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

Part II: Mode Emulation†

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 the companion paper [CoD94], we present a new hardware barrier architecture that provides the full DBM functionality we discussed in [OKD90], but can be implemented with much simpler hardware. In this paper, we show how 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).

Keywords: Dynamic Barrier Synchronization, MIMD/VLIW/SIMD Mixed-Mode Computation, Execution Models, MIMD/SIMD Coding Strategies, 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 of the companion paper [CoD94] 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 how the new barrier mechanism described in [CoD94] can implement various execution modes that can be effectively used with conventional compiler technology.

2. Execution Models

As parallel processing has developed, the execution models used with different systems have not converged. One is tempted to view this divergence as a sign of the field's immaturity; however, we suggest that the use of a range of execution modes is primarily driven by the fact that the relative performance of different system organizations is highly application-dependent [Wat93]. Although the vast majority of parallel computers effectively support only a single execution mode — MIMD, VLIW, or SIMD — supporting multiple modes allows a relatively simple compiler to target whichever mode is most expedient for each language, or even for each program.

Just as languages and complete programs often exhibit strong execution mode preferences, so too can individual functions within a single parallel program desire specific execution modes [NiS90]. By constructing hardware that can rapidly switch between multiple execution modes, one gains the opportunity to improve performance by using the execution mode that is most appropriate for each portion of a program [Wat93].

In this section, we describe how the proposed barrier mechanism can be used to efficiently implement three different execution modes: MIMD, VLIW, and SIMD. Note that the proposed implementations also allow very rapid switching between modes. Finally, we discuss the issue of granularity of execution.

2.1. Implementation Of MIMD Execution

MIMD (Multiple Instruction stream, Multiple Data stream) execution is essentially the native execution mode for a machine that is constructed as a collection of conventional sequential processors. Each processing element asynchronously executes code from its own local memory, as shown in Figure 1. Most often, the same code image is replicated in each processor; this variation on MIMD is known as SPMD (Single Program, Multiple Data) execution.

Because processing elements are executing asynchronously, even if there are no external reasons for processing elements to loose synchronization (e.g., no interrupts), `while` loops and pipeline bubbles can cause significant differences in execution rates. When relative timing information across processing elements is required, a barrier synchronization can be explicitly performed to restore static time constraints.

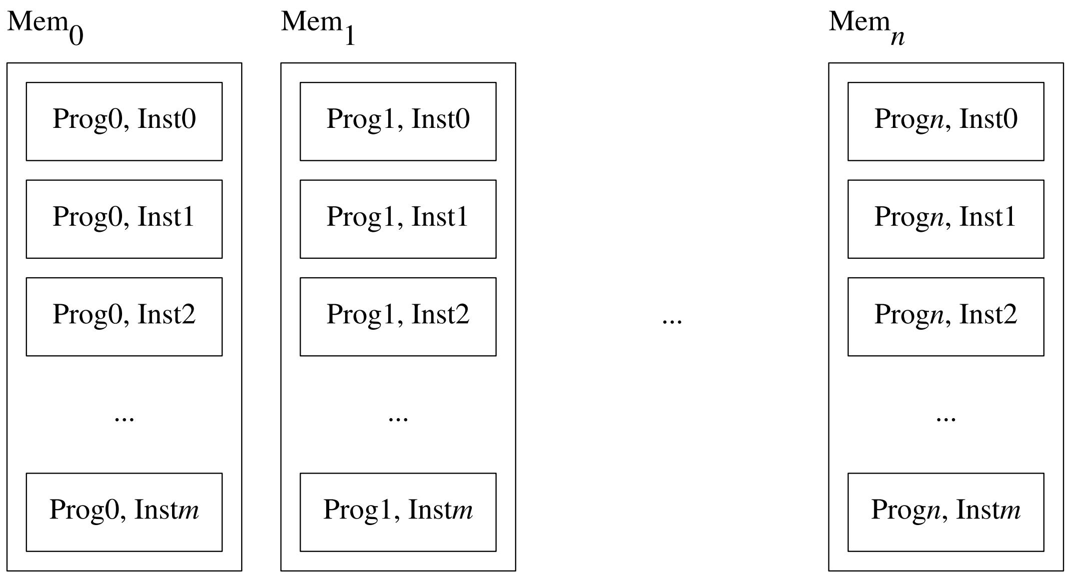

Figure 1: Memory Layout For MIMD Execution

There are a wide range of techniques that can be used by a processing element to signal its arrival at a barrier, and to cause the processing element to wait until the barrier has fired. For example, in our scheme, a barrier synchronization is accomplished in a single instruction by issuing a `LOAD` from an address decoded as a barrier synchronization request. The memory reference does not complete until the barrier has fired.

2.2. Implementation Of VLIW Execution

VLIW, or Very Long Instruction Word, computation is based on use of a single instruction sequence with an instruction format that allows a potentially different opcode for each function unit. Treating each processing element as a function unit, genuine VLIW [Ell85] [CoN88] execution can be obtained without any hardware changes. Each long instruction word is striped across the various local (instruction) memories, as shown in Figure 2. I.e., the VLIW instruction at address α is encoded by having address α in instruction memory β be the opcode field for function unit β.

Although synchronization would normally be maintained by the way the code is scheduled, variable-time operations and interrupts could cause synchronization to be lost. This problem can be averted by simply imposing a barrier before each instruction.

An additional feature that many VLIW scheduling techniques use is the ability to combine multiple conditional-`JUMP` instructions into a single multi-way branch executed by all processors. There are a variety of possible implementations. In [BrN90], Brownhill and Nicolau discuss this problem in detail and propose "Setbit" hardware to accumulate the results of VLIW branch conditions computed on various processors — the barrier architecture proposed in this paper directly implements a superset of the Setbit functionality.

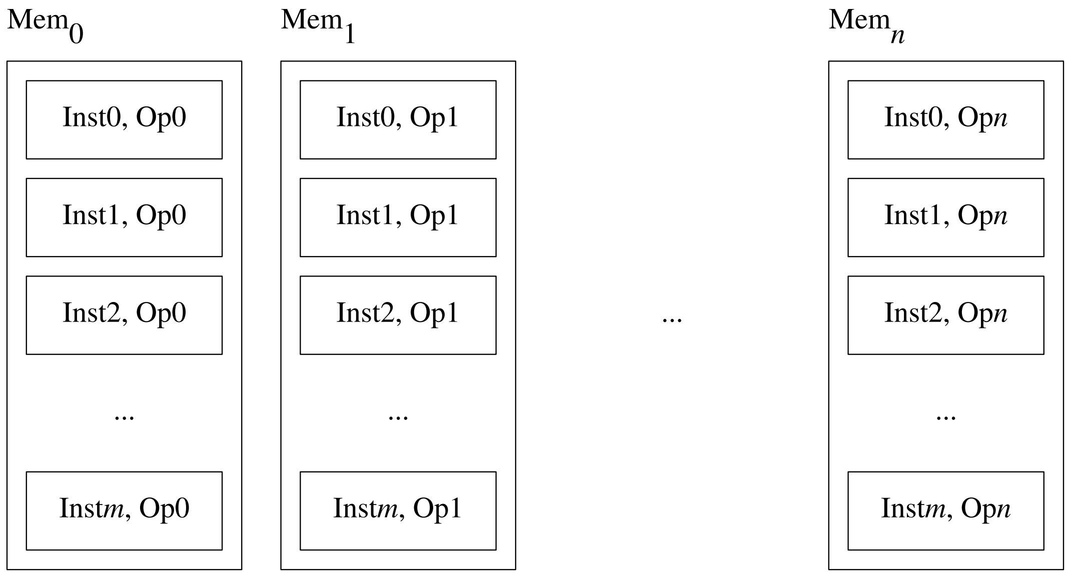

Figure 2: Memory Layout For VLIW Execution

2.3. Implementation Of SIMD Execution

SIMD (Single Instruction stream, Multiple Data stream) execution is very similar to VLIW execution. The primary difference is that, while VLIW instructions allow a different opcode for each function unit (i.e., processing element), SIMD provides only one opcode field in each instruction. There are two basic schemes by which this can be implemented:

[1] Treat SIMD as a "degenerate" form of VLIW. This is done by replicating the SIMD instruction stream in each local memory and executing that program in VLIW mode (Figure 2).

[2] Directly implement SIMD execution from a single copy of the instruction stream. Rather than having processing elements fetch instructions from local memory, have all processors fetch instructions from a shared "broadcast memory" (Figure 3), with each instruction fetch implying a barrier synchronization. Because SIMD execution implies that all the processing elements will be fetching from the same location at the same time, the same broadcast memory address will be fetched by all processing elements, effectively broadcasting the opcode. In fact, since the fetch address must be the same for all processing elements, the address could be ignored and the broadcast memory could simply be a FIFO queue [SiN87] or even a single register that holds the next broadcast instruction.

In addition to the SIMD concept of a single opcode, the SIMD execution model is generally expected to provide support for enable masking: the ability to “turn off” selected processing elements for a sequence of operations.

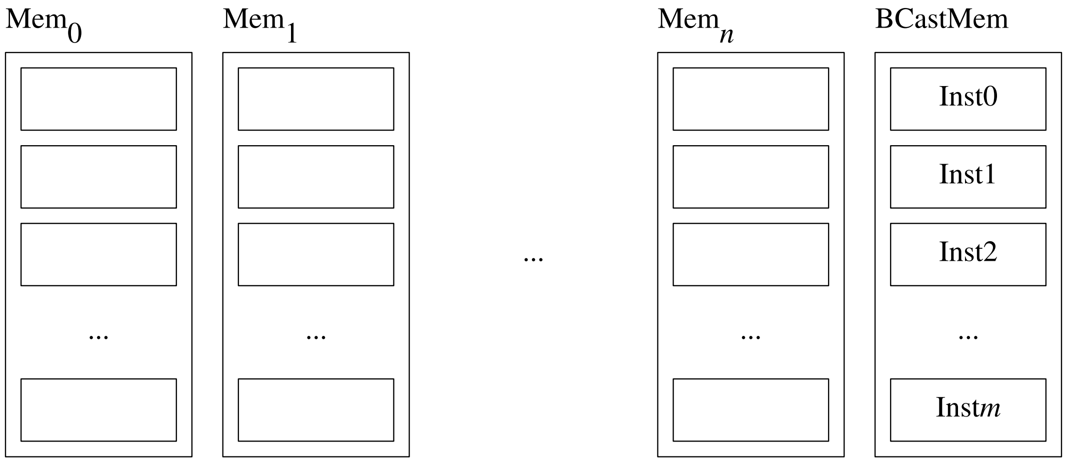

Figure 3: Memory Layout For SIMD Execution

There are a variety of simple and effective ways in which hardware could disable particular processing elements, but the most practical implementation is probably to have all processing elements perform every operation, nulling the effects on the processing elements that should not have been active. For example, some processors have conditional store instructions that would allow “disabled” processing elements to perform computations, but not update variables with the results. With somewhat greater cost, this nulling can even be done using conventional arithmetic; for example, the parallel assignment:

```
c = a + b;
```

Can be made to execute only when `enabled` is a bit mask containing all `1`s (for true, as opposed to `0` for false) by all processing elements simultaneously executing:

```
c = (((a + b) & enabled) | (c & ~enabled));
```

Another implementation would be to simply `JUMP` over code for which `enabled` is false, but this must be done very carefully to avoid accidentally skipping operations that should be executed (enable mask operations, scalar code, function calls, etc.). Further, using `JUMP`s can cause pipeline bubbles that seriously degrade performance.

There is also the issue of “stacking” enable status as nested constructs change the set of enabled processors. Although the `enabled` status could simply be pushed on the stack for each construct, perhaps the best method is that proposed by Keryell and Paris [KeP93]. In their scheme, a counter in each processor is used to track the number of properly nested constructs for which each processor has been disabled. If this counter is zero, the processor is enabled; otherwise, the processor is disabled. This scheme is both fast and memory efficient, since entering a new enable scope requires only update of the counter (presumably a register; thus implying no memory references).

Just as enable masking does not require additional hardware, the fact that the traditional view of SIMD incorporates a control unit (CU) does not require the hardware to have such a processor. The primary function of the CU is to broadcast instructions; which is managed by the mechanisms described earlier. The secondary function of the CU is to perform scalar computations. This also can be done without additional hardware by simply designating one processing element to act as the CU. Alternatively, it is easy to add a processor to the broadcast memory interface to act as a dedicated CU. In fact, most parallel computers have a serial front-end host that could be treated as a dedicated CU.

2.4. Granularity

In the above discussion of execution modes, this paper refers to parallel execution of individual operations or instructions. This seems to imply very fine grain: instruction-level parallelism. However, there is no reason to require that an operation or instruction is an atomic function for the hardware; it could instead be an entire subroutine or sequence of machine instructions.

In this light, it can be seen that the execution mode does not imply granularity. Normally one thinks of MIMD as coarse-grain, and SIMD and VLIW as fine-grain, but any combination of mode and granularity is possible. For example, one can imagine a coarse-grain SIMD machine, in which each operation is actually an entire subroutine — yielding a grain size that could be tens of thousands of machine cycles.

We define the granularity of a program's execution as the amount of time spent computing locally before an interaction with another processing element is required. The simplest possible interaction with another processing element is to synchronize with it; thus, a machine's grain size is similarly defined to be the amount of time taken to perform a synchronization. Equivalently, this is the maximum error in one processor's estimate of what another processor is currently executing. As such, it forms the basic unit for (compile-time) code scheduling.

To simplify the discussion, consider a machine with grain size x that must execute a program with grain size $y \,|\, y<x$. In such a case, the program will always execute at least a factor of x/y slower than the machine's peak speed due to interaction delays. Put another way, in order to achieve a program grain size of x, one could attempt to pack x/y grains of computation that interact with each other onto each processing element. Thus, the x/y factor also can be viewed as a reduction in the usable parallelism width of the program, and a larger machine grain size reduces the probability that programs will have enough parallelism to get good performance from massively-parallel hardware.

The ideal grain size results when synchronization can occur at the level of simple data movement, i.e., at the level of register moves or `LOAD/STORE` instructions. This is the level of the barrier mechanism presented in the current work. In contrast, for most machines, the grain size ranges from hundreds of instructions (e.g., [TMC92]) to thousands of instructions (e.g., [Ncu90] or [Int92]); which severely restricts the class of programs that will be able to make effective use of the hardware.

3. Coding Strategy

As suggested in Section 1, the most efficient way to use the proposed hardware involves detailed static timing analysis and compiler code scheduling [DiO92]; however, reasonable performance can be obtained for a wide range of programming models without resorting to such complex compiler techniques. For this paper, we limit the discussion to three of the most common parallel programming models that are suitable for expressing scalable fine-grain parallelism:

[1] Programming models based on groups of worker processes that asynchronously obtain and execute tasks from a worklist. A good example is The Force [Jor87]; however, most SPMD-oriented languages follow these semantics.

[2] Programming models based on data parallelism with SIMD-oriented ordering constraints and structured control of the active set of processing elements. A good example is C* [RoS87]; however, most data-parallel languages, including most parallel dialects of Fortran 90 [ANS89] (CM Fortran, MasPar Fortran, etc.), follow these semantics.

[3] Programming models based on data parallelism with provision for unstructured control of the active set of processing elements; this is essentially a variation on the SIMD semantics of [2]. A good example is MPL [MCC91] with its `all` statement or the new C* [TMC90] with its `everywhere` statement.

For each of these programming models, we present a simple description of the corresponding barrier execution model and examples showing how common language constructs could be coded for the target barrier machine. Since a wide range of programming languages and processor instruction sets can be supported, we have chosen to represent both the programming language constructs and the translated code in a notation based on C. The notation is described in the following section.

3.1. Notation

Although each of the programming models provides a wide range of constructs, the basic semantic differences between the models are effectively illustrated by examining the coding of a parallel conditional construct (`if-else`) and a parallel loop construct (`while`). Since we are focusing on the semantic differences, the syntax of the language constructs is relatively unimportant; for the purposes of this paper, we will refer to the MPL syntax given in Listings 1 and 2.

```
if (parallel_expr){
    stat_a;
}else{
    stat_b;
}
```

Listing 1: Parallel `if-else`

```
while(parallel_expr){
    stat;
}
```

Listing 2: Parallel `while`

The target barrier code for our examples is given as raw C code that would be replicated on each of the processors. It is important to note that all variables referenced in the target code can be placed in registers and that the macros `WAIT()`, `WAIT_GATHER()`, `PUSH()`, and `POP()` are used only because they are more mnemonic than the instructions they represent:

`WAIT()`

The `WAIT()` operation takes one argument, which is the barrier address containing the bit vector that describes the processors to participate in the barrier synchronization. A processing element will not continue past the `WAIT()` until all the processing elements participating in that barrier are executing corresponding `WAIT()` operations. The CARDBoard's Am29050 processors implement `WAIT()` as a single `LOAD` instruction; the generic C definition is:

```
#define  WAIT(bar)  *(bar)
```

`WAIT_GATHER()`

The `WAIT_GATHER()` operation has two characteristics which differentiate it from the `WAIT()` macro — it takes a one bit flag and returns a bit vector. The one bit flag can be used to communicate information to the other processing elements participating in the barrier, usually the result of a test operation. The value returned is a word value that looks like the barrier address, except in that the bit vector contains the flag values gathered from each processor rather than bits indicating which processors participate in the barrier. Return value bit positions corresponding to processors not participating in the barrier have undefined values. The CARDBoard's Am29050 processors implement `WAIT_GATHER()` as an `OR` instruction followed by a `LOAD`; however, many processors can implement this operation using a single `LOAD` with an addressing mode that adds two registers to form the data address. The generic C definition is:

```
#define  WAIT_GATHER(bar, flag)  *((bar) + (flag))
```

`PUSH()`

The `PUSH()` operation simply saves its argument on the runtime stack.

`POP()`

The `POP()` operation simply returns the value from the top of the runtime stack and removes that value from the stack.

Thus, although the target code might appear to be invoking relatively expensive functions to perform barrier synchronizations, the actual cost of executing these macros is typically either one or two instructions with at most one off-chip reference.

Throughout the code examples, we make use of the following variables:

`barrier_address`

This is the address to reference to cause a barrier across the currently selected set of processors, as discussed in Section 3.3.1 of the companion paper.

`flag_vector_mask`

This is a (machine-specific constant) mask that covers all the bits in the flag vector.

3.2. The SPMD Worker Model

Of the three programming models, the least constrained is the SPMD worker model, in which barrier synchronizations are directly visible to the user as "barrier statements." In general, barrier synchronizations are invoked only when the programmer explicitly calls for them; however, the system must automatically track which processors may synchronize with each other, and this can introduce additional barriers for the purpose of broadcasting information about a new partitioning of the workers.

Thus, we can imagine a parallel construct being used to partition the current set of workers into two sets: those where some parallel expression evaluates as true and those in which the expression evaluates as false. When such a construct is encountered, each processor executing the construct can select which set will contain it. However, these sets are not fully specified until all processors that will execute the construct have made their individual decisions. We suggest that a single `WAIT_GATHER()` operation suffices to both ensure that the sets are fully specified and to notify each of the participating processors of the complete membership for the set that contains them. Because the `WAIT_GATHER()` operation returns bits for all processors, even those that did not participate in the barrier, it is a simple matter of appropriately masking the return value to convert the set membership information into a barrier mask for that set.

This semantic definition and coding technique covers the partitioning actions implied by both `if-else` and `while` statements in a SPMD worker model. Notice that, if the constructs did not imply partitioning, no barrier synchronization operations would be needed beyond those explicitly requested by the user. The following sections detail the behavior and coding of these partitioning constructs.

3.2.1. Parallel `if-else`

The semantics of an SPMD `if-else` construct do not imply an ordering between the "then" and the `else` clauses of the construct. In fact, the two alternatives are asynchronously executed by mutually exclusive sets of processors. Thus, if such a construct is used to partition processors into two independent groups that can synchronize within themselves independently, code must be introduced to create the new barrier masks. Notice that this operation is a true partitioning of the previous barrier mask; each processor will join either the "then" barrier group or the `else` barrier group. Further, notice that this partitioning cannot affect any processors that were not a member of the original barrier group. A sample implementation of these semantics is given in Listing 3.

```
    PUSH(barrier_address);
    t = parallel_expr;
    gather_result = WAIT_GATHER(barrier_address, t);
    if (t == 0) goto Else;
    barrier_address &= gather_result;
  stat_a;
    goto Exit;
Else:
    barrier_address &= (flag_vector_mask ^ gather_result);
  stat_b;
Exit:
    barrier_address = POP();
```

Listing 3: SPMD worker model SIMD parallel `if-else`

This code first saves the current `barrier_address`, because within the construct each processor will use a `barrier_address` that is a subset of the original, yet the original must be restored at the end of the construct (by the `POP()`). The next step is to evaluate *parallel_expr* to determine whether the "then" or `else` clause should be executed; in either case, using a `WAIT_GATHER()`, the result of this evaluation is also sent to other processing elements. Once within the appropriate clause, the gathered evaluation information is used to create the new `barrier_address` by removing processors that selected the other clause. The only remaining task is to execute the code within the selected clause.

Notice that all the code given in `bold` can be removed if the statement is not being used to partition the current barrier group, i.e., if there are no nested references to barrier synchronizations using the partitioned barrier groups.

3.2.2. Parallel `while`

The semantics of a SPMD `while` allow each processor to asynchronously execute as many iterations of the loop as it desires. However, if the `while` is being used to partition the current barrier group, a synchronization must be inserted at the top of each iteration of the loop in order to compute a new partition of the barrier group. Thus, there is only one *active* barrier group within the loop, and that group monotonically decreases in size until it has no members, at which time the original barrier group is restored. A sample implementation of these semantics is given in Listing 4.

```
    PUSH(barrier_address);
Loop:
    t = parallel_expr;
    gather_result = WAIT_GATHER(barrier_address, t);
    barrier_address &= gather_result;
    if (t == 0) goto Exit;
  stat;
    goto Loop;
Exit:
    barrier_address = POP();
```

Listing 4: SPMD worker model parallel `while` loop

As for the `if-else` example, the looping code is enclosed by instructions saving and restoring the original `barrier_address` and `WAIT_GATHER()` is used to restrict the current `barrier_address`. Notice that processors that exit the loop early will be given a `barrier_address` that is a superset of those still executing within the loop; however, this causes no conflicts. If one of these early processors attempts a barrier synchronization, the fact that it is waiting will be ignored by the processors in the loop.

3.3. The Structured SIMD Model

Compared to the SPMD worker model, the structured SIMD model imposes an additional ordering constraint: the execution of constructs is serially ordered such that operations that appear later in the source program are executed only after all processors have moved past all earlier code. This essentially corresponds to the concept of structured code being executed only when the single program counter in a SIMD machine has reached that code. Notice, however, that this model does not require that all processors actually execute all code.

The following section describes in detail the execution of the parallel `if-else` and the parallel `while` statement for the structured SIMD model and how these constructs would be implemented using the barrier mechanism.

3.3.1. Parallel `if-else`

Structured SIMD semantics require an ordering that first evaluates *parallel_expr*, then *stat_a*, and finally, *stat_b*. Thus, although processors may be waiting at the `else` clause as soon as the first processors reach the "then" clause, barrier synchronizations ensure that there is no overlap in the execution of these clauses. An implementation of these semantics is given in Listing 5.

```
      PUSH(barrier_address);
      t = parallel_expr;
      gather_result = WAIT_GATHER(barrier_address, t);
      if (t == 0) goto Else;
      barrier_address &= gather_result;
    stat_a;
      barrier_address = POP();
   A: WAIT(barrier_address); /* pairs with B */
      goto Exit;
   Else:
   B: WAIT(barrier_address); /* pairs with A */
      barrier_address &= ( flag_vector_mask ^ gather_result);
    stat_b;
      barrier_address = POP();
   Exit:
      WAIT(barrier_address); /* all processors here */
```

Listing 5: Structured SIMD parallel `if-else`

The first five lines of this code are identical to the code using the SPMD model because processors are allowed to execute the "then" clause under the same conditions that applied for the SPMD model. However, the remaining code must ensure that the `else` clause is not executed until all processors that entered the `if` are either ready to execute the `else` clause or have completed executing the "then clause. Further, when that second barrier has fired, yet another barrier must be imposed to ensure that the processors that executed the "then" clause will not execute past the `if` statement until the last processor has completed executing the `else` clause.

One might wonder why the effort of partitioning the barrier mask and executing two additional synchronizations would be desirable. The answer is in the code that is not in the example — there is no SIMD-style enable masking needed to disable inactive processors. Simulation of SIMD-style enable masking using conventional processors is likely to be slower than performing these barrier operations (see Section 3.4).

3.3.2. Parallel `While`

Structured SIMD code differs from SPMD only in that it requires a complete ordering between blocks of parallel code. Because the SPMD encoding of `while` enforces such an ordering with just one exception, it is not surprising that the structured SIMD code can be nearly identical. As seen in Listing 6, the one difference is that the structured SIMD code must ensure that no processor can execute past the `while` until all processors that entered the loop have completed the loop.

```
      PUSH(barrier_address);
Loop:
      t = parallel_expr;
      gather_result = WAIT_GATHER(barrier_address, t);
      barrier_address &= gather_result;
      if (t == 0) goto Exit;
   stat;
      goto Loop;
Exit:
      barrier_address = POP();
      WAIT(barrier_address);
```

Listing 6: Structured SIMD parallel `while` loop

3.4. The Unstructured SIMD Model

In most high level languages designed for SIMD machines, the programmer does not have direct control of the enable status of the processors; rather, changes in the enable status are implied by structured control constructs. The rule is that the code within a construct cannot be executed by processors that were disabled at entry to the construct. Thus, the handling of enable status for each processor acts as a stack where an enable status is recorded for each active scope and the stack always has the property that once a disable is placed on the stack, all items placed on the stack after that item will also be disables [KeP93]. These semantics yield the structured SIMD model described above.

However, most SIMD hardware does not restrict processors to manage enable status in that way. Thus, some SIMD languages provide features that allow "unstructured" changes to processor enable status. For example, MPL's `all` construct enables all processing elements for execution of the following statement — enabling processors that were disabled at entry to the enclosing region of code. In fact, a statement affected by an `all` can be a compound statement that contains additional structured and unstructured masking. Another example is MPL's `proc` construct; `proc[i].j` refers to the value of `j` on processor `i`, reguardless of whether processor `i` was enabled in the enclosing region of code. Because disabled processing elements can be enabled at any time, all processing elements must synchronously follow the same control flow paths.

Thus, unlike the other two programming models, the set of processors in the barrier mask never changes. All barriers are performed with the `all_processors` barrier address, which represents a barrier mask including all processors in the portion of the machine that is executing this program.

Since the barrier mask is not used to track the SIMD enable set (as it is in the structured SIMD code), some other mechanism must be used. Thus, processors must simulate enable masking using one of the techniques described in Section 2.3. The next section defines how enable masking is represented in the `if-else` and `while` examples in the sections that follow.

3.4.1. Enable Mask Simulation

For simplicity, we will denote the current enable status of each processor as the value of the variable `enabled`; this value is `0` if the processor is disabled, `1` if it is enabled. Although there are a variety of different approaches to simulation of enable masking, there just three basic operations needed to track the processor's enable status:

`PUSH_ENABLED()`
: This operation saves the current value of `enabled` onto the processor's stack.

`TOP_OF_ENABLED_STACK()`
: This operation returns the `enabled` value on top of the stack, but does not remove that value from the stack.

`POP_ENABLED()`
: This operation returns the `enabled` value on top of the stack, but also removes that value from the stack.

Alternatively, the references to the `enabled` variable and these operations can be replaced by the less intuitive, but more efficient, "activity counter" equivalents described in [KeP93]. Note that the "activity counter" technique by itself assumes that a processor cannot be enabled from within a construct if it was disabled at entry to that construct, but constructs like MPL's `all` can be correctly handled by stacking the "activity counter" value at entry to an `all` and unstacking the value at exit.

In addition to tracking the processor's enable status, it is necessary to ensure that disabled processors do not change the values of variables to reflect the results of operations in their path. This is further complicated by the fact that disabled processors should not be prevented from executing operations that may alter their enable status. Because the method used (e.g., arithmetic nulling, as described in Section 2.3) can literally change how code is generated for every operation within a statement, we have chosen to simply indicate which regions of the generated code must be processed by such a transformation. Any code within a `EVAL_ENABLED()` is to be generated such that operations not involved in computing enable status have no effect if `enabled` is `0` (false).

3.4.2. Parallel `if-else`

Unlike the other execution models, the unstructured SIMD semantics do not make a parallel `if-else` construct change the control flow of the program. Rather, the construct changes enable/disable status of processors as all the processors pass through the code for both the "then" and `else` clauses.

Although both the new C* [TMC90] and MPL [MCC91] use unstructured SIMD execution, the C* language more literally adheres to the principle that control flow cannot be altered by a parallel construct. Even if no processors are enabled, C* semantics suggest that the complete code sequence should be "executed." In contrast, MPL semantics suggest that code for which no processor is enabled should not be "executed." Thus, before each section of conditionally-executed code, MPL inserts instructions that jump over the code if no processors would be enabled for its execution. This difference in semantics is reflected by the fact that the code of Listing 7 implements the MPL semantics as is, and implements the new C* semantics if the `bold` code is removed.

```
   PUSH_ENABLED();
   t = 0;
   EVAL_ENABLED(t = parallel_expr);
   enabled = t;
   any = WAIT_GATHER(all_processors, enabled);
   if ((any & flag_vector_mask) == 0) goto Else;
   EVAL_ENABLED(stat_a);
Else:
   enabled = TOP_OF_ENABLED_STACK();
   enabled &= ~t;
   any = WAIT_GATHER(all_processors, enabled);
   if ((any & flag_vector_mask) == 0) goto Exit;
   EVAL_ENABLED(stat_b);
Exit:
   enabled = POP_ENABLED();
```

Listing 7: Unstructured SIMD parallel `if-else`

3.4.3. Parallel `While`

Because the new C* does not allow control flow to be altered by a parallel construct, there is no parallel `while` in C*. However, there is a parallel `while` construct in MPL. This construct is implemented by the code given in Listing 8.

```
    PUSH_ENABLED();
Loop:
    t = 0;
    EVAL_ENABLED(t = parallel_expr);
    enabled = t;
    any = WAIT_GATHER(all_processors, enabled);
    if ((any & flag_vector_mask) == 0) goto Exit;
    EVAL_ENABLED(stat);
    goto Loop;
Exit:
    enabled = POP_ENABLED();
```

Listing 8: Unstructured SIMD parallel `while`

4. 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 the companion paper [CoD94], we presented a very simple and inexpensive hardware barrier mechanism and described in detail how that mechanism can be interfaced to conventional processors. In this paper, we show how the new hardware allows each portion of the machine to independently select any of a variety of execution modes including MIMD, VLIW, and SIMD models. Detailed coding strategies for three commonly used execution modes are presented: a SPMD worker model, a structured SIMD model, and an unstructured SIMD model. Reguardless of execution model, grain sizes as small as a few instructions are efficiently supported — because barrier synchronization cost is just a single `LOAD` instruction.

We are currently working on more sophisticated compiler techniques based on timing analysis [DiO92].

References

[ANS89] *American National Standard for Information Systems Programming Language Fortran*, Draft S8, Version 112, June 1989.

[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.

[BrN90] C.J. Brownhill and A. Nicolau, *Percolation Scheduling for Non-VLIW Machines*, Technical Report 90-02, University of California at Irvine, Irvine, California, January 1990.

[CoD94] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part I: Barrier Architecture," 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.

[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.

[Int92] Intel Corporation, *Paragon Supercomputers: Paragon XP/S Supercomputer*, Literature packet, Intel Corporation, Beaverton, Oregon, 1992.

[Jor87] H. F. Jordon, *The Force*, Technical Report, University of Colorado, January 1987.

[KeP93] R. Keryell and N. Paris. "Activity Counter: New Optimization for the Dynamic Scheduling of SIMD Control Flow,*Proc. Int'l Conf. Parallel Processing,* pp. II 184-187, August 1993.

[MCC91] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 2.2*, Document Number 9302-0001, Sunnyvale, California, November 1991.

[Ncu90] nCUBE Corporation, *nCUBE 2 Programmer's Guide*, nCUBE Corporation, Beaverton, Oregon, December 1990.

[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 (DBM)," *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 43-46, August 1990.

[RoS87] J. Rose and G. Steele, *C*: An Extended C Language for Data Parallel Programming*, Thinking Machines Corporation, Technical Report PL87-5, Cambridge, Massachusetts, April, 1987.

[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.

[TMC90] Thinking Machines Corporation, *C* Programming Guide*, Thinking Machines Corporation, Cambridge, Massachusetts, November 1990.

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

[Wat93] D. W. Watson, *Compile-Time Selection of Parallel Modes in an SIMD/SPMD Heterogeneous Parallel Environment*, Ph.D. Dissertation, Purdue University School of Electrical Engineering, August 1993.

Table of Contents

1. Introduction 2
2. Execution Models 4
2.1. Implementation Of MIMD Execution 4
2.2. Implementation Of VLIW Execution 5
2.3. Implementation Of SIMD Execution 6
2.4. Granularity 9
3. Coding Strategy 10
3.1. Notation 10
3.2. The SPMD Worker Model 12
3.2.1. Parallel `if-else` 13
3.2.2. Parallel `while` 13
3.3. The Structured SIMD Model 15
3.3.1. Parallel `if-else` 15
3.3.2. Parallel `While` 16
3.4. The Unstructured SIMD Model 17
3.4.1. Enable Mask Simulation 18
3.4.2. Parallel `if-else` 18
3.4.3. Parallel `While` 19
4. Conclusion 20