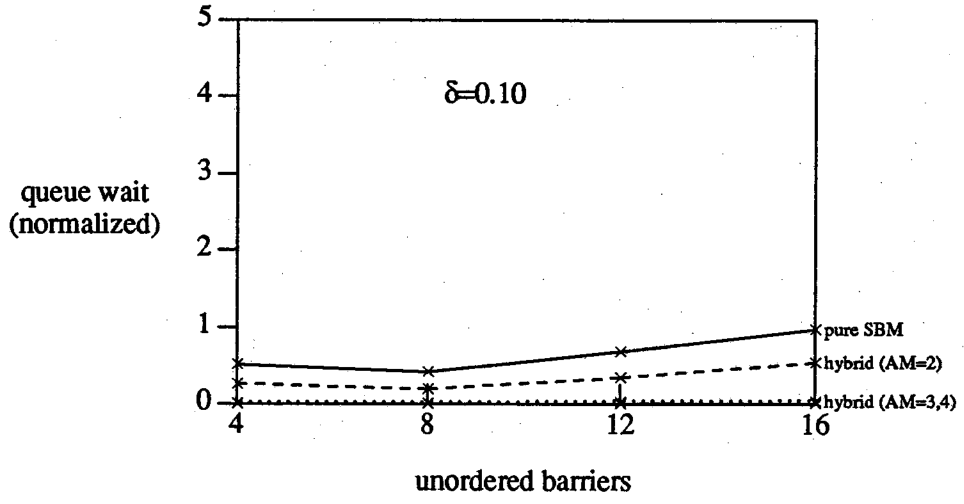

**Figure 16**

Figure 16 shows the results when staggered scheduling is employed with $\delta = 0.10$ and $\phi = 1$. The effects of staggering alone reduce the delays significantly.

These results suggest that it is possible to significantly reduce the effects of barrier blocking in the SBM and HBM by proper scheduling. However, it should be noted that the results apply only to sets of unordered barriers (antichains). Barrier embeddings with long, independent synchronization streams pose serious problems to both the SBM and HBM architectures. In essence, these independent streams are "serialized" in the barrier queue. This can potentially introduce large delays due to blocking effects. The *dynamic barrier MIMD* architecture proposed in a companion paper (part II) supports multiple, independent synchronization streams, avoiding these problems and efficiently supporting a broad class of partial orderings.

## 6. Conclusions

This paper presented designs for two different flavors of barrier MIMD architecture: the SBM and a slightly enhanced version called the HBM (Hybrid Barrier MIMD).

The primary motivation for these barrier MIMD architectures is the use of static (compile-time) code scheduling for eliminating some synchronizations, and the proposed designs are built around the features needed to support this scheduling. As discussed in [ZaDO90], a significant fraction (>77%) of the synchronizations in synthetic benchmark programs were removed through static scheduling for an

SBM. However, this was not the only benefit found.

Because the proposed SBM and HBM architectures are so simple and can operate within a few clock ticks, these architectures also show great promise as a mechanism for synchronization in general — even where sophisticated static scheduling techniques are not applied. As discussed in this paper, the SBM and HBM designs given are at least as capable as previous hardware barrier mechanisms, yet permit more efficient implementation.

As mentioned in the abstract, the SBM (and HBM) architectures are more restrictive than the DBM which we have also proposed, but SBM hardware is far simpler. The performance analysis presented in this paper suggests that, provided that static scheduling can be applied across the entire SBM, the extra complexity of the DBM is not needed. This further suggests that a highly scalable parallel computer system might consist of SBM processor clusters which synchronize across clusters using a DBM mechanism, and such an architecture is under consideration within CARP (the Compiler-oriented Architecture Research group at Purdue).

Other ongoing research includes techniques for parallelizing and scheduling complete programs, more performance analysis, and also the actual implementation of a VLSI SBM.

## 7. Acknowledgements

The authors would like to thank Dr. Thomas Schwederski, the chief architect of the PASM prototype, for his extensive help both in the original conception of the barrier MIMD idea and in the design alternatives for dynamic barrier MIMD.

## 8. References

[AcKT86] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. Comput.*, vol. C-35, no. 9, pp. 815-828.

[AdCr84] L. M. Adams and T. W. Crockett, "Modeling Algorithm Execution Time on Processor Arrays," *IEEE Computer, vol. 17, no. 7, pp. 38-44, July 1984.*

[AlWo75] W. G. Alexander and D. B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *IEEE Computer*, Nov. 1975, pp. 41-46.

[ArJo87] N. S. Arenstorf and H. F. Jordan, "Comparing Barrier Algorithms," *ICASE Rept. No. 87-65*, Inst. Comp. Applications Sci. Eng. (ICASE), NASA Langley Research Center, Hampton, VA, Sept. 1987.

[BaLu81] G. H. Barnes and S. F. Lundstrom, "Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems," *IEEE Computer*, vol. 14, pp. 31-41, Dec. 1981.

[Beck89] C. J. Beckmann, "Reducing Synchronization and Scheduling Overhead in Parallel Loops," *MSEE Thesis*, U. of Illinois at Urbana-Champaign, July 1989.

[BePo89] C. J. Beckmann and C. Polychronopolous, "The Effect of Barrier Synchronization and Scheduling Overhead on Parallel Loops," *Proc. 1989 Int. Conf. Parallel Processing,* vol. II, pp. 200-204, Aug. 1989.

[BrCJ89] E. C. Bronson, T. L. Casavant, L. H. Jamieson, "Experimental Application-Driven Architecture Analysis of a SIMD/MIMD Parallel Processing System," *Proc. 1989 Int. Conf. Parallel Processing,* vol. I, pp. 59-67, Aug. 1989. Also to appear in *IEEE Transactions on Parallel and Distributed Systems,* Spring, 1990.

[Broo86] E. D. Brooks III, "The Butterfly Barrier," *Int. Jour. Parallel Programming,* vol. 15, no. 4, pp. 295-307, 1986.

[Burr79] Burroughs Corp., Federal and Special Systems, *Final Report, Numerical Aerodynamic Simulation Facility Feasibility Study,* NASA CR-152284 and CR-152285, Paoli, PA, 1979.

[Burr81] Burroughs Corp., Federal and Special Systems, *Final Report, Numerical Aerodynamic Simulator Processing System, Final Report - System Design Study,* NASA CR-166133, Paoli, PA, 1981.

[Call87] C.D. Callahan II, *A Global Approach to the Detection of Parallelism, Ph.D. Dissertation,* Rice University, March 1987.

[CNOPR88]R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. on Computers,* vol. C-37, no. 8, pp. 967-979, Aug. 1988.

[DSOZ89] H. G. Dietz, T. Schwederski, M. T. O'Keefe, and A. Zaafrani, "Extending Static Synchronization Beyond VLIW," *Supercomputing 89,* pp. 416-425, Reno, NV, Nov. 1989.

[DiSc88] H. G. Dietz and T. Schwederski, "Extending Static Synchronization Beyond SIMD and VLIW," Tech. Report TR-EE 88-25, Purdue University, School of Electrical Engineering, June 1988.

[Elli85] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: MIT Press, 1986.

[FCSS88] S. A. Fineberg, T. L. Casavant, T. Schwederski, and H. J. Siegel, "Non-Deterministic Instruction Time Experiments on the PASM System Prototype," *Proc. Int. Conf. Parallel Processing,* vol I, pp. 444-451, Aug. 1988.

[Fish85] P. C. Fishburn, *Interval Orders and Interval Graphs: A Study of Partially Ordered Sets.* New York, NY: John Wiley and Sons, 1985.

[Fost76] C. C. Foster, *Content Addressable Parallel Processors.* New York: Van Nostrand Reinhold, 1976.

[GoVW89] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *ASPLOS-III,* pp. 64-75, April 1989.

[Gott83] A. Gottlieb, et al., "The NYU Ultracomputer - Designing a MIMD Shared Memory Parallel Computer," *IEEE Trans. Comput.,* vol. C-32, no. 2, pp. 175-189, Feb. 1983.

[Gupt89a] R. Gupta, "The Fuzzy Barrier: A Mechanism for the High Speed Synchronization of Processors," Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA, April 1989, pp. 54-63.

[Gupt89b] R. Gupta and M. Epstein, "Achieving Low Cost Synchronization in a Multiprocessor System," *1989 Parallel Architectures and Languages Europe*, published as *Lecture Notes Comp. Sci. #365*, Springer-Verlag, 1989.

[HeFM88] D. Hensgen, R. Finkel, and U. Manber, "Two Algorithms for Barrier Synchronization," *Int. Jour. Parallel Programming*, vol. 17, no. 1, pp. 1-17, 1988.

[Jord78] H. F. Jordan, "A Special Purpose Architecture for Finite Element Analysis," *Proc. Int. Conf. Parallel Processing*, 1978, pp. 263-266.

[KrWe84] C. P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *Int. Conf. on Parallel Processing*, pp. 236-240, 1984.

[Lee89] G. Lee, "A Performance Bound of Multistage Combining Networks," *IEEE Trans. Comput.*, vol. 38, no. 10, pp. 1387-1395, October 1989.

[Luba89] B. D. Lubachevsky, "Synchronization Barrier and Related Tools for Shared Memory Parallel Programming," *Proc. Int. Conf. Parallel Processing*, vol. 2, pp. 175-179, St. Charles, IL, August 1989.

[Lund80] S. F. Lundstrom, "A Controllable MIMD Architecture," *Proc. Int. Conf. Parallel Processing*, 1980, pp. 19-27.

[Lund85] S. F. Lundstrom, "A Decentralized Control, Highly Concurrent Multiprocessor," *Proc. 12th Annual Int. Symp. Comput. Architecture*, June 1985, pp. 145-151.

[Lund87] S. F. Lundstrom, "Applications Considerations in the System Design of Highly Concurrent Multiprocessors," *IEEE Trans. Comput.*, vol. C-36, no. 11, pp. 1292-1309, Nov. 1987.

[OKDi89] M. T. O'Keefe and H. G. Dietz, "Performance Analysis of Hardware Barrier Synchronization," Tech. Report TR-EE 89-51, Purdue University, School of Electrical Engineering, August 1989.

[OKee90] M. T. O'Keefe, *Barrier MIMD Architectures: Performance Analysis, Design, and Compilation*, Ph.D. dissertation, in preparation, School of Electrical Engineering, Purdue University.

[PaKL80] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. Comput.*, vol. C-29, no. 9, pp. 763-776, Sept. 1980.

[Phil89] M.J. Phillip, *Unification of Synchronous and Asynchronous Models for Parallel Programming Languages*, Masters Thesis, School of Electrical Engineering, Purdue University, May 1989.

[Poly86] C. D. Polychronopolous, "On program restructuring, scheduling, and communication for parallel processor systems," Ph. D. dissertation, Dept. of Comp. Science, U. of Ill. at Urb.-Champ., Aug. 1986; *also available as* CSRD Rpt. No. 595, Center for Supercomp. Res.

Develop., U. of Illinois.

[Poly88] C. D. Polychronopolous, "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design," *IEEE Trans. Comput.*, vol. C-37, no. 8, pp. 991-1004, Aug. 1989.

[ScNa87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "The Implementation of the PASM Prototype Control Hierarchy," *Proceedings of the Second International Conference on Supercomputing*, vol. I, 1987, pp. 418-427.

[SiSi81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, no. 12, pp. 934-947, December 1981.

[ZaDO90] A. Zaafrani, H. G. Dietz, and M. T. O'Keefe, "Scheduling Algorithms and Experiments for Extending Static Synchronization Beyond VLIW," submitted to *1990 Int. Conference on Parallel Processing*.