Loop Distribution with Arbitrary Control Flow

Ken Kennedy
Kathryn McKinley

CRPC-TR90064
August 1990

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Appears in *The Proceedings of Supercomputing '90.*

Loop Distribution with Arbitrary Control Flow *

Ken Kennedy   Kathryn S. McKinley

Rice University
Department of Computer Science
P.O. Box 1892
Houston, TX 77251

Abstract

Loop distribution is an integral part of transforming a sequential program into a parallel one. It is used extensively in parallelization, vectorization, and memory management. For loops with control flow, previous methods for loop distribution have significant drawbacks. We present a new algorithm for loop distribution in the presence of control flow modeled by a control dependence graph. This algorithm is shown optimal in that it generates the minimum number of new arrays and tests possible. We also present a code generation algorithm that produces code for the resulting program without replicating statements or conditions. Although these algorithms are being developed for use in an interactive parallel programming environment for Fortran, they are very general and can be used in automatic parallelization and vectorization systems.

Keywords: parallelization, vectorization, transformation, control dependence, data dependence, loop distribution

1 Introduction

Loop distribution is a fundamental transformation in program restructuring systems designed to support vector and parallel machines. In its simplest form, loop distribution consists of breaking up a single loop into two or more loops, each of which iterates over a distinct subset of the statements in the body of the original loop. The usefulness of this transformation derives from its ability to convert a large loop whose iterations cannot be run in parallel into multiple loops, many of which can be parallelized. Consider the following code.

```
DO I = 2, N
      A(I) = B(I) + C
      D(I) = A(I-1)*E
ENDDO
```

If we wish to retain the original meaning of this code fragment, the iterations cannot be run in parallel without explicit synchronization lest a value of A(I-1) is fetched before the previous iteration has a chance to store it. However, if the loop is distributed, each of the resulting loops can be run in parallel.

```
DOALL I = 2, N
      A(I) = B(I) + C
ENDDO
DOALL I = 2, N
      D(I) = A(I-1)*E
ENDDO
```

In the presence of conditionals, distribution is complicated. Consider, for example the following loop.

```
DO I = 2, N
      IF (A(I).EQ.0) THEN
            A(I) = B(I) + C
            D(I) = A(I-1)*E
      ENDIF
ENDDO
```

In order to place the first assignment in the first loop and the second assignment in the second loop, the result of the IF statement must be known in both loops. The IF cannot be replicated in both loops, because the first assignment changes the value of A. One solution to this problem is to convert all IF statements to conditional assignment statements, as follows:

```
DO I = 2, N
      P(I) = A(I).EQ.0
      IF (P(I)) A(I) = B(I) + C
      IF (P(I)) D(I) = A(I-1)*E
ENDDO
```

The resulting loop can be distributed by considering only data dependence, because the control dependence has been converted to a data dependence involving the logical array P. This approach, called *if-conversion*

*This research is supported by the National Science Foundation under grants ASC8518578 and CDA8619893, and by the IBM Corporation.

[AKPW83, All83], has been used successfully in a variety of vectorization systems [AK87, SK86, KKLW84]. However, it has several drawbacks. If vectorization fails, it is not easy to reconstruct efficient branching code. In addition, if-conversion may cause significant increases in the code space to hold conditionals.

For these reasons, research in automatic parallelization has concentrated on an alternative approach that uses *control dependences* [FOW87, ABC+87, ABC+88] to model control flow. Reconstructing sequential code from a control dependence graph is not trivial, but it is easier than reconstructing from code that has been subject to if-conversion [FM85, FMS88, CFS90].

Unfortunately, the control dependence representation complicates loop distribution. Although control dependences can be used like data dependences for determining the placement of statements in loops, a problem arises in generating code for a distribution: "How does one generate two loops that have a control dependence between statements in their respective bodies?" The only way to accomplish this is to record the results of evaluating the predicate in a logical array and test the logical array in the second loop.

This paper presents a method for performing loop distribution in the presence of control flow based on control dependences. The approach is optimal in the sense that it introduces the fewest possible new logical arrays and tests. In particular, it introduces one array for each conditional node upon which some node in another loop in the distribution depends. In addition, we show an algorithm for generating code for the body of a loop after distribution. This algorithm, which is intended for use in an interactive program transformation system called ParaScope, generates code that is very close to the original. Although this approach was inspired by the ParaScope transformation system [BKK+89], it is also suitable for use in automatic parallelization and vectorization systems. The algorithms are very fast, both asymptotically and practically.

2 Loop Distribution

Distribution is a program transformation, introduced by Muraoka [Mur71], that converts a single loop into multiple loops. The placement of statements into loops must preserve the data[1] and control dependences of the original loop. We use the following definitions of postdominance and control dependence, which are taken from the literature [FOW87, CFS90].

[1] A data dependence exists between two statements if they reference the same memory location and at least one of them is a write [Ber66, Ban88, AK84, Wol82].

Def: x is *postdominated* by y in the control flow graph G_f if every path from x to the exit node of G_f contains y.

Def: Given two statements x, $y \in G_f$, y is *control dependent* on x if and only if:

1. $\exists$ a non-null path p, $x \rightarrow y$, such that y postdominates every node between x and y on p, and
2. y does not postdominate x.

Based on these definitions, a control dependence graph G_{cd} can be built with the control dependence edges $(x,\ y)_l$ where l is the label of the first edge on path $x \rightarrow y$. Intuitively, control dependence between two statements, x and y, indicates that the source of the control dependence x directly determines whether the sink y will execute.

For our purposes, a node in G_f usually represents a single statement. Exceptions to the single statement per node rule are inner loops and irreducible regions; all of their statements are represented with a single node. If G_f is structured, rooted and acyclic, the resulting G_{cd} is a tree, where structured is as defined by Böhm and Jacopini [BJ66]. Also, if G_f is unstructured, rooted and acyclic, the resulting G_{cd} is a DAG [CFS90].

For the purposes of this paper, loop distribution is separated into a three-stage process: (1) the statements in the loop body are partitioned into groups to be placed in different output loops; (2) the control and data dependence graphs are restructured to effect the new loop organization and (3) an equivalent program is generated from the dependence graphs. The method we present is designed to work on any partition that is *legal*, i.e., any partition that preserves the control and data dependences of the original program. A partition can preserve all dependences if and only if there exists no dependence cycle spanning more than one output loop [KKP+81, AK87]. If there is a cycle involving control and/or data dependences, it must be contained entirely within a single partition (there are a few additional considerations for loops with exit branches, which are discussed in Section 3.4).

This condition is necessary and sufficient. Consider what must be done to generate code given a partitioning into loops: some linear order for the loops must be chosen. If we treat each output loop as a single node and define dependence between loops to be inherited in the natural way from dependences between statements, then the resulting graphs will be acyclic if and only if each original recurrence is confined to a single loop. Since an acyclic graph can always be ordered using topological sort and a cyclic graph can never be ordered, the condition is established.

Because our algorithm accepts any legal partition as input, it is as general as possible. It can be used for

vectorization, which seeks a partition of the finest possible granularity, or for MIMD parallelization, which seeks the coarsest possible granularity without sacrificing parallelism.

3 Restructuring for Loop Distribution

In the original program, control decisions are made and used in the same loop on the same iteration, but a partition may specify that decisions that are made in one loop be used in another. This problem is illustrated below by Example 1. Its corresponding G_{cd} and data dependence graph are shown in Figure 1.

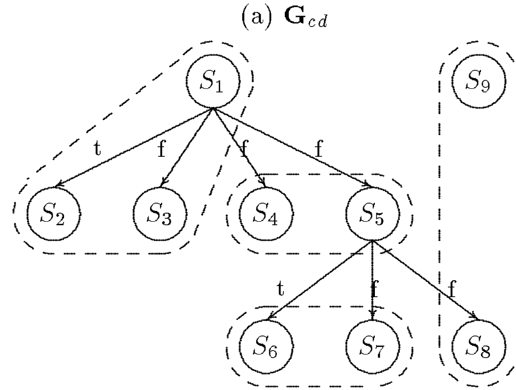
Example 1

```
        DO I = 1, N
S1          IF (A(I) .GT. T) THEN
S2              A(I) = I
            ELSE
S3              T = T + 1
S4              F(I) = A(I)
S5              IF (B(I) .NE. 0) THEN
S6                  U = A(I) / B(I)
                ELSE
S7                  U = A(I) - U
S8                  C(I) = B(I) + C(I)
                ENDIF
            ENDIF
S9          D(I) = D(I) + C(I)
        ENDDO
```

The data dependence graph in Figure 1(b) shows true dependences with solid lines and anti dependences with dashed lines. Loop carried edges are labeled with *lc*. In this example, output dependences are redundant and are not included. Given the data and control dependences in Figure 1, the statements may be placed in four partitions: (S_1, S_2, S_3), (S_4, S_5), (S_6, S_7), and (S_8, S_9). This particular partition is chosen solely for exposition of the algorithm, and in Figure 1(a) it is superimposed on G_{cd} such that each partition is enclosed by dashed lines.

Given this partition, some statements are no longer in the same loop with statements upon which they are control dependent. For example, S_4 is control dependent on S_1, but S_1 and S_4 are not in the same partition. In Figure 1 the G_{cd} edges that cross partitions represent decisions made in one loop, and used in a later loop. There may be a chain of decisions on which a node n is control dependent, but given a legal partition, all of n's predecessors and ancestors in G_{cd} are guaranteed either to be in n's partition, or in an earlier one. Therefore the execution of n may be determined solely from the execution of n's predecessors. We introduce *execution variables* to compute and store decisions that cross partitions in G_{cd} for both structured and unstructured code.

(a) $\mathbf{G}_{cd}$



(b) **Data Dependence**

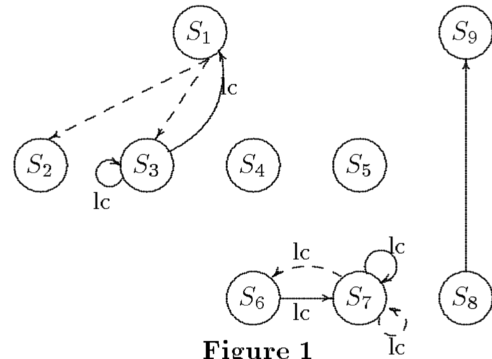


Figure 1

3.1 Execution Variables

Execution variables are only needed for *branch nodes*, because they correspond to control decisions in the original program. Any node in G_{cd} that has a successor must be a branch node, but only branch nodes with at least one successor in a different partition are of interest here. For each branch in this restricted set, a unique execution variable is created. Only one execution variable is created, regardless of the number of successors or the number of different partitions to which the successors belong. The execution variable is assigned the value of the test at the branch, capturing the branch decision. Later this variable will be tested to determine control flow in a subsequent partition. Hence, the creation of an execution variable will replace control dependences between partitions with data dependences. Execution variables are arrays, with one value for each iteration of the loop, because each iteration can give rise to a different control decision. If desired, loop invariant decisions can be detected [AC72] and represented with scalar execution variables.

All other known techniques, whether they are G_{cd} based or not, use boolean logic when introducing arrays to record branch decisions. This requires either testing and recording the path taken in previous loops or introducing additional arrays. In Example 1 in the

loop with statements (S_6, S_7), either S_6, or S_7, or neither may execute on a given iteration. Because there are three possibilities, the correct decision cannot be made with a single boolean variable. For example, if S_1 takes the true branch, then neither S_6 nor S_7 should execute. If just S_5's decision is stored, then one of S_6 or S_7 will mistakenly be executed, because the branch recording array for S_5 must either be true or false, regardless of S_1's decision.

Given this drawback, we have formulated execution variables to have three possible values: *true*, *false* and $\top$, which represents "undefined". Every execution variable is initialized to $\top$ at the beginning of the loop in which it will be assigned, indicating that the branch has not yet been executed. Because of the existence of a "not executed" value, the control dependent successors in different partitions need only test the value of the execution variables for their predecessors; they do not need to test the entire path of their control dependence ancestors. This is true whether the control flow is unstructured or structured. Execution variables completely capture the control decision at a node, making them extremely powerful.

3.2 Restructuring

The restructuring algorithm in Figure 2 creates and inserts execution variables and guards, given a distribution partition. It also updates the control and data dependence graphs to reflect the changes it makes. The algorithm is applied in partition order and, within a partition, in statement order over G_{cd} (statement order can be the original lexical order or interval order). The algorithm can be subdivided into three parts. First, execution variables for a branch node n are created where needed. Next, guard expressions are inserted for any nodes control dependent on n. Then the control and data dependences are updated, reflecting the new guards and execution variables.

The need for an execution variable for n is determined by considering n's successors. If there is an outgoing edge from n to a node that is not in n's partition, an execution variable is created. In Example 1, execution variables are needed for S_1 and S_5. The initialization of the execution variable is inserted at the beginning of n's partition, ensuring it will always be executed. Next, an assignment of the execution variable to n's test is inserted in node n. If n has successors in its partition, its branch is changed to test the execution variable. Otherwise, its branch is deleted.

For each partition P_k that contains a successor of n, a guard on n's execution variable is built. Here the successors of n are also considered in statement order. A guard is built for every distinct label from n into P_k. Each guard compares n's execution variable, $EV_n(I)$, to the distinct label l. All of n's successors in G_{cd} in P_k on label l are severed from n, and connected to the newly created corresponding guard. Our examples have only two labels, true and false, but any number of branch targets can be handled.

Execution Variable and Guard Creation

INPUT: partitions, G_{cd}, statement order
OUTPUT: modified G_{cd} with execution variables

```
for each partition, P
  for each n ∈ P, in order
    if (∃ an edge (n, o)_l ∈ G_cd, where o ∉ P)
      insert "EV_n(I) = ⊤" into P at top
      let test be n's branch condition
      if (∃ (n, m)_l where m ∈ P)
        replace n with { "EV_n(I) = test"
                       { "IF (EV_n(I) .EQ. true)"
      else
        replace n with "EV_n(I) = test"
      for each P_k ≠ P containing a successor of n
      {* Build guards, and modify G_cd *}
        for each l where ∃ (n, p)_l with p ∈ P_k
          create new statement N:
              "IF (EV_n(I) .EQ. l)",
          add N to P_k {* N is new and unique *}
          insert data dependences for EV_n
          for each (n, q)_l where q ∈ P_k
          {* Update control dependences *}
            delete (n, q)_l from G_cd
            add (N, q)_true to G_cd
          endfor
        endfor
      endfor
  endfor
endfor
```

Figure 2 Restructuring for Distribution

Consider Example 1. S_5 has successors in two partitions, (S_6, S_7) and (S_8, S_9). The successors in (S_6, S_7) are on different branches. S_6 is on the true branch, so the guard expression created is "$EV_5(I)$ .EQ. true." S_7 is on the false branch, so its guard expression is "$EV_5(I)$ .EQ. false." The old edges (5, 6) and (5, 7) are deleted from G_{cd}, and new edges attaching 6 and 7 to their corresponding guards are created. Similarly a guard is created for and connected to S_8.

The following simple optimization is included in the algorithm and examples but, for clarity, does not appear in the statement of the algorithm. Determining whether the initialization of an execution variable is necessary, can be accomplished when an execution variable is created for a node n. If n is not control dependent on any other node, that is, a root in the control dependence graph, then there is no need for initialization to be inserted. During guard creation for the successors of this node, the execution variable is known to have a value other than $\top$. Therefore, if control flow is structured, only one guard is needed for

each successor partition, instead of for each label.

After restructuring is applied, each partition has a correct G_{cd}, a correct data dependence graph, and possibly some new statements (execution variable assignments and guards). At this point the code for the distribution partition can be generated. We use a simple code generation algorithm, which is described in Section 4. Given the distribution in Figure 1 for Example 1, restructuring and code generation results in the following code.

```
      DO I = 1, N
          EV1(I) = A(I) .GT. T
S1        IF (EV1(I) .EQ true) THEN
S2            A(I) = I
          ELSE
S3            T = T + 1
          ENDIF
      ENDDO
      DO I = 1, N
          EV5(I) = ⊤
          IF (EV1(I) .EQ. false) THEN
S4            F(I) = A(I)
S5            EV5(I) = B(I) .EQ. 0
          ENDIF
      ENDDO
      DO I = 1, N
S6        IF (EV5(I) .EQ. true) U = A(I) / B(I)
S7        ELSE IF (EV5(I) .EQ. false) U = A(I) - U
      ENDDO
      DO I = 1, N
S8        IF (EV5(I) .EQ. false) C(I) = B(I) + C(I)
S9        D(I) = D(I) + C(I)
      ENDDO
```

The advantages of three-valued logic are illustrated by the concise guards for S_6 and S_7. As shown in Section 3.1, EV_5(I) must be explicitly tested for true or false, because if S_1 evaluated to true, then EV_5(I) will be $\top$ and neither S_6 nor S_7 should execute. Not only do we avoid testing EV_1(I) here, if S_4 and S_5 were in S_1's partition, there would be no need to store S_1's decision at all, even though S_6 are S_7 *indirectly* dependent on S_1 and S_1 remains in a different loop.

3.3 Optimality

Given a distribution, this section proves that our algorithm creates the minimal number of execution variables needed to track control decisions affecting statement execution in other loops. It also establishes that the algorithm produces the minimal number of guards on the values of an execution variable required to correctly execute the distributed code. Therefore, our algorithm is optimal for a given distribution partition.

Lemma 1: Each execution variable represents a unique decision that must be communicated between two loops.

Proof: An execution variable is only created when a decision in one partition directly affects the execution of a statement in another partition, as specified by G_{cd}. The definition of G_{cd} guarantees that no decision node subsumes another, and therefore any decisions represented by execution variables are unique. □

The restructuring algorithm creates the minimal number of guards on the values of an execution variable required to correctly determine execution. Let

p = the number of distinct partitions, P, and
m = the number of distinct branch labels, l,

that contain successors of node n. There are at most k tests on the value of an execution variable EV_n, where

$$k = \sum_{i=1}^{p} \sum_{j=1}^{m} (l_j \in P_i).$$

k is the sum of distinct labels into every distinct partition, and is bounded by the number of n's successors that are in separate partitions P_i.

Theorem 1: The number of guards that test an execution variable is the minimal required to preserve correctness for the given distribution.

Proof by contradiction. If there exists a version of the distribution with fewer guards, then guards would be produced that were either unnecessary or redundant. If there were unnecessary guards, then Lemma 1 would be violated. If there were redundant guards, then there would be multiple guards for nodes in the same partition with the same label. However the algorithm produces at most one guard per label used in a partition. □

3.4 Exit Branches

Because exit branches determine the number of iterations that are executed for an entire loop, they are somewhat sequential in nature. It is possible to perform distribution on such loops in a limited form by placing all exit branches in the first partition. Of course any other statements involved in recurrences with these statements must also be in the first partition. This forces the number of iterations to be completely determined by the first loop. If there are any statements left, any legal partitioning of them may be performed. The control dependences for each of the subsequent partitions can be satisfied with execution variables as described above. However, during code generation their loop bounds must be adjusted. If an exit branch was taken, any statements preceding it in the original loop must execute the same number of times as the first loop, later statements must execute one less time than the first loop. Otherwise, when no exit branch is taken, all loops must execute the same number of times as the first loop.

4 Code Generation

To review, there are three phases to distribution in the presence of control flow. The first step determines a partitioning based on data and control dependences.

The second step inserts execution variables and guards to effect the partition and updates the control and data dependences. The third step is code generation.

In step two the only changes to the data dependence graph are the addition of edges that connect the definitions of execution variables to their uses. A G_{cd} is built for each new loop during this phase. In each new loop's G_{cd} there are no control dependences between guards. However, there may be relationships between execution variables that may be exploited and inserted during code generation.

Now we consider code generation for unstructured or structured control flow without exit branches (Section 3.4 contains the extensions necessary for exit branches). Because the data and control dependence graphs, as well as the program statements are correct on entry to the code generation phase, a variety of code generation algorithms could be used. For example, any of the code generation algorithms based on the *program dependence graph* [FM85, FMS88, CFS90, BB89] could be used in conjunction with the above algorithm. A very simple code generation scheme is described here. It is designed to be used in ParaScope, an interactive parallelizing environment.

When transformations are applied in an interactive environment it is important to retain as much similarity to the original program as possible. The programmer can more easily recognize and understand transformed code when it resembles the original. For this reason, although partitioning may cause statements to change order, the original statement order and control structure *within* a partition is maintained. If the original loop is structured, the resulting code will be structured. If the original loop was unstructured and difficult to understand, so most likely will be the distributed loop.

To maintain the original statement ordering, an ordering number is computed and stored in $order[n]$. All the nodes in G_{cd} are numbered relative to their original lexical order, from one to the number of nodes. All of the execution variable initialization nodes are numbered zero, so they will always be generated before any other node in their partition. The newly created guard nodes have an order number and a relative number, $rel[n]$. Their order numbers are the number of the node whose execution variable appears in the guard expression. Their relative numbers $rel[n]$ are the number of the guard's lowest numbered successor. Both of these numbers can be computed when the guard is created. To simplify the discussion, branches are assumed to have only two label values, true and false, but the algorithm may be easily extended for multi-valued branches.

The rest of this section is divided into three parts. First relabeling, which corrects and renames statement labels, is described. Then the code generation discussion is separated into sections for structured and unstructured code.

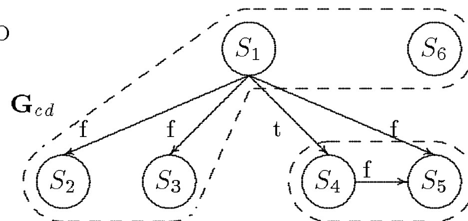
4.1 Label Renaming

A distribution partition may specify that the destination of a GOTO, that is, a labeled statement, be in a different loop from the GOTO. Replication and label renaming of GOTOs of this type must be performed to compensate for this after restructuring and before code generation. Renaming is easily accomplished by replacing the destination of a GOTO that is no longer in the same loop with an existing label or a new label, l_{P_j}, which may require a CONTINUE. The new destination has the same relative ordering as the original label. Often this will be the last statement in the partition. Reuse of labels is done whenever possible.

Example 2

```
        DO I = 1, N
S1          IF (p1) GOTO 4
            S2
S3          GOTO 5
        4   IF (p4) GOTO 6
        5   S5
        6   S6
        ENDDO
```

G$_{cd}$



Consider Example 2 with a distribution partition (S_1, S_2, S_3, S_6) and (S_4, S_5). The destination of S_1's GOTO, S_4, is not in the same partition as S_1, therefore the GOTO's label must be renamed. In this case, the new destination of S_1's jump must not interfere with the execution of S_6. To determine the destination and new label, the statement number of the original labeled statement (in this case 4) is compared to each statement in the partition following S_1 in order. When a statement number greater than the original is found (S_6 in our example), its label is used or a new one is created for it. Any empty jumps are deleted. A straightforward relabeling of the first partition in Example 2 after restructuring results in the following.

```
        DO I = 1, N
            EV1[I] = p1
S1          IF (EV1[I] .EQ. true) GOTO 6
            S2
        6   S6
        ENDDO
```

4.2 Structured Code Generation

Because code generation based on G_{cd} when it is a tree, is relatively simple [FM85, FMS88, BB89], this discussion emphasizes properly selecting and inserting the appropriate control structures for newly created

guards. Other G_{cd} code generation algorithms must select and create control structures for all branches. Because we use the original control structures for all but the newly created guards, only they are of interest here. When the guards are created they are identified by setting *guard*[n] to true. For all other nodes, *guard*[n] evaluates to false. With structured control flow the only two control structures that need be inserted when generating guards are IF-THEN and IF-THEN-ELSE.

Our algorithm for code generation given structured or unstructured code appears in Figure 3. It considers each partition and its nodes based on their order number, from lowest to highest. If a node n is not a guard node, it is generated with its original control structure followed by any descendents using depth-first recursion on G_{cd}. Given a tree G_{cd}, and that all control dependences are satisfied, the ancestors of a node n in G_{cd} must be generated before n is. If the node is a guard node, the control structure for it must be selected and created. This work is done in the procedure *genguard*.

If the guard node has true and false branches, an IF-THEN-ELSE is generated, where the conditional is the guard expression. For each successor on the true branch, it and its descendents are generated recursively, in order. The false successors are generated similarly under the ELSE. If there are two guards with the same order number, they are ordered by their relative number, and an IF-THEN-ELSE-IF-THEN is generated. The first guard expression becomes the first conditional, and its successors and their descendents are generated in the corresponding THEN. The second

Figure 3
Code Generation after Distribution

```
INPUT:   Gcd, ordered partitions, order[n], rel[n],
         guard[n], goto[n]
OUTPUT:  The distributed loops

for each partition, P
  gen (DO) {* The original loop header *}
  while (∃ n ∈ P)
    choose n with smallest order[n] and
      if goto[n] and not only predecessor,
      with greatest rel[n], otherwise smallest rel[n]
    done = false
    delete n from P
    if (guard[n])
      genguard (n)
    else
      gen (n)   {* all matches any branch label *}
      gensuccessors (n, all)
  endwhile
endfor

procedure gensuccessors (n, l)
  while (done = false and ∃ (n, m)_l ∈ Gcd and P)
    choose m with smallest order[m]
    if (∃ (p, m) where p ≠ n)
    {* In structured code m has one predecessor *}
    {* so this will never occur *}
      done = true
    else
      delete (m) from P
      gen (m)
      gensuccessors (m, all)
  endwhile
end
```

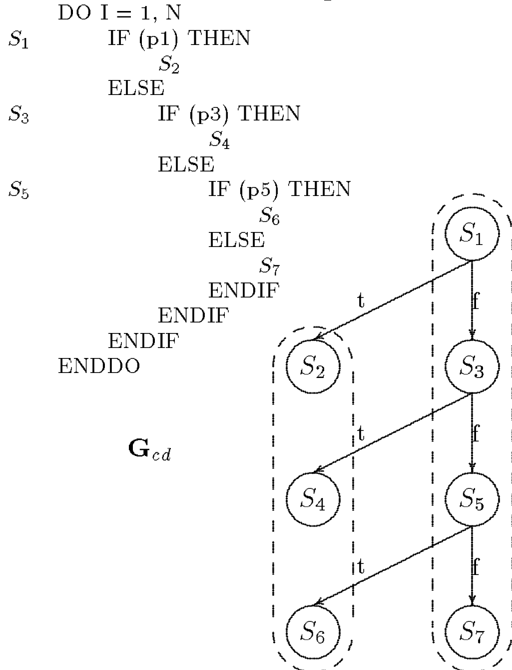
FIGURE 3: (continued)

```
procedure genguard (n)
  if (∃ (p, rel[n]) p ≠ n and p still ∈ P)
  {* Generate unstructured code *}
    let L be the statement label of node rel[n]
    gen (IF n GOTO L)
  {* Generate structured constructs *}
  else if (∃ (n, q)_true and (n, r)_false
  where order[q] < order[r])
  {* The original conditional was structured *}
    gen (IF n THEN)
    gensuccessors (n, true)
    gen (ELSE)
    gensuccessors (n, false)
    gen (ENDIF)
  else if (∃ o where order[o] = order[n])
  {* n chosen s.t. rel[n] < rel[o] *}
  {* The original conditional was structured *}
    gen (IF n THEN)
    gensuccessors (n, true)
    delete o from P
    gen (ELSE IF o THEN)
    gensuccessors (o, true)
    gen (ENDIF)
  else
  {* Unstructured or structured *}
  {* original conditional *}
    gen (IF n THEN)
    gensuccessors (n, true)
    gen (ENDIF)
end
```

guard expression conditions the ELSE-IF-THEN, and is followed by its descendents. Otherwise the guard is the only node with this order number, and an IF-THEN is generated for the guard and its descendents.

In Example 3 the control dependence graph is a long narrow tree.

Example 3



After performing the above algorithms, the code below results.

```
        DO I = 1, N
           EV3[I] = ⊤
           EV5[I] = ⊤
S1         EV1[I] = p1
           IF (EV1[I] .EQ. false) THEN
S3            EV3[I] = p3
              IF (EV3[I] .EQ. false) THEN
S5               EV5[I] = p5
                 IF (EV5[I] .EQ. false) THEN
                    S7
                 ENDIF
              ENDIF
           ENDIF
        ENDDO
        DO I = 1, N
           IF (EV1[I] .EQ. true) S2
           IF (EV3[I] .EQ. true) S4
           IF (EV5[I] .EQ. true) S6
        ENDDO
```

The first loop shows the dead branch optimization. The second loop illustrates that it is possible to generate correct code without adding control dependences between guards. More efficient code could be generated by noticing in the second loop nest if EV_1[I] is true then neither EV_3[I] or EV_5[I] can be true, and similarly if EV_3[I] is true then EV_5[I] cannot be true. This code would not have fewer tests, but would be more efficient and have a different structure.

4.3 Unstructured Code Generation

We can avoid the usual problems when generating code with a DAG G_{cd} for unstructured control flow by using the original structure and computing some additional information about the origin of the new guards. This information can be computed during code generation, or when the guards are created. If a guard is the only predecessor of its successors, the ordering and structure selection for structured control flow can be used. For guards that have successors with multiple predecessors, GOTO's are generated.

The key insight is that, although a node can be control dependent on many nodes, only one of these dependences may be from a structured construct. Observe that in a connected subpart of G_{cd}, when guards are created from GOTOs outside the partition into the subpart, the guards with the highest order numbers will be generated first. One or two GOTOs may result. When a GOTO will result in a guarded GOTO and a structured construct, care is taken to generate the GOTO first. In this case the node with larger relative number between the two guards will be selected, and a GOTO for it is generated.

The recursive generation of successors and their descendents must choose the lowest numbered successor to generate first. In structured code this is guaranteed to be the true branch, but with an IF-GOTO the false branch is lower. In structured code, the generation of successors is immediately preceded by their one and only predecessor. In unstructured code, to ensure all control dependences are satisfied, the recursion must cease if a node has other predecessors that have not yet been generated. When there are multiple GOTO's this situation may arise.

Now returning to Example 2, and applying code generation results in the code below.

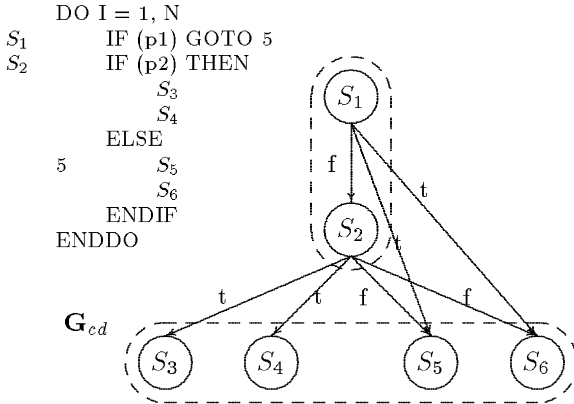
```
        DO I = 1, N
           EV1[I] = p1
S1         IF (EV1[I] .EQ. true) GOTO 6
           S2
        6  S6
        ENDDO
        DO I = 1, N
           IF (EV1[I] .EQ. false) GOTO 5
           IF (EV1 .EQ true) THEN
S4            IF (p4) GOTO P2
        5     S5
P2         CONTINUE
        ENDDO
```

Notice that when the second partition is generated the GOTO is generated first. The guards for S_4 and S_5 have the same order number, i.e. 1, but because S_1 was a GOTO, the jump to S_5 is generated first. Then S_4's guard, S_4, and S_5 are generated. Here and in Example 4 there are jumps into structured constructs. Although these jumps are non-standard Fortran, many compilers accept them, and regardless can

be implemented with GOTO's.

Finally, consider Example 4 with a distribution partition (S_1, S_2) and (S_3, S_4, S_5, S_6).

Example 4



Distribution restructuring, label renaming, and code generation performed on the above results in the following code.

```
        DO I = 1, N
              EV2[I] = ⊤
S1            EV1[I] = p1
              IF (EV1[I] .EQ. true) GOTO P1
S2            EV2[I] = p2
        P1    CONTINUE
        ENDDO
        DO I = 1, N
              IF (EV1[I] .EQ. true) GOTO 5
              IF (EV2[I] .EQ. true) THEN
                    S3
                    S4
              ELSE IF (EV2[I] .EQ. false) THEN
        5           S5
                    S6
              ENDIF
        ENDDO
```

5 Related Work

Callahan and Kalem present two methods for generating loop distributions in the presence of control flow [CK87]. The first, which works for structured or unstructured control flow, replicates the control flow of the original loop in each of the new loops by using G_f. *Branch variables* are inserted to record decisions made in one loop and used in other loops. An additional pass then trims the new loops of any empty control flow. Dietz uses a very similar approach [Die88]. It has some of the same drawbacks of if-conversion.

Callahan and Kalem's second method, which works only for structured control flow, uses G_f, G_{cd}, and boolean execution variables. Their execution variables indicate if a particular node in G_f is reached and are created for edges in G_{cd} that cross between partitions. Their execution variables are assigned true at the successor indicating the successor will execute, rather than assigning the decision made at the predecessor. Also, one execution variables may be needed for every successor in the descendent partition. Because their code generation algorithm is based on G_f, rather than G_{cd}, the proof of how an execution variable is used is much more difficult and is not given. Towle [Tow76] and Baxter and Bauer [BB89] use similar approaches for inserting conditional arrays.

Ferrante, Mace, and Simons present related algorithms whose goals are to avoid replication and branch variables when possible [FM85, FMS88]. Their code generation algorithms convert parallel programs into sequential ones, and like ours, are based on G_{cd}. They discuss three transformations that restructure control flow: loop fusion, dead code elimination, and branch deletion.

Other research concerned with the definition and use of the program dependence graph does not address distribution [FOW87, FM85, FMS88]. The papers describing the PTRAN project [CFS90, ABC+87], which also performs code generation based on G_{cd}, do not address distribution. Work in memory management and name space adjustment [KKP+81, Por89] uses distribution, but only when no control dependences are present.

The Stardent compiler [All90] distributes loops with structured control flow by keeping groups of statements with the same control flow constraints together. For example, all the statements in the true branch of a block IF must stay together, so only the outer level of IF nests can be considered. This limits effectiveness of distribution because partitions are artificially made larger, possibly by grouping parallel statements with sequential ones.

6 Conclusions and Future Work

We have presented a very general and optimal algorithm for loop distribution when control flow is present. The algorithm can be used to enhance the effectiveness of vectorizers, parallelizers and programming environments, alike. The generality of our system will allow future research to focus on discovering partitioning algorithms that are effective in deciding if and when a distribution can be profitably used.

This work was motivated by the desire to handle loops with control flow in the ParaScope Editor[BKK+89], which supports a variety of transformations, including loop distribution. An implementation of this work is in progress.

7 Acknowledgments

We would like to thank the reviewers for their valuable suggestions and comments, all of which were in-

corporated. We are also grateful to Marina Kalem and Chau-Wen Tseng for their significant contributions to this work.

References

[ABC+87] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing.* Springer-Verlag, Athens, Greece, June 1987.

[ABC+88] F. Allen, M. Burke, P. Charles, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for detecting useful parallelism. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[AC72] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, 1972.

[AK84] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AKPW83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1983.

[All83] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[All90] J. R. Allen. Private communication, February 1990.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[BB89] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, October 1966.

[BJ66] C. Böhm and G. Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Communications of the ACM*, 19(5), May 1966.

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[CFS90] R. Cytron, J. Ferrante, and V. Sarkar. Experiences using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, 1990.

[CK87] D. Callahan and M. Kalem. Control dependences. Supercomputer Software Newsletter 15, Dept. of Computer Science, Rice University, October 1987.

[Die88] H. Dietz. Finding large-grain parallelism in loops with serial control dependences. *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[FM85] J. Ferrante and M. Mace. On linearizing parallel code. In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, New Orleans, LA, January 1985.

[FMS88] J. Ferrante, M. Mace, and B. Simons. Generating sequential code from parallel code. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[KKLW84] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. The structure of an advanced retargetable vectorizer. In *Supercomputers: Design and Applications*, pages 163–178. IEEE Computer Society Press, Silver Spring, MD, 1984.

[KKP+81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[Mur71] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971. Report No. 71-424.

[Por89] A. Porterfield. *Software Methods for Improvement of Cache Performance.* PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[SK86] R. G. Scarborough and H. G. Kolsky. A vectorizing Fortran compiler. *IBM Journal of Research and Development*, 30(2):163–171, March 1986.

[Tow76] R. A. Towle. *Control and Data Dependence for Program Transformation.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March 1976.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.