Percolation Scheduling for Non-VLIW Machines

Technical Report 90-02

Carrie J. Brownhill and Alexandru Nicolau
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

January 15, 1990

Abstract

Percolation Scheduling, a technique for compile-time code parallelization, has proven very successful for exploiting fine-grain irregular parallelism in ordinary programs. Currently, this technology is targeted only to VLIW (Very Long Instruction Word) machines, which have the advantages of 'free' synchronization and communication. Shared memory multi-processors can simulate the execution characteristics of VLIW machines with the use of static barriers. Preliminary results show that Percolation Scheduling can be used with good results on this type of architecture by increasing the granularity from operation level to source statement level, removing any redundant synchronization, and providing an efficient implementation of multi-way jumps.

Keywords - Static code transformations, parallelizing compilers, barrier synchronization, VLIW, MIMD.

1 Introduction

Static code transformation, such as Percolation Scheduling, has been shown to be very successful at parallelizing code at a fine-grain (machine operation) level [6, 7]. This type of parallelization has been targeted to VLIW (Very Long Instruction Word) machines. On VLIW machines, synchronization is built into the architecture, and is essentially free. There is also no communication cost, because all memory is accessable by all operations. In the past, the relatively high cost of communication and synchronization between processes on MIMD (Multiple Instruction Multiple Data) architectures has excluded the exploitation of fine-grain parallelism. Most research has therefore been concentrated on higher level parallelization [1, 8, 9]. However, the use of shared memory, and fast hardware barriers provides the opportunity to use VLIW compiler technology on traditional MIMD architectures.

In [2], Dietz, et. al. has proposed the use of hardware barriers on MIMD machines to enforce the same synchronization that VLIW machines provide. In VLIW machines, parallelism occurs at the machine operation level. Multiple operations and operands are explicitly coded into the same instruction. The next instruction does not begin execution until all of the operations in the previous instruction have completed. To give an MIMD operational characteristics similar to those of a VLIW machine, three things are necessary. First, some type of fast communication is needed. This is easily implemented using shared memory. Second, VLIW-like synchronization is necessary. In order to achieve this across multiple asynchronous processors, barriers must be used. A barrier causes each processor to stop and wait until all processors reach the barrier, then all processors are allowed to proceed. Third, an efficient implementation of multi-way jumps must be available. Solutions to fill this need are discussed later in this paper.

Current VLIW compilers parallelize code at the machine operation level. This is practical for VLIW machines because they are synchronized after every operation. Even with very fast barriers, the cost of synchronization after each operation on an MIMD machine is likely to be too high. However, Percolation Scheduling can be used to exploit irregular parallelism on MIMD machines, if two adaptations are made to reduce the amount of synchronization necessary. First, Percolation Scheduling is modified to provide source statement level parallelism instead of operation level parallelism. This means that synchronization is required between

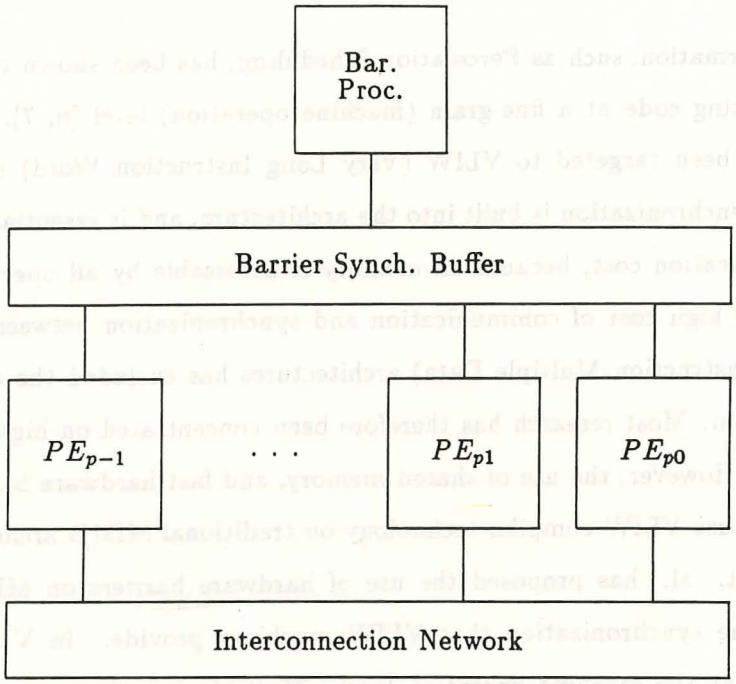


Figure 1: Barrier MIMD (Dietz)

statements, instead of between operations. Second, barriers which are redundant due to previous synchronization are removed. Preliminary results indicate that, with these changes, Percolation Scheduling can be used to exploit irregular parallelism on MIMD machines.

2 Hardware Barriers

Fast barriers are essential for making Static Barrier Machines (SBM) feasible. Dietz has proposed using a specialized barrier processor and barrier synchronization buffer. This design has the benefit of allowing the barrier mechanism to be used on subsets of processors. The barrier processor generates a barrier pattern, or mask, indicating which processors will participate in the barrier [Figure 1].

A simpler hardware barrier can be implemented by providing each processor with a dedicated one-bit register. When a processor reaches a barrier, it sets its register to one. The

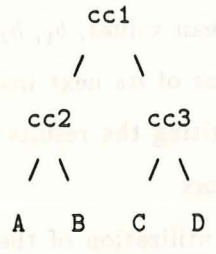


Figure 2: Program Execution Path With Multiple Branch Tests

registers are all ANDed together to produce a *result bit* that all of the processors can read. When the *result bit* turns to one, all the processors proceed. The *result bit* must be stored until all the processors have exited the barrier, and then it can be reset. This implementation involves a small amount of additional hardware, but is efficient and has been used in several high performance multi-processors.

3 Multi-way Jumps

Along with 'free' synchronization and communication, VLIW machines, such as the IBM VLIW machine [3], have another notable feature. They allow multiple branch-tests to be executed in parallel. For example, three conditionals, cc_1, cc_2, and cc_3, nested as in Figure 2, can be scheduled in the same instruction. The result is a multi-way jump. In one instruction, the conditionals are evaluated and the address of the next instruction is determined.

In contrast, on an MIMD machine, the architecture does not support this type of operation. Therefore, the parallelization of conditional tests is not straight-forward. In order to make MIMD machines truly compatible with VLIW compiler technology, it is necessary to have an efficient way of performing multi-way jumps.

One solution would be to execute any conditional branches sequentially on one or all processors. Clearly, this would reduce the effective parallelism. Studies on actual programs have shown the number of IF statements ranging from 10% in FORTRAN programs to 43% in C programs[11]. Performing these statements sequentially would produce a severe bottleneck in the parallel execution of the program.

On the other hand, if cc_1, cc_2, and cc_3 are each evaluated in parallel on separate processors, then the results of the tests, a set of boolean values, b_1, b_2, and b_3, must be read by each processor in order to determine the address of its next instruction. On a shared memory machine, this is easily accomplished by writing the results of each test to shared memory, where it can be accessed by all the processors.

As with the barriers, the storage and utilization of the set of boolean values must be fast, if the parallelization of the tests is to result in a speedup. If there are P processors available, then there will be up to P boolean results to store. If each result is stored in a separate location, then the processors can write their results concurrently without conflict. The storage will take constant time (assuming that the bus isn't saturated). However, once they are stored, then the P booleans must be read and tested in order to determine the execution path. This, in effect, is recreating the sequential execution of the conditionals. In the best case, this is $O(log(N))$ where N is the number of conditionals. The execution time saved by executing the conditionals in parallel may easily be spent writing and testing the results.

It is more efficient to store each boolean result as a bit in a *results word.* If P is less than the word length, only one *results* word is required. Each processor sets a predetermined bit, then the whole word can be use as an index into a lookup or jump table to determine the next instructions. Each processor can have its own table of addresses. The lookup is constant, but now the storage time must be reconsidered. On most machines, only one processor may write the *results word* at a time. As a result, the processors have to wait for exclusive control of the *results word.* In the worst case, the processors will all queue up, and the storage time is proportional to P. Since the results cannot be used until all of the conditionals are written, all of the processors must wait for the slowest one to finish.

Parallel reduction can be used, to combine results together in binary tree fashion, reducing the storage to only $O(log(N))$ time [5]. However, if the number of processors is smaller than the number of addressable memory units per word, then a simple trick can be used. Each processor is allocated a byte (or smallest addressable unit) in the *results word* (or double word). The processors can concurrently write their result bit in their own byte, then the consecutive bytes can be read together as an integer. See Figure 3. This solution takes

```
/* iam: the number of this processor                          */
/* results_buffer: Word to buffer results                     */
/* results: Word to store results of conditionals             */
/*      results is declared as a union type, or variant       */
/*      record with one variant (result_byte) being an array */
/*      of bytes and the other (result_word) being a word     */

if (conditional)
results.result_byte[iam] = 1;
Wait (); /* Make sure all bits written */
if (iam == 0) /* This code only done by one processor */
{
results_buffer = results.result_word;  /* Results buffered */
results.result_word = 0;
}
Wait ();                                  /* This barrier is needed to make */
                                          /* sure that housekeeping is done */
/* results_buffer can now be read */
```

Figure 3: Parallel Conditional Tests on Exclusive Write Machines Using Byte Addressing

constant storage time and constant lookup time. Unfortunately, the number of processors is severely limited, depending on the number of addressable memory units per word.

4 Setbit Instruction and Implementation

With the addition of a small amount of hardware, an $O(1)$ solution for a larger number of processors can be implemented. The number of processors is limited to the number of bits in a word. The idea is similar to the hardwired barrier, in that each processor has a dedicated one bit register. When the processor executes a conditional test, it writes the result to its

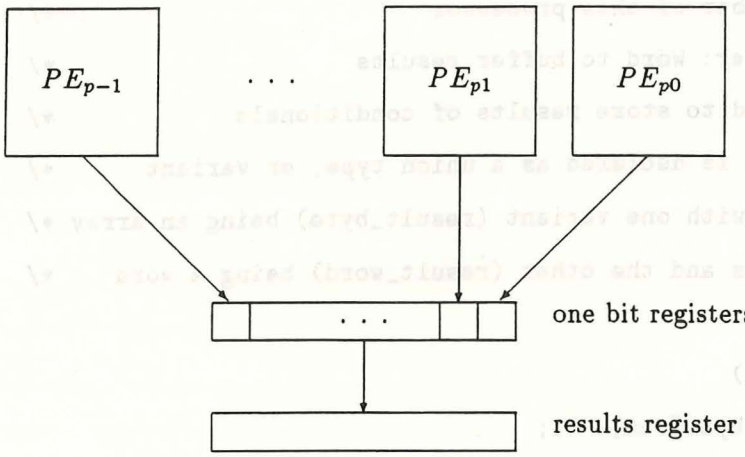


Figure 4: Registers for Setbit Instruction

```
Setbit (conditional);
Wait ();                                /* Make sure all bits written */
                                        /* Results buffered automatically */
/* results_buffer can now be read */
```

Figure 5: Parallel Conditional Tests on Using Setbit Instruction

register and proceeds to the next barrier. The other processors behave similarly, writing to their own registers. When all of the processors reach the barrier, then the contents of the one bit registers are moved to a *results register*. If each processor has an associated number P, ranging from zero to the (number of processors - 1), then the contents of the one bit register of processor P is moved into the P^{th} bit of the *results register* [Figure 4]. The one bit registers are then reset, and the processors are allowed to proceed. They can then read (test) the *results register* concurrently. Alternatively, if two sets of one bit registers are available, the processors can take turns between them, writing one set while reading the other.

The processor calls the instruction *Setbit(boolean_value)* to set its one bit register to *boolean_value* [Figure 5].

The one bit registers must be stored and reset while the processes are waiting at the

barrier. Therefore, the implementation of the Setbit instruction must be based on the implementation of the barrier, if it is to be efficient. If a hardware barrier using dedicated registers and AND gates is used, then microcode is used to preform register moves and loads. If a separate barrier processor and bus are used, then the barrier processor can perform the housekeeping.

5 Statement Level Percolation Scheduling

Percolation scheduling used for fine-grain, automatic parallelization gives good results for tightly-coupled machine models like VLIW[6, 7]. In these models, synchronization is essentially free. Multiple operations can be scheduled to execute in parallel, with all data reads occurring before all data writes of the same variable. However, for loosely coupled machine models, operations on separate processors cannot be guaranteed to occur in a particular order without explicit synchronization. Even with efficient barriers, it is worthwhile reducing synchronization costs by reducing the number of barriers required. This can be done by adapting Percolation Scheduling to work on medium-grain, statement level transformations, instead of the usual operation level transformations.

To define the statement level transformations, we start with a program graph. Initially, each node in the program graph contains one high level language statement. A statement is treated as an indivisible unit, and can be an assignment statement, procedure call, loop, or a branch statement. A branch statement is the conditional test of an IF or CASE statement. The arcs in the program graph represent execution paths. If a node contains one or more branch statements, then there may be a set of exit paths pointing to the nodes which may be executed next. The path taken depends on the runtime results of the conditional tests.

Formally, a node is a four-tuple of the form $n =< S_n, test_n, BNext_n, Next_n >$. S_n is a set of statements, $S_n = s_1, \ldots$, such as assignment statements or procedure calls. Each statement s_i contains a set of written variables, a set of read variables and a set of operations. $Test_n$ is a set of tests, $Test_n = t_1, \ldots$. Each test is on a set of read variables. $BNext_n$ is a set of control-flow pointers that correspond to the evaluation of $Test_n$. One member of $BNext_n$ may correspond to all results which are not explicitly enumerated. This allows for an OTHERWISE path. $Next_n$ represents the unconditional continuation of the node, which

is used if $Test_n$ is empty.

5.1 Transformations

The transformations are analogous to the core transformations used in normal Percolation Scheduling, except that they are applied at the statement level instead of the operation level. Correctness of the Percolation Scheduling transformations has previously been proven[6]. Correctness of the statement level transformations can be shown in a similar manner. Statements are 'percolated' up the program flow graph into successively higher nodes. Statements which are in the same node may be executed concurrently.

There are four core transformations. See Appendix A for illustrations of the transformations.

- Move-statement transformation: Statement s is moved from N_2 to N_1. If more than one execution path ($Next_n$) points to s, then a copy of s is made. All $Next_n$ which point to node s, besides the one from N_1, are changed to point to the copy. The copy may later be moved up the program graph, along the other execution paths.

- Move-branch transformation: This transformation is used to move the conditional test in an IF or CASE statement. If more than one $Next_n$ points to the statement, then it is copied, as in the move-statement transformation. In addition, if the branch moves above another node, then that node must be copied and placed on each of the paths exiting the branch.

- Unification transformation: As statements are moved up the program graph, they may be copied. If a statement moves up all the paths of a branch statement, it will block itself from moving past the branch. The unification transformation allows the copies of the statement to be unified into one, so that it can continue moving up the graph.

- Delete-node: If node N has no components, then N may be deleted. All nodes which point to N are changed to point to $Next_n$.

5.2 Execution Semantics

The execution semantics of Percolation Scheduling state that all reads for a particular variable must occur before any writes to that variable in the same node. On VLIW machines, this is built into the execution order of the operations. On asynchronous processors, the order of reads and writes across processors cannot be determined. Therefore, in order to insure program correctness, the following must be true:

- The operations in a statement must be executed in correct sequential order, with all data reads occurring before any writes. This is easily guaranteed by executing each statement on a single processor.
- A node n, may contain multiple statements which read a particular variable, if no statements in n write that variable.
- If one statement in a n writes a variable, then it is the only statement which can read or write that variable. Since the order of execution within a statement is fixed, it is okay to allow a statement to read and write the same variable.
- All statements in the current node must be executed before the execution of the next node starts.

5.3 General Algorithm for Statement Level Transformations

Each node in the program graph initially contains one statement. The flow graph is analyzed to determine which data is written and read at each node. In addition, the liveness (or deadness) of data at each node is stored in the form of a set, D_d , of nodes where the data d is dead. This information must be updated as each transformation occurs.

The transformations are applied to the statements according to the following heuristic: The statement with the longest dependency chain is moved first. It is advanced as far up the graph as possible. If more than one statement has the same length dependency chain, then the statement with the highest probability of being executed is moved first. Alternatively, the selection may be randomly made from those statements having the same length chain.

If the statement being moved is a branch-statement, then the move-branch transformation is repeatedly applied until the statement cannot move any farther up the graph. Otherwise, the move-statement transformation is applied until the statement is stopped because of a data dependency. If the move-statement transformation fails because of a write-live dependency, then the unification transformation is attempted. If it succeeds, then the move-statement transformation is once again applied. If the movement of the statement the node empty, then the delete-node transformation is used to remove the node.

The following preconditions exist for moving statement S from node N_2 to node N_1:

- There exists a path from N_1 to N_2.
- For all data D written in S:
  1. No statement in N_1 writes D.
  2. D is dead at all nodes in N_1.
- For all data D read in S:
  1. No statement in N_1 writes D.

If the preconditions are satisfied, the following things occur:

1. If more than one path points to S, then a copy of $S(S')$ is made. S' is set to point to N_2. All paths pointing to N_2 except the one from N_1 are changed to point to S'. S' is put on the list of nodes to be moved.
2. S is added to N_1.
3. S is deleted from N_2.
4. The dead data list is updated to reflect the changes.

5.4 Removing Redundant Barriers

In his discussion, Dietz has suggested that some synchronization barriers would be redundant because of the synchronization provided by previous barriers. In general, if the processors are

perfectly synchronized when they execute a barrier, and if the shortest and longest possible execution times of each operation are known, then a schedule can be analyzed to determine where synchronization is necessary. In addition, possible drift in clock times of asynchronous processors must be taken into account. This idea can be adapted for use by Percolation Scheduling using the following algorithm.

1. Use PS to get an 'ideal' schedule.
2. In a top-down manner, schedule processors according to resource limits.
3. For each operation scheduled, keep a *last possible finish time* in terms of delta from the last barrier. (Previous last finish time + possible clock drift + longest execution time for operation.)
4. Keep running total for each processor *earliest possible execution time.*
5. When an operation is scheduled, the current *earliest possible execution time* must be greater than the *last possible finish time* tag on all dependent operations since the last synchronization *except* for those scheduled on the same processor.
6. If an operation cannot yet be scheduled because it may conflict, then check to see if another operation may be scheduled first.
7. If no operation can be scheduled, then perform a barrier synchronization.

6 B-tree Application

Since the Setbit instruction provides an efficient way to evaluate conditionals in parallel, it is useful for other applications besides programs compiled using statement level Percolation Scheduling. In particular, decision trees, sorting, or searches involving large amounts of data could use parallelized tests advantageously.

As an example, the use of the Setbit instruction in a B-tree search is demonstrated. B-trees are multi-way trees that are well-suited to applications where large amounts of data are stored on secondary storage devices. They minimize the number of external memory reads. Database applications often use B-tree data structures [10].

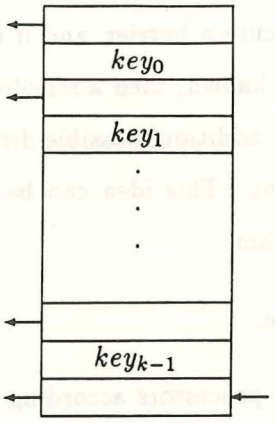


Figure 6: B-tree node with k keys

Each internal node in a B-tree has K ordered keys, and $K + 1$ branches. Each key holds the value of the largest key located down the associated branch [Figure 6]. When the B-tree is searched for a specific value, the values of the keys in the root node are compared to the search key until the appropriate branch is found. Then the node associated with that branch is searched [4].

K is typically chosen either to keep the maximum path length within a desired bound, or to make the size of a node equal to a memory page. Therefore, the K is usually greater than 100. If K happens to be less than the number of processors available (P), then parallelization of the search through the tree is straight-forward. Each processor compares the search key to one of the keys in the node. The result of the comparison is stored using the Setbit instruction. After the barrier is cleared, each processor can read the *result word* and use a switch or jump table to determine the correct branch. The time to search each node is constant. If K is larger than the number of processors, then the algorithm operates on P keys at a time, using an offset into the node [Figure 6].

The speedup produced by parallelizing the conditional tests depends on the number of processors, the execution time of the conditional tests, the time for the Setbit and barrier instructions and the time to lookup the next node address.

The parallelized version will produce speedup if the following relationship holds.

```
/* iam: the number of this processor                          */
/* btree: an array of records                                 */
/*    each record contains a branch field and a value field   */
/* offset: indicates the current search point into the node    */
do
{   Setbit (btree[offset + iam].value < search_key);
    Wait ();
    switch (results_register) ,
    {   case 0:
            correct_branch = offset;
            break;
        case 0x00000001:
            correct_branch = offset + 1;
            break;
        case 0x00000101:
            correct_branch = offset + 2;
            break;
        case 0x00010101:
            correct_branch = offset + 3;
            break;
        case 0x01010101:
            if (offset < (K - P))
                offset = offset + P;
            else
                correct_branch = offset + P;
    }
}
while (!correct_branch);
next_node = btree.index[correct_branch].branch;
```

Figure 7: Parallel Search of a B-tree Node Using Setbit Instruction

$$\frac{K}{P} * (C + B) < K * C$$

or

$$P > \frac{B}{C} + 1$$

where:

K = Number of keys

P = Processors

C = Time for comparison

B = Time for Setbit plus barrier plus table lookup

7 Examples and Experiment Results

Statement level Percolation Scheduling was tested by scheduling real code selected from a machine tool controller program. Because of the decision-making nature of the program, the code contains a high percentage of conditional branches. In real-time environments, like factory automation, the execution speed of software can be critical. However, the conditional branches present in this code make parallelization by current methods difficult.

The code was written in C and run on a Sequent Symmetry with four processors. The Sequent does not provide any type of fast barriers. Even with the slow software barriers available, one of the Percolation Scheduling tests showed 21% speedup. To our knowledge, no other method of parallelization would have produced any speedup on this code.

We wanted to see what kind of results would be possible if the test code was executed on a machine with reasonably fast barriers, instead of the slow barriers of the Sequent. To do this, we timed the execution of the slow barriers. In order to determine what a reasonable execution time might be, we timed the execution with three instructions substituted for each barrier. This is a generous allocation of time, as a hardware barrier should execute much faster than three instructions. The wasted time was found by subtracting the execution time of the simulated fast barriers from the execution time of the slow barriers. This wasted

time was subtracted from the execution times of the parallelized programs. The resulting execution times are given in Figure 8. The percent speedup is given in parentheses and was calculated using the normal formula.

$$speedup = \frac{sequential - parallel}{sequential} * 100$$

In addition, we wanted to see what speedup was possible if a hardware implementation of Setbit was available. This was done in a similar manner, with the execution time of a hardware implementation determined by timing the execution of two register writes and a register read. This was subtracted from the execution time of the two memory writes and memory read used by the software implementation. In addition, only one barrier is needed to insure correctness with a hardware implementation, instead of the two required for the software implementation. The resulting execution times and speedup percentages are also given in Figure 8.

The B-tree application was also coded and timed. These tests did not involve the use of Percolation Scheduling, just the Setbit instruction. The results were calculated in the same manner as the PS tests.

The results show that with a reasonably fast barrier mechanism, statement level Percolation Scheduling can provide good speedup on code that contains a high percentage of conditional tests. These results were obtained using just four processors. With the software implementation of the Setbit instruction, the speedup ranged from 19 to 54%. The results simulating a hardware implementation of the Setbit instruction, showed speedups of up to 60%.

The B-tree application tests showed approximately 40% speedup with the software implementation, and about 50% with the hardware implementation. These are very good results with just four processors used.

8 Conclusion

Fine-grain irregular parallelism can be exploited using automatic code transformations such as Percolation Scheduling. Currently, this compiler technology is only available when the

	Sequential	Parallel with Byte Addressing and Fast Barriers	Parallel with Hardware Setbit and Fast Barriers
B-tree with float keys k = 100	523078	347195 (34%)	289201 (45%)
B-tree with string keys k = 100	620458	344099 (46%)	286105 (54%)
Percolation Scheduling Test 1	42369	34327 (19%)	27612 (35%)
Percolation Scheduling Test 2	10381	4835 (54%)	4163 (60%)

Figure 8: Execution Times and Speedup for PS and B-tree Tests (ms)

target machine has a VLIW architecture. However, our preliminary results show that it is feasible to use an adapted Percolation Scheduling on MIMD machines which possess shared memory and fast barriers. Percolation scheduling can be adapted for use on such machines by increasing the granularity from machine operation level to source statement level, removing any redundant synchronization and providing an efficient implementation of multi-way jumps.

References

[1] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[2] Hank Dietz, T. Schwederski, M. O'Keefe, and A. Zaafrani. Static synchronization beyond VLIW. In *Proceeding Supercomputing '89*, pages 416–425. ACM, November 1989.

[3] Kemal Ebcioglu and Toshio Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Proceedings of the Second Workshop on Programming Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing. MIT-Press/Pitman, 1989. University of Illinois, Champagne-Urbana.

[4] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Co., Reading, MA, 1973.

[5] Boris D. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. In *1989 International Conference on Parallel Processing*, pages 175–179. The Pennsylvania State University Press, August 1989.

[6] Alexandru Nicolau. Percolation scheduling: A parallel compilation technique. Technical Report TR85-678, Department of Computer Science, Cornell University, 1985.

[7] Alexandru Nicolau. Uniform parallelism exploitation in ordinary programs. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[8] D.A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[9] C. D. Polychronopoulos. Towards auto-scheduling compilers. *The Journal of Supercomputing*, July 1988.

[10] Akira Sekino, Ken Takeuchi, Takenori Makino, et al. Design considerations for an information query computer. In David K. Hsiao, editor, *Advanced Database Machine Architecture*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1983.

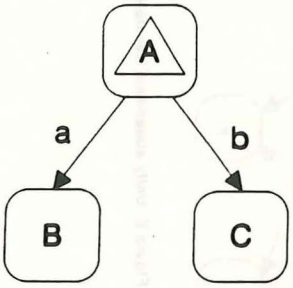
[11] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 3 edition, 1990.

Appendix A

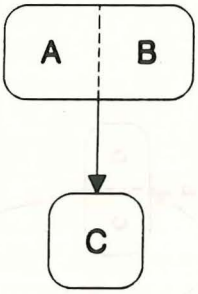
Definitions of Symbols

Statement node - 'A' represents a statement such as an assignment statement.

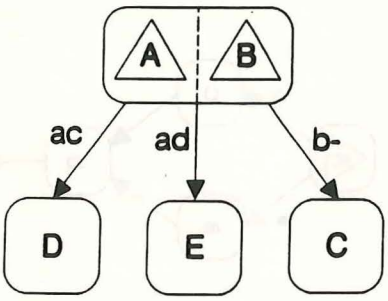
Sequencing arrow - Nodes located before the arrow must finish execution before the nodes after the arrow may begin execution.



Branch node - A branch node causes the program to follow a particular execution path depending on the result of a test. 'A' represents a branch statement, such as an IF/THEN or CASE statement. 'a' and 'b' represent the possible test values.



Tier - A tier contains one or more nodes that may be executed in parallel (indicated by the dotted line). All nodes in a tier must finish execution before the next tier, as indicated by the sequencing arrow, may begin execution.



Multiple branches located in one tier - Each exiting arrow is marked with the combination of branch values which will cause the program to follow that path. The value associated with the leftmost branch in the tier is positioned first. The other values follow in the same order as the branch statements. A hyphen indicates a value of "don't care".

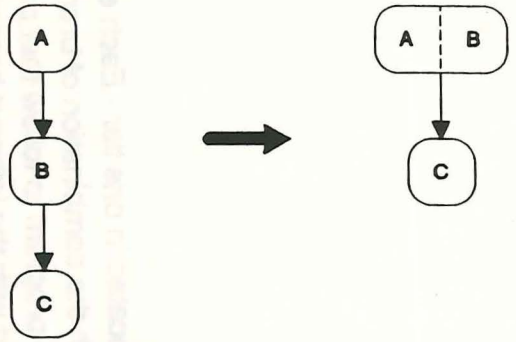


Figure A: Move-statement transformation B not dependent on A.

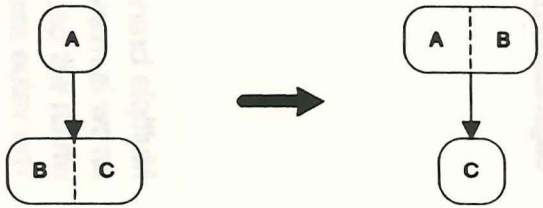


Figure B: Move-statement transformation B not dependent on A.

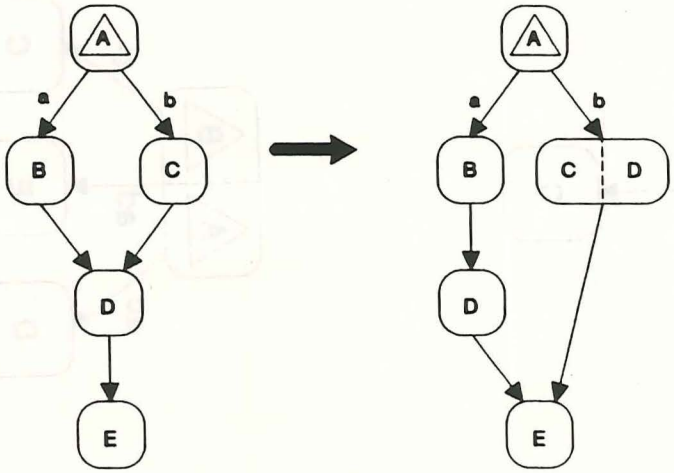


Figure C: Move-statement transformation D not dependent on C.

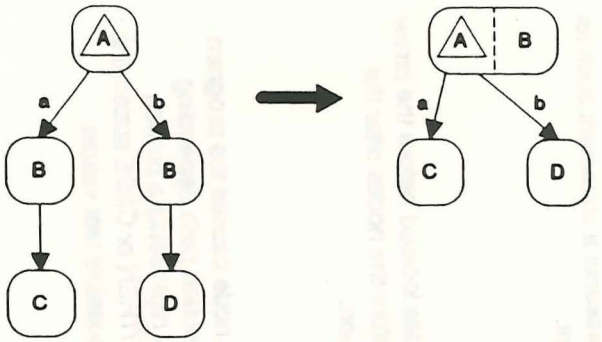


Figure D: Unify-statements transformation B not dependent on A.

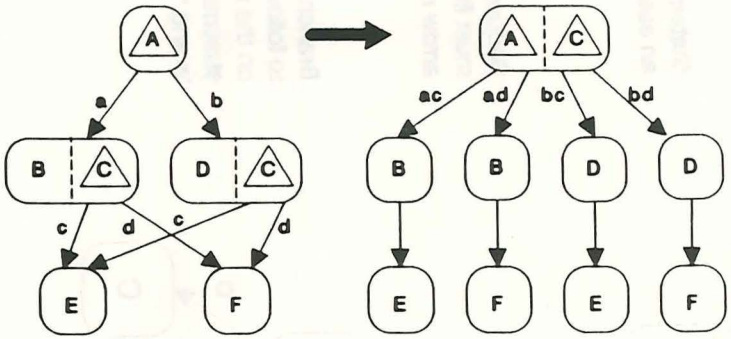


Figure E: Unify-statements transformation C not dependent on A.