CHAPTER 7

Control Dependence

7.1 Introduction

In this chapter we return to the issue of conditional execution. Conditional execution causes problems for parallelization because it affects the pattern of dependence. For instance, consider the following loop:

```
     DO 100 I = 1, N
        IF (A(I-1).GT. 0.0) GO TO 100
S1         A(I) = A(I) + B(I)*C
100  CONTINUE
```

If statement S_1 appeared alone in the loop, it would be directly vectorizable. However, once its execution is controlled by the conditional it cannot be vectorized because, in each iteration, we need to wait until the previous iteration is complete to determine whether or not to execute statement S_1. In a sense, statement S_1 the conditional. Once this is understood, it is easy to see that the conditional and statement S_1 form a recurrence, because there is clearly a loop-carried dependence from S_1 to the IF-statement in the next iteration.

The situation is much clearer if the loop is written as follows:

```
DO I = 1, N
   IF (A(I-1).GT. 0.0) A(I) = A(I) + B(I)*C
ENDDO
```

Now it seems quite natural to view the conditional expression as another input to the statement, making the recurrence clearly recognizable.

This observation raises the question: can we eliminate the problem of conditionals by converting every statement to a conditional statement—that is, by making the conditions for execution of each statement part of the list of inputs for that statement. As it happens, the answer is *yes*, and the resulting transformation—known as *if conversion*—is widely used in vectorizing compilers. The beauty of if conversion is that it neatly finesses the issue of control dependence because all dependences arising from control are converted to data dependences.

In this chapter we will present if conversion and the various algorithms that support it. We will then examine why if conversion is not as well suited for parallel computation as it is for vector computation. Finally we will introduce control dependence and the algorithms for computing it and using it in parallel code generation.

7.2 If Conversion

7.2.1 Forward Branches

We will begin our study of if conversion by considering a simple subproblem: elimination of forward branches within a single loop. This will permit us to temporarily postpone examination of more complicated branching structures.

We have already seen that conditional assignment statements can be easily handled by a vectorizing compiler. For example, it is easy to see that the loop

```
DO I = 1, N
   IF (X(I).GT. 0.0) X(I) = X(I) + Y(I)
ENDDO
```

can be directly vectorized by rewriting in Fortran 90:

```
WHERE(X(1:N).GT. 0.0) X(1:N) = X(1:N) + Y(1:N)
```

However, it is not as easy to see how to vectorize

```
    DO 100 I = 1, N
S0     IF (X(I).LE. 0.0) GO TO 100
S1        X(I) = X(I) + Y(I)
100 CONTINUE
```

To do this, we must capture the outcome of the conditional statement S_0 in a logical array variable. Then we will use that array variable in a conditional that "guards" the execution of S_1.

```
    DO 100 I = 1, N
S0     m(I) = X(I).LE. 0.0
S1     IF (.NOT.m(I)) X(I) = X(I) + Y(I)
100 CONTINUE
```

In turn, this can be directly vectorized:

```
S0  m(1:N) = X(1:N).LE. 0.0
S1  IF (.NOT.m(1:N)) X(1:N) = X(1:N) + Y(1:N)
```

Now consider a slightly more complicated example:

```
    DO 100 I = 1,N
C1     IF (A(I).GT.10) GO TO 60
20        A(I) = A(I) + 10
C2        IF (B(I).GT.10) GO TO 80
40           B(I) = B(I) + 10
60     A(I) = B(I) + A(I)
80     B(I) = A(I) - 5
    ENDDO
```

To correctly translate this loop to one in which all assignments are conditional, you must determine the condition that holds at any assignment statement. Statement 20 is clearly executed if and only if the conditional in C_1 is false. Similarly, statement 40 is executed if and only if both the conditionals from C_1 and C_2 are false. However, it is not so simple for statements 60 and 80. Statement 60 can be reached by direct fall-through (C_1 and C_2 both false) or by transfer from statement C_1 (C_1 true). Statement 80 can be reached by fall-through from 60 (hence under the same conditions) or by direct transfer from C_2 (C_1 false and C_2 true). Thus the loop can be translated to the following:

```
    DO 100 I = 1,N
       m1(I) = A(I).GT.10
20     IF(.NOT.m1(I)) A(I) = A(I) + 10
```

```
      IF(.NOT.m1(I)) m2(I) = B(I).GT.10
40    IF(.NOT.m1(I).AND..NOT.m2(I))
         B(I) = B(I) + 10
60    IF(.NOT.m1(I).AND..NOT.m2(I).OR.m1(I))
         A(I) = B(I) + A(I)
80    IF(.NOT.m1(I).AND..NOT.m2(I).OR.m1(I).OR.
       .NOT.m1(I).AND.m2(I)) B(I) = A(I) - 5
   ENDDO
```

These conditions seem to get very complicated at the end of the loop, raising the possibility that the code generated will be very complicated. However, a little symbolic simplification reveals that the condition on statement 60 can be slightly improved and the condition on statement 80 is equivalent to *true*—that is, statement 80 is unconditionally executed. After simplification, the loop looks like this:

```
   DO 100 I = 1,N
      m1(I) = A(I).GT.10
20    IF(.NOT.m1(I)) A(I) = A(I) + 10
      IF(.NOT.m1(I)) m2(I) = B(I).GT.10
40    IF(.NOT.m1(I).AND..NOT.m2(I))
         B(I) = B(I) + 10
60    IF(m1(I).OR..NOT.m2(I))
         A(I) = B(I) + A(I)
80    B(I) = A(I) - 5
   ENDDO
```

The procedure used in this example can be stated as a simple algorithm for forward if conversion, given in Figure 7.1.

FIGURE 7.1

Forward if conversion algorithm.

Input: a single loop with a list of body statements *B*

Output: A loop without conditional transfers.

current := *true*;
for each labeled statement *S* in *B* **do** *cond*(*S*) := *true*;

for each statement *S* in *B* in order **do begin**
    if *S* is labelled **then** *current* := *Simplify* (*current* `.OR.`*cond*(*S*));
    if *S* is a conditional transfer**:** `IF (`*C*`) GO TO` *L* **then begin**
        let `m` be a new compiler-generated logical variable,
        initialized to *true*;
        convert it to `IF (`*current*`) m :=` *C*;
        current := *current* `.AND..NOT.m`

cond(*L*) := *cond*(*L*) .OR. (*current*.AND.m)
end
else if *S* is of the form: IF (*C*) *S*, where *S* is not a transfer **then**
convert it to IF (*C*.AND.*current)* *S*
else
convert *S* to IF (*current*) *S*
endif
end

Note that this method relies on scalar expansion to convert the compiler-generated logical variables to arrays. We will discuss the boolean simplification algorithm use in *Simplify* shortly.

7.2.2 Branch Classification

For the purpose of analysis, every branch can be categorized as one of three types:

1. a *forward branch* transfers control to a target that occurs after the branch but a the same loop nesting level;
2. an *exit branch* terminates one or more loops by transferring control to a target outside a loop nest; and
3. a *backward branch* transfers control to a statement occurring lexically before the branch but at the same level of nesting.

We have already seen a number of examples of forward branches. Here is a code fragment containing an exit branch:

```
      DO I = 1, N
         IF (ABS(A(I)-B(I)).LE.DEL) GO TO 200
      ENDDO
      ...
200   CONTINUE
```

In this "search loop" the exit occurs when a pair of values from the arrays A and B is found to be sufficiently close together in value.

Backward branches are often used, particularly in Fortran 66 and 77, to encode "while" loops:

```
10    I = NEXT(I)
      A(I) = A(I) + B(I)
      IF (I.LT.1000) GO TO 10
```

If conversion uses two different transformations to eliminate branches within a program:

1. *Branch relocation* moves branches out of loops until the branch and its target are nested in the same number of DO-loops—the procedure converts each exit branch into either a forward branch or a backward branch.
2. *Branch removal* eliminates forward branches by computing guard expressions for action statements under their control and conditioning execution on these expressions.

The following sections present these two techniques in more detail.

7.2.3 Exit Branches

Exit branches differ from other branches in that they affect the execution of statements lexically before the branch in the same loop, as well as those that follow. This is because a branch out of a loop terminates execution of all statements in that loop on iterations after the exit. Consider a loop of the following pattern:

```
     DO I = 1, N
        S1
        IF (p(I)) GO TO 100
        S2
     ENDDO
100  S4
```

If the jump is taken on iteration `I`, neither S_1 nor S_2 will be executed on iterations `I+1` through `N`. Thus, exit branches are more complicated than forward branches, because eliminating them requires modifying the guards of all statements within the loop exited.

To get a sense of how we might handle exit branches, let us begin with an example:

```
     DO J = 1, M
        DO I = 1, N
           A(I,J) = B(I,J) + X
           IF (.NOT.L(I,J)) GO TO 200
           C(I,J) = A(I,J) + Y
        ENDDO
        D(J) = A(N,J)
200     F(J) = C(10,J)
     ENDDO
```

To eliminate the exit branch in this example, we must introduce a compiler-generated boolean variable `lm`, defined as follows:

$$\mathbf{lm} = \bigcap_{k=1}^{\mathbf{I}} \mathbf{L}(k, \mathbf{J}) \qquad \textbf{(EQ 7.1)}$$

Using this new variable, the code can be converted to:

```
     DO J = 1, M
        lm = .TRUE.
        DO I = 1, N
           IF (lm) A(I,J) = B(I,J) + X
           IF (lm) LM = LM.AND.L(I,J)
           IF (lm) C(I,J) = A(I,J) + Y
        ENDDO
        IF (.NOT.lm) GO TO 200
        D(J) = A(N,J)
200     F(J) = C(10,J)
     ENDDO
```

This transformation effectively moves the branch to statement 200 outside the loop on `I`. If we now apply forward if conversion to the outer loop, we get, with simplification,

```
     DO J = 1, M
        lm = .TRUE.
        DO I = 1, N
           IF (lm) A(I,J) = B(I,J) + X
           IF (lm) LM = LM.AND.L(I,J)
           IF (lm) C(I,J) = A(I,J) + Y
        ENDDO
        m = .NOT.lm
        IF (.NOT.m) D(J) = A(N,J)
200     F(J) = C(10,J)
     ENDDO
```

If we forward substitute for variable `m` and then use scalar expansion to expand `lm` to a two-dimensional array, we get:

```
     DO J = 1, M
        lm(0,J) = .TRUE.
        DO I = 1, N
           IF (lm(I-1,J) A(I,J) = B(I,J) + X
           IF (lm(i-1,J)lm(I,J)=lm(I-1,J)
                 .AND.L(I,J)
           IF (lm(I,J) C(I,J) = A(I,J) + Y
        ENDDO
        IF (lm(N,J)) D(J) = A(N,J)
200     F(J) = C(10,J)
     ENDDO
```

Applying a standard vectorization algorithm to this nest produces:

```
     DO J = 1, M
        lm(0,J) = .TRUE.
        DO I = 1, N
           IF (lm(i-1,J)) lm(I,J)=lm(I-1,J)
                 .AND.L(I,J)
        ENDDO
     ENDDO
     WHERE(lm(0:N-1,1:M)) A(1:N,1:M)=B(1:N,1:M)+X
     WHERE(lm(0:N-1,1:M)) C(1:N,1:M)=A(1:N,1:M)+Y
     WHERE(lm(N,1:M)) D(1:M) = A(N,1:M)
200  F(1:M) = C(10,1:M)
```

This aggressive approach produces conditional vectorization in two dimensions. However, it is not the cleanest possible version of the loop. Suppose we applied scalar expansion in only one dimension of the original loop after if conversion:

```
     DO J = 1, M
        lm(0) = .TRUE.
        DO I = 1, N
           IF (lm(I-1)) A(I,J) = B(I,J) + X
           IF (lm(i-1)) lm(I)=lm(I-1).AND.L(I,J)
           IF (lm(I,J)) C(I,J) = A(I,J) + Y
        ENDDO
        IF (lm(N)) D(J) = A(N,J)
200     F(J) = C(10,J)
     ENDDO
```

After vectorization this becomes:

```
     DO J = 1, M
        lm(0) = .TRUE.
        DO I = 1, N
           IF (lm(i-1)) lm(I)=lm(I-1).AND.L(I,J)
        ENDDO
        IF (lm(I-1)) A(I,J) = B(I,J) + X
        IF (lm(I,J)) C(I,J) = A(I,J) + Y
        IF (lm(N)) D(J) = A(N,J)
200     F(J) = C(10,J)
     ENDDO
```

If the compiler is intelligent enough to recognize that the inner loop is simply finding the first false element in the array `L(1:N,J)`, it can produce the following code:

```
     DO J = 1, M
        n = FirstFalseX(L(1:N,J)) - 1
        A(1:n+1,J) = B(1:n+1,J) + X
        C(1:n,J) = A(1:n,J) + Y
        IF (n.GT.N) D(J) = A(N,J)
```

```
200     F(J) = C(10,J)
      ENDDO
```

where the builtin vector function `FirstFalseX` returns the index in the logical parameter array of the first `.FALSE.` value or, if there are no false values, the length of the parameter array plus 1. This is nearly the best you can do on this example.

The basic procedure used in branch relocation, as in forward branch elimination, is to compute the precise Boolean guard expression that governs the execution of each statement in the loop from which the exit branch is to be eliminated. The algorithm is given in Figure 7.1.

FIGURE 7.2

Branch relocation algorithm.

procedure *relocate_branches*(*l*);

/* *l* is the `DO` statement for the loop */
/* *lg* is the loop exit guard that will be attached to each statement */

S_1: **for each** exit branch "`IF (`*p*`) GO TO` *S*" **do begin**
create a new loop exit flag `x`;
insert the assignment "`x = .TRUE.`" prior to *l*;
if *lx* = **null then** *lg* := "`x`" **else** *lg* := *lg* || "`.AND.x`" **endif**;
insert the branch "`IF (.NOT.x) GO TO` *S*" after the loop;
replace the exit branch by the statement "`x = .NOT.`*p*";
end

for each `DO` statement *d* in the loop *l* **do** *relocate_branches*(*d*);

for each non-`DO` statement *s* in the loop *l* **do**
if guard(*s*) = **null then** guard(*s*) := *lg*
else guard(*s*) := guard(*s*) || "`.AND.`" || *lg* **end if**

end *relocate_branches*

Note that this algorithm also relocates all branches in any nested do loops.

We need to develop a proof of correctness of this algorithm.

7.2.4 Backward Branches

We have already discussed the removal of forward branches, so we now turn to the topic of backward branches, which give rise to implicit loops.

Unfortunately, backward branches create more problems than by creating loops, because they also complicate forward branch removal. The following code illustrates the problem:

```
    IF (P) GO TO 200
    ...
100 S1
    ...
200 S2
    ...
    IF (Q) GO TO 100
```

The forward if conversion algorithm in Figure 7.1 on page 234 would set the guard for S_1 to "`.NOT.P`," which would be incorrect because it would prevent S_1 from being executed when `P` is true and the backward branch to statement 100 is taken.

One approach that avoids many of the complications from backward branches is to isolate these branches, leaving the code under their control, known as an *implicitly iterative region,* untouched. Of course, this would prevent the removal of any forward branches into an implicitly iterative region, a severe limitation. Therefore, we will examine strategies to handle implicit iteration.

Therefore,

To be correct, a guard for S_1 must reflect two alternatives:

1. S_1 is executed on the first pass through the code only if `P` is false.
2. S_1 is always executed when the backward branch is taken.

This suggests that one set of conditions be used to guard the first pass through an implicitly-iterative region and a different set be used to guard subsequent passes. In the example, suppose we introduce a backward branch guard `bb`, which is true whenever the backward branch with which its is associated is taken. Then the example code could be replaced by:

```
    m = P
    ...
    bb = .FALSE.
100 IF (.NOT.m .OR (m.AND.bb)) S1
    ...
200 S2
    ...
```

```
IF (Q) THEN
   bb = .TRUE.
   GO TO 100
ENDIF
```

Note that bb is set to true only if a branch back actually occurs.

Corresponding to the two alternatives, there are two ways that the target of a backward branch can be reached from the start of a program:

1. It can *fall through* from the previous statement, in which case the condition for execution is encoded by the current condition from the predecessor, or
2. It can branch around the statement and then reach it via a backward branch, in which case the correct guard is the logical conjunction of the condition for branching around and the condition for branching back.

Thus, if we denote the current condition at target *y* by cc, the branch condition by m and the backward branch condition by bb, the guard at *y* should be

```
cc.OR.(m.AND.bb)
```

The condition at the target of a forward branch into the iterative region might also have to be modified, as the following code illustrates:

```
    IF (P) GO TO 200
100 S1
    GO TO 300
200 S2
    IF (Q) GO TO 100
300 S3
```

The correct guard for S_2 must be

```
m.AND..NOT.bb
```

because S_2 is executed if and only if the forward branch to 200 is taken and the backward branch has not been taken. Thus, in order to remove the branch just before statement 200, "**.NOT.bb**" must be included in the target condition. In general, the target condition for a forward branch into multiple implicitly iterative regions is the conjunction of the branch flag and the negation of the branch back flag for each region. The negations of the branch back flags in the target condition signifies that control may

pass to the target statement only on the first iterations of these regions. After complete branch removal, the previous example becomes:

```
    m = P
    bb = .FALSE.
100    IF (.NOT.m.OR.(m.AND.bb)) S1
200    IF (.NOT.bb.AND.m) S2
       IF (.NOT.bb.AND.m.AND.Q) THEN
          bb = .TRUE.
          GO TO 100
       ENDIF
300 S3
```

At S_3, the current condition, "`.NOT.bb.AND.m`" is disjoined with the target condition "`.NOT.m.OR.(m.AND.bb)`," which simplifies to true, correctly determining that S_3 should always be executed.

Figure 7.3 contains the general branch removal algorithm incorporating these observations. The only new feature is the check, encapsulated in *process_branch* (Figure 7.4), on whether forward branches jump into implicitly iterative regions. Also, note that block IF statements are not generated at the backward branch, since these would defeat the purpose of IF conversion. Instead, a sequence of equivalent assignments is generated.

FIGURE 7.3

Complete branch removal.

```
procedure remove_branches (S,current) returns condition;
    /* S is the statement under consideration */
    /* current is the current condition        */
    if S is labelled then current := Simplify (current .OR.cond(S))
    endif;

    case statement_type(S) in
       /* IF (P) GOTO L */
       begin
          let m be a new compiler-generated logical variable,
              initialized to true;
          convert it to IF (current) m := C;
          process_branch(S, L, current.AND.m)
          current := current .AND..NOT.m;
       end;
       /* GOTO L */
```

```
            begin
                process_branch(S, L, current.AND.m)
                current := false;
                delete statement S;
            end;
            /* IF (P) GOTO L — backward at same level */
            begin
                let bb be a new compiler-generated logical variable;
                insert "bb = .FALSE." before statement L;
                let tp be a new compiler-generated logical variable;
                replace S with the statements
                    "tp = current"
                    "IF (tp) tp = P"
                    "IF (tp) bb = .TRUE."
                    "IF (tp) GO TO L"
            end;
            /* All other statement types */
            begin
                if S is of the form "IF (P) S1" then
                    replace it with "IF (current.AND.P) S1"
                else
                    replace it with "IF (current) S"
                end if
            end;
        end case;
        return current
    end remove_branches
```

FIGURE 7.4 Forward branch processing.

```
procedure process_branch(S, L, cond);

    /* S is the branch and L is the target */
    /* cond is the condition for branch taken */

    stmt_guard := true;
    for each implicitly iterative region that S jumps into do begin
        let bb be the branch back flag controlling the region;
        let x be the target of the backward branch;
        cond(x) := cond(x) .OR. (cond.AND.bb);
        stmt_guard := stmt_guard .AND..NOT.bb
    end
    cond(L) := cond(L) .OR. (cond.AND.stmt_guard)
```

end *process_branch*

Develop full branch removal in the presence of backward branches. Include a section on converting backward gotos to while loops. Also prove the correctness of the algorithm here.

7.2.5 Iterative Dependences

Once IF conversion has been applied to a program, all dependences due to action statements, branch statements and placeholder statements are representable as data dependences. However, iterative statements can also give rise to control dependences, as the next example illustrates.

```
 20 DO 100 I = 1, 100
 40    L = 2*I
 60    DO 100 J= 1,L
 80       A(I,J) = 0
100 CONTINUE
```

Without dependences on iterative statements, this might be vectorized as follows:

```
 20 DO 100 I = 1, 100
 40    L = 2*I
100 CONTINUE
 80 A(1:100,1:L) = 0
```

This is incorrect because the value of `L` is different for each different value of the outer loop induction variable `I`. The inner loop header depends on the computation of the variable `L`, and the assignment statement in the inner loop depends on the loop header because it determines how many times the assignment is executed. Thus, there is a control dependence from the inner loop header to the assignment.

In general, dependences on iterative statements are caused by exactly this effect; namely, a DO statement controls the number of times a statement within the loop is executed. We would like to convert this type of control dependence to data dependences. One way to do this is to a DO statements as an assignment statement, in which the left hand side is some unique (compiler-generated) variable and the right hand side is an expression that uses the upper bound, lower bound and step to compute the number of iterations in the loop. Then assume that the right hand side variable is an input to each statement in the range of the DO loop.

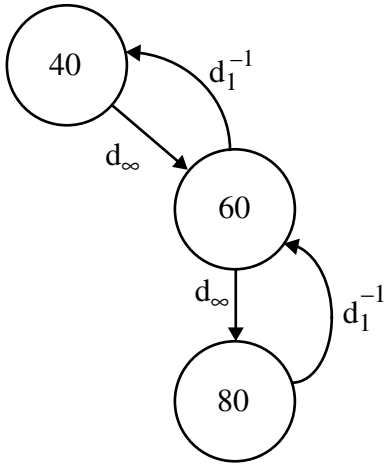
Since the purpose of the assignment statement is to capture the evaluation of number of iteration, the statement must be evaluated a the same point where the bounds are evaluated. In Fortran, the bounds are evaluated just prior to entering the loop, so the assignment should be nested just outside the loop it controls. In the case of a conditional loop, such as a WHILE loop, the control condition is evaluated on each pass through the loop; therefore, the assignment for a WHILE loop should be nested at the same level as the statements within the loop.

If we replace the loop statements in our example with these special assignments, we get the following code:

```
    ICNT1 = 100
 40    L = 2*I (ICNT1)
 60    ICNT2 = L
 80       A(I,J) = 0 (ICNT2)
100 CONTINUE
```

Here the variables in parentheses are considered implicit inputs to statements. The assignment for the outer loop does not participate in dependence because forward substitution of the constant right-hand side eliminates its use. However, the assignment for the inner loop cannot be eliminated. The result of these transformations is the dependence pattern shown in Figure 7.5.

FIGURE 7.5 Iterative dependence example.



Here we see that the outer loop carries a recurrence and cannot be vectorized. However, if the outer loop is executed sequentially, the inner loop can be vectorized to produce:

```
 20 DO 100 I = 1, 100
 40    L = 2*I
 80       A(I,1:L) = 0
100 CONTINUE
```

To achieve maximum performance, it is desirable to move the count computations out of as many loops as possible, as the following example demonstrates.

```
DO I = 1, L
   IUB = 2*I
   DO J = 1, M
      DO K = 1, N
         A(I,J,K) = 0.0
      ENDDO
   ENDDO
ENDDO
```

After conversion, this becomes:

```
ICNT1 = L
```

```
      K = 2*I (ICNT1)
      ICNT2 = M (ICNT1)
         ICNT3 = N (ICNT2)
            A(I,J,K) = 0.0 (ICNT3)
         ENDDO
      ENDDO
   ENDDO
```

Here the count computations for the loop at level 2 can be moved out of the loop at level 1 and the count computation for level 3 can be moved out of the loop at level 2:

```
ICNT1 = L
ICNT2 = M (ICNT1)
   K = 2*I (ICNT1)
   ICNT3 = N (ICNT2)
         A(I,J,K) = 0.0 (ICNT3)
      ENDDO
   ENDDO
ENDDO
```

This permits the vectorization of the assignment statement in two dimensions:

```
DO I = 1, L
   K = 2*I (ICNT1)
   A(I,1:M,1:N) = 0.0 (ICNT3)
ENDDO
```

Once the dependence graph of the program has been computed, this code motion is easy to perform. All dependences in the program will be preserved if the iteration count assignment is moved out to the deepest level containing a statement (other than another generated iteration count assignment, on which it depends.

Iterative dependences arising from while loops can be handled similarly, except that the condition is evaluated within the loop. For example:

```
DO I = 1, L
   M = I*2
   J = 0
   DO WHILE (J.LT.M)
      J = J + 1
      A(I,J) = 0
   ENDDO
```

```
ENDDO
```

The conversion in this case would yield

```
icnt1 = L
   M = I*2 (icnt1)
   J = 0 (icnt1)
      icnt2 = (J.LT.M) (icnt1)
      J = J + 1 (icnt2)
      A(I,J) = 0 (icnt2)
   ENDDO
ENDDO
```

In this case the condition is recomputed each time the inner loop is executed, so no vectorization is possible. In the next section, we illustrate how IF loops can be converted to WHILE loops and WHILE loops can be converted to DO loops.

7.2.6 Conversion to WHILE Loops

7.2.7 Simplification

One of the key components of the methods described thus far is simplification of statement guards. The Boolean simplifier is called each time a statement to which a branch might be made is processed. At each point, the condition to be simplified is a logical expression over the compiler-generated logical variables produced by IF conversion.

One well-known simplification procedure, due to Quine and McCluskey, operates in three phases:

1. The logical expression is reduced to a conjunction of *minterms*, which are disjunctions in which each logical expression in the original expression is present in either negated or non-negated form.
2. The collection of *prime implicants*, terms which exactly represent a disjunction of minterms in the original expression but which are not contained in any other prime implicant, is computed.
3. A minimal subset of the prime implicants is selected, with the restriction that a subset is legal only if it covers the original expression. That is, the conjunction of a legal selection of prime implicants is true if and only if the original Boolean expression is true.

Unfortunately, this well-known algorithm is exponential in the size of the original expression, making it impractical for general IF conversion.

A faster algorithm can be derived if we sacrifice economy of representation and concentrate on the real reason for simplification: the determination of when a compiler-generated branch variable can be factored out of the current condition. To understand the problem, let us consider an example.

```
    DO I = 1, N
1      IF (P) GO TO 10
2         A(I) = B(I)
3         IF (Q) GO TO 20
10           C(I) = 0
20     A(I) = A(I) + C(I)
    ENDDO
```

IF conversion will translate this to:

```
    DO I = 1, N
1      m1 = P
2         IF(.NOT.m1) A(I) = B(I)
3         IF(.NOT.m1) m2 = Q
10           IF(.NOT.m2.AND..NOT.m1.OR.m1)C(I) = 0
20     A(I) = A(I) + C(I)
    ENDDO
```

Observe that, even though the branch target for variable `m1` is reached at statement 10, variable `m1` does not simplify out of the equation until the branch target for `m2` is reached. In general, logical branch variables can only be simplified out of an expression in the reverse of the order that they are introduced by the algorithm. In other words, the most recently introduced logical variable will simplify out first. Thus, `m1` cannot be simplified out until statement 20, the branch target for `m2`, is reached. At that time, both variables can be eliminated by simplification and the resulting statement is unguarded.

Note that the price paid for this streamlined simplifier is that some conditions, like the one for statement 10, are left in a form longer than needed. For example, the guard for 10 could be represented by the Boolean expression

```
.NOT.m2.OR.m1
```

This is a small price to pay for avoiding an exponential simplifier.

Now we present the algorithm for simplification that results from these observations.

1. Each condition in the form of a term has a *key factor*, which is the most recently introduced variable or its negation. This factor is prepended on the current condition at the time the variable is introduced. Note that the current condition may be a disjunction of several terms.
2. Whenever a new variable is introduced because of a conditional branch, two copies of the current condition are made. The compiler-generated variable associated with the new condition is prepended to one of these copies and the result is added to a list associated with the branch target statement. The negation of the variable is anded onto the other condition, which becomes the new current condition.
3. When an unconditional branch is reached, the current condition is added to the list of conditions for the branch target statement.
4. When a branch target is reached, consider the collection of conditions associated with that target in step 2, the terms of any condition associated with the target in step 3 and each of the terms in the current condition. While there exists a pair of conditions with the same key variable (if this happens, it can be shown that one must be the exact negation of the other), strip the key variable off of one of the conditions and introduce each of its terms into consideration. When this process terminates, the disjunction of the collection of terms remaining is the new current condition.

To illustrate this procedure, consider the example above. Before we process statement 1, the current condition is .TRUE., which we represent by an empty condition. After statement 1 is processed, statement 10 has the condition "`m1`" on its list and the current condition is "`.NOT.m1`". At statement 3, we add the condition

```
m2.AND..NOT.m1
```

to the list for statement 20 and we make

```
.NOT.m2.AND..NOT.m1
```

the new current condition. When we reach statement 10, we have the conditions:

```
.NOT.m2.AND..NOT.m1  and

m1
```

which are disjoined to make the new current condition. A simple examination of the key variables tells us that no simplification is possible here. Finally, we reach statement 20, with the terms:

```
m2.AND..NOT.m1,

.NOT.m2.AND..NOT.m1 and

m1
```

The first two conditions have the same key variable, and are combined to yield:

```
.NOT.m1 and

m1
```

which have the same key variable. The result simplifies to `.TRUE.`.

FIGURE 7.6 Simplification.

```
procedure Simplify(cond);
    /* cond is a disjunction of terms, each term having a key variable */
    /* seen is a set of variables that have been seen as key variables  */
    /*      in the current condition.                                   */
    /* sofar is the set of terms that have been added so far to the list */
    /*      for the output condition.                                   */
    /* worklist is the collection of terms currently under consideration */
    /*      the output current condition.                               */

    seen := Ø; sofar := Ø;
    worklist := the set of terms in cond;
    while worklist≠Ø do begin
        let t be an arbitrary element of worklist;
        worklist := worklist – {t};
        let k be the key variable for t;
        if k ∈ seen then
            take the term q corresponding to k from sofar;
            sofar := sofar – {q};
            strip the key variable factor from q;
            put all the disjoined terms in the result back into worklist;
        else
            sofar := sofar ∪{t};
        endif;
    end;
```

return the disjunction of terms in *sofar*;
end *Simplify*;

The simplification algorithm, for use in the forward IF conversion procedure, is given in Figure 7.6. At each branch target, this algorithm does an amount of work proportional to the number of terms in the current condition. This is roughly proportional to the number of control flow paths past the current point in the program—the number of edges severed by a cut through the control flow graph at this point. Although this can grow large, it is unlikely to be more than a small integer in real programs.

7.2.8 IF reconstruction

Describe the fundamentals of reconstruction of forward branches. Leave exit branches for the section on recurrence breaking.

7.2.9 Interaction with Other Transformations

Discuss scalar expansion.Point out the effects of IF conversion on the calculation of preventing properties, by developing the properties of scalar dependences in the absence of IF conversion. (Include triangular loops from LINPACK).

7.2.10 Impact on Code Generation

This section needs to go somewhere else.

Programs with conditionals. This brings up the issue of what a reordering transformation is in the presence of control dependence. For the moment, we will solve this problem by assuming that each source of a control dependence stores the result of evaluation of the conditional that determines the control flow into an auxiliary memory location. This can never require more memory locations than the number of statement instantiations in the program and its loops. Each sink of a control dependence then tests the result of the conditional to determine whether it should execute. If its execution condition is false, it fails to read any other inputs or to write any outputs. This has the effect of reducing the problem of control dependence to a problem of data dependence, in which each control dependence becomes a true data dependence. By the results in the previous proof sections, each instruction in the reordered program will see exactly the same inputs as they did in the original program and hence the results will be exactly the same. QED.

7.3 Control Dependence

While IF conversion was very successful in vectorization, it has a number of disadvantages. First, it generated far too many conditional statements, which evaluate complex sets of conditions on each statement execution. If parallelization is unsuccessful, some sort of IF reconstruction is needed to reduce the cost of these conditional evaluations, not to mention improve the looks of the generated code. Second, the representation of conditions in the code is not particularly compact.

In general, it would be far better if we could do the analysis for parallelism before we actually performed IF conversion, which would make it possible to do the transformation only when it would have benefits for parallelization. In this section, we introduce "control dependence," a type of dependence that reflects control. We also show how both vectorization and parallelization algorithms can be adapted to make use of control dependence.

Definition 7.1. A statement y is said to be *control dependent* on another statement x if

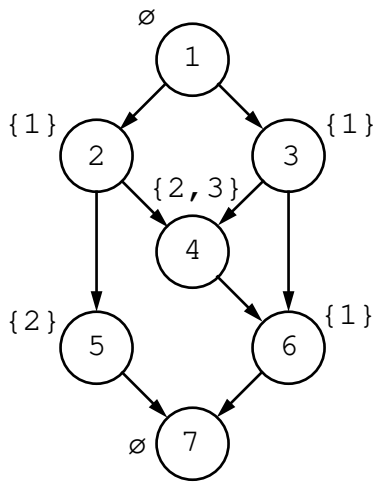
(1) there exists a non-trivial path from x to y such that every statement $z \neq x$ in the path is postdominated by y and

(2) x is not postdominated by y.

The intuition behind this definition is that x must be a conditional branch such that, if the branch is taken in one direction, y must be executed, while if it goes the other way, control can avoid y.

In discussing control dependence, it is convenient to view it as if it were a property of blocks of statements, because every statement in the same basic block in a program has the same control dependences. Consider the example in Figure 7.7.

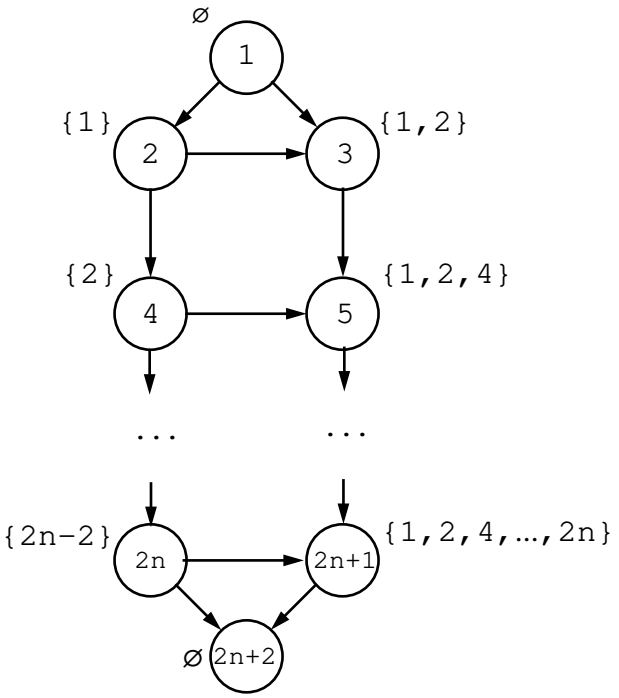
FIGURE 7.7 Control dependence example.



In this figure, each node represents a basic block and each edge represents a possible transfer of control. The set of blocks on which a given block is control dependent is specified by each node. It is interesting to note that neither block 4 or block 5 depends on block 1, because 1 cannot force execution of either block by transferring to one of its successors. Furthermore, block 6 does not depend on block 3 because there is no way to avoid executing block 6 once control reaches block 3.

It is possible for the size of the control dependence graph to be much larger than the control flow graph. For example, the family of graphs depicted in Figure 7.8.

FIGURE 7.8 Quadratic growth of the control dependence graph..



In this example, each node on the left hand sided of the diagram is control dependent on only one other node. However, each node on the right hand side is control dependent on node 1 plus every even-numbered node that can reach it. Thus the total number of control dependences for a graph with $2n$+2 nodes is n for the left hand side plus

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

For a total of

$$n + \sum_{i=1}^{n} i = \frac{n(n+3)}{2}$$

or $O(n^2)$ control dependences.

7.3.1 Control Dependence Construction

We now turn to the problem of automatically generating a control dependence graph from a control flow graph. The following algorithm, due to Cytron et. al. [Cytron-SSA] is the fastest known algorithm for the general construction.

FIGURE 7.9

Control dependence construction algorithm.

ConstructCDG(G,CD)

INPUT:
G is the input control flow graph
OUTPUT:
CD(*x*) is the set of blocks on which *x* is control dependent
INTERMEDIATE:
ipostdom(*x*) is the immediate postdominator of block *x* in the control flow graph *G*.
ALGORITHM:

L_1: find the immediate postdominator relation *ipostdom* for the control flow graph *G*; (For a control flow graph with a single exit, this relation forms a tree, with the exit node as the root.)
let *l* be a topological listing of the postdominator tree such that, if *x* postdominates *y*, then *x* comes before *y* in *l*.

L_2: **while** $l \neq \emptyset$ **do begin**
let *x* be the first element of *l*;
remove *x* from *l*;

L_3: **for all** control flow predecessors *y* of *x* **do**
if *ipostdom*(*y*) ≠ *x* **then** *CD*(*x*) = *CD*(*x*) ∪ {*y*} **end if;**

L_4: **for all** *z* such that *ipostdom*(*z*) = *x* **do**
for all *y* ∈ *CD*(*z*) **do**
if *ipostdom*(*y*) ≠ *x* **then** *CD*(*x*) = *CD*(*x*) ∪ {*y*} **end if**
end while
end *ConstructCD*

Note that *CostructCD* depends on the construction of a postdominator tree.

Definition 7.2. A node *x* in directed graph *G* with a single exit node *postdominates* node *y* in *G* if any path from *y* to the exit node of *G* must pass through *x*.

Algorithms for computing postdominators in a directed graph have been developed by a number of researchers [papers]. The fastest of these requires $O(E\alpha(N,E))$ time in the worst case, where N is the number of nodes and E is the number of edges in the control flow graph and α is a function that grows more slowly that any log power and is related to an inverse of Ackerman's function . Experiments have shown that algorithms with this time bound are effectively linear in the size of the input graph.

Theorem 7.1. Algorithm *ConstructCD* correctly constructs the control dependence relation. That is, $y \in CD(x)$ if and only if x is control dependent on y.

Proof. If. Only if. □

Theorem 7.2. Not counting the construction of postdominators, algorithm *ConstructCD* requires $O(\text{MAX}(N+E, |CD|))$ time in the worst case.

Proof. The topological sort takes $O(N+E)$ time, while the header of the loop at label L_2 is executed once for every node in the control flow graph or $O(N)$ times. The loop at label L_3 is entered once for each control flow predecessor of each node, for a total of $O(E)$ times, and since its body can be implemented in constant time, the total time required by the loop is $O(E)$.

The loop at label L_4 is more complicated. The loop is executed once for each edge in the postdominator tree, but since each node has only one postdominator, the loop header is executed only $O(N)$ times. The inner loop is executed at most once for each edge in the control dependence graph, so the loop nest takes $O(|CD|)$ time. The overall time bound follows immediately. □

7.3.2 Code Generation in the Presence of Control Dependence

Let us now turn to the problem of code generation in the presence of control dependence. For this purpose, there are two issues:

1. How can we adapt the transformations used in code generation to work with control dependence?
2. Once we have transformed a program, how do we generate code for a program represented by statements, data dependences and control dependences? In other words, how do we construct control flow from control dependences.

7.3.2.1 Control Dependence and Transformations

For the moment, we will restrict ourselves to the consideration of loop-independent control dependences, which arise from control flow within a single loop. Loop-carried control dependences arise only from loop-exit branches discussed in Section 7.2.3.

It is fairly easy to see that the correctness of most loop transformations are not affected. by control flow within the loop. This is true for loop skewing, loop reversal, strip mining and index-set splitting (if renaming is used to avoid duplicate labels). It is also true for loop interchange if the control dependences are loop-independent with respect to both loops—that is, control flow is contained within both loops. Finally, fusion of two loops with no exit branches offers no problems.

The principle difficulty in generalizing loop transformations arises from loop distribution. Recall that most code generation schemes begin by distributing a single loop into a collection of loops of the finest granularity allowed by the pattern of dependences in the original loop. This is problematic if the resulting code has a control dependence between the two of the new loops. For example, in the code below

```
     DO I = 1, N
S1      IF (A(I).LT.B(I)) GOTO 20
S2         B(I) = B(I) + C(I)
20      CONTINUE
     ENDDO
```

The only dependence in this loop is a control dependence from S_1 to S_2.Therefore, a maximal distribution would leave these two statements in separate loops. How can we express the program that results from this transformation? Clearly, the following would be wrong because we do not want to skip all iterations of S_2 simply because one iteration of S_1 happens to take the branch:

```
     DO I = 1, N
S1      IF (A(I).LT.B(I)) GOTO 20
     ENDDO

     DO I = 1, N
S2         B(I) = B(I) + C(I)
     ENDDO

20      CONTINUE
```

Somehow, we need to capture the decision made at S_1 and use it to control the execution of S_2. This suggests a transformation similar to if con-

version which would save the result of the conditional in a logical array which could be interrogated later. The following code might result from such a process.

```
     DO I = 1, N
S1      e(I) = A(I).LT.B(I)
     ENDDO

     DO I = 1, N
S2      IF (e(I).EQ..FALSE.) B(I) = B(I) + C(I)
     ENDDO
```

This transformation differs from if conversion in that it only need be invoked when the final loop distribution has been chosen and control dependences that were previously loop independent cross loop boundaries.

A more complex illustration of the problem that control dependences present for loop distribution is provided by the example code in Figure 7.10

FIGURE 7.10 An example for loop distribution in the presence of control dependence.

```
     DO I = 1, N
1       IF (A(I).NE.0) THEN
2          IF (B(I)/A(I).GT.1) GOTO 4
        ENDIF
3       A(I) = B(I)
        GOTO 8
4       IF (A(I).GT.T) THEN
5          T = (B(I) - A(I)) + T
        ELSE
6          T = (T + B(I)) - A(I)
7          B(I) = A(I)
        ENDIF
8       C(I) = B(I) + C(I)
     ENDDO
```

The control flow graph for this example is given in Figure 7.11 and the corresponding control dependence graph is shown in Figure Figure 7.12. When we add data dependences, marked by the dotted arrows in

FIGURE 7.11 Control flow graph for Figure 7.10.

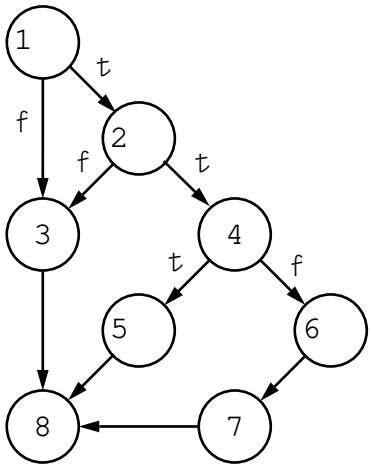


FIGURE 7.12 Control dependence graph for Figure 7.10.

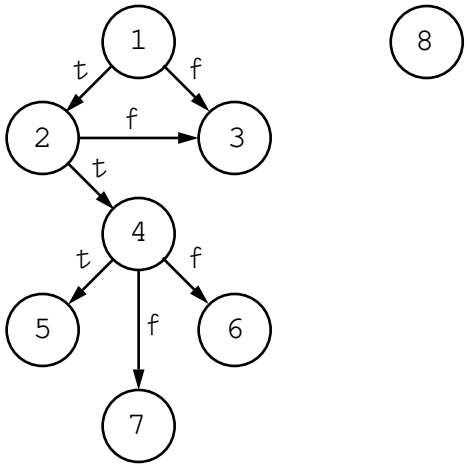
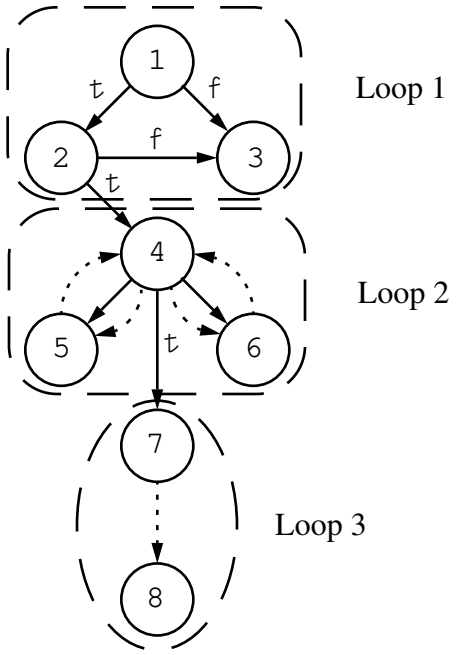


FIGURE 7.13 Control and data dependence graph for Figure 7.10.



In Figure 7.13, the dashed regions indicate the loops suggested by loop distribution. The problem arises because of the control flow edges that cross loop boundaries. These must be converted to data dependences so that correct code can be generated for the three loops.

To do this we will need to capture, on each iteration, the branch decision made at statements that are the sources of loop-crossing control dependence arc. We will do this by the use of arrays called *execution variables.* In the example above, we will need two execution variables: E2(I), to hold the result of branches at statement 2, and E4(I), to hold the result of branches at statement 4. To understand the values that an execution variable may take on, consider the cases that may hold at each of these two statements. At statement 2, three cases may hold:

1. statement 2 is executed and the true branch to statement 4 is taken,
2. statement 2 is executed and the false branch to statement 3 is taken, or
3. statement 2 is never executed because the false branch is taken at statement 1.

In general these three conditions can hold at any statement that is the source of a loop-crossing control dependence. Therefore, it seems natural

to use three values for an execution variable element: *true, false* or *undefined* (sometimes represented by the symbol "⊤"), corresponding to the three conditions above. Once it is determined which execution variables will be needed, all such variables are initialized to ⊤. Then the program is converted to set the execution variables at appropriate places in the code. The algorithm in Figure 7.14 performs the conversion. Note that this algorithm does not generate parallel code, but selectively converts control dependences to data dependences, changing some statements in the process.

FIGURE 7.14

Execution variable and guard creation.

DistributeCDG(G,G_{cd},P)

INPUT:

G is the input control flow graph
G_{cd} is the input control dependence graph
$P = \{P_1,P_2, ...,P_k\}$ is a list of partitions corresponding to loops after distribution

OUTPUT:

modified G_{cd} with execution variables

ALGORITHM:

L_1: **for each** partition P_i **do begin**
for each $n \in P_i$ such that there exists an edge $(n,o)_l \in G_{cd}$, where l is the true/false label of the control dependence edge, such that $o \notin P_i$
do begin
insert "`EV(I)= ⊤`" into P_i at the beginning;
let *test* be the branch condition for *n;*
if there exists an edge $(n,m)_l \in G_{cd}$, where $m \in P_i$ **then**
replace *n* with
"`EV`$_n$`(I) =` *test*"
"`IF (EV`$_n$`(I).EQ..TRUE.)`"
/* Conditions now depend on this latter test */
else
replace *n* with
"`EV`$_n$`(I) =` *test*"
endif
for each $P_k \neq P_i$ containing a successor of *n* **do begin**
create a new, unique statement *N*:
"`IF (EV`$_n$`(I).EQ.`*l*`)`";
add *N* to P_k;
insert data dependences for `EV`$_n$;

```
                    for each (n,q)_l such that q ∈ P_k do begin
                        /* Update control dependences */
                        delete (n,q)_l from G_cd;
                        add (N,q)_true to G_cd;
                    end
                end
            end
        end
end DistributeCDG
```

If we apply this algorithm to the example in Figure 7.10 and then generate the corresponding code, we get the program shown in Figure 7.15.

FIGURE 7.15 Example code from Figure 7.10 after loop distribution.

```
     PARALLEL DO I = 1, N
        E2(I) = ⊤;
1       IF (A(I).NE.0) THEN
2          E2(I) = (B(I)/A(I).GT.1)
        ENDIF
3       IF(E2(I).NE..TRUE.) A(I) = B(I)
     ENDDO
     DO I = 1, N
        E4(I) = ⊤;
        IF (E2(I).EQ..TRUE.) THEN
           E4(I) = (A(I).GT.T)
4          IF (E4(I).EQ..TRUE.) THEN
5             T = (B(I) - A(I)) + T
           ELSE
6             T = (T + B(I)) - A(I)
           ENDIF
        ENDIF
     ENDDO
     PARALLEL DO I = 1, N
        IF (E4(I).EQ..FALSE.) THEN
7          B(I) = A(I)
        ENDIF
8       C(I) = B(I) + C(I)
     ENDDO
```

It is not entirely clear how the code in Figure 7.15 was generated. That is the subject of the next section.

7.3.2.2 Generating Code from a Program with Control Dependence

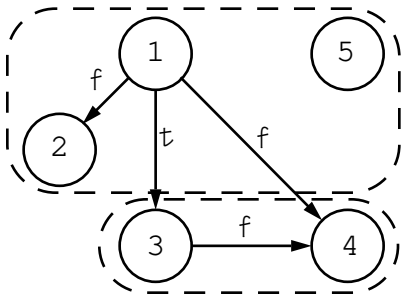
Once all the transformations above, including loop distribution, have been performed, we need to generate code that a machine can execute.

For the purpose of exposition, we let our target language be Fortran itself. It is a surprisingly difficult problem to generate a correct Fortran program from a dependence graph as it exists after loop distribution and other transformations in the presence of control flow. To see why, let us consider an example.

```
     DO I = 1, N
S1      IF (p1) GOTO 3
        S2
        GOTO 4
3       IF (p3) GOTO 5
4       S4
5       S5
     ENDDO
```

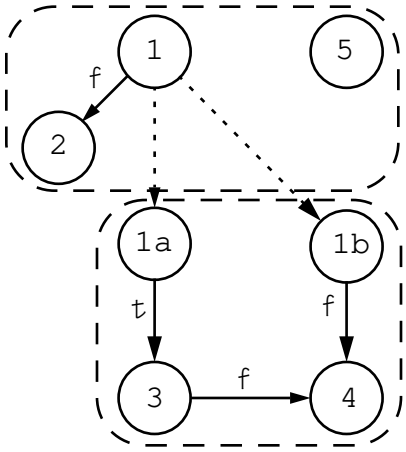
For this code fragment, we have the control dependence graph shown in Figure 7.16, which includes partitions desired after loop distribution.

FIGURE 7.16 Control dependence graph for sample code fragment after distribution.



Loop distribution will produce the control dependence graph in Figure 7.17. Nodes 1a and 1b are two new nodes, data dependent on node 1, generated in the second partition by the distribution algorithm.

FIGURE 7.17 Control dependence graph for sample code fragment.



Now if we attempt to generate code for these two partitions, the first partition becomes:

```
DO I = 1, N
   E1(I) = p1
   IF (E1(I).EQ.FALSE) THEN
      S2
   ENDIF
   S5
ENDDO
```

This is pretty straightforward from the distributed control flow graph. However, the second partition is problematic because statement 4 is in two different conditionals, so we cannot put it in a single structured conditional statement. At least one of the two conditionals 1a and 1b must be unstructured. In this case, we pick 1b—the guard with the highest numbered control dependence successor—to generate as a GOTO. The rest of the statement can be generated as a structured IF statement. Note we are taking advantage of an ugly property of Fortran that allows you to branch into the range of a conditional statement.

```
      DO I = 1, N
         IF(E1(I).EQ..FALSE.) GOTO 4
         IF(E1(I).EQ..TRUE.) THEN
            IF (.NOT.p4) THEN
4              S4
            ENDIF
```

```
      ENDIF
   ENDDO
```

Our goal is to develop a general algorithm for correct code generation for code with control dependences. Such an algorithm would be easy if we were only dealing with "structured code" arising from DO-loops and IF statements, without any GOTO statements, because of the following Theorem.

> **Theorem 7.3.** The body of a DO-loop that contains no gotos or other DO-loop will give rise to a control dependence graph in which no statement is control dependent on more than one other statement in the loop body.

Proof. The code in the loop body consists entirely of normal statements or IF-statements. If there are no IF-statements, no statement in the body has a control dependence on any other statement. Within an IF-statement, each statement in the THEN or ELSE part of the statement that is not contained in a nested IF is control dependent *only* on the IF part, which includes the condition. □

This theorem leads to a very simple, recursive code generation procedure for structured code. We begin with a numbering of the vertices in the program that is based on the original ordering of nodes in the source program. This ordering has the property that if node m is control dependent on node n, then $order[n] < order[m]$.

FIGURE 7.18

Structured code generation.

CDGgenCode(G_{cd},*P*)

INPUT:

G_{cd} is the input control dependence graph
P is the collection of statements in the loop body

OUTPUT:

modified G_{cd} with execution variables

ALGORITHM:

procedure *gencode*(n)

/* Generate code recursively from top to bottom */
if n is of the form IF(p) **then**
let T be the set of statements m such that $(n,m)_{true} \in G_{cd}$ **;**
let F be the set of statements m such that $(n,m)_{false} \in G_{cd}$ **;**
if $T \neq \emptyset$ **then**

```
                generate "IF (p) THEN"
                genset(T);
                if F ≠ ∅ then
                    generate "ELSE";
                    genset(F);
                 endif;
                generate "ENDIF"
            else if F ≠ ∅ then
                generate "IF (.NOT. p) THEN"
                genset(F);
                generate "ENDIF"
            endif;

        else /* n is not a conditional */
            generate n
        endif;
end gencode;

procedure genset(S);

        /* Generate code for each statement in a set, in order */
        while S ≠ ∅ do begin
            let m be the element of S such that order[m] is minimal;
            delete m from S;
            gencode(m);
        end
end genset

let P0 be the set of statements in P that have no control dependence
    predecessors;
genset(P0);

end CFGgenCode
```

This routine simply generates code for each of the subtrees forming the control dependence graph in the order that the roots of these subtrees appeared in the original program.

How can we generalize this approach to the case of unstructured code. The scheme we shall use is to examine the successors of each IF node to see if either successor has more than one control dependence predecessor. Suppose that the true branch has such a successor *m*. In this case, we will generate a conditional GOTO statement "IF (P) GOTO *label*(*m*)," where the special routine *label*(*m*) returns the label for a given control dependence graph node, creating one if it does not already exist.

This approach leads to the following procedure for unstructured code generation.

FIGURE 7.19 Structured code generation.

```
CDGgenUCode(G_cd,P)
  INPUT:
      G_cd is the input control dependence graph
      P is the collection of statements in the loop body
  OUTPUT:
      Fortran program corresponding to G_cd
  ALGORITHM:

procedure gencode(n)

    /* Generate code recursively from top to bottom */
    if n is of the form IF(p) then
        let T be the set of statements m such that (n,m)_true ∈ G_cd ;
        let F be the set of statements m such that (n,m)_false ∈ G_cd ;
        if T ≠ ∅ then
            generate "IF (p) THEN"
            genset(T, n);
            if F ≠ ∅ then
                generate "ELSE";
                genset(F, n);
             endif;
            generate "ENDIF"
        else if F ≠ ∅ then
            generate "IF (.NOT. p) THEN"
            genset(F, n);
            generate "ENDIF"
        endif;

    else /* n is not a conditional */
        generate n
    endif;
end gencode;

procedure genset(S,n);

    /* Generate code for each statement in a set, in order */
    /* Node n is the control dependence predecessor */
    while S ≠ ∅ do begin
        let m be the element of S such that order[m] is minimal;
```

```
            delete m from S;
            if m has more than one predecessor in G_cd then
                generate "GOTO label(m);
                delete (n,m)_l from G_cd
            else
                gencode(m);
            endif
        end
end genset

let P_0 be the set of statements in P that have no control dependence
    predecessors;
genset(P_0, null);

end CDGgenUCode
```

Note that the change from the structured case is restricted to the routine *genset* and its calling sequence, which is modified to generate a GOTO statement to any statement in its range that has more than one control dependence predecessor.

Theorem 7.4. Procedure *CDGgenUCode* in Figure 7.19 generates correct code for an arbitrary control dependence graph of a loop body with no nested DO loops.

Proof. First we must establish that the program visits and generates code for every statement in the control dependence graph. This follows from the observation that every statement with no control dependence predecessor is visited and every other node is visited after the last of its control dependence predecessors is visited, at which point it has only one predecessor because previous invocations of *genset* have deleted any others.

It should be clear that, in the case of a control dependence graph that is a collection of trees, the code generated is the same as that generated for the unstructured case. So the key question is: Do we get correct behavior for unstructured code? Since the code in the loop body must be a directed acyclic graph, there can only be one control flow path by which a particular node is reached in any given loop iteration. If that path passes through the first of several control dependence predecessors of a given statement *m* it will reach *m* via the goto that is inserted in *genset.* Since nodes that are unrelated by control dependence will be generated in the order in which they appeared in the original program, every execution path through the loop body will be correctly preserved.

One question remains: Suppose that *genset* is invoked on a set with more than one statement and its execution results in a GOTO being inserted before the list of statements has been exhausted. We claim that all of the same statements will be generated together at the target of the GOTO. This is true because the set consists of a collection of statements that form a block of statements in the control dependence graph. This is established by the following Lemma.

> **Lemma 7.1.** Suppose there exist four statements n_1, n_2, m_1 and m_2 and true/false labels *l1* and *l2* such that $(n_1,m_1)_{l1}$, $(n_1,m_2)_{l1}$ and $(n_2,m_1)_{l2}$ are all in G_{cd}. Suppose also that, in the original program, m_1 always executes before m_2 if they both execute. Then $(n_2,m_2)_{l2}$ must also be in G_{cd}.

Proof. $(n_1,m_1)_{l1}$, $(n_1,m_2)_{l1} \in G_{cd}$ means that there is a successor of n_1 on the *l1* branch that is postdominated by both m_1 and m_2. Hence, either m_1 *postdominates* m_2 or vice versa, but since m_2 always executes after m_1, m_2 postdominates m_1.

$(n_2,m_1)_{l2} \in G_{cd}$ means that the successor of n_2 on the *l2* branch is postdominated by m_1. If $(n_2,m_2)_{l2} \notin G_{cd}$, then the same successor of n_2 is not postdominated by m_2 but this is impossible since m_2 postdominates m_1. QED.

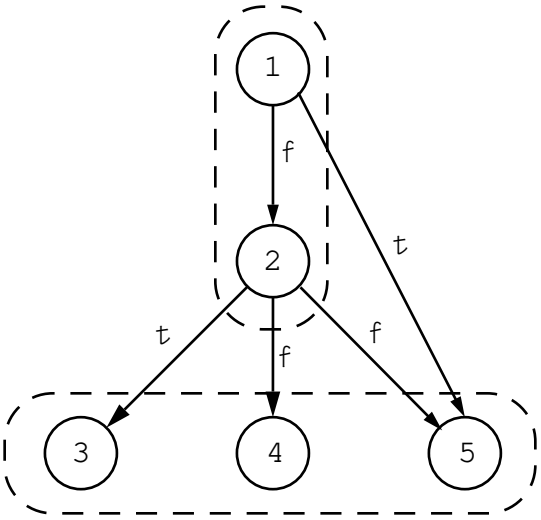
This lemma also establishes Theorem 7.4. QED.

We conclude our treatment of code generation with a final example.The original code is as follows:

```
     DO I = 1, N
S1      IF (p1) GOTO 5
S2      IF (p2) THEN
           S3
        ELSE
           S4
5          S5
        ENDIF
     ENDDO
```

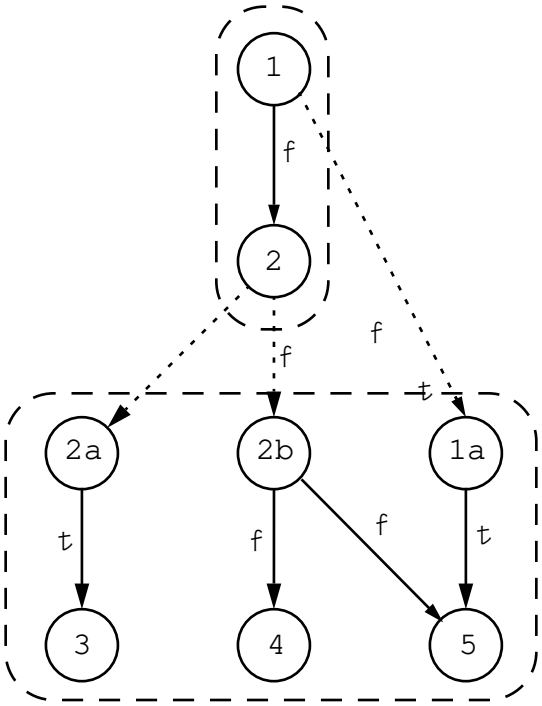
The control dependence graph for this example program is shown in Figure 7.20, where the dashed regions represent the required data distribution.

FIGURE 7.20 Control dependence graph for example code fragment.



After distribution according to Algorithm *DistributeCDG* in Figure 7.14 this becomes

FIGURE 7.21 Control dependence graph for example code fragment.



The code for the first of these two loops is as follows:

```
     DO I = 1, N
        E2(I) = ⊤
S1      E1(I) = p1
        IF (E1(I).EQ..FALSE.) THEN
S2         E2(I) = p2
        ENDIF
     ENDDO
```

Before showing the code generated for the second loop, we need to make a small point about the order assigned to nodes generated as part of the distribution. This ordering is determined by two principles:

1. Generated nodes inherit the ordering of the nodes from which they are derived. Thus node 1a is chosen before either node 2a or 2b.
2. When selecting between two nodes derived from the same original node (e.g., 2a and 2b), if the lowest-numbered successor of both nodes has multiple predecessors in the control dependence graph pick the

node with the highest-numbered first successor, otherwise pick the node with the lowest-numbered first successor.

Rule 1 is essential, but Rule 2 is not—its only purpose is to make the final result slightly more readable. If ties are broken randomly, the result will still be correct.

Here is the final code for the second loop, as generated by Algorithm *CDGgenUCode* in Figure 7.19.

```
     DO I = 1, N
        IF (E1(I).EQ..TRUE.) GOTO 5
        IF (E2(I).EQ..TRUE.) THEN
           S3
        ELSE
           S4
5          S5
        ENDIF
     ENDDO
```

This code is consistent with the input code, a satisfying result.

7.4 Historical Notes

If conversion was developed by Allen and Kennedy. Control dependence has a long history. The algorithm for construction is due to Cytron et al. Loop distribution in the presence of control dependence and the code generation algorithm are due to Kennedy and McKinley.