Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors

Part I: Barrier Architecture†

W. E. Cohen, H. G. Dietz, and J. B. Sponaugle

Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu

Abstract

Parallel computers constructed using conventional processors offer the potential to achieve large improvements in execution speed at reasonable cost, however, these machines tend to efficiently implement only coarse-grain MIMD parallelism. To achieve the best possible speedup through parallel execution, a computer must be capable of effectively using all the different types of parallelism that exist in each program. A combination of SIMD, VLIW, and MIMD parallelism, at a variety of granularity levels, exists in most applications; thus, hardware that can support multiple types of parallelism can achieve better performance with a wider range of codes.

In this paper, we introduce a new hardware barrier architecture that provides the full DBM functionality we discussed in [OKD90a], but can be implemented with much simpler hardware. This mechanism can be used to efficiently support multi-mode moderate-width parallelism with instruction-level granularity (i.e., synchronization cost is approximately one `LOAD` instruction), as described in the companion paper [CoD94].

Keywords: Parallel Architecture, Dynamic Barrier Synchronization, MIMD/VLIW/SIMD Mixed-Mode Computation, Partitionable Systems, Instruction-Level Parallelism.

† This work was supported in part by the Office of Naval Research (ONR) under grant number N00014-91-J-4013 and by the National Science Foundation (NSF) under award number 9015696-CDA.

1. Introduction

While the market for massively parallel supercomputers has remained limited, the demand for more conventional computers has grown to the extent that many microprocessors are now considered "commodities." The high sales volume for these parts has justified heroic design and fabrication efforts, with the result that some microprocessors offer uniprocessor performance that is simply beyond what can be achieved with the more modest resources available for designing specialized processors for supercomputers. A similar argument applies to the development of system software. Thus, building massively parallel supercomputers using standard microprocessors seems very attractive. The problem is simply that these chips were not designed to be used in a speedup-oriented parallel configuration.

Nonetheless, using these microprocessors, it is easy to build a massively parallel system that can achieve reasonable performance for very large-grain parallel applications; one might even be able to use workstations connected by local area networks (LANs). The challenge is to achieve good performance with fine-grain parallel applications. It is possible to build systems that use standard chips yet yield good fine-grain performance; however, the *system* must be designed very carefully to achieve this goal.

Some of the system attributes most important to achieving good fine-grain performance are:

[1] Good processing element (uniprocessor) performance.

[2] Near-zero cost synchronization (and hence the flexibility of choice of execution mode).

[3] High-performance communications with both low latency and high bandwidth.

[4] Compilers that can effectively use the system, as opposed to individual processors.

It is a relatively simple matter to use a commodity microprocessor to cheaply provide [1], but cost-effective ways to achieve [2], [3], and [4] are surprisingly elusive. The key question is why?

Obtaining near-zero cost synchronization is difficult for a number of reasons. Perhaps the most fundamental cause is the propagation delay of electrical (or optical) signals in a parallel machine; however, even the most distant processing elements within a massively-parallel machine can be just tens of nanoseconds apart. Thus, the synchronization speed of most parallel machines is not limited by distance, but rather by the inappropriateness of their synchronization model. For example, synchronization methods that require synchronization signals to be "routed" between processors are generally slow because dynamic routing implies delays due to switching and propagation through active components.

Further, synchronization cannot be efficient if invoking each synchronization requires execution of an expensive sequence of instructions. Any synchronization operation across multiple processors requires each processor to notify the processors that it should synchronize with and to obtain an acknowledgement that the synchronization has completed; both of these operations require off-chip communication. In modern microprocessors, the combination of a high internal clock rate, deep pipelining, and out-of-order instruction execution, makes off-chip references expensive and their timing imprecise. Thus, it is vital that the number and cost of off-chip references needed to implement a synchronization operation be minimized.

From an engineering point of view, construction of high-performance communication hardware is not particularly difficult; low latency and high bandwidth are both relatively easy to obtain at reasonable cost. The problem lies not in actual communication of data, but in the processes of sending, routing, and receiving data. Sending and receiving data must both require execution of very few instructions; routing (including arbitration of shared paths) must be made as efficient as possible.

Finally, compilers for single processing elements leave management of interactions between processing elements to the application programmer, but the level of detail needed to efficiently schedule these interactions is not accessible in a high-level language. Even if that level of detail were visible to the programmer, making the best use of such a machine involves complex VLIW-like code scheduling based on detailed machine-dependent timing analysis — a burden that few programmers would be able to bear. Without a compiler that treats the system as a system, the programmer can expect to spend a lot of time writing, and performance-tuning, highly non-portable code.

In this paper, we suggest that all these problems can be cheaply solved by implementing a very simple and fast barrier synchronization hardware mechanism. Relative to the above issues:

[1] The new mechanism can efficiently use conventional processors as processing elements.

[2] This synchronization hardware is not PE-to-PE, but PE-to-synchronizer; therefore, there is no routing. Using conventional processors, the cost of invoking a synchronization is essentially one off-chip operation (i.e., a `LOAD` operation). The result is that fine-grain synchronization time is dominated by signal propagation delay.

[3] Because the synchronization is so precise, the communication mechanism can be *statically scheduled*. This can greatly simplify the system by minimizing routing and arbitration hardware. It also improves performance by reducing the send/receive software overhead, as was observed in the PASM prototype (see Section 4.4 for a discussion of PASM's barrier mechanism) [BeS91].

[4] Although the compiler must aggressively use timing analysis to determine how to optimally schedule code for such an architecture, the hardware allows the compiler to view the machine as a much simpler target, e.g., as a straightforward partitionable SIMD. Although some efficiency might be sacrificed, building a good compiler for such a simple execution model is much easier than building a compiler to achieve comparable performance for the more conventional, loosely-coupled, MIMD models. The ability to switch execution modes on demand also allows for optimization of the execution model to match the program structure.

Thus, the way to achieve a familiar environment with better performance and lower cost per unit performance is to use compiler timing analysis and code scheduling technology in concert with careful architectural design so that a set of commodity microprocessors can behave as a tightly-coupled parallel system with a nominal amount of "glue logic." We suggest that the key to this is the use of a particular type of hardware barrier synchronization in conjunction with compiler code scheduling.

In this paper, we ignore the "fancy" compiler technology [DiO92]. The focus of this paper is the new barrier mechanism. The companion paper [CoD94] details how this mechanism can implement various execution modes that can be effectively used with conventional compiler technology.

2. Barrier Synchronization

Although not commonly used in the way we suggest, barrier synchronization has long been an important mechanism for coordinating parallel processes. For this reason, many research efforts have focused on efficient implementations in both hardware [Lun87] [Pol88] [Gup89] and software [Bro86] [ArJ87] [HeF88] [Lub89]. In 1987, based on observing properties of the PASM prototype [SiN87], we proposed a new class of fast barrier synchronization architectures [DiS89] [OKD90] [OKD90a].

A traditional barrier is a synchronization point. A processor typically performs the following three steps upon reaching a barrier:

[1] Marks itself as present at the barrier.

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

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

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

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

This subtle difference is actually the enabling condition for the use of static timing analysis and compile-time code scheduling. In essence, immediately after executing a barrier, the compiler sees exactly the same scheduling constraints for the MIMD hardware that it would have had for a similar VLIW machine. Thus, static instruction scheduling for SIMD and VLIW [Ell85] [CoN88] machines can be extended into the MIMD domain [DiS89] [DiO92] — and to a set of commodity processors. This implies that conceptual timing hazards can be resolved at compile-time, without the use of runtime hardware for synchronization or arbitration — the most costly components in most parallel machines.

3. The New Barrier Architecture

As discussed in the companion paper [CoD94], to support multi-mode instruction-level parallelism, the barrier mechanism must operate quickly and provide precise timing constraints.

However, there are actually two separate classes of barrier mechanisms that can provide these properties: *static* [OKD90] and *dynamic* [OKD90a]. The difference between these techniques involves how the hardware determines which barrier synchronization should be the next to fire. The static version assumes that there is a complete order for all barrier synchronizations, whereas the dynamic version allows barrier synchronizations to be specified as a partial order. Thus, a dynamic barrier mechanism allows barrier synchronizations involving disjoint portions of the machine to fire in any order.

To illustrate the difference between static and dynamic barriers, consider the simultaneous execution of two different programs on a four processor machine such that program *A* is executed by processors 0 and 1, and program *B* is executed by processors 2 and 3. Since these two programs are independent, each may contain any number of internal barrier synchronizations, as shown in Figure 1.[1] While either the static or dynamic barrier mechanism can be used, the static mechanism requires a static complete ordering of the barriers, while the dynamic mechanism allows a partial ordering.

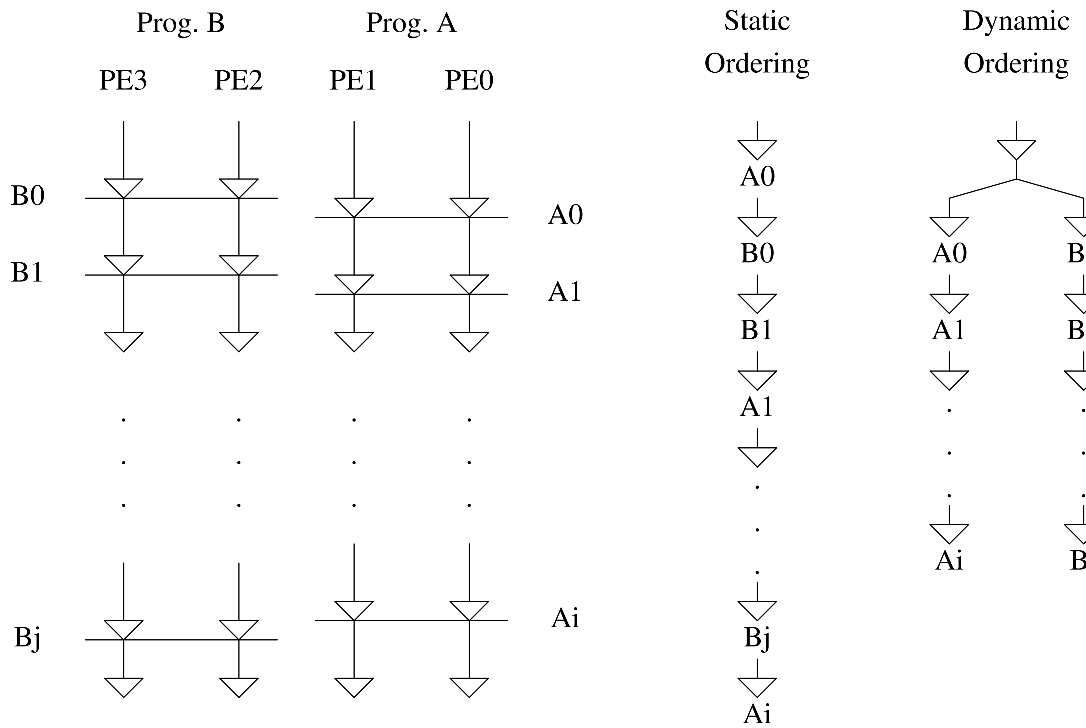

Figure 1: Difference Between Static & Dynamic Barriers

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

[1] Notice that Figures 1 and 2 use the abbreviation "PE" to refer to a processing element, i.e., a processor.

can be introduced by blocking. Thus, the dynamic mechanism is clearly superior to the static mechanism.

There is, however, one problem with the dynamic mechanism: until the design presented in this paper, it appeared that the only way to implement a dynamic mechanism would require an associative memory and sophisticated system to enqueue barrier patterns from a partial ordering [OKD90a]. This led us to conclude that only the very simple static barrier mechanism was practical.

The primary contribution of this paper is a new dynamic barrier design that is effectively implemented by replicating the very simple static barrier architecture, presented in [OKD90], for each processor. Notice that the distributed nature of the new barrier synchronization hardware necessitates a distributed method for enqueuing barriers; the new barrier architecture solves this problem by associating a "return value" with each barrier that encodes the next barrier to be "enqueued." A complete dynamic barrier synchronization is accomplished by this mechanism with a cost approximately equal to execution of a single `LOAD` instruction.

Although the new architecture can be used with most conventional processors, it does place some restrictions on the processor interface. These restrictions are discussed in Section 3.1. The architecture itself is presented in Section 3.2. Although the architecture is very simple, it is not immediately obvious how it can be used to manipulate the masks that determine which processors should participate in each barrier. Thus, Section 3.3 describes the basic handling of barrier masks. Although this barrier architecture is intended primarily for parallel systems with modest numbers of processors, Section 3.4 covers issues relating to how the approach can be scaled to massively-parallel systems.

3.1. Interface To Conventional Processors

Unless the processor provides an additional user-controllable off-chip interface (for example, the Am29050's `LOAD` and `STORE` instructions provide three "OPT" bits [AMD91]), the processor address and data lines must implement the complete interface with the barrier mechanism. This results in requirements for the address and data busses, as well as a number of restrictions on the internal ordering and operation of memory accesses.

3.1.1. Address And Data Bus Sizes

The sizes of the address and data busses determine the maximum number of processors that can be synchronized using a *single* `LOAD` instruction.

As Sections 3.2 and 3.3 explain, fields in the `LOAD` instruction's read address are used both to inform other processors that this processor is waiting at a barrier and to broadcast at least one bit that can be used to construct the mask for the next barrier in which this processor will participate. Thus, if there are n processors, a barrier synchronization will directly use n+1 address bits. In addition, the address space must be arranged such that the hardware can distinguish between a barrier reference and a normal memory reference. Thus, some portion of the address map —

large enough to accommodate n+1 address bits — must be reserved for the barrier mechanism. This can be done by any of a variety of techniques, the simplest of which is to dedicate one additional address bit to distinguish between barrier and normal memory references.

When a `LOAD` instruction implementing a barrier completes, the value loaded contains bits gathered from each of the processors that participated in the barrier synchronization. Since at least one bit is gathered from each processor, the value loaded must be at least n data bits in length. While this can be done using a data bus of fewer than n bits, we suggest that it is most practical to constrain n to not exceed the width of the data bus. Notice that these data bits are used only for barrier operations; there is no impact on the number of data bits available for ordinary processor operations.

3.1.2. Internal Ordering And Access Restrictions

In addition to the relatively mechanical restrictions on the address and data busses, there are several other processor requirements that relate to internal properties of the processor. Broadly, these requirements fall into three categories: address mapping, pipeline structure, and processor stalling.

Address mapping becomes an issue for several reasons. The first is that on-chip caches could cause a `LOAD` to be completely internal to the processor chip, thus not effecting the desired synchronization. However, even more bizarre behavior could result: caches often prefetch the next few words after a cache miss, and this could cause a barrier `LOAD` to be issued multiple times, perhaps with the barrier mask bits "incremented" — whatever that might mean. There is also the minor difficulty that the barrier mechanism can only examine the address bits that exit the chip, i.e., the physical address; the complication is that the barrier fields in the virtual address might or might not directly correspond to the barrier fields in the physical address. Thus, care must be taken to ensure that a barrier `LOAD` is sent off-chip in the correct format. Fortunately, most modern microprocessors allow portions of the address map to be marked as non-cacheable, and a virtual-to-physical mapping that does not interfere with barrier operations can usually be created. However, processors that require the entire cache to be flushed or disabled for such references cannot efficiently use our barrier mechanism. Likewise, if the same address is not used to represent the same barrier on different processors, extra instructions may be needed to form new barrier patterns.

Pipeline structure is an issue because barrier synchronization is used to ensure that all participating processors have reached the barrier before any can execute past the barrier, yet there are two basic ways in which pipelining can cause later instructions to be partially executed before the barrier has fired. Perhaps the most obvious of these reorderings occurs in processors supporting out-of-order execution, however, there can also be order violations in that different pipeline stages may cause other operations to occur out-of-order with respect to the barrier `LOAD`'s off-chip reference. These problems occur because the processor pipeline interlock hardware does not know that all instructions after a `LOAD` that implements a barrier actually depend on that `LOAD`, reguardless of register usage. The solution is generally to either force a dependence that the pipeline hardware can detect or to simply add NOPs to ensure that order around barrier synchronizations is preserved (compiler optimizations can sometimes reduce the number of NOPs required).

Processor stalling is perhaps the most fundamental problem in implementing a barrier synchronization, since there must be some way to stop the processor until the barrier has fired. There are many different ways in which this can be done — the problem is that most techniques would increase the minimum time to synchronize well beyond the cost of a single LOAD. Thus, we suggest that the ideal technique is to have the barrier hardware simply insert what appears as memory wait states until the barrier LOAD has fired. However, some processors do not allow an arbitrarily long sequence of memory wait states to be inserted on a LOAD. In such cases, alternative solutions range from software polling to use of an interrupt signal, but performance will be lower.

3.2. Barrier Architecture

Given a processor that provides the facilities described above, the interface to the new barrier mechanism is trivial: the address referenced by the LOAD must be decoded as a barrier synchronization and all relevant signals must be appropriately latched. The new barrier architecture, which is replicated for each processor, is depicted in Figure 2.

As suggested in the introduction to Section 3, the reader will notice that the logic tree in Figure 2 is virtually identical to Figure 6 in [OKD90] — the original static barrier mechanism. The new architecture is even simpler in that it lacks the barrier mask queue. However, unlike the static design which uses only a single tree, the new dynamic design replicates this tree for each processor.

The purpose of this tree is to determine whether the processor it is associated with needs to wait for the barrier synchronization to complete. The processor needs to wait if and only if at least one processor it should synchronize with has not notified this processor's tree that it is synchronized. The interesting twist is that each of the processors is responsible not only for determining when it may proceed past the barrier, but also for informing all and only the other processors in that barrier that it is waiting. Both these functions are accomplished by use of the barrier mask, which is extracted from the LOAD address. Thus, instead of having just one output line from each processor, there is one output line for each of the other processors in the machine (i.e., for n processors, $O(n^2)$ wiring complexity), and it is the responsibility of each processor to set these lines appropriately for each barrier.

If the set of processors participating in each barrier was known at compile time, or did not change during execution, the above portion of the hardware would be sufficient. However, to support enqueuing of different barrier masks, the barrier architecture also constructs a return value for the LOAD. The value LOADed is identical to the barrier address referenced in the LOAD, except in that the `flag` field is forced to `0` and the barrier mask is replaced by the `flag` bits gathered from all the processors. This gathering is implemented by direct wiring.

The best way to understand the new architecture is to examine how the combination of processor and barrier hardware implement basic operations on barrier masks, as presented in the following section.

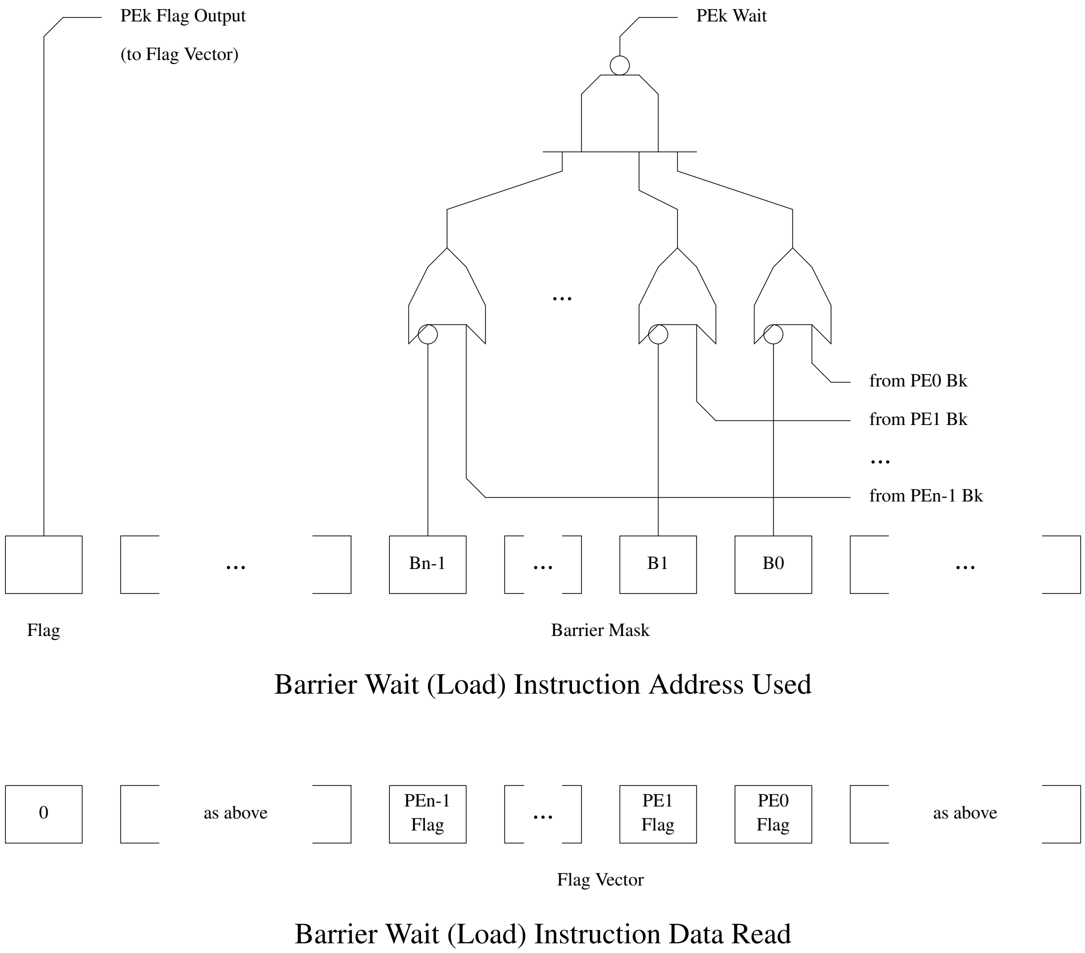

Figure 2: The New Barrier Architecture

3.3. Manipulation Of Barrier Masks

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

3.3.1. Dynamic Barrier Synchronization

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

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

[2] Internal to the processor, the bit mask is aligned to the "Barrier Mask" positions B0 through Bn-1 (as shown in Figure 2), and is inserted into a base address that will be decoded as a barrier synchronization request.

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

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

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

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

[7] Upon the tree's signal becoming a `0`, the barrier hardware is reset (the barrier mask latches are cleared) and the processor is allowed to complete the access that initiated the barrier. The value read from the data bus is ignored.

In practice, steps [1] and [2] can be combined by simply having the processor maintain the barrier address rather than constructing the address from the mask for each barrier. This optimization is used in the code examples given in the companion paper [CoD94].

It is also useful to note that, since all processors determine their firing conditions independently, there is no conflict in firing multiple non-overlapping barriers simultaneously. This constitutes yet another improvement over the original dynamic design [OKD90a].

3.3.2. Partitioning A Barrier Group

Because it is the responsibility of each processor to know the set of processors with which it will synchronize, a method is needed to notify all processors that will participate in a barrier as to the complete set of processors in its barrier group. If the partitioning into groups is known at compile time, as it might be in ELP [NiS90], then this notification is accomplished by simply placing appropriate lists of barrier addresses within the code generated for each processor. However, it is more common that this partitioning is not statically known. Instead, new barrier groups are most often created by partitioning an existing barrier group into two groups based on the run-time evaluation of a conditional expression. Those processors for which the condition evaluates as true form one barrier group and those for which it evaluates as false form the other.

The dynamic partitioning of a barrier group is accomplished by using a barrier synchronization to ensure that all processors in the original barrier group have evaluated their conditional expressions. This same barrier is also used to gather and broadcast the results of the conditions for all the processors. The barrier operation sequence is as described in Section 3.3.1, with the following changes:

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

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

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

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

Change step [7] to:

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

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

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

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

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

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

3.3.3. Enlarging A Barrier Group

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

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

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

Notice that there is nothing to prevent processors from partitioning or enlarging barrier groups using whatever communication hardware mechanisms are available, because barrier masks/addresses can be transmitted by any mechanism capable of sending integer/address values. Because these mechanisms are highly machine specific and are not needed to support the instruction-level multi-mode operations described in the current work, the use of such mechanisms is beyond the scope of this paper.

3.4. Scalability

As indicated in Section 3.1.1, there are limits to the number of processors that can be synchronized with a single `LOAD` instruction. For most modern processors, the direct implementation of the above architecture is limited to systems with fewer than about 32 processors. However, in a machine using many more than 32 high-performance processors, signal propagation delays alone are likely to extend the cost of synchronization well beyond the cost of a single `LOAD`. Thus, the most reasonable scaling method is to use this barrier architecture within a cluster and another method across clusters; in fact, this is exactly how we are implementing the CARD machine (see Section 4.1).

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

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

4. Performance Of Barrier Implementations

The use of barrier synchronization to support static scheduling either through aggressive compile-time code scheduling or through implementation of statically-scheduled execution modes (e.g., SIMD) has many proponents, and a number of machines have been built to take advantage of these abilities. In this section, we briefly review some of the competing designs to determine the range of execution models and grain sizes for which each implementation is effective; the more models supported and the finer the grain size, the better the performance of the barrier mechanism.

To aid in this evaluation, we first describe the CARD system — which incorporates the mechanism described in this paper. The succeeding sections discuss the FEM, FMP, PASM, and CM-5, each of which was the first to introduce some aspect of this type of barrier synchronization. Finally, we offer very brief comments on a few other machines that contain barrier hardware.

4.1. CARD

The CARD system is based on the CARDBoard, a Compiler-oriented Architecture Research Demonstration Board. It was designed specifically to demonstrate a variety of new compiler and architecture interactions, including the new barrier architecture described in this paper.

CARDBoard will be the first complete computer system built around a full implementation of the dynamic barrier mechanism described in [OKD90a]. For this reason, we felt it was wise to begin by developing a prototype of the barrier hardware that we could experimentally evaluate and modify. This barrier-only prototype is called PAPERS, Purdue's Adapter for Parallel Execution and Rapid Synchronization; it is designed to connect multiple PCs using only their standard parallel printer ports. Because we are still experimenting with the PAPERS prototype, the details of the CARDBoard barrier mechanism are subject to change, but the basic system design is firm.

Unlike most academic prototypes, the CARDBoard is intended to be an easily replicated public domain design, i.e., it is designed to be cheap enough for universities to afford multiple large configurations, and for individual students to afford the minimum configuration. Toward this goal, the system is hosted by a personal computer running Linux (a freeware version of UNIX), and the basic CARDBoard is a single ISA bus card integrating four processors.

Although the CARDBoard design can accommodate any of a variety of processors, the version currently being prototyped uses Am29050 processors at 25 MHz. These chips are inexpensive and can complete up to one floating point multiply-add per cycle; however, they are also pin-compatible with the Am29000, making the cost for an integer-only CARDBoard very low. Each processor has its own local memory made of fast static RAM, but can also connect to a shared bus to access a shared memory, another processor's local memory, the ISA bus interface, or registers used to communicate with other CARDBoards. The CARDBoard's internal shared bus and interface circuitry is implemented using a field programmable gate array. Although multiple CARDBoards can be used to make a parallel machine within a single ISA bus chassis, a low-cost fiber optic interface board will facilitate making a single system span multiple ISA bus chassis containing up to 256 CARDBoards.

The current CARDBoard design actually has two separate barrier synchronization mechanisms, each using a four-bit barrier mask and closely following the architecture described in this paper. The local barrier mechanism operates exclusively within an individual CARDBoard, allowing arbitrary dynamic barrier partitioning. The global barrier mechanism operates across CARDBoards, and will probably use each barrier mask bit to represent a group of boards; however, this aspect of the design has not been finalized. In either case, a barrier synchronization is accomplished in a single `LOAD` operation with delay typically within a few cycles of signal propagation time.

Perhaps most importantly, the barrier synchronization support on each CARDBoard does not significantly increase the complexity of the board, yet provides full dynamic barrier operation supporting MIMD, VLIW, and SIMD execution models. In fact, the CARDBoard compilers use static timing analysis to arbitrate and schedule most shared bus accesses; without barrier hardware, the shared bus implementation would have to be much more complex to achieve similar performance.

4.2. FEM

The term “barrier synchronization” was first used in a paper by Harry Jordon [Jor78]. This paper described the FEM — Finite Element Machine — a MIMD machine designed to efficiently manage problems with an SPMD structure such that all processors must complete one phase of the program before any can enter the next.

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

4.3. FMP

The Burroughs FMP [LuB80] was designed to be the Flow Model Processor in a system for performing aerodynamic simulations. Although it was never built, it is the first machine design to incorporate hardware barrier synchronization with timing properties and hardware structure similar to the barrier mechanism discussed in this paper.

The FMP’s barriers are implemented using an `AND` tree that spans all 512 processing elements. When a PE executes a `WAIT` instruction, that instruction does not terminate until a `GO` signal is received. The `GO` signal is received by all PEs within 160ns after the last PE has begun to execute a `WAIT` instruction. Given that each PE has a peak performance of 3 MFLOPS, this synchronization cost is only about half the time taken to perform a floating point operation — very fine grain.

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

4.4. PASM

The PASM prototype [SiN87], a PArtitionable Simd Mimd machine designed and built at Purdue, has the distinction of being the first machine to implement both instruction-level SIMD and MIMD execution using conventional processors and special barrier synchronization hardware.

Although PASM’s design is said to scale to 1,024 processors, the PASM prototype implements just 16 PEs, each of which is a standard Motorola MC68000 microprocessor. These 16 PEs are divided into four groups; each group has a separate control unit incorporating a Motorola

MC68000 and custom hardware implementing a SIMD fetch unit, enable masking, and barrier synchronization. Thus, PASM can be partitioned into at most four barrier groups (or 32 groups for a 1,024 PE machine), with partitioning restrictions similar to those of the FMP.

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

Although the above sounds like an extension of the proposed barrier mechanism (as in the CARDBoard), it provides only a fraction of the functionality. One reason is that only the static barrier ordering is permitted. However, the more severe limitation is that only the control units can enqueue barrier patterns and return values. Thus, PASM is very inefficient if mask patterns must be derived from the result of runtime evaluation of parallel expressions — empirically, the most common case. Stated differently, PASM's barrier mechanism is a very powerful and efficient implementation of static barrier synchronization hardware, but PASM's barrier enqueue hardware is too centralized.

4.5. CM-5

The Thinking Machines CM-5 [TMC92] is a commercial supercomputer that combines conventional processors in an architecture supporting both MIMD and SIMD execution. It has the distinction of being the first machine to implement SIMD execution without a traditional SIMD control unit broadcasting instructions — also a feature of our SIMD execution models, as discussed in the companion paper [CoD94].

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

A barrier synchronization is initiated by sending a control message noting arrival at a barrier. The synchronization is terminated by receiving a barrier-completed message. This decoupling is sometimes called a “fuzzy barrier” mechanism [Gup89]. Although the control network has the ability to be partitioned at configuration time, there is no support for partitioning barrier groups under program control. The result is a static barrier hardware structure resembling that of the FMP.

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

4.6. Other Barrier Machines

There are a variety of other machines implementing some type of hardware barrier mechanism. The following section summarizes the relevant characteristics for some of the better known machines.

4.6.1. Triton/1

The Triton/1 [PhW93] is a 260 PE SIMD/MIMD machine closely resembling the PASM prototype. There are many differences, but the similarities are striking: PEs are based on MC68010 microprocessors, the interconnection network is very fast, and the mechanism for SIMD instruction broadcast is much like that in PASM. However, barrier synchronization is implemented using a “global wired-OR” across the processors. Thus, Triton/1 supports only static barrier synchronization in which all processors participate.

4.6.2. OPSILA

Unlike the other machines described in this paper, OPSILA [AuB86] employs PEs constructed using bit-slice processors (AMD 29116). Despite this difference, the methods used to support SIMD and SPMD are again similar to those used in the PASM prototype. The most serious difference between OPSILA and PASM is that OPSILA’s interconnection network is only operable in SIMD mode; in SPMD mode, PEs cannot communicate. Although SIMD execution is directly implemented in hardware, a barrier synchronization (referred to as a “join” operation) is used to regain synchronization when switching from SPMD into SIMD execution mode.

4.6.3. OSCAR

OSCAR [KaH91], the Optimally SCheduled Advanced multiprocessoR, differs from the other machines in that it is explicitly oriented toward compile-time static scheduling of MIMD code rather than toward implementing a simpler fine-grain execution model.

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

Although OSCAR's barrier mechanism appears to be dynamic, it is actually just three static mechanisms within a single machine. Further, making changes to a barrier group is not addressed in [KaH91], and it appears that making any change would require multiple communication operations among the processors.

4.6.4. Cray T3D

The Cray T3D [Cra93] is a shared-memory MIMD machine incorporating up to 2,048 nodes. Each node contains an Alpha processor and a large amount of custom support circuitry, including an interface to a barrier mechanism most closely resembling eight separate copies of the FMP's mechanism. Since the use of the eight lines is software defined, it may be possible to emulate the operation of the architecture proposed in this paper, but insufficient details are available at the writing of this paper.

5. Conclusion

Although the use of conventional processors to construct parallel computers has become commonplace, very few designs provide support for fine-grain parallelism or for an execution mode other than MIMD. We suggest that there is no reason for parallel machines based on conventional processors — or even distributed machines comprised of ordinary workstations — to suffer these restrictions.

In this paper, we have presented a very simple and inexpensive hardware barrier mechanism and described in detail how it can be interfaced to conventional processors. Because this hardware implements the full dynamic barrier synchronization mechanism, it easily allows a parallel machine to be partitioned at runtime. Thus, the new hardware is significantly more powerful than other approaches (as outlined in Section 4).

The companion paper [CoD94], discusses how the new mechanism allows each portion of the machine to independently select any of a variety of execution modes including MIMD, VLIW, and SIMD models.

We are currently building the CARD machine at Purdue University to demonstrate the properties and abilities of the proposed hardware and software.

References

[AMD91] Advanced Micro Devices, *Am29050 Microprocessor User's Manual*, Advanced Micro Devices, Inc., Sunnyvale, California, 1991.

[ArJ87] 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.

[AuB86] M. Auguin and F. Boeri, "The OPSILA Computer," *Parallel Languages and Architectures*, edited by M. Consard, Elsevier Science Publishers, Holland, pp. 143-153, 1986.

[BeS91] T.B. Berg and H.J. Siegel, "Instruction Execution Trade-Offs for SIMD vs. MIMD vs. Mixed Mode Parallelism," *5th International Parallel Processing Symposium*, April 1991, pp. 301-308.

[Bro86] E. D. Brooks III, "The Butterfly Barrier," *Int. Jour. Parallel Programming,* vol. 15, no. 4, pp. 295-307, 1986.

[CoD94] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part II: Mode Emulation," Submitted to *Int'l Conf. on Parallel Processing*, 1994.

[CoN88] 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.

[Cra93] Cray Research, Inc., *Cray T3D System Architecture Overview*, Cray Research, Inc., Mendota Heights, Minnesota, September 1993.

[DiS89] 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.

[DiO92] H. G. Dietz, M.T. O'Keefe, and A. Zaafrani, "Static Scheduling for Barrier MIMD Architectures," *The Journal of Supercomputing*, vol. 5, pp. 263-289, 1992.

[Ell85] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: MIT Press, 1985.

[Gup89] 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.

[HeF88] D. Hensgen, R. Finkel, and U. Manber, "Two Algorithms for Barrier Synchronization," *Int. Jour. Parallel Programming,* vol. 17, no. 1, pp. 1-17, 1988.

[Jor78] H. F. Jordon, "A Special Purpose Architecture for Finite Element Analysis," *Proc. Int'l Conf. on Parallel Processing*, pp. 263-266, 1978.

[KaH91] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita, "A Multi-Grain Parallelizing Compilation Scheme for OSCAR (Optimally Scheduled Advanced Multiprocessor)," *Languages and Compilers for Parallel Computing*, edited by U. Banerjee, D. Gelertner, A. Nicolau, and D. Padua, Springer-Verlag, New York, pp. 283-297, 1991.

[LuB80] S. F. Lundstrom and G. H. Barnes, "A Controllable MIMD Architecture," *Proc. Int'l Conf. on Parallel Processing*, pp. 19-27, 1980.

[Lub89] B. D. Lubachevsky, "Synchronization Barrier and Related Tools for Shared Memory Parallel Programming," *Proc. Int'l Conf. Parallel Processing,* pp. II 175-179, August 1989.

[Lun87] 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.

[NiS90] M. A. Nichols, H. J. Siegel, and H. G. Dietz, "Data Management and Control-Flow Aspects of an SIMD/SPMD Parallel Language/Compiler," *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, pp. 397-406, October 1990.

[OKD90] M. T. O'Keefe and H. G. Dietz, "Hardware barrier synchronization: static barrier MIMD (SBM)," *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 35-42, August 1990.

[OKD90a] M. T. O'Keefe and H. G. Dietz, "Hardware barrier synchronization: static barrier MIMD (DBM)," *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 43-46, August 1990.

[PhW93] M. Philippsen, T. Warschko, W. F. Tichy, and C. Herter, "Project Triton: Towards Improved Programmability of Parallel Machines," *26th Hawaii International Conference on System Sciences*, Vol. 1, pp. 192-201, January 1993.

[Pol88] 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.

[SiN87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "The Implementation of the PASM Prototype Control Hierarchy," *Proc. of Second Int'l Conf. on Supercomputing,* pp. I 418-427, 1987.

[TMC92] Thinking Machines Corporation, *Connection Machine CM-5 Technical Summary*, Thinking Machines Corporation, Cambridge, Massachusetts, November 1992.

Table of Contents