1-1-1990

Static Scheduling for Barrier MIMD Architectures

Abderrazek Zaafrani
Purdue University

Henry G. Dietz
Purdue University, hankd@ecn.purdue.edu

Matthew T. O'Keefe
Purdue University

Static Scheduling for Barrier MIMD Architectures

Abderrazek Zaafrani
Henry G. Dietz
Matthew T. O'Keefe

TR-EE 90-10
January, 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Static Scheduling for Barrier MIMD Architectures

Abderrazek Zaafrani, Henry G. Dietz, and Matthew T. O'Keefe

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

ABSTRACT

Barrier MIMDs are asynchronous Multiple Instruction stream Multiple Data stream architectures capable of parallel execution of variable-execution-time instructions and arbitrary control flow (e.g., `while` loops and calls); however, they differ from conventional MIMDs in that the need for run-time synchronization is significantly reduced. Whenever a group of processors within a barrier MIMD encounters a synchronization point (barrier), static timing constraints become precise, hence, conceptual synchronizations between the processors often can be statically resolved with zero cost — as in a SIMD or VLIW and using similar compiler technology. Unlike these machines, however, as execution continues past the synchronization point the accuracy within which the compiler can track the relative timing between processors is reduced. 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 static, implicit synchronization.

This paper describes new scheduling and barrier placement algorithms for barrier MIMDs that are based loosely on the list scheduling approach employed for VLIWs [Elli85]. In addition, the experimental results from scheduling more than 3500 synthetic benchmark programs for a parameterized barrier MIMD machine are presented.

Keywords: Static Barrier MIMD (SBM), Dynamic Barrier MIMD (DBM), barrier synchronization, code scheduling, compiler optimization.

1. Introduction

Runtime synchronization overhead is a critical factor in achieving high speedup using parallel computers. 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) [Elli85], [CNOPR88] machines are successful in large part because they preserve this property while providing more flexibility in terms of the operations that can be parallelized. Unfortunately, VLIWs cannot tolerate any asynchrony in their operation; hence, they are incapable of parallel execution of multiple flow-paths, subprogram calls, and variable-execution-time instructions. In a recent paper [DSOZ89], a new architecture was proposed that extended the static synchronization properties of the SIMD and VLIW class of parallel machines into the MIMD domain. The new architecture is called the *Static Barrier MIMD*, or *SBM*.

In this paper, we describe scheduling and barrier placement algorithms for barrier MIMDs, as well as extensive results from scheduling synthetic benchmarks using the new algorithms. An SBM is a MIMD computer which has specialized hardware implementing a new type of barrier synchronization which allows the compiler to perform static scheduling. If an SBM barrier is placed across a set of processes, then no process can execute past that barrier until all have reached the barrier. Unlike other barrier mechanisms, all processes will resume execution in exact synchrony[1]. Hence, immediately after executing an SBM barrier, the machine can be treated as a VLIW, using static scheduling to eliminate the need for further runtime synchronization.

However, VLIW machines do not allow MIMD code structures (e.g., multiple flow paths) nor even variable-time instructions. In an SBM, static scheduling tracks both minimum and maximum completion times for each processes' code; runtime synchronization is needed *iff* the minimum time for the consumer of an object is less than the maximum time for that object's producer. If the timing constraints cannot be met statically, this implies that the static timing information has become too "fuzzy." Inserting another barrier effectively reduces this fuzziness to zero. Based on the static scheduling work described in this paper, more than 77% of all synchronizations which would occur in execution on a conventional MIMD will be accomplished without runtime synchronization in a barrier MIMD.

When barrier MIMD architectures were originally proposed [DiSc88], [DSOZ89] , an algorithm for inserting barriers while scheduling code was given. The algorithm attempted to minimize the number of barriers required by using the static timing constraints inherent in the barrier synchronization operation. In these previous papers, no implementation of the barrier insertion algorithm was attempted nor was there any algorithm proposed for the actual code scheduling [DSOZ89]. This work contains three new results: a code scheduling algorithm for barrier MIMDs, an "optimal" barrier insertion algorithm, and extensive scheduling experiments on synthetic benchmarks using the new algorithms.

[1] Machines with this constraint will be called *barrier MIMDs* in this paper.

The paper is organized as follows. Section two describes the structure of the synthetic benchmark programs, while section three explains the principles of operation for a barrier MIMD. Section four gives the scheduling and barrier insertion algorithms, and is followed by the description of the scheduling experiments in section five. Section six provides a comparison between VLIW and SBM performance for the synthetic benchmarks used in this paper. Finally, section seven gives conclusions and describes current research efforts.

2. Structure of the Synthetic Benchmark Programs

This study focuses on fine-grain scheduling of a single-chip multiprocessor RISC node [DSCO89] that employs the barrier mechanism discussed in this paper. Expensive operations such as multiplication and division are implemented as data-dependent code sequences that introduce asynchrony into the chip operation. Memory accesses across a shared bus or interconnection network involve contention that also involves stochastic delays. It is shown that static scheduling may still be used to advantage within this framework.

In this work, we wished to characterize and study the extent to which static scheduling can be employed in barrier MIMDs. In particular, measurements of the number of synchronizations that are satisfied statically, at compile-time, versus the number that require explicit synchronization instructions executed at run-time were desired. To this end, a compiler was developed for a simple language consisting of basic blocks of code with no control flow constructs.

The programs to be scheduled on barrier machines were automatically generated using common instruction execution frequencies [AlWo75]. This allowed us to automatically generate a very large number of *synthetic benchmarks* from which summary statistics were obtained. It also made it quite simple to change the various characteristics of generated programs to observe the effects on the statistics of scheduled programs. The drawback, of course, is that it is not possible to take real benchmark programs directly as input. Current efforts include prototype compiler development for generating code for barrier MIMDs from a standard set of programming language constructs, including control structures, array and pointer data structures, and subroutine calls [OKee90]. We view the scheduling results for the synthetic benchmarks as conservative in terms of comparing the performance of VLIWs and the SBM since it is precisely for such programming constructs that the SBM is superior.

2.1. Benchmark Instruction Set

The scheduling algorithm takes as input a basic block of instructions. A basic block is a region of code that contains a sequence of consecutive statements. This region should have a single entry point and no embedded control structures [AhSU86]. There are nine instructions generated from the synthetic code sequences in the instruction set: four of these nine instructions have variable execution time. The variable-time instructions are as follows:

Instruction	Execution Freq.	Min. Time	Max. Time
Load	—	1	4
Store	—	1	1
Add	45.8%	1	1
Sub	33.9%	1	1
And	8.8%	1	1
Or	5.2%	1	1
Mul	2.9%	16	24
Div	2.2%	24	32
Mod	1.2%	24	32

Table 1: Instruction Frequencies and Execution Time Ranges

Load Execution time varies from one to four time units. In a shared-bus multiprocessor, this difference is mainly due to different access times between local cache and main memory. Typically, an access to the main memory is anywhere from four to twenty times longer than an access to cache. A more pronounced difference between local and non-local memory is often found in multiprocessors that require that non-local accesses to go through single- or multistage-interconnection networks.

Mul Execution time varies from 16 to 24 units of time. This assumes that the multiplier operation is either implemented using shift and add instructions, or in an asynchronous hardware design. In either case, execution time is variable to take advantage of data-dependent optimizations. Synchronous designs with constant execution time are possible, but require more hardware, typically in the form of pipelines. Since multiplication is a commonly executed instruction, this additional hardware sometimes can be justified.

Div Execution time varies from 24 and 32 time units, for much the same reason as multiply. Asynchronous designs are much more common for division, as the operation is inherently harder to pipeline and the additional hardware is not justified by its typically low execution frequency.

Mod The execution time of the modulus operation Mod varies between 24 and 32 time units, for the same reasons as division.

For the other operations, it is realistic to assume that they have a constant execution time of one unit. These operations are Or, And, Add, Sub, and Store. Table 1 summarizes the instruction frequencies and execution time ranges.

Tuple No.	Instruction	Min. Time	Max. Time
0	`Load i`	1	4
1	`Load a`	1	4
2	`Add 0,1`	2	5
3	`Store b,2`	3	6
4	`Load f`	1	4
24	`Load d`	1	4
5	`Load j`	1	4
12	`Load c`	1	4
26	`And 4,24`	2	5
6	`Add 4,5`	2	5
30	`Sub 26,4`	3	6
18	`Sub 6,0`	3	6
22	`Add 1,2`	2	5
38	`Add 12,30`	4	7
19	`Store i,18`	4	7
23	`Store a,22`	3	6
27	`Store h,26`	3	6
31	`Store e,30`	4	7
39	`Store g,38`	5	8

Figure 1: Instructions from Example Synthetic Benchmark

2.2. Benchmark Synthesis

A C program was developed to randomly generate the basic blocks according to the statistics described below. This program requires as input the number of statements, variables, and constants desired in the generated code. It then generates a random sequence of assignment statements satisfying the desired conditions. The frequency of the assignment statements corresponds loosely to the instruction frequency distributions found in [AlWo75]. Note that in table 1 the frequencies of load and store are not given. These instructions are provided as necessary during code generation and optimization: the first reference to a variable causes a load for that variable to be generated, and a store is generated when a variable is assigned a value.

During code generation, the randomly-generated assignment statements are optimized using standard local optimizations, including common subexpression elimination, constant folding and value propagation, and dead code elimination [AhSU86]. Hence, the resulting synthetic benchmark does not contain "redundant" parallelism that might skew the results.

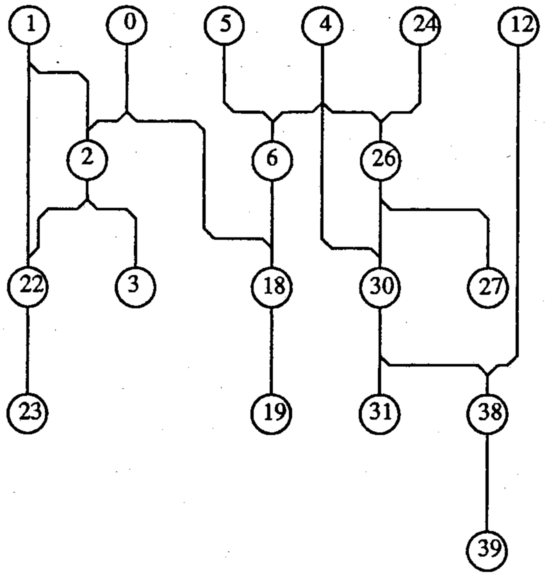


Figure 2: Instruction DAG for Example Synthetic Benchmark.

An example synthetic benchmark is shown in figure 1, and its corresponding DAG (directed acyclic graph) is shown in figure 2. In figure 1, the leftmost column represents the tuple number. Each tuple is incrementally assigned a number as it is generated by the code generator. Many tuples are not represented because they were removed by the optimizer. The two rightmost columns represent the minimum finish time and maximum finish time on an infinite number of processors. This columns will help in ordering the tuples as it will be explained in section 4. In the instruction DAG the instructions are represented as nodes and edges represent the precedence constraints between instructions. This DAG is important to both the code optimizations and scheduling algorithms.

Parameters for both the generated code and the scheduling algorithms can be varied. These machine size for the scheduling algorithm can be varied from 2 to 128 processing elements. The parameters for the random sequences of assignment statements include the number of variables and statements. The number of variables corresponds roughly to the parallelism width of the generated benchmark after optimization. For a fixed number of processors, the number of variables can be varied from 2 to 15 variables. For a fixed number of processors and a fixed number of variables, the number of statements can be varied from 5 to 60 statements. The larger basic blocks sizes approximate a long instruction trace found in VLIW scheduling [Elli85].

3. Barrier MIMD Principles of Operation

Figures 3 through 8 are used to illustrate the basic scheduling concepts behind barrier MIMD architectures. Figure 3 depicts the use of conventional directed synchronization to insure that the producer executes before the consumer. Here, at run-time, a synchronization object is transmitted from the producer to the consumer. This transmission could take a potentially unbounded amount of time dependent on routing and traffic through a network; hence, the only timing information available at compile-time is that the consumer will execute at a time after the producer.

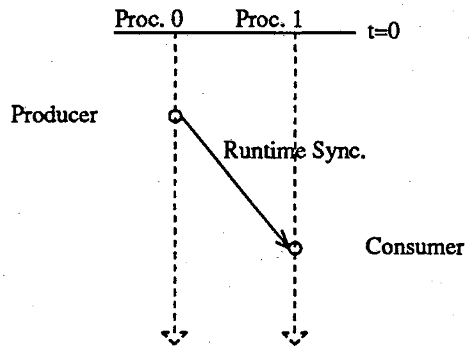


Figure 3

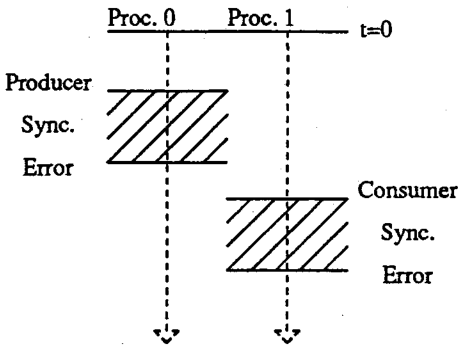


Figure 4

Instead, suppose that compiler analysis attempts to precisely track the minimum and maximum times at which the producer and consumer would execute without using any runtime synchronization. As shown in figure 4, if the minimum consumer time is greater than the maximum producer time, then no runtime synchronization is required. If not, as in figure 5, then it is necessary to insert a barrier to impose timing constraints which will be known to satisfy the producer/consumer relationship. This is shown in figure 6.

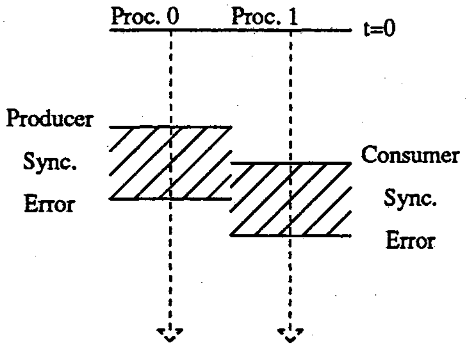


Figure 5

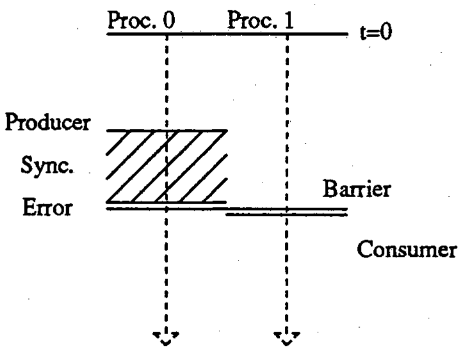


Figure 6

Shaffer [Shaf89] and others have applied a transitive reduction [AhHU74] to task graphs to remove redundant synchronization in code executing on MIMD architectures. Callahan [Call87] proposed a similar method for reducing the number of (conventional) barrier synchronizations required in scheduling nested loop constructs. However, these techniques will only remove synchronizations based on graph structure, rather than on knowledge of minimum and maximum execution time bounds as we propose. In addition to removing synchronizations based on the structure of the task graph, we can safely remove those synchronizations constrained by timing to be satisfied despite the lack of a subsuming synchronization.

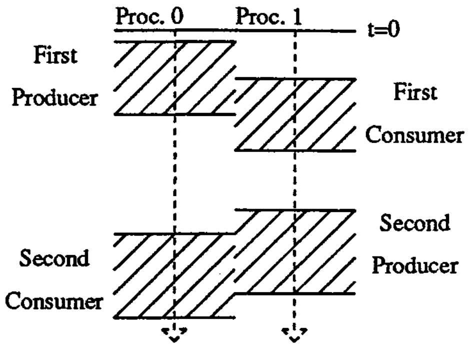


Figure 7

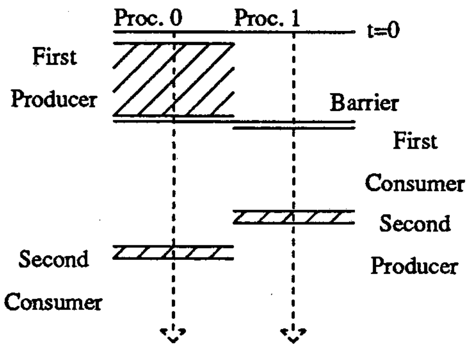


Figure 8

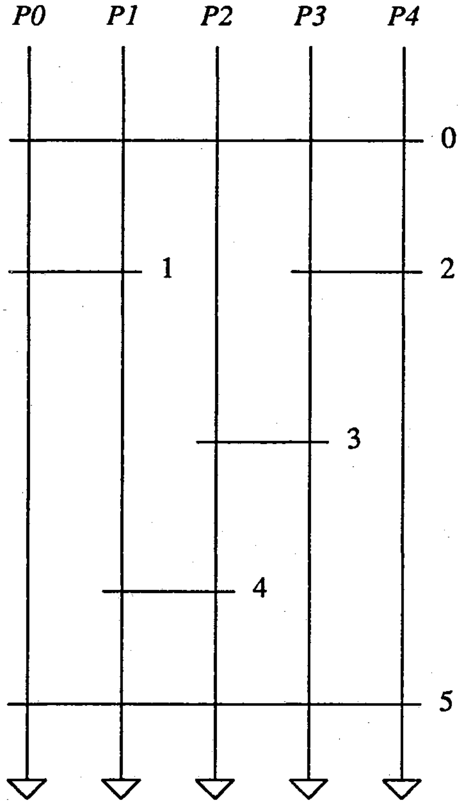
There is also a secondary effect unique to our scheduling for the proposed barrier architectures. Typically, there will be a sequence of several producers and consumers in the code being analyzed and scheduled, as shown in figure 7. Although at first it seems that each such producer/consumer pair will require a runtime synchronization (hence, two barriers for figure 7), the insertion of a barrier satisfying the first producer/consumer constraint causes the timing of later producer/consumer pairs to be more precisely known. This often (about 28% of the time in our current studies) allows the compiler to avoid inserting further barriers, as shown in figure 8.

3.1. Models for Barrier Synchronization

We now introduce representations for barrier synchronization in concurrent processors. These representations will help in understanding barrier MIMD execution and design alternatives. In this work, the *barrier embedding* for a set of concurrent processors will be represented as in figure 9. The vertical lines represent concurrently executing processors while the horizontal lines represent barriers across the processors they intersect. The semantics of these barriers are that the participating processors cannot proceed until all have arrived at the barrier, e.g., in figure 9, processors *P0, P1, ..., P4* cannot proceed past barrier 0 until all have arrived there. At that point, they all start execution of the instruction following the barrier *simultaneously*. Process execution proceeds in the downward direction.

Several concepts and results from the theory of partially ordered sets are useful in understanding barrier embeddings within concurrent processors. Recall that a *binary relation* R on a set P is a subset of the Cartesian product X^2, that is $R \subseteq X \times X$. Let xRy correspond to $(x, y) \in R$, and not(xRy) represent $(x, y) \notin R$. The binary relation $<_b$ on a set of barriers B is a *partial ordering* because $<_b$ is both *irreflexive* and *transitive*[2] [Fish85]. The partially ordered set $(B, <_b)$ may be illustrated by a directed acyclic graph (dag), with the graph nodes representing barriers and edges representing the ordering relations $<_b$ among the

[2] A binary relation R on X is *irreflexive* if not xRx for every x in X. It is *transitive* if $(xRy, yRz) \Rightarrow xRz$ for all x, y, z in X.



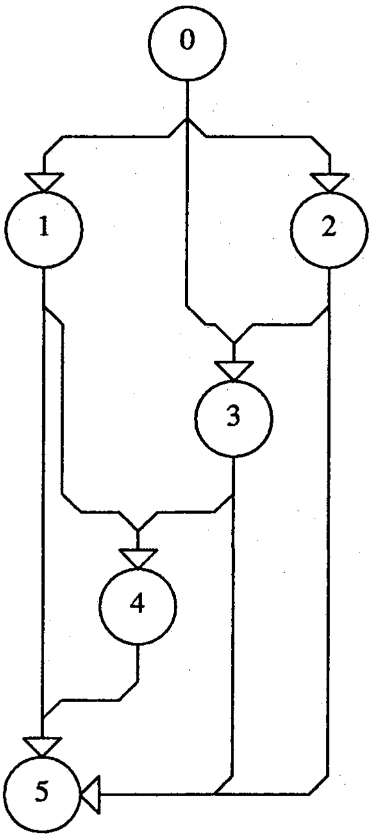


Figure 9

Figure 10

barriers. The *initial barrier* is defined as the barrier that extends across all processors and precedes all other barriers. A barrier dag for the barrier embedding in figure 9 is shown in figure 10. The initial barrier for this dag is b_0 (barrier 0). Here we see that b_2 (barrier 2) must execute before b_3 (barrier 3), hence $b_2 <_b b_3$, and similarly $b_3 <_b b_4$. Transitivity implies $b_2 <_b b_4$. These properties are derived from the barrier semantics: barrier b_3 must be executed after the process *P3* has encountered barrier b_2. Similarly, b_4 must be executed after the process *P2* has encountered b_3.

Several terms that will be used frequently in this paper are now defined.

Total Implied Synchronizations:

The number of edges in the directed acyclic graph (DAG) corresponding to the code generated from a basic block. Each edge is considered to be a producer/consumer synchronization pair.

Barrier Synchronization Fraction:

The number of barriers in the schedule divided by the *Total Implied Synchronizations*.

Serialized Synchronization Fraction:

The number of synchronizations satisfied by serialization, i.e., consumer assigned to same processor as a producer, divided by the *Total Implied Synchronizations.*

Static Scheduling Fraction:

The remaining fraction of total implied synchronizations after the barrier and serialized fractions are removed. This represents the synchronizations that are scheduled away by tracking static timing constraints after a barrier executes, as in the second producer/consumer synchronization of figure 8. In this case, no explicit synchronization instruction need be generated.

3.2. Barrier MIMD Hardware

Two forms of *barrier MIMD* are discussed in this paper: *static barrier MIMD (SBM)* and *dynamic barrier MIMD (DBM)*. The difference between the two lies in the run-time ordering of the barriers: the SBM imposes an ordering at compile-time, and barriers may be delayed if this compile-time (static) ordering differs from the run-time ordering. The DBM executes the barriers in whatever run-time order they occur. It requires an associative matching memory to achieve this, whereas the SBM requires only a hardware queue [OKDi90].

Although, intuitively, a synchronization operation which can span an arbitrary set of processes appears to imply high overhead, we believe that the new barriers can be implemented with lower overhead than conventional binary producer/consumer synchronization. In fact, the PASM prototype [ScNa87] has hardware capable of implementing SBM execution and preliminary benchmarks have demonstrated very good performance [BrCJ89]. These benchmarks have shown the barrier execution mode to consistently outperform both SIMD and MIMD modes.

The detailed hardware design and performance analysis for hardware barrier synchronization is discussed in a companion paper [OKDi90]. We briefly outline it here. The SBM barrier hardware has a very simple structure closely resembling the enable/disable mask logic of a SIMD control unit. Each barrier is represented by a bit mask indicating which processors participate in that barrier; these bit masks are enqueued into a FIFO queue in the sequence in which they will be executed, as shown in figure 11. Processors activate a *WAIT* signal when they execute a `wait` instruction. When the set of processors waiting for a barrier becomes a subset of the waiting processors in the top barrier mask, the top barrier executes and is removed from the queue. Processors participating in the executed barrier then proceed past the `wait` instruction. If a processor not participating in the top barrier executes a wait, the wait instruction will not complete until a barrier in which that processor participates becomes top and fires. When an barrier executes, all participating processors resume execution simultaneously (on the next clock tick).

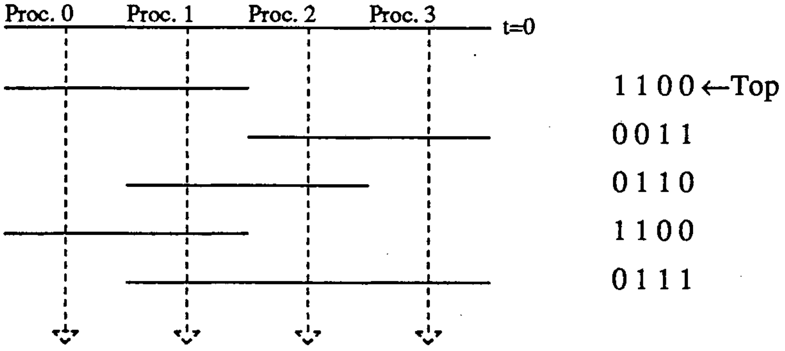


Figure 11

4. Code Scheduling and Barrier Insertion Algorithms

The code scheduling algorithm will be described for barrier MIMDs in general. The appropriate restrictions for the static barrier MIMD (SBM) are given at the end of this section. Although minimum and maximum times are known for each of the instructions instead of a fixed execution time, it is straightforward to adapt the scheduling heuristics commonly used for fixed-execution-time tasks — and this has been our approach. It is well-known that even for simple or relaxed cases the optimal static scheduling of a partially ordered set of tasks on parallel processors is NP-hard, and hence, computationally intractable [KaNa84]. However, several heuristics with bounded worst-case performance degradation (from optimal) have been found to be effective for this problem [Hu82]. In particular, the *critical path* method exhibits good performance at reasonable computational cost.

Section 4.1 outlines a technique for labeling operations and section 4.2 applies these labels to generate an ordering for list scheduling. Using the list ordering, section 4.3 details the assignment of operations to specific processors. Upon assigning each operation to a processor, it may be necessary to insert a barrier; algorithms for this purpose are given in section 4.4.

The scheduling algorithm proceeds in two phases: ordering of the nodes based on height information, followed by the node assignment to processors, with barrier synchronizations inserted as necessary during node assignment.

4.1. Node Labeling

The scheduling algorithm assumes that the instructions are represented in an instruction DAG[3] (Directed Acyclic Graph) $G(N, A)$, where N is the set of n instruction nodes and A is the set of edges

[3] The directed acyclic graph for instructions will be specified in upper-case (DAG) whereas the directed acyclic graph for barriers will be given in lower-case (dag).

representing the precedence (producer/consumer) constraints between instructions. If an edge is directed from node i to node j, node j is said to be a *successor* (or *consumer*) of node i. Similarly, node i is said to be a *predecessor* (or *producer*) of node j.

The DAG is assumed to have one *entry* and one *exit* node; *dummy* nodes (with zero execution time) are added if necessary. Let $t(i)$ represent the execution time for node (instruction) i, which is assumed to take integral values. For variable-execution-time instructions, $t_{\min}(i)$ and $t_{\max}(i)$ represent the minimum and maximum execution times, respectively, for node i. For the instruction DAG, the critical path is defined as the longest path from the entry node to the exit node, expressed mathematically as

$$t_{cr} \equiv \max_{k} \sum_{i \in \phi_k} t(i)$$

where ϕ_k represents the kth path from the entry node to the exit node. Clearly, t_{cr} represents a lower bound on the execution time of the instruction DAG, regardless of the number of processors that execute it. The *height* of node i is defined as the length of the longest path from the exit node to node i where the orientation, or direction, of the edges are reversed, i.e.,

$$h(i) \equiv \max_{k} \sum_{i \in \pi_k} t(j)$$

where π_k represents the kth path from the exit node to the node i.

For the variable-execution-time instructions in the DAG, the *minimum* and *maximum height* for a node i, $h_{\min}(i)$ and $h_{\max}(i)$, are defined as follows:

$$h_{\min}(i) \equiv \max_{k} \sum_{i \in \pi_k} t_{\min}(j)$$

and

$$h_{\max}(i) \equiv \max_{k} \sum_{i \in \pi_k} t_{\max}(j) \quad .$$

The minimum height corresponds to the height for node i assuming all nodes in the DAG take their minimum execution time, and similarly for the maximum height.

4.2. Node Ordering

The maximum height and minimum height are computed for all nodes. This can be done in $O(n^2)$ time, since the problem reduces to finding longest paths from the exit node to all other nodes [Hu82] (with edge orientations reversed.) The nodes are first sorted into a list in descending order using the maximum height as the key, followed by another sort (on nodes with equal maximum height) in descending order using the minimum height as the key. The complexity of each sort procedure is no worse than $O(n\log_2 n)$ [AhHU74].

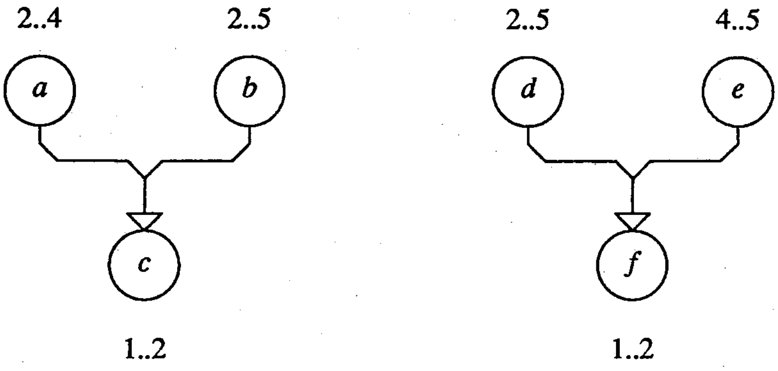


Figure 12: DAG Height Examples.

The maximum height is employed as a key first in an attempt to minimize the worst-case execution time, i.e., when all instructions take maximum time. The minimum height is used to break ties as it represents, in some sense, an attempt to optimize for the best case. For example, in figure 12, for the DAG on the left, $h_{\max}(b) > h_{\max}(a)$, so node b is placed ahead of node a in the list. In the DAG on the right $h_{\max}(d) = h_{\max}(e)$, and $h_{\min}$ must break the tie. Since $h_{\min}(e) > h_{\min}(d)$, node e is given priority in the list.

4.3. Node Assignment

During this phase, the nodes are removed (in order) from the sorted list and assigned to particular processors. Some nodes should be placed in a processor that includes a predecessor (producer) for that node. This *serialization* of the nodes increases efficiency because it reduces the number of processors required and may eliminate a run-time synchronization operation. On the other hand, too much serialization can increase the schedule length. The node assignment algorithm attempts to strike a balance between these two competing aims.

The current node being scheduled is referred to as node *i*. The processor to which some node *j* is assigned is denoted as *Processor(j)*.

[1] The first step in node assignment is to determine the set of processors in which the predecessors of node *i*, denoted as *Preds(i)*, are scheduled. These are referred to as the *producer processors* for *i*, or *ProdProc(i)*, and are computed as follows:

```
proc ProdProc(i)
    for each node j in Preds(i)
        add Processor(j) to the set of producer processors
```

For each processor in *ProdProc(i)*, determine if no other nodes are scheduled after *Pred(i)* on that processor. If a single producer processor meets this condition, place node *i* in that processor, and insert a barrier if necessary, as described later in this section. If more than one producer processor meets this condition, assign node *i* to the producer processor with the largest current maximum time (to possibly avoid inserting a barrier.) If all processors in *ProdProc(i)* have the same current maximum time, choose one at random and assign *i* to it.

[2] If none of the processors satisfy the condition in Step 1, assign node *i* to a processor such that it is scheduled as early as possible. In case of ties between processors, choose one at random. This helps balance the number of nodes assigned to each processor. Insert a barrier as necessary.

4.4. Barrier Insertion

Two algorithms for barrier insertion are described. The first algorithm is conservative in that it always adds a barrier synchronization when one is necessary, but it may add unnecessary, redundant barriers. The other barrier insertion algorithm is ''optimal'' in the sense that a barrier is not inserted unless it is absolutely necessary. The barrier dag $(B\ ,\ <_b)$ is constructed incrementally as the nodes are assigned to processors and barriers inserted into the schedule. It embodies the precedence constraints among the barriers, as described in section 3.1.

The notion of one barrier ''dominating'' another is useful in constructing the barrier dag [AhSU86]. A barrier *x* *dominates* barrier *y*, written *x dom y*, if every path from the initial node of the barrier dag to *y* goes through *x*. With this definition, the initial barrier dominates all other barriers in the dag and every barrier dominates itself.

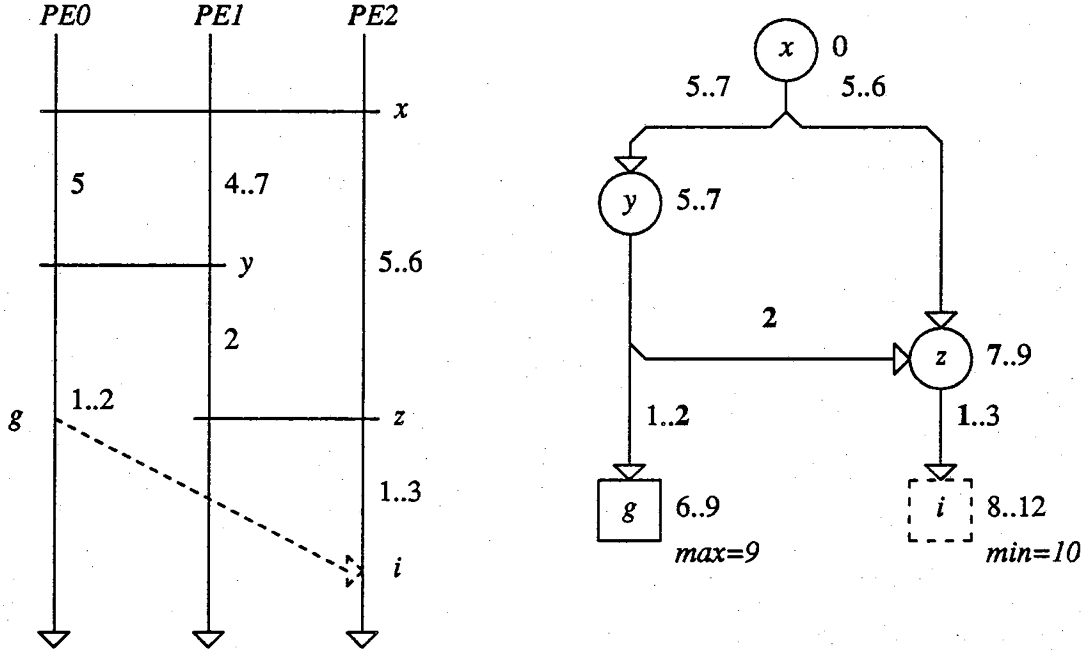


Figure 13: Barrier Embedding where Conservative Algorithm Fails.

Each edge (u,v) between barriers *u* and *v* in a barrier dag contains the minimum and maximum execution time for the code between the barriers. Note that the minimum execution time for edge (u,v) is actually the maximum of the minimum times for all code regions between *u* and *v*. For example, in figure 13, the minimum execution time of the code between barriers *x* and *y* is five, not four; recall that no processor proceeds past the barrier until all have arrived there. This constraint means that the even if *PE1* (processing element 1) executes the code between barriers *x* and *y* in 4 time units, the barrier would still need to wait for *PE0*, which requires 5 time units. The maximum time for the edge (*x*,*y*) is 7 units.

After consumer node *i* has been assigned to a processor *C*, it is necessary to check all producers for *i* to determine if a barrier is necessary. Suppose that node *g* is a member of the set *Preds(i)*, the predecessors, or producers, for instruction *i*, and that it is assigned to processor *P*.

4.4.1. Conservative Barrier Insertion

To determine if a barrier is needed between instruction nodes *i* and *g*, assigned to processors *C* and *P*, respectively, the following steps are performed:

[1] Define *LastBar(g)* as the last barrier to execute before node *g*. Define *NextBar(g)* to be the next barrier to execute after node *g*. This step determines if a barrier or chain of barriers already exists between nodes *NextBar(g)* and *LastBar(i)*. This would guarantee that node *g* executes before node *i*. This step consists of checking for a path between *NextBar(g)* and *LastBar(i)*. Let us call this procedure *PathFind(g,i)*[4]. If a path is found, no barrier is needed; otherwise, continue with step [2].

[2] In this step, the static timing constraints inherent in the barrier dag are examined to check if these constraints resolve the synchronization statically. First, the nearest common dominating barrier for barriers *LastBar(g)* and *LastBar(i)*, written as *CommonDom(g,i)* is found. The dominator information can be stored in a *dominator tree*. The initial node is the root of the dominator tree, and a node dominates only its descendants in the tree. Hence, the common dominator *CommonDom(g,i)* is the nearest common ancestor in the dominator tree. It is the last common synchronization point for processors *P* and *C*.

[3] From this common dominator barrier, timing information is propagated up to the producer and consumer instructions *g* and *i*. The length of the longest path, assuming maximum execution times for code regions between barriers, from *CommonDom(g,i)* to *LastBar(g)* is computed. Denote this longest path as $\psi_{\max}(CommonDom(g,i),LastBar(g))$, and its length as $l(\psi_{\max}(CommonDom(g,i),LastBar(g)))$. Add the maximum time necessary to execute all instructions after *LastBar(g)* up to and including *g*, denoted as $\delta_{\max}(g)$, to this length to yield the maximum time $T_{\max}(g)$ to execute node *g* relative to the common dominating barrier.

[4] Similarly, the longest path, assuming minimum execution times for code regions, from the common dominator to *LastBar(i)* is computed. Denote this longest path as $\psi_{\min}(CommonDom(g,i),LastBar(i))$, and its length as $l(\psi_{\min}(CommonDom(g,i),LastBar(i)))$. Let i^- represent the instruction before *i* on processor *C*. The minimum time to execute all instructions up to but *not* including *i*, denoted as $\delta_{\min}(i^-)$, is added to this length, yielding the minimum time $T_{\min}(i^-)$ to start executing node *i* relative to the common dominator.

[5] If $T_{\min}(i^-) \geq T_{\max}(g)$, then no barrier is needed; otherwise, go to step [6].

[6] A barrier is inserted across processor P somewhere after the producer node g, and across processor C just before node i. To determine where the barrier is placed on P, we compute $l_{\max}(CommonDom(g,i), LastBar(i))$, and then add in the maximum execution times of all instructions on C after *LastBar(i)* up to node i^-, yielding $T_{\max}(i^-)$. If $T_{\max}(i^-) \leq T_{\max}(g)$, then the barrier is inserted right after the producer node g on processor P. However, if $T_{\max}(i^-) > T_{\max}(g)$, and if there is some instruction g^+ after instruction g, such that $T_{\max}(i^-)$ falls into the execution time range (assuming maximum times) of g^+, then the barrier is inserted after g^+ on processor P. This approach allows the producer processor to execute a bit more work after the producer instruction g.

This barrier insertion algorithm will sometimes add unnecessary barriers. For example, in the barrier embedding given in figure 13, the conservative insertion algorithm will insert a barrier across processors 0 and 2, after the producer node g and before consumer node i. In the figure, *LastBar(g)* is y and *LastBar(i)* is z. The common dominating barrier for y and z is x. It can be seen that $T_{\max}(g) = 9$, while $T_{\min}(i^-) = 8$, so it would appear that a barrier is necessary.

However, the longest path from x to z, $\psi_{\min}(x,z)$, overlaps with the longest path from x to y, $\psi_{\max}(x,y)$, on edge (x,y); recall that different assumptions have been made about the execution time for this edge on the different paths. If this is taken into account, then $\psi_{\min}(x,z)$ should be computed, as before, assuming minimum execution times for edges *except* for the edges which intersect $\psi_{\max}(x,y)$. For these edges, use their maximum execution time when computing the longest path. In figure 13, this means that edge (x,y) has value 7, (y,z) has value 2, and the minimum time for i^- is 1, yielding an actual minimum time for node i of 10. Thus, i always executes after g and no barrier is required. In the next section, an optimal barrier insertion algorithm that does not generate these unnecessary barriers is briefly discussed.

4.4.2. Optimal Barrier Insertion

From the previous example, it is clear that the problem with the conservative insertion algorithm is that it does not take into account the possibility that the longest paths from the common dominator to the producer and consumer nodes may overlap. In such cases, assuming maximum execution times on edges that overlap may increase the minimum execution time for the consumer node just enough to resolve the synchronization statically.

But resolving the synchronization is not quite that simple. The "second" longest path (to the producer node) must also be checked: if the execution time on this path is also greater than the consumer node maximum execution time, then the same check for path overlap and resulting timing adjustments must be made. This process continues for the decreasing longest paths to the producer node until the length of the *i*th longest path to the producer is less than the longest path to the consumer node (assuming minimum execution times.)

Let u be the nearest common dominator for barriers v and w, where v is *LastBar(g)* and w is *LastBar(i)*. Recall that node i is being scheduled, node g is a producer for node i, and it is necessary to determine if a barrier is required between these instructions.

The relationship between the various path lengths can be expressed as

$$l(\psi_{\max}(u,v)) \geq l(\psi^2_{\max}(u,v)) \geq l(\psi^3_{\max}(u,v)) \geq \cdots$$
$$\cdots \geq l(\psi^{k-1}_{\max}(u,v)) \geq l(\psi_{\min}(u,w)) + (\delta_{\min}(i^-) - \delta_{\max}(g)) \geq l(\psi^k_{\max}(u,v)) \;.$$

where $\psi^j_{\max}(u,v)$, $2 \leq j \leq k$ represents the kth longest path (assuming maximum execution times) from u to v. For each $\psi^j_{\max}(u,v)$, find $\psi^*_{\min}(u,w)$, the longest path from u to w assuming minimum execution times *except* for edges on the path $\psi^j_{\max}(u,v)$, where maximum execution time edges are assumed. If the condition

$$l(\psi^j_{\max}(u,v)) + \delta_{\max}(g) \;\leq\; l(\psi^*_{\min}(u,w)) + \delta_{\min}(i^-)$$

is satisfied, consider the next longest path $\psi^{j+1}_{\max}(u,v)$ and repeat the process. If the condition is not met, then a barrier must be inserted as described as described in step [3] of the conservative barrier insertion algorithm, and the scheduling algorithm starts again with the next node in the list.

The process of checking the jth longest path continues until

$$l(\psi^j_{\max}(u,v)) + \delta_{\max}(g) \;\leq\; l(\psi_{\min}(u,w)) + \delta_{\min}(i^-)$$

is met for the jth longest path, where $j = k$, proving that the synchronization is satisfied statically and no barrier need be inserted.

4.4.3. Barrier Merging

An additional merging step is performed when inserting barriers into an SBM schedule. If the execution time range of the new barrier overlaps with any other barriers currently scheduled, and if the overlapping barriers are not ordered with respect to the barrier dag, then they are merged into a single barrier. For one set of benchmarks studied (with 10 variables and 80 statements) the merging of barriers resulted in 35% fewer barriers in the resulting schedules. The static scheduling fraction also increased as a result of the larger barriers in the SBM schedule. The merging of barriers increased the completion time for the SBM compared to the DBM, although these times are quite close for the benchmarks studied. More scheduling results are given in the next section.

5. Scheduling Experiments

The scheduling algorithms discussed in the last section[5] were applied to the synthetic benchmark programs. The effects of varying different parameters that are related to the architecture of the machine

[5] Although an optimal algorithm was presented for barrier insertion, the conservative algorithm was used for all the scheduling experiments. This was done because the conservative algorithm is much simpler and the results were very good.

and the structure of the synthetic benchmarks have been studied. Architecture parameters that were varied include the number of processors and timing assigned to each instruction; barriers were assumed to always execute immediately upon arrival of the last participating processor. Benchmark parameters included the number of instructions and variables in generated programs. Particular attention has been paid to the different synchronization fractions and how they vary as the parameters change. These results have provided good feedback concerning the performance of the scheduling algorithms.

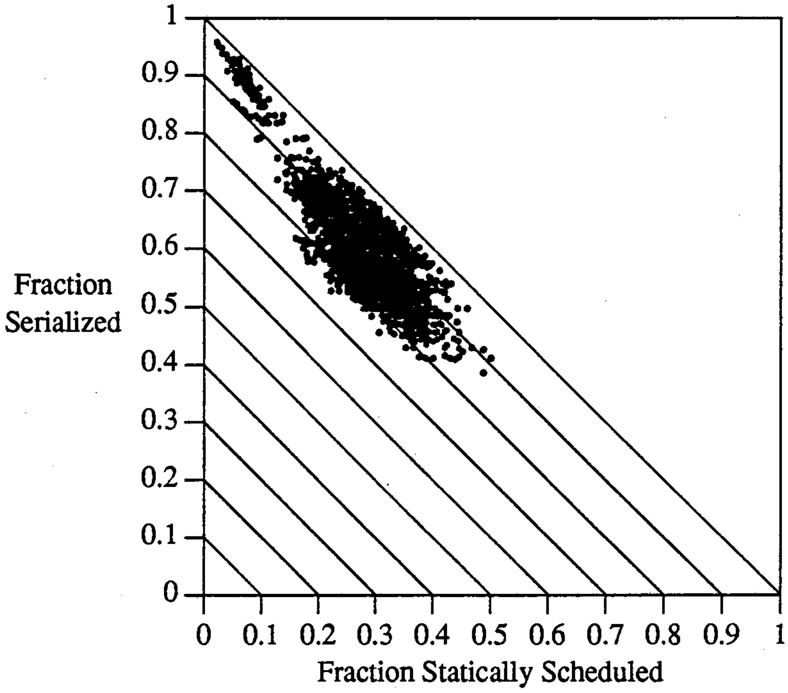


Figure 14: Scatter Plot (Benchmarks Contain From 65 to 132 Syncs.)

One-hundred synthetic benchmarks were generated for each set of parameters and the results averaged to yield each point on the curves shown in this section. Over 3500 benchmarks were generated. Over all these programs, the results fell into the following ranges:

- The barrier fraction varies from 3% to 23%
- The serialization fraction varies from 50% to 90%
- The fraction of barriers statically scheduled away varies from 8% to 40%.

Note that the last fraction represents a feature unique to barrier architectures. A scatter plot with the serialization fraction on the vertical axis and the statically scheduled fraction on the horizontal axis is shown in figure 14. The results for more than 2000 of the synthetic benchmarks are given in the figure, and it can be seen that the ''center of mass'' of the points lies near the 85% line; hence, about 85% of the synchronizations are either serialized or statically scheduled away.

5.1. Number of Assignment Statements

In this section, the number of processors and variables are fixed (8 processors and 15 variables), while the number of instructions varies. This section investigates the effect of changing the number of instructions in the synthetic benchmarks.

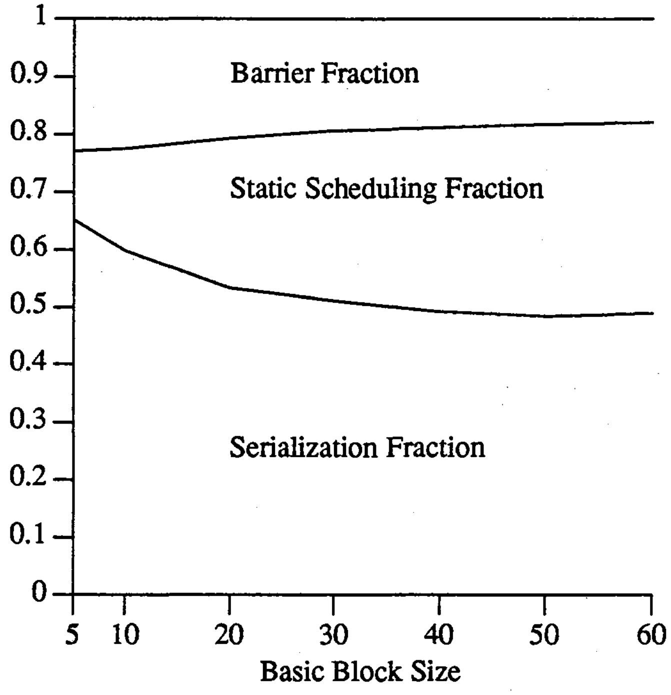


Figure 15: Sync. Frac. for 8 Processors (PEs) and 15 Variables (Vars.)

As can be seen in figure 15, the fraction of barrier synchronization decreases as the number of statements in the generated basic blocks increases. This decrease is relatively large when the number of statements varies from 5 to 20. Generally speaking, the `Load` operations are scheduled for early execution. Since `Load` is a variable-execution-time instruction (from 1 and 4 time units), barriers are often required after a `Load`. Hence, there is a concentration of barriers at the the beginning of the execution of the basic block. The barrier fraction decreases when we increase the number of statements because the percentage of `Loads` becomes smaller. This effect is counter-balanced at larger benchmark sizes because `Mul`, `Div`, and `Mod` instructions begin appearing in generated benchmarks, and these instructions have large execution time variation.

Notice that the serialization fraction decreases as the benchmark size increases. Larger benchmarks make it less likely that a consumer can be placed directly after a producer because another instruction has probably been scheduled there already.

5.2. Number of Variables

In this section, the benchmark size and number of processors are fixed (60 statements and 8 processors), while number of variables is changed. Referring to figure 16, as the number of variables increases, the barrier fraction first increases, then remains constant. Since the number of variables corresponds closely to the parallelism width of the benchmarks, the parallelism width increases with the number of variables, and the scheduling algorithm employs more processors in the schedule. The result is that more barriers are generated until the parallelism width of the benchmark exceeds the number of processors, and the barrier fraction then becomes constant.

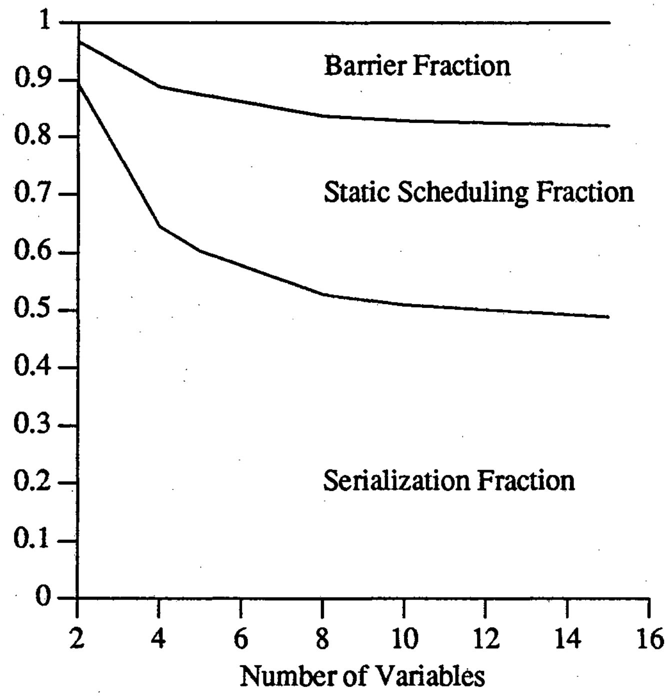


Figure 16: Sync Fractions for 8 PEs and 60 Instructions (Instrs.)

The serialization fraction decreases when more variables are used because for a large number of variables, the parallelism width is large, and fewer opportunities for serialization exist.

5.3. Number of Processors

In this section, the benchmark size and number of variables are fixed, while the number of processors is varied. For two variables, increasing the number of processors did not affect the barrier fraction because the scheduling algorithm keeps almost all of the instructions in two processors. The serialization and static scheduling fractions also remain constant.

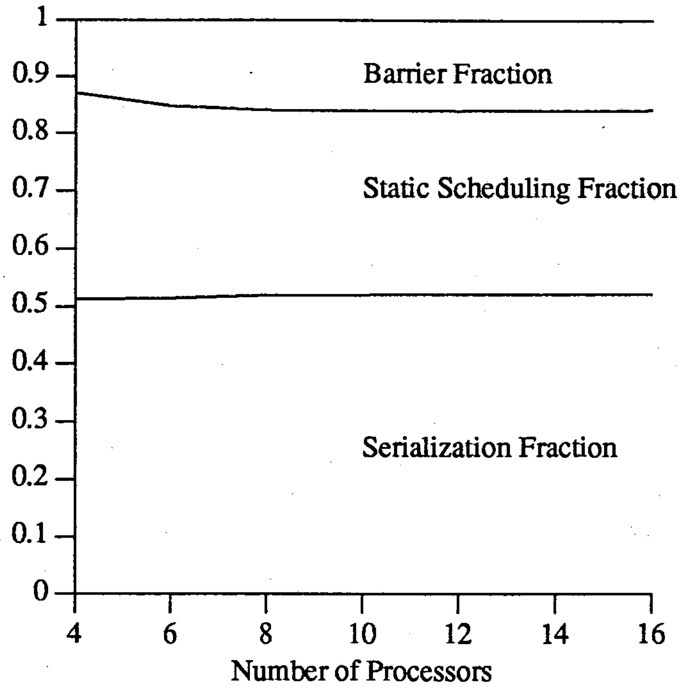


Figure 17: Sync Fractions for 100 Instrs. and 10 Vars.

For five variables, when the number of processors is increased from two to four the barrier fraction increases because more synchronization is required between the different processors. The barrier fraction stabilizes after four processors as the scheduling algorithm employs no more than four processors. For *N* variables, the barrier fraction increases when the number of processors employed is less than the parallel width. When the number of processors used is greater than the parallelism width, then the barrier fraction remains constant.

Figure 17 illustrates the effect of increasing the number of processors on the different synchronization for a barrier machine. this figure is for 100-assignment statement basic blocks with 10 variables.

The serialization fraction remained nearly constant as the number of processors increased. There are two effects contributing to this serialization rate behavior that tend to cancel each other out, resulting in a fixed serialization fraction. The first effect is that for small numbers of processors, the scheduling algorithm often inadvertently serializes a synchronization operation. Conversely, the scheduling algorithm has fewer opportunities to serialize for a barrier machine with a small number of processors, because quite often the "serialization slot" will already be filled. The first effect results in an increase in the serialization fraction for larger numbers of processors while the second effect yields an decrease in the serialization fraction.

5.4. Analysis of the Heuristics

The heuristics employed in code scheduling were analyzed by first modifying them in some way and then scheduling the same synthetic benchmarks on the original and new version of the heuristic. Various characteristics of the resulting schedules, including execution time and synchronization fractions, were compared to gain insights into the performance of each heuristic.

The node assignment step was modified to perform simple *round-robin* scheduling, where the *i*th node in the sorted list is scheduled on processor (i modulo N), where N is the number of processors. As expected, the serialization fraction nearly vanished for large numbers of processors; the barrier fraction also increased significantly, in some cases reaching 50%. Both the minimum and maximum execution times increased, although the execution time difference between list scheduling and pure *round-robin* became smaller for larger numbers of processors. These results suggest that the node assignment heuristics employed in the scheduling algorithm were reasonable.

Another change to the node assignment heuristic switched the ordering priority: the minimum height $h_{\min}$ was employed first in the sort of the instruction nodes, followed by the maximum height to break ties. As expected, the minimum execution time of the benchmarks decreased while the maximum time increased. This is not surprising as the new ordering, in some sense, attempts to optimize the minimum execution time. However, the changes were quite small.

An attempt was made to increase the serialization rate by performing *lookahead* on the sorted instruction list. Instructions in a window of size p on the top of the list were examined before a node was assigned to a processor to see if such an assignment would preclude a later *serialization* assignment. The effects were interesting: the serialization fraction increased as expected, but not very much for large numbers of processors, as the scheduling algorithm keeps the serial streams on a single path without lookahead. For small numbers of processors, the execution time increased from 10% to 30% due to an increase in the critical path length brought on by the additional serializations. This increase disappears for a large number of processors.

Another parameter of interest is the timing variation of the instructions. Synthetic programs were generated with instructions having very large timing variations. The results showed that the barrier sync fraction was not very sensitive to increases in instruction timing variation, increasing only slightly for large variations.

6. VLIW vs Barrier Architecture

A comparison between barrier MIMD and VLIW execution of synthetic benchmark programs was undertaken. The results are given in figure 18.

The VLIW execution mode used in scheduling the instructions assumed that all instructions required their maximum time to execute. No asynchrony was allowed in VLIW execution. As can be seen in the figure, the maximum times for both the barrier MIMD and VLIW were nearly identical. Execution time as displayed in figure 18 has been normalized to VLIW execution time. The barrier machine

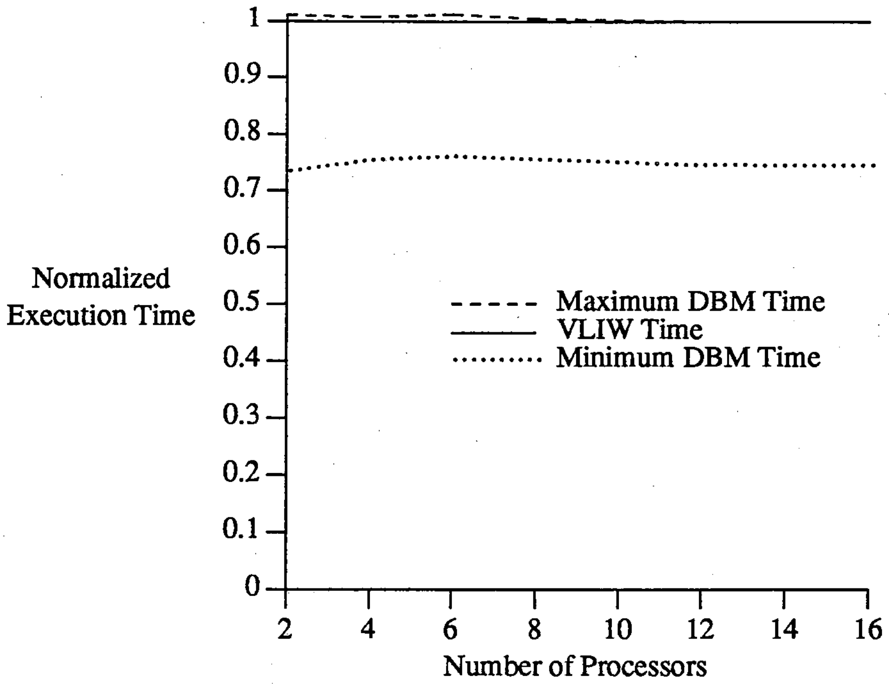


Figure 18: VLIW vs Barrier Architecture (60 Instrs. and 10 Vars.)

took slightly longer to complete execution for smaller numbers of processors because more barriers were required due to instruction execution variation.

The minimum barrier MIMD completion time was about 25% lower than the VLIW completion time, and average barrier completion time will fall between the minimum and maximum times, the exact value being determined by the probability distributions of the variable-execution-time instructions. It should be noted that an optimal schedule (completion time equal to the critical path time) was determined for almost all the synthetic benchmarks for the comparison.

These results suggest that it is important to reduce the maximum execution time for instructions, and this is traditionally done in VLIWs by adding extra hardware for such operations as integer and floating-point multiplies. Also, memory reference delays are reduced by having multiple paths to memory banks and through careful scheduling of memory references to avoid bank conflicts [CNOPR88].

7. Conclusions

VLIW architectures have proven to be capable of providing consistently good performance over a larger range of programs than vector processors; the barrier MIMD architectures hold great promise for extending this range even further. Whereas VLIW architectures cannot achieve efficient parallel execution of `while` loops, subroutine calls, and variable-execution-time instructions, barrier MIMDs provide a hardware mechanism which allows VLIW-like static scheduling to be applied to all these constructs.

Barrier MIMD hardware has been shown to be easily and efficiently implementable [OKDi90]; hence, the prime question is whether a reasonable approximate solution to the NP-complete problem of static code scheduling for such machines can be found. In this paper, we have presented a set of algorithms which efficiently implements this code scheduling. Further, we have shown the proposed algorithms to perform very well, citing the results from applying these algorithms to over 3500 synthetic benchmark programs.

Ongoing work includes the extension of the basic scheduling techniques to more complex code structures (including arbitrary control flow) and the possible application of the barrier scheduling techniques to remove some synchronizations in conventional MIMD architectures. We are also working toward a complete machine design, the CARP (Compiler-oriented Architecture Research at Purdue) machine [DSCO89], that incorporates the barrier MIMD mechanism as well as various other novel compiler/architecture interactions.

References

[AhoSU86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[AhHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

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

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

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

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

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

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

[DSCO89] H.G. Dietz, H.J. Siegel, W.E. Cohen, M.T. O'Keefe, *et al.*, "A Compiler-Oriented Architecture: The CARP Machine," *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.

[Elli85] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: MIT Press, 1985.

[Hu82] T. C. Hu, *Combinatorial Algorithms*. Addison-Wesley: Reading, MA, 1982.

[KaNa84] H. Kasahara and S. Narita, ''Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing,'' *IEEE Trans. on Computers*, vol. C-33, no. 11, pp. 1023-1029, November 1984.

[OKee90] M. T. O'Keefe, *Barrier MIMD Architectures: Performance Analysis, Design, and Compilation*, Ph.D. dissertation, in preparation. School of Electrical Engineering, Purdue University, Spring 1990.

[OKDi90] M. T. O'Keefe and H. G. Dietz, ''Hardware Barrier Synchronization,'' submitted to *1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[ScNa87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, ''The Implementation of the PASM Prototype Control Hierarchy,'' *Proceedings of the Second International Conference on Supercomputing*, vol. I, 1987, pp. 418-427.

[Shaf89] P.L. Shaffer, ''Minimization of Interprocessor Synchronization in Multiprocessors with Shared and Private Memory,'' *Proc. 1989 Int. Conf. Parallel Processing*, vol. III, pp. 138-142, Aug. 1989, St. Charles, IL.