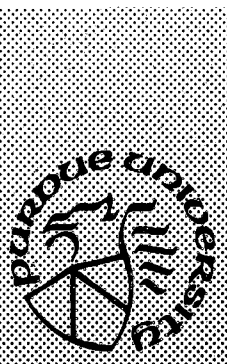
1-1-1990

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: Dynamic Barrier MIMD (DBM)

Matthew T. O'Keefe
Henry G. Dietz

TR-EE 90-9
January, 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM)

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

Keywords: Synchronization hardware, barrier synchronization, dynamic barrier MIMD (DBM), associative memory.

---

1. The companion paper is "Hardware Barrier Synchronization: Static Barrier MIMD (SBM)," 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. The design and performance of *Dynamic Barrier MIMD (DBM)* is described in section 5, followed by conclusions and a description of current work in section 6.

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(i) 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.

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 $x \not< y$ 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* =>

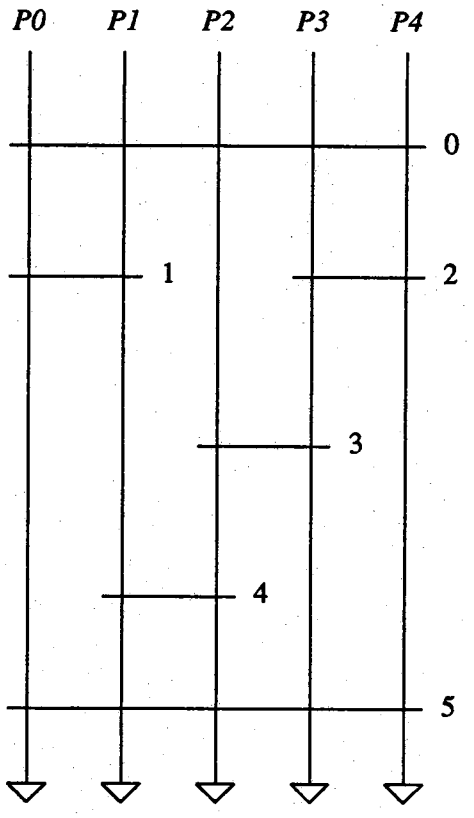


Figure 1

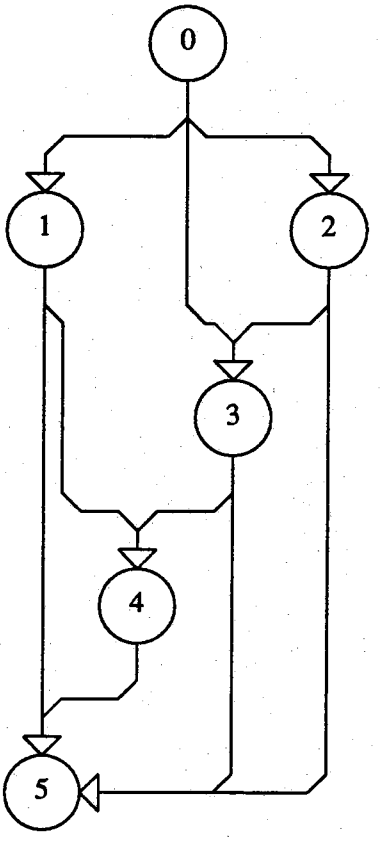


Figure 2

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

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*.

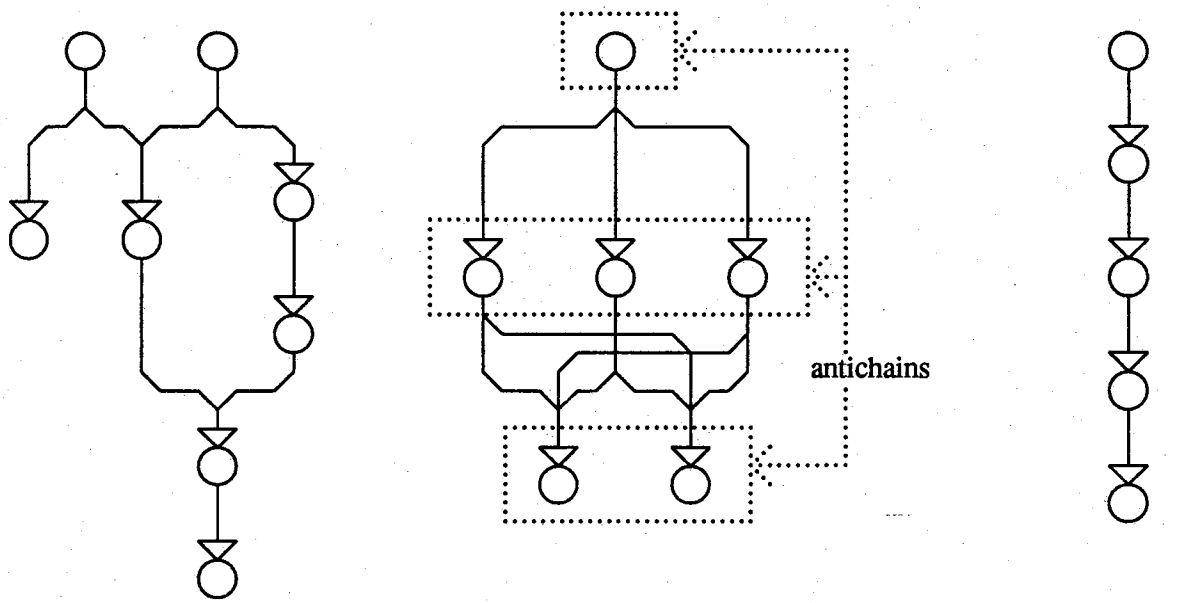


Figure 3

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.

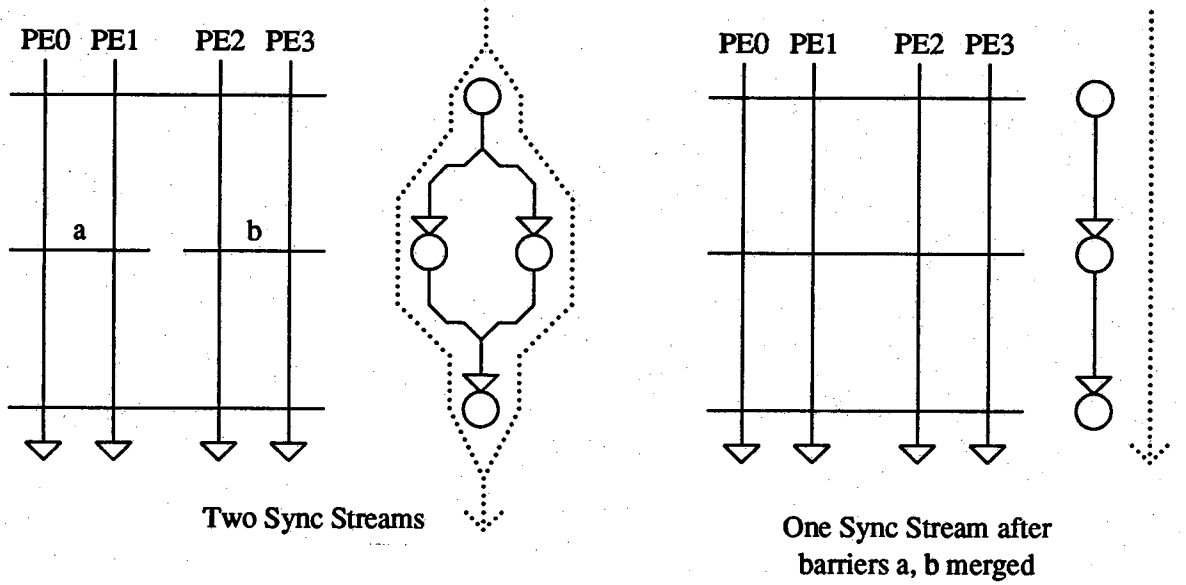


Figure 4: Merging Barriers Reduced Number of Sync Streams

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, as well as the SBM, is described in a companion paper (part I). The implementation and performance of the dynamic barrier MIMD architecture is discussed in more detail later in this paper.

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 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 barrier 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

7. *Trace Scheduling*™ is a trademark of Multiflow Computer, Inc.

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.

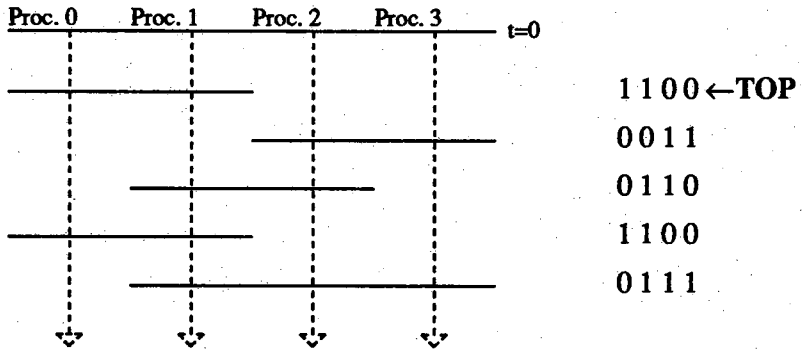


Figure 5

synchronization buffer, and it is this buffer which supports up to *P/2* synchronization streams.

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

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

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. Dynamic Barrier MIMD (DBM)

A *dynamic barrier MIMD* (DBM) machine supports up to *P/2* synchronization streams. This removes the delays inherent in the SBM and HBM due to the ordering (a linear and weak order,

respectively) imposed on the barriers at run-time. Some barriers experience delays in the SBM or HBM when the compile-time and run-time orderings are different. The DBM imposes no particular ordering at run-time, and can execute any partial ordering corresponding to a barrier dag. However, the hardware design is more complex. The DBM synchronization buffer consists of an associative memory and a sophisticated load mechanism. Two possible designs are discussed in this section.

5.1. Lookahead DBM

One possible DBM design employs a large associative memory and attempts to lookahead to the barriers that are most likely to be executed in the near future. The associative memory contains $O(kP/2)$ words, where P is the number of processors and k is an integer constant corresponding to the depth of lookahead into the synchronization stream. This constant determines the number of barriers loaded into the associative memory from each independent stream.

To distinguish between identical barrier masks, a count field exists in each barrier word in associative memory, and this field is decremented as "preceding barriers" match the current PE WAIT pattern. This could be accomplished with the associative memory hardware [Fost76] or with additional circuitry to implement a down counter. In either case, the decrements are performed in parallel for all barriers that are ordered with respect to the barrier that just executed. Only barriers with count fields at zero are eligible for execution.

A similar associative memory design was used in the *dispatch stack* instruction issuing mechanism for multiple functional unit processors [AcKT86]. In the dispatch stack issuing unit dependencies among instructions were tracked using a count field. The operation of the lookahead DBM is now described in more detail using an example barrier embedding.

Figure 6 shows a barrier embedding in eight processors. Let us assume that in figure 6 barriers at the same "depth" in relation to barrier zero (e.g., barriers two and ten) have the same expected execution time. Hence, there is no reason to skew lookahead of the synchronization streams to favor a particular stream, and an interleaved loading approach is appropriate (i.e., barriers are loaded as follows: 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, ...). The barrier dag corresponding to the embedding in figure 6 is given in figure 7. Figure 8 shows the state of a lookahead DBM buffer with lookahead constant $k = 2$ containing the example barrier dag in figure 7 after barrier zero has been executed.

It can be seen that barriers 1, 2, 5, 6, 9, 10, and 13, 14 are loaded into the associative memory. The count fields of barriers 1, 5, 9, and 13 are zero, so these barriers can execute once they match the current PE WAIT pattern; the count fields of barriers 2, 6, 10, and 14 are one, indicating that at least one other barrier currently in the associative memory must execute before they are eligible for execution themselves. For example, barrier 2 must not execute until after barrier one: as can be seen in figure 8, its count field equals one. When barrier 1 matches[9] the WAIT pattern in the WAIT/Input buffer, the count field of

9. Note that during a barrier *match* the zeros in the barrier masks in the associative memory are considered don't cares, i.e., only the ones are important when determining if a barrier mask matches the current PE

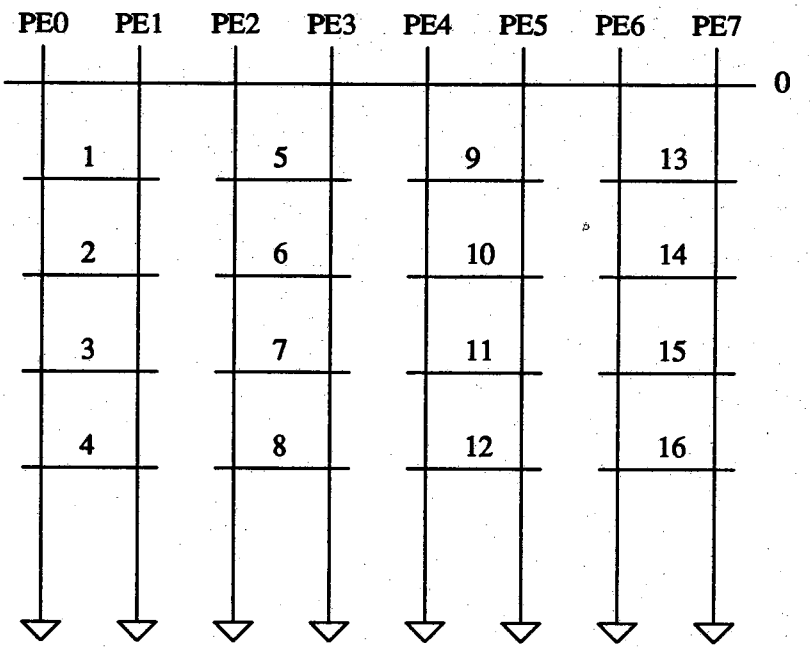


Figure 6: Example Barrier Embedding in Eight Processors

barrier 2 is decremented by one, and it becomes eligible for execution. The count field and loading order implicitly maintain the precedence relations between barriers, as can be seen from the barrier dag in figure 7. This is in contrast to other schemes that require tags to explicitly identify and order different barriers [Gupta89a], [Beck89].

The lookahead DBM operates in two phases: the *match phase* and the *update phase*. During the *match phase*, the associative memory determines if any barrier masks (with count fields equal to zero) match the current PE WAIT pattern. If so, then the masks that match are "OR"ed together and the result is placed in the *OR buffer*. The contents of the *OR buffer* are then output as the PE GO signals, enabling processors stopped at a barrier that has been matched to proceed past the barrier. The ability to "OR" the masks together means that multiple barriers can be executed[10] simultaneously.

After a barrier or barriers have been matched and executed, the *update phase* commences. During this phase, the masks of the barriers that have executed are compared to the current contents of the associative memory. This comparison operation determines if a mask in the memory *overlaps* an executed barrier, i.e., a mask in memory and the executed barrier mask have ones in the same bit position. In this case, there is a precedence relation between the barriers, and the count field of the corresponding masks are decremented.

WAIT pattern.

10. In this work, a barrier is considered to be *executed* when the processors formerly waiting at the barrier have proceeded past it.

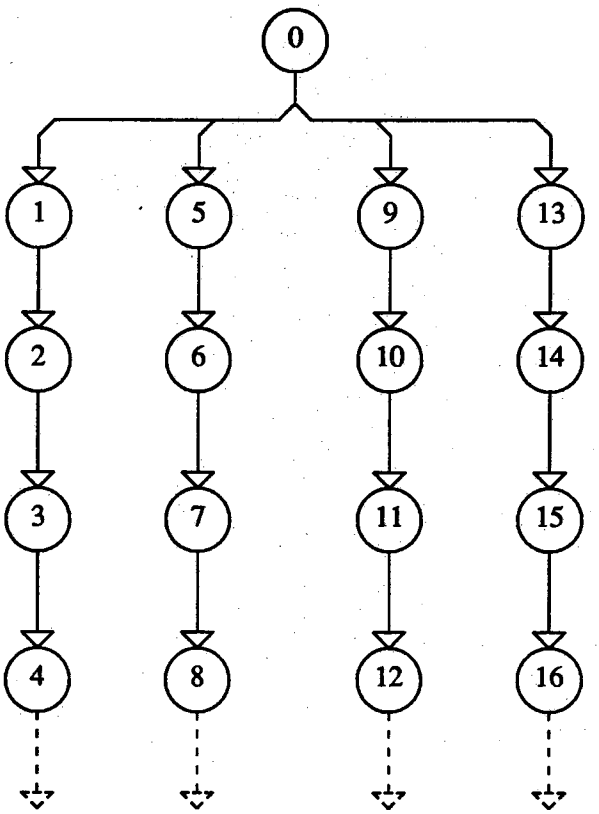


Figure 7: Barrier Dag for Embedding in Figure 6.

The overlap check and decrement operations can be done in parallel across all barrier masks in the associative memory. These steps must be repeated for all barriers that fired during the preceding match phase, so a maximum of *P/2* barrier update operations may be necessary. During this phase, some barriers that were previously unable to execute will become eligible for execution in the next match phase. The time spent in the update phase should be masked by the execution of code following the barriers that executed in the previous match phase.

The empty locations that are left in the associative memory after executed barriers are removed are filled by inserting barriers from the queue. The count field for the inserted barriers is determined by performing a comparison to determine overlap in one bits with barriers currently in the memory. This *overlap comparison* is identical to that used to identify those barriers with count fields that must be updated after a barrier executes. The associative memory computes the number of barriers with overlap, and this result is placed in the count field. The number of barriers with overlap corresponds to the number of barriers that must be executed prior to the execution of the inserted barrier.

As stated previously, if multiple barriers "match" the current WAIT pattern, they are "combined" into a larger enable pattern (the PE GO signals). This barrier combining can be accomplished in constant time in the associative memory, but the count fields for the remaining barriers must then be updated.

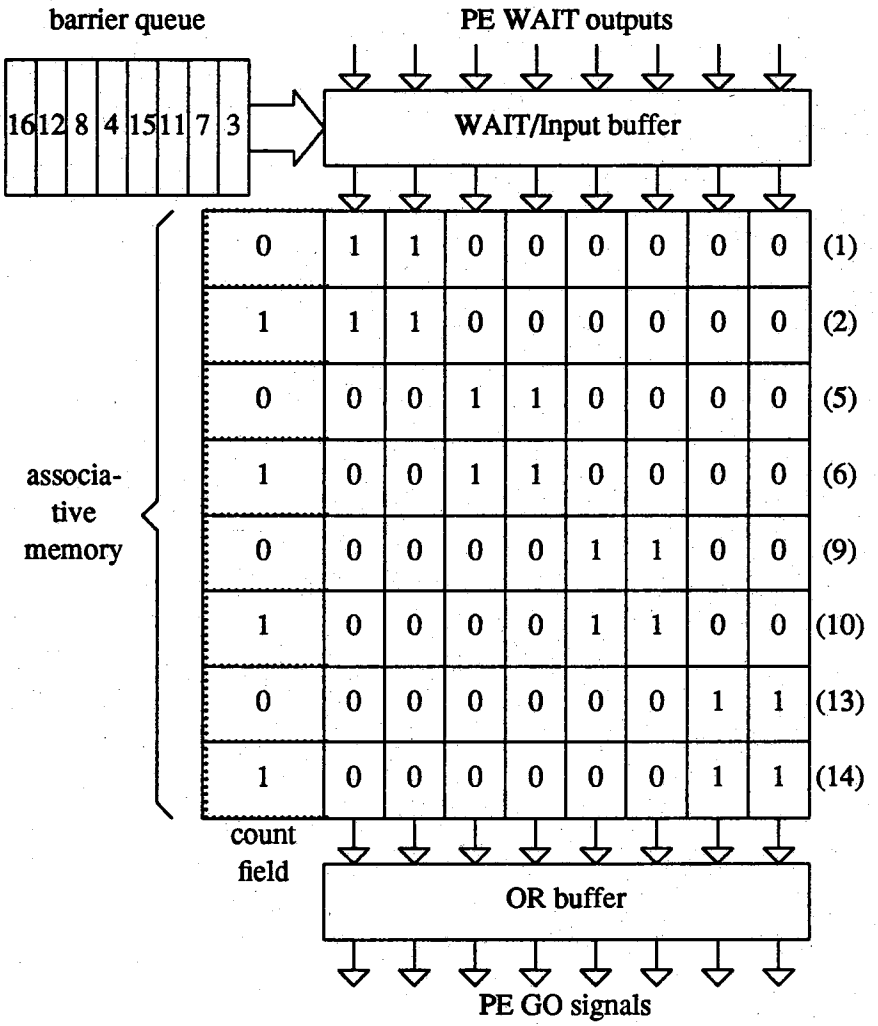


Figure 8: Lookahead Dynamic Barrier MIMD.

Also, barriers can be removed from the queue and placed in the associative memory to replace barrier masks that have been executed. These operations can be seen more clearly in figure 9, which shows the state of the lookahead DBM in figure 8 after barriers 1 and 13 have been executed.

In figure 9, it can be seen that barrier 3 replaces barrier 1, and its count field is set to one, as it is preceded by barrier 2. Note also that barrier 13 has been replaced by barrier 7, and that the count field for barrier 7 is set to two, since it is preceded by barriers 5 and 6. Figure 10 shows the state of the lookahead DBM after barrier 14 is executed. Barrier 14 is replaced by barrier 11, and its count field set to two.

A weakness in the lookahead scheme is that it is sensitive to the order in which the associative memory is loaded, i.e., the order of the barriers in the barrier queue. For example, if a single synchronization stream executes faster the others it may experience delays if the "next" barrier in that stream to be executed is

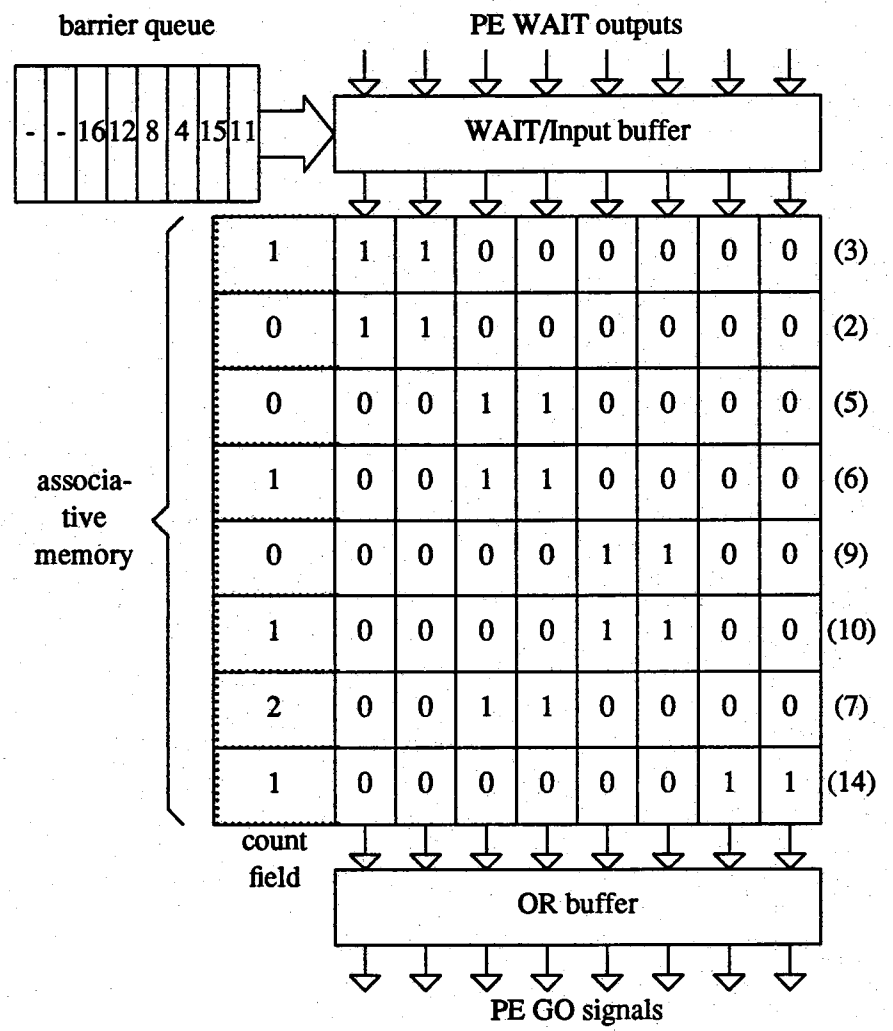


Figure 9: DBM State after Barriers 1 and 13 are Executed.

still in the barrier queue. For example, given the state of the lookahead DBM in figure 10, if processors 6 and 7 encounter barrier 15 then they will be delayed until another barrier currently in the associative memory (e.g., barriers 2, 5, or 9) executes and is removed. In this case, it can be seen that the synchronization stream corresponding to barriers 13, 14, 15, and 16 has executed faster than the other streams, and will be forced to delay until barrier 15 is loaded.

There are several approaches to reducing or removing these delays in a lookahead DBM architecture. One approach is simply to make the associative memory larger, increasing the lookahead depth and making it unlikely that a particular synchronization stream will get this much farther ahead of the the others. Merging barriers can also be used to advantage, as in the static barrier MIMD architecture. Merging barriers on separate streams has the advantage of reducing the possibility of one stream getting far ahead of the others. It also reduces the overhead in the update phase, as fewer executed barriers need to be

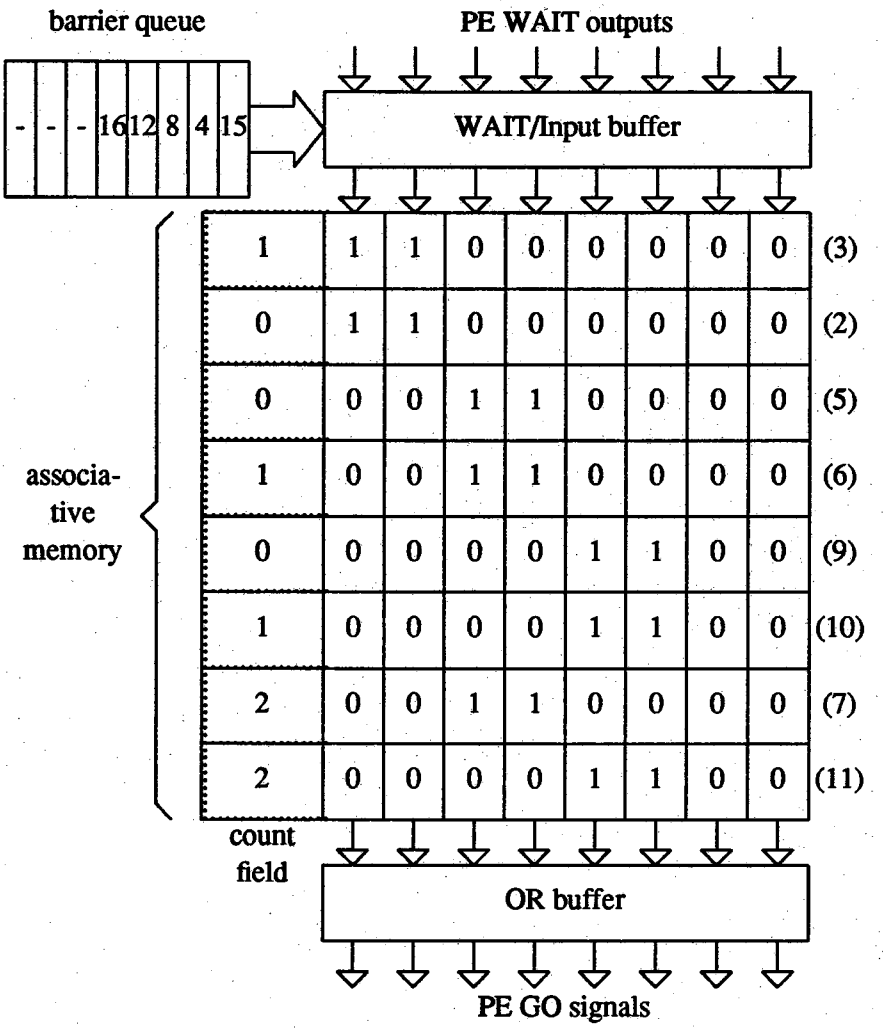


Figure 10: Lookahead DBM after Barrier 14 Executes.

compared to barriers in associative memory. Both these approaches have limits. Clearly, the size of the associative memory cannot be increased without limit, and merging barriers results in longer average delays for the processors involved in the barriers.

Another possible problem occurs when a barrier is generated conditionally, depending on the control flow of the program. For example, consider the barrier dag in figure 7: suppose that barriers 13 through 16 are associated with the iterations of a `while` loop, and that after barrier 16 control returns to the beginning of the loop, and barriers 13, 14, 15, and 16 must be generated and placed in the associative memory. These *conditionally-generated* barriers must be placed in the associative memory early enough to avoid additional delays for the processors associated with these barriers.

The root cause of most of the difficulties in the lookahead DBM design is the existence of the barrier queue from which the associative memory is loaded. The memory helps hide the ordering inherent in

this queue by lookahead barriers, but the problem remains.

5.2. Full DBM

An alternative *pointer-based* dynamic barrier MIMD design, referred to as the *full DBM*, that avoids the problems associated with the lookahead DBM is now described. Instead of looking ahead into the synchronization stream and guessing which barrier masks should be loaded into the associative memory, the barrier dag currently being executed is stored in the barrier processor. Each barrier is stored in a record structure containing the barrier mask and pointers to succeeding barriers in the dag.

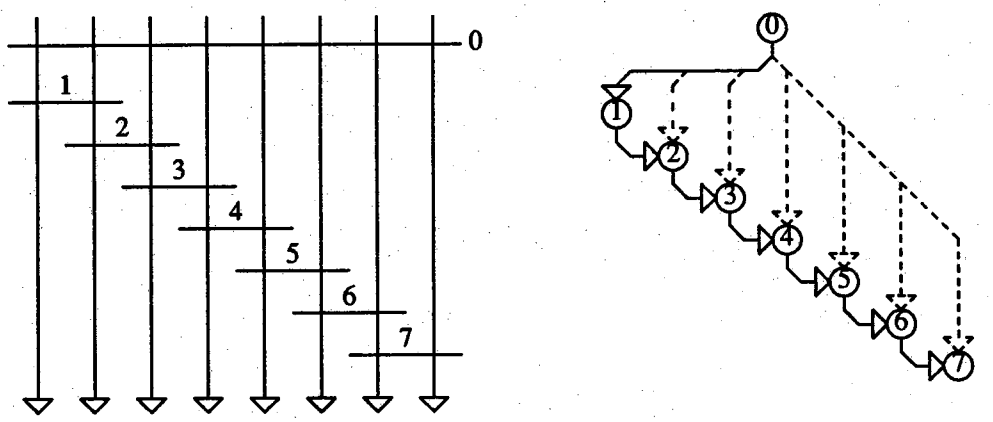


Figure 11: Transitive Reduction of Example Barrier Dag Removes Dashed Edges.

Note that it is only necessary to store the *transitive reduction* of the barrier dag. The transitive reduction of a dag $(B,<_b)$ is defined as the dag $(B,<_b')$ with as few edges as possible, such that the transitive closure of $(B,<_b')$ is equal to the transitive closure of $(B,<_b)$. The transitive reduction of the barrier dag could be computed in a pre-processing phase before barrier execution. In the example barrier embedding given in figure 11, nearly 50% of the edges (dashed edges in the figure) are removed in the transitive reduction of the barrier dag.

Figure 12 represents a full DBM containing the barrier dag from figure 7. Note that the barrier dag is stored in the barrier processor memory: for example, the record for barrier 0 contains pointers to barriers 1, 5, 9, and 13. Barrier 0 has executed (represented as an asterisk to the right of the memory record) and it can be seen that barriers 1, 5, 9, and 13 are currently in the associative memory. The records for these barriers contain pointers to 2, 6, 10, and 14, which have not yet been loaded into the associative memory since they are not yet eligible for execution (represented as an "?" to the right of the memory record.)

In this full DBM, the associative memory size is fixed at $P/2$, and each word contains only the barrier mask as no count field is necessary. This is in contrast to the memory size in the lookahead DBM,

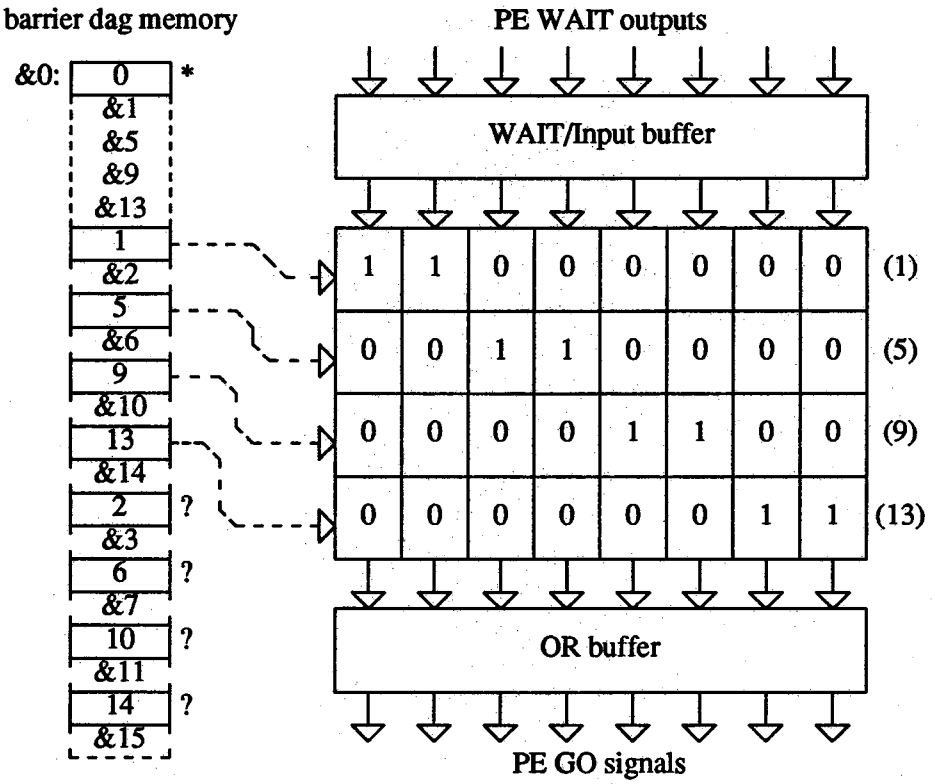


Figure 12: Full DBM.

which can be varied to change the amount of lookahead.

The full DBM also operates in two phases. In the *match phase*, the current PE WAIT pattern is compared to the barrier masks in the associative memory. When one or more barriers "match"[11] the current WAIT pattern, the masks that match are "OR"ed together and placed in the *OR buffer*. The contents of the *OR buffer* are then output as the PE GO signals, enabling processors stopped at a barrier that has been matched to proceed past the barrier. The ability to "OR" the masks together means that multiple barriers can be executed. Except for the lack of a count field in the compare operations, the match phases of the lookahead and full DBM are identical.

11. As in the match operation in the lookahead DBM, the zeros in the barrier masks in the associative memory are treated as *don't cares* during a *match* operation, i.e., only the ones are considered when determining if a barrier mask matches the current PE WAIT pattern.

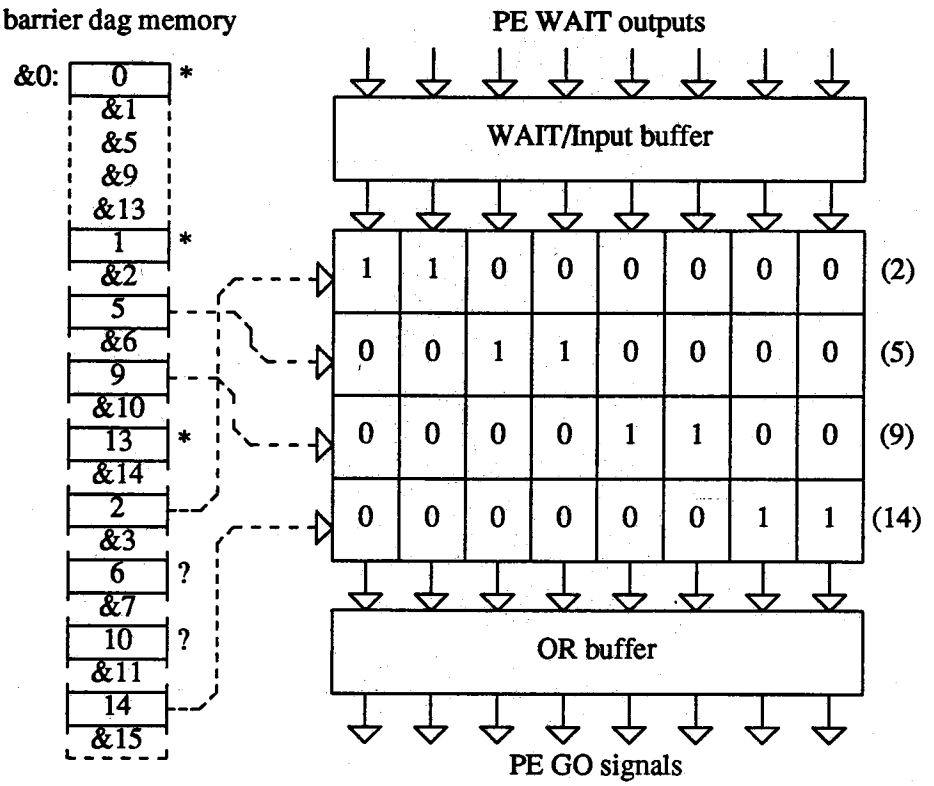


Figure 13: Full DBM after Barriers 1 and 13 are Executed.

During the *update phase* following the match phase, an attempt is made to load the succeeding barriers pointed to in the record structures of the executed barriers. Each succeeding barrier for every barrier executed in the previous match phase is compared to active barriers remaining in the associative memory. The compare operation is an *overlap match* to determine, as in the lookahead barrier, whether a succeeding barrier has any one bits that overlap with one bits in active barriers. This overlap implies a precedence relation between barriers and that means some active barrier must execute before the succeeding barrier can be placed in associative memory. If there is no overlap, then this succeeding barrier can be placed in the associative memory and made active, as all its predecessors have executed. Thus, the associative memory is limited to $P/2$ words, where P is the number of processors, as this is the largest number of barriers that may be active simultaneously.

Given the state of the full DBM in figure 12, suppose that processors 0,1 and 7,8[12] encounter barriers 1 and 13. These barriers will be matched and executed, and the succeeding barriers 2 and 14 will be

12. In this case, the PE WAIT output pattern would be "11000011." This same pattern will be output on the PE GO signal lines to enable processors 0, 1, 7, and 8 to proceed past their barriers.

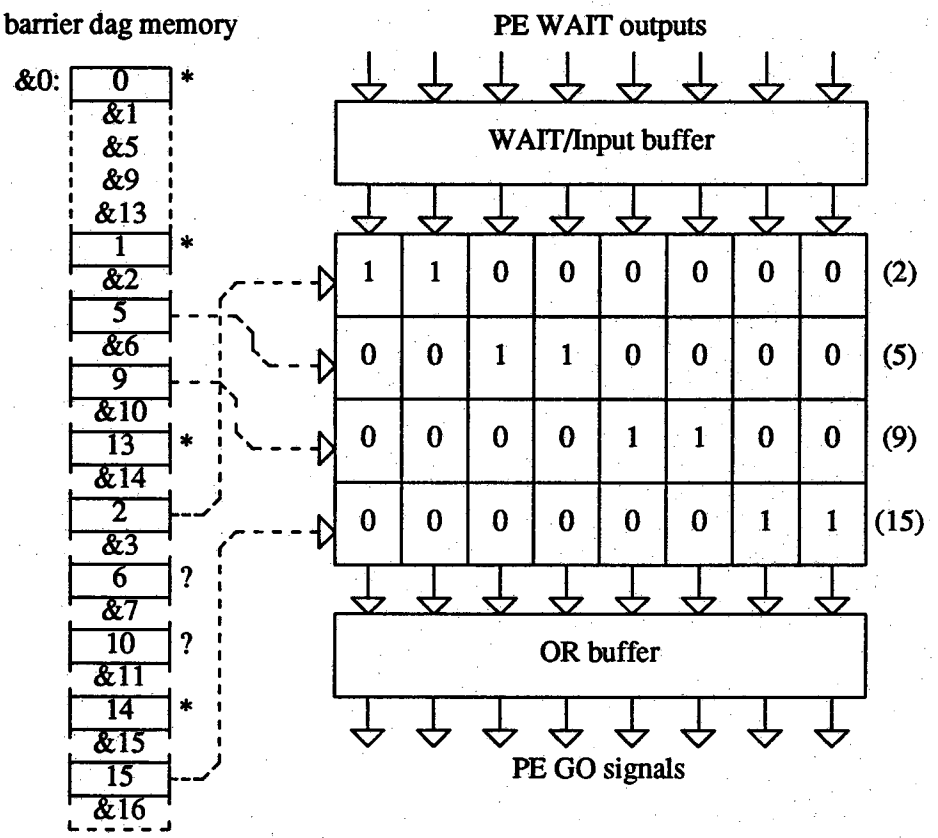


Figure 14: Full DBM after Barrier 14 Executes.

overlap matched and then loaded into the associative memory. The new state of the full DBM is shown in figure 13. Suppose now that processors 7 and 8 reach barrier 14 and this barrier is executed. Its succeeding barrier is 15, and it is loaded into the memory. The new state is shown in figure 14. Note that, unlike the state of the lookahead DBM for this same sequence of barrier executions, barrier 15 is now loaded into the memory and is ready to execute.

This pointer-based scheme is not sensitive to any compile-time ordering of the barrier dag as it stores this structure directly in the barrier processor memory. Complications may arise when barriers that are conditionally generated according to program control flow must be "spliced" into the barrier dag data structure in memory. It does not appear that this will be a significant problem for synchronization structures resulting from the parallelization of most common programming language constructs, but further research in this area is required.

6. Conclusions and Current Work

This paper presented designs for two different flavors of barrier MIMD architecture: the full DBM and an approximation to the DBM called the Lookahead DBM.

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 a DBM. However, this was not the only benefit found.

Although either type of DBM requires significantly more complex circuitry than an SBM, the DBM architectures also can easily execute each barrier within a few clock ticks. For this reason, these architectures 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 DBM and lookahead DBM designs given are at least as capable as previous hardware barrier mechanisms, yet permit a fast implementation.

Because the DBM, unlike the SBM, allows multiple synchronization streams, it is very well suited to an environment where multiple independent parallel programs are permitted to share a machine. For each parallel program, a barrier schedule can be statically determined independent of all other program schedules. When that program is to be executed, the static schedule for its processes is simply appended to the DBM schedule which exists at that time. This ability is not directly supported by any of the other hardware barrier mechanisms, yet, it greatly increases the utility of the DBM as a general-purpose synchronization mechanism.

Other ongoing research includes techniques for parallelizing and scheduling complete programs and performance analysis of DBM as a general-purpose synchronization mechanism.

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.*