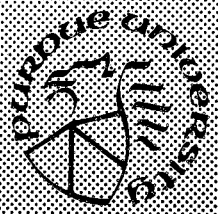
6-1-1988

Extending Static Synchronization Beyond SIMD and VLIW

Henry G. Dietz
Purdue University

Thomas Schwederski
Purdue University



Extending Static Synchronization Beyond SIMD and VLIW

Henry G. Dietz
Thomas Schwederski

TR-EE 88-25
June 1988

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Extending Static Synchronization Beyond SIMD and VLIW

Henry G. Dietz and Thomas Schwederski

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
June 1988

Abstract

A key advantage of SIMD (Single Instruction stream, Multiple Data stream) architectures is that synchronization is effected statically at compile-time, hence the execution-time cost of synchronization between "processes" is essentially zero. VLIW (Very Long Instruction Word) machines are successful in large part because they preserve this property while providing more flexibility in terms of what kinds of operations can be parallelized. In this paper, we propose a new kind of architecture — the "static barrier MIMD" or SBM — which can be viewed as a further generalization of the parallel execution abilities of static synchronization machines.

Barrier MIMDs are asynchronous Multiple Instruction stream Multiple Data stream architectures capable of parallel execution of loops, subprogram calls, and variable-execution-time instructions; however, little or no run-time synchronization is needed. When a group of processors within a barrier MIMD has just encountered a barrier, any conceptual synchronizations between the processors are statically accomplished with zero cost — as in a SIMD or VLIW and using similar compiler technology. Unlike these machines, however, as execution continues the relative timing of processors may become less precisely knowable as a static, compile-time, quantity. Where this imprecision becomes too large, the compiler simply inserts a synchronization barrier to insure that timing imprecision at that point is zero, and again employs purely static, implicit, synchronization. Both the architecture and the supporting compiler technology are discused in detail.

Keywords: SIMD, VLIW, LSM, SBM, DBM, MIMD, barrier-synchronization, code-scheduling, compiler-optimization.

1. Introduction

PASM is the PArtitionable Simd/Mimd system designed by H. J. Siegel et. al. [SiS81] and the PASM prototype which was constructed at Purdue University is a 16 processing-element implementation [ScN87]. The work presented in this paper is largely the result of considering implementation of a VLIW execution model, and associated compiler technology, for the PASM prototype.

It quickly became clear that PASM could not easily support VLIW execution, however, it is capable of executing a model which is not SIMD, MIMD, nor alternately or in partitions SIMD and MIMD, but rather something *between* the two. Processors would run and communicate in MIMD mode, however, the logic that normally enables/disables processors in SIMD mode would be used to create a barrier synchronization mechanism. An arbitrary subset of the processors could be specified, using the enable/disable logic, to participate in each barrier synchronization. The key realization was that code for this model could be generated using VLIW-like compiler technology.

Some simple benchmarks have been run using the PASM prototype in this mode [FiC87] [FiC88], allbeit without taking full advantage of VLIW-like code scheduling techniques. Preliminary results have been very promising.

In the meantime, work continued within CARP — the Compiler-oriented Architecture Research group at Purdue — to define both the new compiler technology and the characteristics of the architectures between SIMD and MIMD which constrain the compiler's model in generating efficient parallel code.

In this paper, we present an overview of the new taxonomy, the architectural concepts, and the compiler technology. Section 2 defines the classification scheme and uses it to evaluate which unusual architecture(s) are worthy of further investigation. The most useful of these architectures, barrier MIMDs (the SBM and DBM models), are described in detail in Section 3. Section 4 discusses the compilation technology needed in support of barrier machines, and presents algorithms for implementing the key compiler analysis and optimization routines. Finally, Section 5 summarizes the contributions of this paper and suggests directions for further research.

2. Motivation and Classification

Flynn's traditional classification of architectures separates machines along the dimensions of how many instruction streams and how many data streams can be executed/operated on simultaneously. The classification has become so widely accepted that it is commonly held that describing a machine in these terms defines the architecture sufficiently well that one may evaluate its properties. However, recent developments, such as VLIW (Very Long Instruction Word) computers [Ell85], are not adequately described by classification as SIMD or MIMD.

Motivated by the inadequacy of the SIMD and MIMD labels in describing the properties of VLIW, we propose a classification based on the contiguous spectrum of properties between SIMD and MIMD. This spectrum is based on the concept of SIMD differing from MIMD in that SIMD places more constraints on the parallelism structures which the hardware is able to execute, yet it is superior to traditional MIMD models in that there is no runtime synchronization cost. This classification is summarized in Table 1.

Across the top of Table 1 are listed the names of the various machine types between conventional SIMD and MIMD machine models. Down the left side of Table 1 are listed the the various characteristics which we used to define the differences between these machine types. Before describing the primary concern of this paper — the static barrier MIMD (or SBM) architecture — it is useful to describe these features since they lead to the realization that a static barrier MIMD would be an especially useful design.

2.1. Architectural Features

The "simultaneous operations" row indicates how many *different* operations can be performed simultaneously on a machine with N processors. This is primarily a constraint on parallel execution; the larger the number, the more different kinds of parallelism the machine will be able to employ.

The number of "control flow threads" is the number of independent program counters in a machine of width N. Again, the larger this number, the more different kinds of parallelism the machine will be able to employ. For example, if this number is greater than one it is possible for the machine to execute a loop in parallel with straight-

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

Table 1: Hardware Parallelism Constraints for SIMD → MIMD

line code.

"Relative time synchronization error" specifies the time error with which *the compiler* can know which instruction is executing on one processor when a particular instruction is executing on another processor. In SIMD and VLIW execution, the fact that this this error is very small (essentially zero) enables *static scheduling* of instructions to be used to perform conceptual synchronizations without runtime overhead; this property makes fine-grain parallelism usable. A barrier MIMD (of either kind) also has this property, and hence it can also be instruction scheduled with good efficiency. The reasoning is that when a barrier is encountered, relative timing error between processors participating in the barrier is reset to zero, hence, even if processors execute code which has dynamically varying execution time for some processor relative to the other processors, the compiler can always insert a barrier to reduce this time error to zero. It is a very new way in

which to view barriers: a barrier is not an implementation of synchronization, but merely a method for forcing relative execution time ambiguities to zero when they otherwise might dynamically exceed an arbitrary constant, k. This implies that barriers are needed only to resolve timing ambiguities, and not to implement synchronization; typically, only a small fraction of all conceptual synchronizations will actually require that a barrier be generated. A more traditional MIMD requires larger-grain processes because the relative timing error between processes cannot be made zero: instruction scheduling alone cannot be used to implement all conceptual synchronizations.

The number of "synchronization control flow threads" is how many different synchronization operations are candidates for the next synchronization operation to occur. If this number is not zero (no synchronization), then larger values imply less waste in performing multiple synchronizations. If, for example, a four processor machine requires processors 0 and 1 to synchronize and processors 2 and 3 to also synchronize, one needs to know which of these pairs synchronizes first. If this cannot be predicted at compile time, a machine which permits multiple synchronization control flow threads will insure that the synchronizations occur in the correct order. A machine which permits only one such thread will sometimes suffer a delay due to, for example, processors 0 and 1 waiting for 2 and 3 because the compiler incorrectly guessed that the synchronization of 2 and 3 would occur first. In fact, one could avoid this waste by merging both synchronizations into a single barrier across processors 0, 1, 2, and 3 if the machine is a static barrier MIMD. This yields the same delay, but leaves the compiler with fewer relative time errors (e.g., relative timings between 0 and 2 would be known).

Finally, there is the issue of whether synchronization primitives are directed or not. A directed synchronization is an operation whereby one processor is forced to wait for some action of another, but the processor performing the action need not wait upon performing the action. In other words, if A is to wait for B and B arrives before A does, B is allowed to continue immediately. Undirected synchronization causes all involved processes to wait, hence it is somewhat less efficient as a synchronization mechanism.

2.2. Architectures

Having outlined what the features are which distinguish the various architectures between SIMD and MIMD, this section attempts to discuss the practical utility of each conceptual architecture.

The first, and probably oldest, of these architectures is the SIMD model. However, it takes little insight to see that, according to the features given above, SIMD is less general than it could be and still provide all the same benefits; SIMD is distinguished from VLIW only by being a less general kind of parallelism. Even the hardware implementation is nearly identical. Given this, it is not surprising that SIMD is rapidly being replaced by VLIW in commercial product offerings; in fact, MSIMD (Multiple SIMD) machines such as the Connection Machine [Thi87] — which are essentially VLIWs by another name — have also found wide acceptance. Perhaps the only reason "straight" SIMD architectures have survived this long is that the more constrained parallelism model yields a simpler programming and debugging methodology, although it does so at the cost of loosing much parallelism in typical applications.

As discussed in the above paragraph, VLIW architectures have many benefits and are very effective machines. The largest problem is that the programming model is too complex to be directly expressed in high-level language, hence more sophisticated compiler technology is needed. About 1982, Fisher and others at Yale University proposed a compiler technology called "trace scheduling" to manage VLIW coding. This technique is a very clever extension of basic block (DAG) flow analysis which allows parallelization of code across control flow constructs such as `if` or `case` statements (but not across loops or subprogram invocations). The simplicity of the compiler analysis and the generality of the hardware are an very good match. VLIWs will probably continue to gain support.

Lock-step MIMDs, or LSMs, are essentially VLIWs where each processor has its own program counter, hence it would be possible to execute a loop in parallel with straight-line code — something a VLIW can't do. The hardware is also quite similar in complexity to that of a VLIW. The problem is that parallel operations *must be known at compile time to take exactly the same amount of time to execute.* This means that each loop must

iterate a known number of times — but if this is true, the compiler could simply *unroll* the loop and achieve the same parallel execution using a VLIW . . . only the code size would distinguish the two architectures. Further, the compiler technology to parallelize for a lock-step MIMD would be relatively very complex compared to that for a VLIW. In summary, lock-step MIMD isn't a bad architecture, but it is very unlikely that it would achieve any better parallelism than a VLIW, and it would be harder to use (the compiler would be harder to write and would compile slower).

A static barrier MIMD, or SBM, loosens the parallelism constraints of a lock-step MIMD just a little — a static barrier MIMD can simultaneously perform runtime-variable-execution-time operations. In other words, implicit synchronization can be made "fuzzy" and then sharpened at arbitrary points in a program's execution. This means that, for example, `while` loops, subprogram invocations, conditionals, and straight-line code can all be executed simultaneously; one can even perform dynamic load balancing. In effect, a barrier MIMD can parallel execute all of the parallelism structures typically generated by automatic parallelizing compilers (although it cannot efficiently execute some explicitly-parallel programs because they rely on point-to-point synchronization operations for which no static order can be determined). Further, both the hardware and the compiler algorithms are relatively simple because one can always fall back on generating a barrier for each synchronization operation. Of all the architectures discussed here, static barrier MIMDs should yield the cheapest, most efficient, machine capable of using nearly all the parallelism in an application — this is why static barrier MIMD is the primary topic of the current work.

Dynamic barrier MIMDs, or DBMs, differ from static barrier MIMDs in that they can require slightly less time to execute a set of barriers where the relative times at which the barriers are encountered are not known at compile time. However, the hardware appears to be significantly more complex and seems to require an associative matching of processors awaiting a barrier to the barrier to occur next. In addition, the compiler technology for a static barrier MIMD offers a solution for those machines which is nearly as good — if a group of barriers are so close that the sequence of them cannot be statically determined, one would simply merge the barriers into a single barrier on the static barrier machine. This does result in a slightly longer average delay in barrier execution, but

it also provides much tighter bounds on interprocessor timing, hence it may eliminate the need for some future synchronization barriers. In summary, it is difficult to be certain that dynamic barrier MIMDs would perform noticeably better than static barrier MIMDs, hence the additional hardware complexity probably isn't worthwhile.

Finally, the traditional directed-synchronization MIMD has the obvious advantage of being able to parallel execute completely arbitrary parallel code structures. However, synchronization cost is much higher, and this implies larger process granularity is needed. Synchronization cost makes some parallelism structures unbeneficial, even though all can be parallel executed. For this reason, hybrids or reconfigurables which provide both MIMD and one of the finer-grain parallel architecture models, especially VLIW, make particularly good sense. Further, directed synchronization permits the creation of *races* and *deadlocks* — parallel debugging horrors which ***do not occur*** using any of the other parallel machine types listed above. Hence, directed-synchronization MIMDs have a firm reason for being, but are not always the most desirable parallel architecture.

3. Barrier Hardware

As discussed above, there is good reason to believe that the special properties of a static barrier MIMD will result in very good performance on a wide range of codes — especially on those generated by automatic parallelization of sequential programs. In this section, some architectural and implementation details of barrier MIMD machines are given. First, existing machines are considered, then an idealized static barrier MIMD design is proposed.

3.1. Barrier Mechanisms in Existing Machines

Despite the common use of barrier synchronization in parallel application codes, there are very few references to barrier synchronization as a fundamental, hardware-supported, synchronization mechanism. There are at least a couple of reasons for this:

[1] Barriers are not as general as directed synchronization primitives (it is easy to simulate barrier synchronization using multiple directed synchronizations such as counting semaphores, but the reverse simulation is impossible) and

[2] Barrier synchronization has generally been viewed as a software issue — a programming style concern.

Actually, reason [1] isn't valid unless one ignores the fact that a barrier conceptually synchronizes processors at the clock-cycle level whereas most implementations of barriers using directed synchronization only approximately synchronize the processors. This approximation is mainly due to variations in network traversal times of synchronization requests and/or the fact that a tree-structured collection of synchronization operations is used. In other words, the directed-synchronization-based simulation of a barrier does not provide the key feature of barrier synchronization as discussed in this paper: simulated barriers do not yield the primary benefit of permitting fine-grain parallelism without requiring runtime synchronization for each conceptual synchronization operation.

Of course, reason [2] is simply a matter of convention.

The *only* machine the authors have found to deliberately implement barrier synchronization as the only hardware-supported synchronization mechanism is the "Controlable MIMD" machine described by Lundstrom and Barnes in 1980 [LuB80]. Apparently, Burroughs Corporation never built this machine, however, it was described in detail to NASA as a proposal for "the Flow Model Processor (FMP) in the Numerical Aerodynamic Simulator."

Rather than discussing barriers per se, the Burroughs proposal discussed hardware support for DOALL constructs. A DOALL is a loop such that the body can be executed simultaneously for all iterations, i.e., no serializing dependencies exist within the loop. The typical program was expected to consist of a sequence of DOALL loops where the body of each loop was arbitrary chunk of code. Since such a chunk of code would be likely to contain several control flow paths — each examining a particular special-case involving boundary conditions or making special-case simplifications to the computation — these DOALL loop "instances" could not be parallel executed using a SIMD machine. On the other hand, using a traditional MIMD model also would be a problem, because all processors must complete processing one DOALL loop before any can begin to execute code from the next DOALL, and this machine-wide synchronization would take a significant amount of time using conventional, directed, synchronization primitives. Their solution was to propose hardware which implemented a P-way synchronization

primitive (where P is the number of processors in the machine).

The P-way mechanism they presented is a machine-width barrier mechanism implemented by a processor instruction called `wait`. When each processor reaches the end of its work in parallel execution of a `DOALL`, it executes a `wait` instruction. As each processor executes a `wait` instruction, it is halted until all processors have executed a `wait` instruction. This halting is effected using synchronization lines independent from the network which interconnects processors and memory. Except for the constraint that *all* processors must participate in the synchronization instead of any arbitrary subset, this is precisely the static barrier mechanism we propose. Of course, the use of this mechanism for `DOALL` loops does not take advantage of the fact that immediately after processors have executed a barrier they may be scheduled as a VLIW.

Interestingly, the Burroughs proposal also references the design of PASM — the same machine which led us to study barrier synchronization. PASM's contribution in this respect is that the architecture allows both fine-grain and large-grain parallelism to be executed, although the fine-grain parallelism must conform to SIMD constraints. The burroughs proposal appreciated this property, and recognized that a barrier synchronization model could reduce the constraints on fine-grain parallel executable code structures while still supporting large grain parallelism.

3.2. The Proposed Barrier Mechanism

As the design of PASM and the proposal of Burroughs suggest, it is very difficult to achieve very low-cost, high time-precision, P-way synchronization in the context of using the communications network of a large multiprocessor. Further complicating matters, to achieve the maximum benefit the hardware should permit any arbitrary subset of the processors to synchronize; this implies that some hardware mechanism is able to specify for each barrier what subset of the processors should participate.

As we noticed in studying the PASM prototype architecture, this problem of generating the subset of processors to participate in each barrier synchronization operation is actually identical in nature to that of determining an enable pattern for SIMD processors. Hence, as a SIMD has a control processor which is responsible for generating enable

masks, an SBM or DBM machine incorporates a *barrier processor* whose sole responsibility to to generate the sequence of processor subsets for barrier synchronization.

As an example, a typical MIMD system design is given in Figure 1.

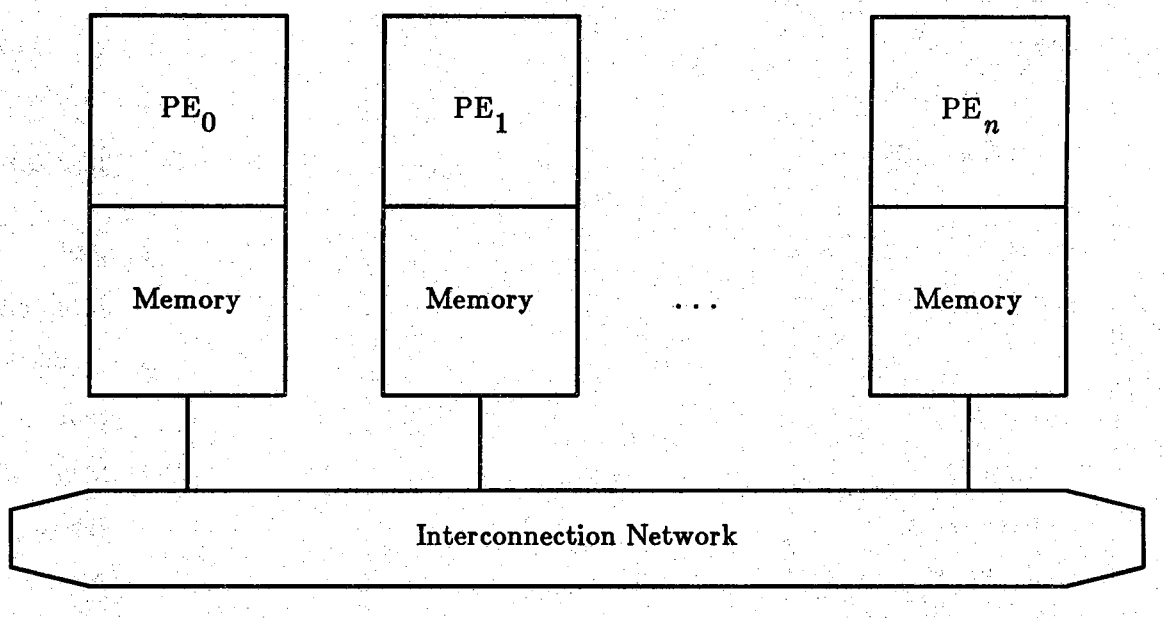


Figure 1: Conventional MIMD (with Local memory)

This design should be compared with the SBM/DBM design given in Figure 2.

Issues of, for example, shared vs. non-shared memory address space are irrelevant to the design because the general communication network is not used for barrier synchronization.

In Figure 1 — a typical MIMD — it is generally impossible to achieve *exact* synchronization between multiple processors since synchronization time-accuracy is affected by possible variations in network traversal time. Other stochastic delays are introduced either when "smart" combining (especially fetch-and-op [GoG83], RFM [Kla80], or RFM+ [Par86]) network switches are used or when a tree of binary semaphore operations is used to simulate a P-way tree. These properties have the effect of making timing ambiguity in synchronization approximately a log factor worse than if a separate, single-level, scheme is used.

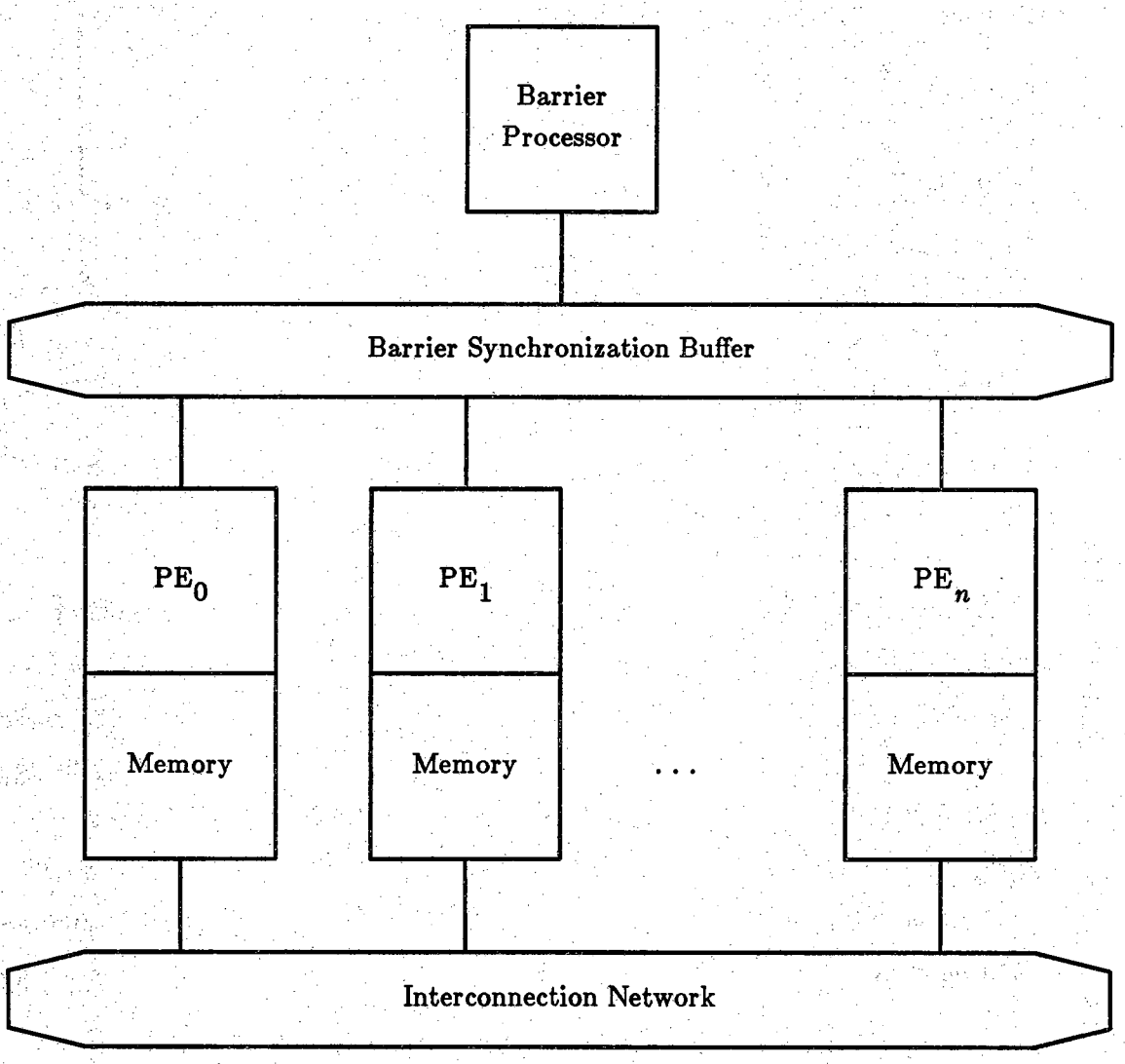


Figure 2: Barrier MIMD (with Local memory)

In contrast, the barrier MIMD of Figure 2 employs an independent barrier processor to generate barrier patterns. Each barrier pattern is a vector containing one bit per processor. The value of a bit determines whether the corresponding processor will participate in that synchronization barrier. These patterns are generated into a barrier synchronization buffer where each is held until it has been executed. In the SBM execution model, the barrier synchronization buffer acts as a simple FIFO queue; in the DBM execution model, barriers are executed and removed from the barrier synchronization buffer in the order in which barriers are encountered at runtime (implying an associative match

process). Since, in a DBM, there may be as many as $P/2$ possibly next barriers in a P-processor machine, it is the associative action of the buffer which implements the $P/2$ virtual synchronization control flow threads — the SBM and DBM barrier processors are identical.

In either SBM or DBM model, processors execute `wait` instructions (or instructions tagged with a `wait` bit) and are halted until the halted processor pattern completes the next barrier. A processor which is not involved in the current SBM barrier need not execute a `wait` for that barrier — if a `wait` is issued by a processor not involved in the current barrier, the SBM simply ignores that signal until a barrier including that processor becomes the current barrier. Since barrier patterns can be created asynchronously by the barrier processor and buffered awaiting their use, the main processors see no overhead in specification of barrier patterns. Hence, both SBM and DBM machines can achieve essentially perfect synchronization of any subset of processors with only a very small, roughly constant, overhead.

Of course, in addition to generating code for the main processors, in either SBM or DBM the compiler must precompute the order[1] and patterns of all barriers required for the computation and must generate code which the barrier processor will execute to produce these barriers. The code for the main processors also must contain the appropriate `wait` instructions or instruction tags. Separate `wait` instructions are probably easier to implement than tags, but tags would permit more frequent use of barriers . . . the trade-off depends on how often conceptual synchronizations occur in the code as compared to the time between variable-length operations or chunks of code.

4. Software (Compilation) Strategy

An SBM machine is, in every way, a superset of a VLIW machine. Hence, it is not surprising that one can compile code for an SBM using precisely the same techniques used in VLIW compilers, especially *trace scheduling*. Of course, using *exactly* the VLIW model would yield no better results for an SBM than for a VLIW. In this section, we outline

1. For SBMs, this is a complete order — a sequence. For DBMs, it is a partial ordering of maximum width $P/2$ for a P-processor machine.

two different approaches to compiling for SBMs, both based on VLIW scheduling. The first is a very simple add-on to the standard VLIW trace scheduling mechanism and the second is a more complex technique which may make better use of SBM hardware by explicitly considering "fuzzy" timing relationships.

4.1. VLIW Trace Scheduling

Trace scheduling for VLIWs is eloquently described in [Ell85], and we shall not review the technique here. Instead, this section defines the changes needed to adapt trace scheduling (and related scheduling, such as [Die87]) to generation of code for an SBM model.

To demonstrate the difference between pure VLIW scheduling and the minimal extension to VLIW scheduling for SBM machines, a simple example will be used. Figure 3 shows a set of regions of code[2] for this example. The regions are named A, B, C, D, E, F, G, and H, and each region is labeled with the *exact* amount of time required to execute that region. Parallel-execution precedence constraints are given by arrows which, as a matter of convention, point from the following process to its predecessor. In other words, if all eight code regions, A through H, were to be submitted for simultaneous execution, each arc would represent a directed synchronization operation where the consumer points to the producer.

It is easy to see that spawning eight processes and using directed synchronizations could be quite inefficient — there would be 9 directed synchronizations and only 18 units of useful work, also, only two of the code regions are ready to execute at any time. Hence, it is useful to conceptually re-package these regions into two sequential processes which may be parallel-executed: this grouping is indicated in Figure 3 by the dotted boxes. Once this has been done, only the three inter-process synchronization arcs need be considered because the other synchronizations are inherent in sequential order of execution within each process[3]. If inter-process synchronization were cheap enough, this

2. A region is an arbitrary grain size chunk of code having certain properties; see [Die87] for a precise definition. As a simplification, one may consider each region to be a sequence of a few instructions.
3. This is a general principle which we refer to as the **principle of selective**

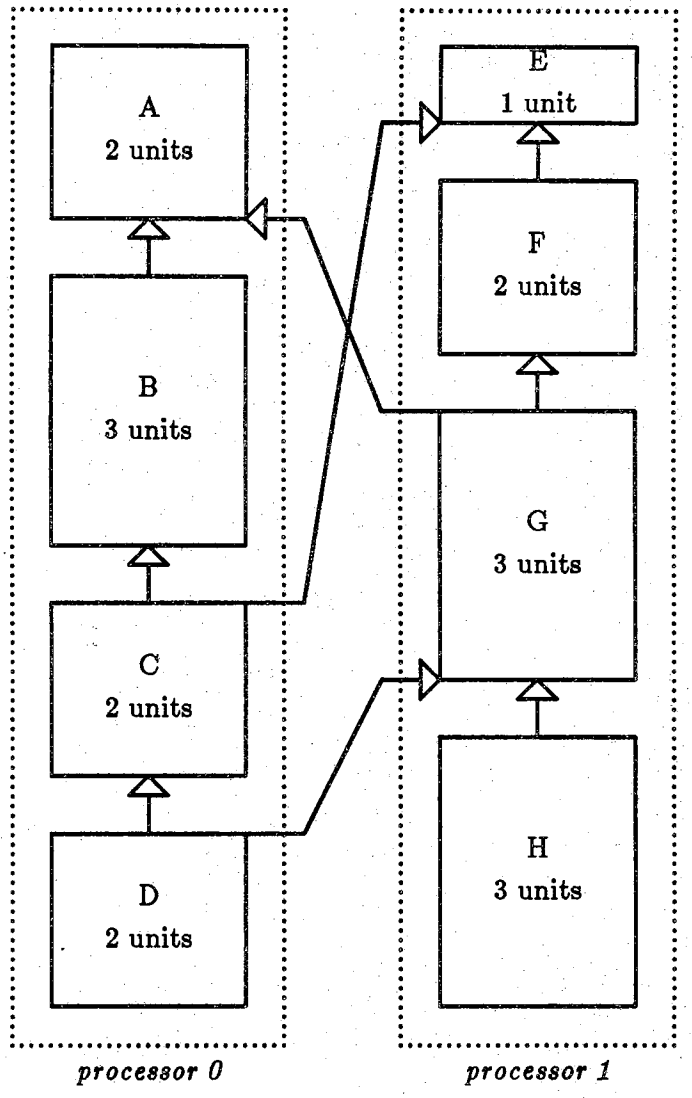


Figure 3: Sample VLIW Code

structure would be the optimal encoding for a traditional, directed synchronization, MIMD. Unfortunately, the cost of the directed synchronizations might easily make the completely serial version faster; if the three remaining synchronizations delayed the computation by more than 9 units of time, a completely sequential order such as A, B, E, C,

serialization. Serializing synchronization/communication arcs typically reduces or zeroes their cost, hence, for a given parallelism width it is best to package code regions into processes such that the fewest/lowest-cost arcs are inter-process.

F, G, D, H would execute faster. In summary, a directed-synchronization MIMD probably would find no useful parallelism at this grain level.

On the other hand, this is precisely the kind of code that VLIW execution was designed for. The VLIW execution of this code would merely observe that all "inter-process" synchronizations are satisfied by the static timing constraints (i.e., all the arcs point backward in time), and no synchronization cost would ever be incurred. Since SBM is a superset of VLIW, the same would be true of SBM execution of the code. A diagram of this is shown in Figure 4 (the inter-process arcs are drawn for reference purposes only).

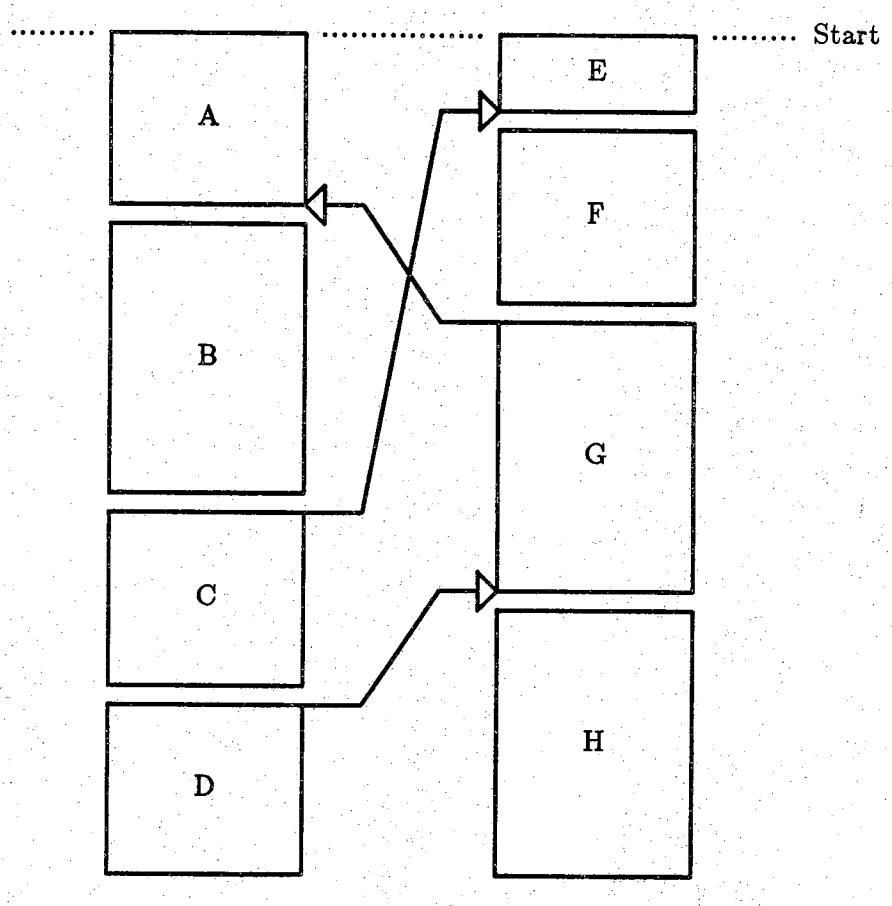


Figure 4: VLIW Code Executed using SBM

4.2. Simplified SBM Scheduling

The SBM model of execution is, however, more general than the VLIW model, and this difference can be demonstrated easily.

In order for a VLIW to execute the ordering given in Figure 4, the control flow after trace analysis for all the regions A through H would have to be the same — a VLIW machine has only one program counter. This implies, for example, that if B contains a loop, then a VLIW could not execute the above parallel structure[4]. The SBM model, on the other hand, can execute the parallel structure of Figure 4 no matter what control flow appears within each code region. Not only does the SBM permit loops and conditionals within each code region, but subroutine/function calls are permitted as well. To take full advantage of this, unlike a VLIW compiler which cannot parallelize calls, an SBM compiler may need to be able to examine the complete program or flow analysis results representing it.

A more insidious VLIW constraint is, however, that the compiler *must know exactly* what the execution time will be for each region of code. Without such perfect knowledge, operations must be completely serialized such that *at any given time, only a single region is executing.* Further, the compiler's time estimate may be imperfect for any of three reasons:

[1] There may be variable-time operation(s); operations whose execution time is data dependent or is dependent on other dynamic properties of the execution, such as I/O traffic or interrupts.

[2] A time variation could be due to different control flow paths (while loops or conditional branches) being taken.

[3] The imprecise knowledge could be exactly that — the compiler analysis may fail to discover the exact execution time even though it is theoretically knowable. A good example is that the compiler may may make its execution time estimates before the final code has been generated: unexpected code generation conditions, such as a failure to place a variable in a register or the assembler's recognition that a shorter

4. One could argue that unraveling the loop would be reasonable — and VLIW compilers often take this approach — but the expansion in code size and lessening of locality properties (reduction in cache performance, etc.) limits the performance.

span-dependent instruction [Szy78] could be used in a particular case, may cause the compiler's estimate to be in error.

Current VLIWs minimize these problems by forcing operations to always take their worst-case execution time, but this can cause even more profound damage.

To demonstrate the effect of imprecise time knowledge on VLIW schedules, suppose that code region B might take anywhere from 1 to 3 units of time to execute, and in similar manner region G could take either 2 or 3 units of time. This would result in the (clearly undesirable) VLIW parallelization of Figure 5.

The problem here is simply that VLIW hardware has no mechanism for regaining synchronization if it is ever lost, hence, it cannot permit asynchrony of any kind. SBM machines do not have this constraint, however. Consequently, the minimal extension of a VLIW compiler to take advantage of SBM is simply:

[1] As long as all timing constraints are known, perform ordinary trace scheduling — except that conditionals, loops, and calls are permitted to remain intact.

[2] Whenever a variable-time (or imprecisely known time) code region is encountered, set a compiler-internal flag noting that time is not precisely known for the process containing this region, and continue scheduling (as in [1]) based on a compiler-generated "guesstimate" of the region's average execution time. Upon completing a step in the schedule, if the next step in the schedule is the "consumer" of a synchronization produced by another process, check the imprecise-time flags on both the producer and consumer processes. If either process is flagged as being imprecisely known, insert a barrier before the next region and reset the imprecise-time flags.

Using this scheme, the compiler will generate code which may contain unnecessary barriers, but since barrier synchronization is very fast, this results in only a minor performance loss. For the code whose VLIW schedule is given in Figure 5, Figure 6 presents the SBM schedule derived using the above algorithm.

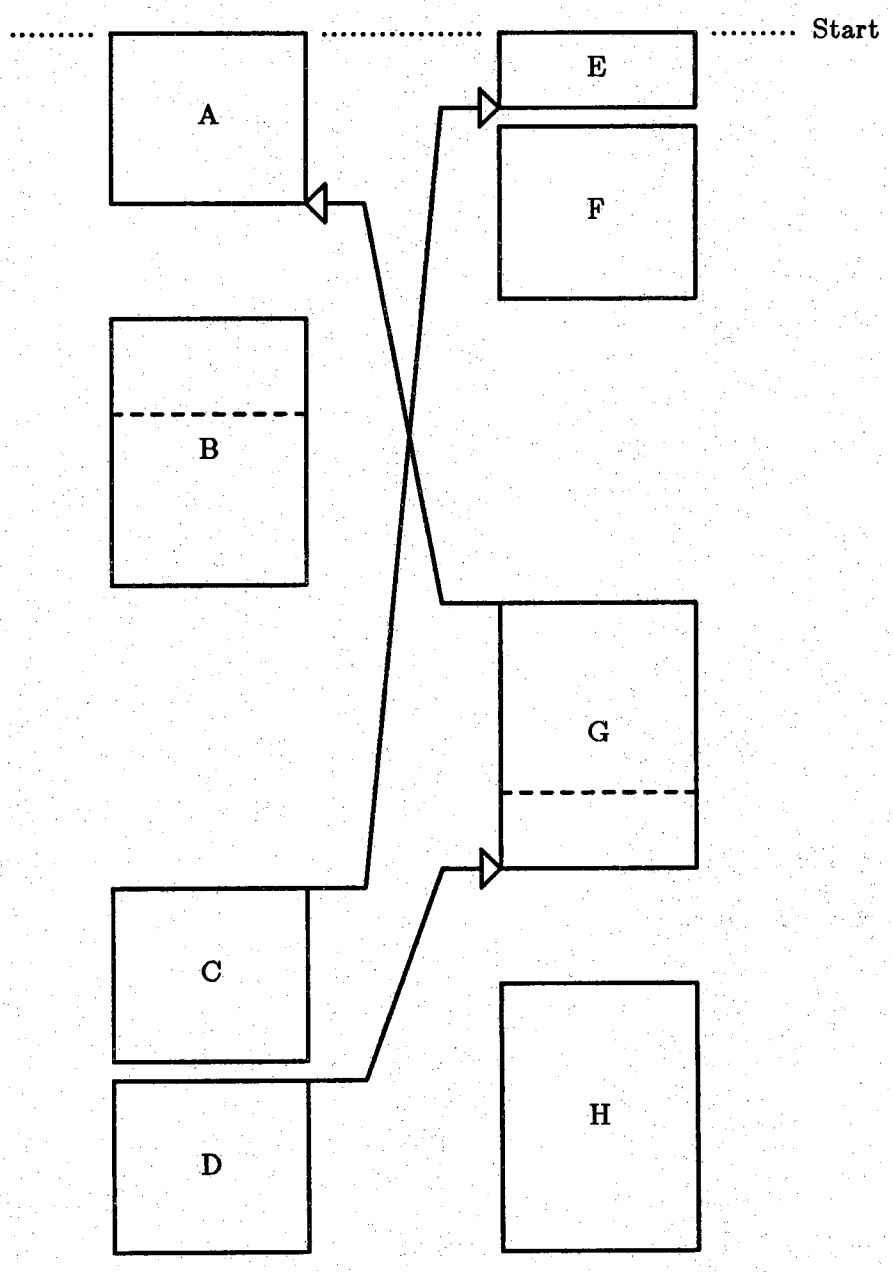


Figure 5: Variable-Time Code Executed using VLIW

4.3. SBM Scheduling

To achieve the maximum possible benefit from the SBM model, it is necessary for the compiler to consider not just that it has imprecise time estimates, but also precisely *how imprecise* the estimates are.

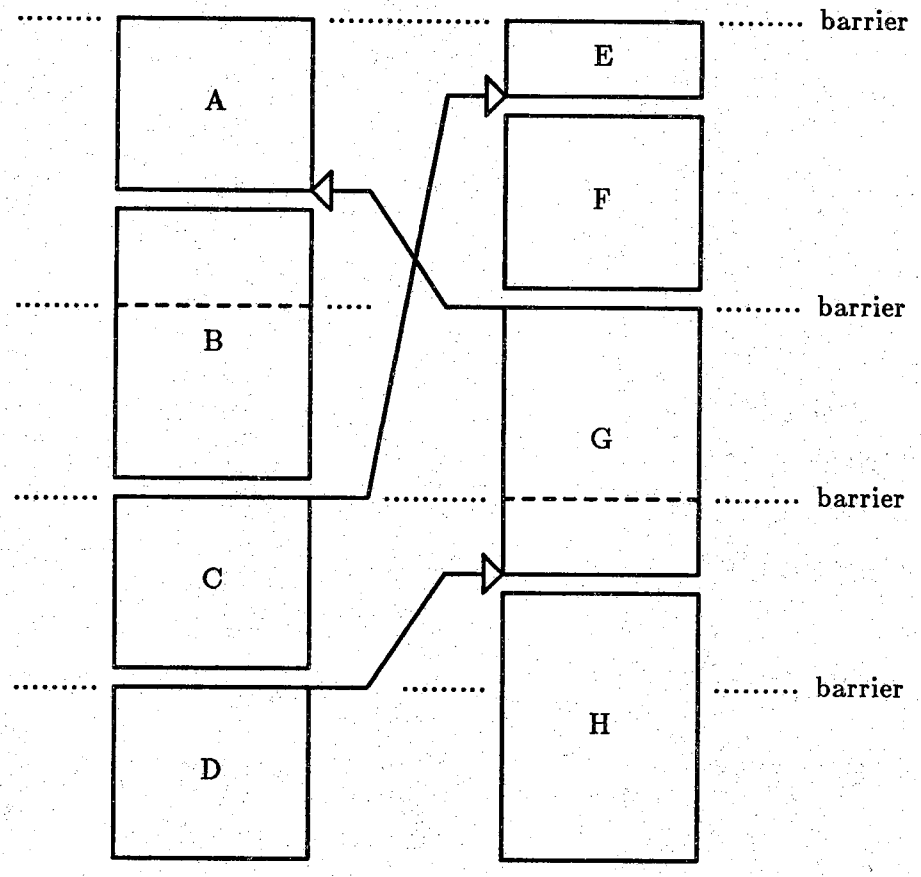


Figure 6: Variable-Time Code Executed using SBM

Consider a single directed synchronization operation which takes place between regions in two different processes. Let T_p be the time at which the producer region completes execution and T_c be the time at which the consumer region begins execution. The synchronization constraint will be satisfied *iff* $T_c \geq T_p$. Hence, we can make the following observations:

[1] Any variable-time operations in the producer's process which occur after the producer region are irrelevant.

[2] The position within the consumer's process at time T_p must be *before* the consumer region begins.

which lead to a very simple algorithm for tracking time imprecisions and inserting barriers. The algorithm to determine whether adding a new step to a schedule requires insertion of a barrier is:

[1] If no region scheduled in this step is a consumer of a synchronization whose producer region is scheduled in another process, no barrier need be inserted before this schedule step. If there is at least one consumer of a synchronization from another process, then perform steps [2] through [6] for each of these consumer regions.

[2] Beginning with the producer code region corresponding to the current consumer region, scan forward looking for a barrier in which both the producer and consumer processes participate. If there is such a barrier, then the synchronization is redundant because an earlier region of the consumer is guaranteed to be executed after the producer code region, and no barrier is needed. If there is no such barrier, go to step [3].

[3] (Steps [3] and [4] simply find the the closest dominating barrier of the producer and consumer.) Beginning with each of the producer and consumer code regions, scan backward to find the last barrier encountered. If the same barrier was found for both producer and consumer processes, go to step [5], else go to step [4].

[4] An irrelevant barrier has been encountered by one of the two processes since the producer and consumer processes last synchronized, hence, the timing error for the occurrence of that barrier must be added in. We say a processes is indirectly involved in a barrier creation problem if it is not a participant in the proposed barrier, but it is a participant in a barrier which propagates timing error to a process in the proposed barrier. To resolve this, continue scanning backward to find the first barrier in which all processes directly or indirectly involved in the proposed barrier were participants. Proceed with step [5].

[5] Beginning with this dominating (common ancestor) barrier, scan forward on both the consumer and producer processes accumulating relative time and time error bounds. If the minimum time since the dominating barrier for the consumer region is greater than or equal to the maximum time since the dominating barrier for the producer region, no barrier is needed. Otherwise, go to step [6].

[6] The result at this point is that a barrier needs to be placed somewhere between the producer and consumer processes anywhere such that the barrier appears after the producer region and before the consumer region. For best performance, one simply remembers these constraints and if the current step required creation of several barriers, one first trys to find an overlap in constraints which will permit a single barrier to be generated instead of several. In other words, two 2-process barriers might become one 4-process barrier, etc.

The final result of applying the above algorithm to the example is given in Figure 7.

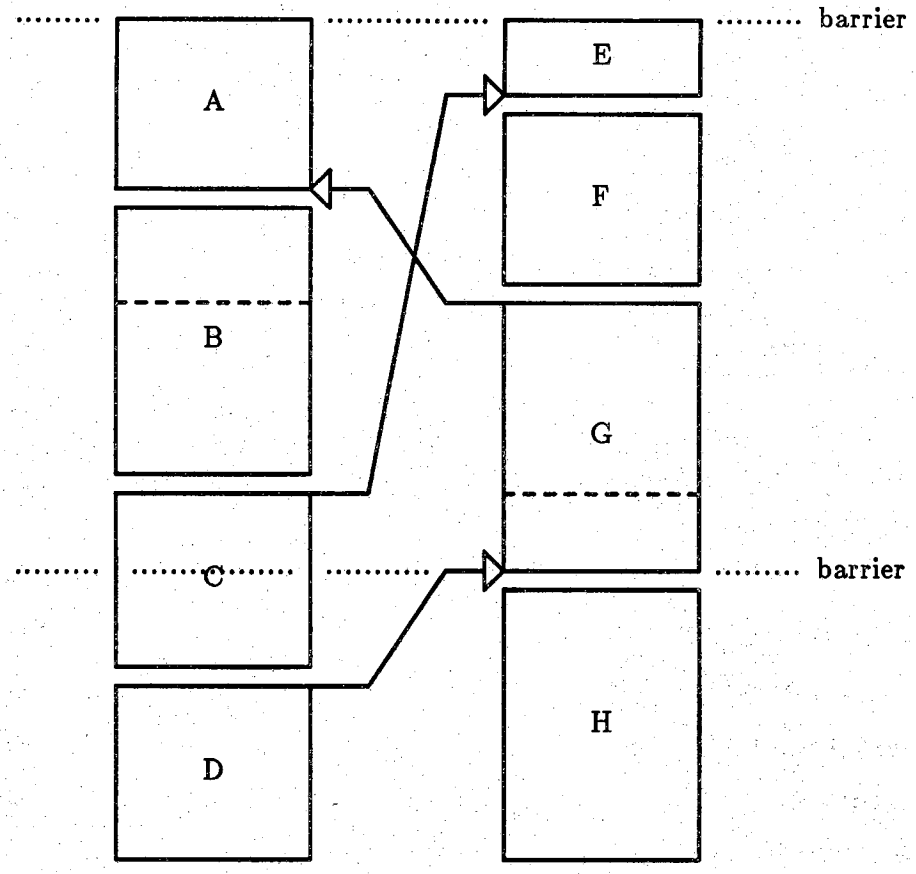


Figure 7: Variable-Time Code Optimally Executed using SBM

5. Conclusions

In this paper, a new classification of parallel computer architectures is presented. Based on this taxonomy, several new and useful architectural concepts — particularly the **static barrier MIMD** (or **SBM**) — are proposed and explored.

Although barrier synchronization has existed as a programming concept for many years, the SBM model recognizes and exploits synchronization barriers not as "cheap approximations" to directed synchronization primitives, but as operations manipulating relative timing constraints which are statically determined (by the compiler). Hence, SBMs can be viewed as relaxing the constraints on parallel structure which were imposed by SIMD and even VLIW models, yet preserving the primary benefits of static scheduling and (in most cases) zero-cost synchronization. The associated compiler technology has also been outlined.

Future research will construct and test SBM compilers using the technologies outlined in Section 4, as well as design and simulate specific SBM and/or DBM architectures. We also believe that the general concept of catagorizing architectures based on what may be considered static (i.e., compile-time) constraints on parallel execution structure will prove valuable in analysis of existing, as well as future, computer architectures.

Acknowledgements

Thanks to the members of CARP (the Compiler-oriented Architecture Research group at Purdue) for their useful comments on and discussion of the architectural catagorization and SBM architecture presented in this paper. Thanks to H. J. Siegel and T. L. Casavant for discussions relating to use of PASM as a SBM machine.

References

[Die87] H. G. Dietz, *The Refined-Language Approach to Compiling for Parallel Supercomputers*, PhD Dissertation, Polytechnic University, June 1987. (An updated version is also available as a Purdue University internal report, May 1987.)

[Ell85] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, 1985 ACM Doctoral Dissertation Award, MIT Press, 1986.

[FiC87] S. A. Fineberg, T. L. Casavant, and T. Schwederski, "Mixed Mode Computing with the PASM System Prototype," 25th Annual Allerton Conference on Communication, Control, and Computing, Monticello, Illinois, 1987, pp. 258-267.

[FiC88] S. Fineberg, T. Casavant, T. Schwederski, and H. J. Siegel, "Non-Deterministic Instruction Time Experiments on the PASM System Prototype," to appear in the Proceedings of the 1988 International Conference on Parallel Processing, August 1988.

[GoG83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer," IEEE Transactions on Computers, Vol. C-32, No. 2, February 1983, pp. 175-189.

[Kla80] D. Klappholz, "An Improved Design for a Stochastically Conflict-Free Memory/Interconnection System," Proceedings of the 14th Asilomar Conference on Circuits, Systems, and Computers, November 1980.

[LuB80] S. F. Lundstrom and G. H. Barnes, "A Controllable MIMD Architecture," IEEE Proceedings of the 1980 International Conference on Parallel Processing, August 1980, pp. 165-173.

[Par86] H-C. Park, *Smart Switching Nodes in an MIMD Architecture*, PhD Dissertation, Polytechnic University, December 1986.

[ScN87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "The Implementation of the PASM Prototype Control Hierarchy," Proceedings of the Second International Conference on Supercomputing, Volume i, 1987, pp. 418-427.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," IEEE Transactions on Computers, Vol. C-30, December 1981, pp. 934-947.

[Szy78] T. G. Szymanski, "Assembling Code for Machines with Span-Dependent Instructions," Communications of the ACM, Vol. 21, No. 4, April 1978, pp. 300-308.

[Thi87] Thinking Machines, *Connection Machine Model CM-2 Technical Summary*, Thinking Machines Technical Report HA87-4, April 1987.