Conversion of Control Dependence to Data Dependence[1]

J.R. Allen
Ken Kennedy
Carrie Porterfield
Joe Warren

Department of Mathematical Sciences
Rice University
Houston, Texas 77251

Abstract

Program analysis methods, especially those which support automatic vectorization, are based on the concept of interstatement dependence, where a dependence holds between two statements when one of the statements computes values needed by the other. Powerful program transformation systems that convert sequential programs to a form more suitable for vector or parallel machines have been developed using this concept [AllK 82, KKLW 80].

The dependence analysis in these systems is based on data dependence. In the presence of complex control flow, data dependence is not sufficient to transform programs because of the introduction of control dependences. A control dependence exists between two statements when the execution of one statement can prevent the execution of the other. Control dependences do not fit conveniently into dependence-based program translators.

One solution is to convert all control dependences to data dependences by eliminating goto statements and introducing logical variables to control the execution of statements in the program. In this scheme, action statements are converted to IF statements. The variables in the conditional expression of an IF statement can be viewed as inputs to the statement being controlled. The result is that control dependences between statements become explicit data dependences expressed through the definitions and uses of the controlling logical variables.

This paper presents a method for systematically converting control dependences to data dependences in this fashion. The algorithms presented here have been implemented in PFC, an experimental vectorizer written at Rice University.

[1]Support for this research was provided by IBM Corporation.

1. Motivation

1.1. Dependence and Vectorization

The development of computer architectures with powerful vector processing units has spawned an interest in languages that permit the explicit specification of vector and array operations. In fact, it seems clear that the next ANSI standard for Fortran (hereafter referred to as Fortran 8x) will contain such explicit vector operations. This language should enable programmers to write high level programs that fully utilize vector hardware [ANSI 81].

Unfortunately, the many millions of lines of Fortran developed prior to Fortran 8x were written without the benefit of explicit vector operations. If this existing code is to use vector hardware effectively, it must be translated to a language from which vector operations may be invoked -- either vector machine language or a high level language with vector operations. This translation must replace the implicit vector operations in the original Fortran program with explicit vector operations. At Rice, we are developing a translator, known as Parallel Fortran Converter (or PFC), that converts Fortran 66 and 77 programs into equivalent vector programs in Fortran 8x [AllK 82].

The natural place to look for vector operations is the Fortran DO loop. Ideally, each assignment in a DO loop would be converted to a vector assignment by translating the subscripts to vector iterators. For example, the loop

```
      DO 100 I = 1, 100
         A(I) = A(I) + C
  100 CONTINUE
```

could be translated to the statement

```
A(1:100) = A(1:100) + C
```

However, the translation process is not quite that simple, because the semantics of vector assignment in Fortran 8x require "fetch before store." That is, while a scalar assignment in a loop intermixes loads and stores, a vector assignment behaves as if all components of the right hand side are fetched before any component of the left hand side is stored. The following loop illustrates this difference.

```
      DO 100 I = 1, 100
         A(I) = A(I-1) + B(I)
  100 CONTINUE
```

Since the intent is that the component of A computed on one iteration be used as input on the next,B the statement cannot be simulated by a vector statement of the form

```
A(1:100) = A(0:99) + B(1:100)
```

with "fetch before store" semantics. By contrast, the statement in our first example loop did not intermix loads and stores in the same array, so its effect can be simulated with vector semantics.

Thus, a statement that computes a value on one iteration of the loop that is used directly or indirectly by the same statement on another iteration cannot be vectorized by transliteration; otherwise, the statement may be vectorized. Correctly distinguishing between these two cases requires a study of the flow of values between uses and definitions.

Classical data flow analysis models the relationship between definitions and uses of variables as a directed graph in which each vertex represents a statement and each edge a data flow link from definition to use; these links are often called def-use chains [Kenn 78]. However, following Kuck [Kuck 77], the term dependence denotes the relationship between a statement S_2 that uses the value that S_1 might have created. S_2 directly depends upon S_1 if the value computed by S_1 might be an input to S_2 at run time. S_2 depends upon S_1 if there exists a sequence of statements $X_1,\ldots,X_n$ such that $X_0=S_1$, $X_n=S_2$, and X_{i+1} depends directly upon X_i for all i, $0\leq i<n$. In these terms, a statement can be vectorized only if it does not depend upon itself.

PFC employs a slightly more sophisticated concept, called loop-carried dependence, which associates each dependence with the iteration of a particular loop. For example, the dependence of S_1 on itself in

```
      DO 100 I = 1, 10
         DO 90 J = 1, 10
S1          A(I,J) = A(I-1,J) + ...
 90      CONTINUE
100   CONTINUE
```

is clearly due to the loop on I. Within any specific iteration of the I loop, S_1 does not use its own results; only when I is incremented does S_1 fetch from a location of A that it has also stored in. Since the dependence (and hence the cycle) disappears when the I loop is run sequentially, S_1 can be correctly vectorized in the J loop to produce

```
      DO 100 I = 1, 10
S1       A(I,1:10) = A(I-1,1:10) + ...
100   CONTINUE
```

Using the concept of loop-carried dependence, Kennedy developed a recursive algorithm to vectorize statements in the maximum number of dimensions permitted by its dependence relations. [Kenn 80].

1.2. Control Dependence

Data dependence alone is not sufficient to describe all important considerations in vectorization. Consider the following loop:

```
      DO 100 I = 1, N
S1       IF (A(I).GT.0) GO TO 100
S2          A(I+1) = B(I) + 10
  100 CONTINUE
```

The theory of data dependence would not preclude vectorization of S_2. Certainly S_1 directly depends on S_2. But S_2 does not depend on itself or on S_1 for any of its inputs. Nevertheless, neither S_1 nor S_2 can be correctly vectorized because of the existence of a control dependence of S_1 on S_2. That is, the outcome of the test in S_1 determines whether S_2 will be executed. When the control dependence is considered, both S_1 and S_2 depend on themselves indirectly.

Unfortunately, control dependence does not fit nicely into the dependence machinery of PFC because the dependence is not associated with any variable. When the same loop is rewritten as

```
      DO 100 I = 1, N
         BR1 = A(I).GT.0
         IF (BR1) A(I+1) = B(I) + 10
  100 CONTINUE
```

(thereby associating the dependence with the variable BR1) the problem becomes much simpler. By viewing the variables in the condition controlling S_2 as inputs to the statement, the relationship of these two statements is now clear in terms of data flow. Control dependence has been completely converted to data dependence.

The beauty of this scheme is that conditional assignments are straightforward to vectorize if the scalar conditions are expanded into arrays. For example, consider a slight variation on our example loop.

```
      DO 100 I = 1, N
         BR1 = A(I).GT.0
         IF (BR1) A(I) = B(I) + 10
  100 CONTINUE
```

This version could be transformed to vector form by using the Fortran 8x WHERE statement.

```
BR1(1:N) = A(1:N).GT.0
WHERE (BR1(1:N)) A(1:N) = B(1:N) + 10
```

Many vector machines have hardware to support conditional vector operations, usually via a logical mask to select the positions in which the computation is to be applied.

By generalizing this idea into a method for converting control dependences to data dependences, PFC can vectorize statements in loops which contain conditional transfers. The IF conversion phase of PFC is responsible for this transformation.

2. Fundamentals of IF Conversion

Central to IF conversion is the notion that Fortran Statements can be classified into four groups:

(1) action statements -- statements which cause some change in the state of the computation or produce some important side effect. Examples: assignment, read, write, call.

(2) branch statements -- statements which make an explicit transfer of control to another location in the program. Examples: goto, computed goto, assigned goto. Note that call is treated as an action statement because within a given module it may be viewed as a macro-action.

(3) iterative statements -- statements which cause another statement or a block of statements to be iterated. Example: DO statement.

(4) placeholder statements -- statements which take no action but which can be used as placeholders for the computation. Example: CONTINUE.

Notice that the Fortran IF statement has no place in our classification. The reason is that we view the IF clause as a qualifier that can be attached to any action or branch statement. In other words, every action or branch statement can be viewed as a conditional statement.

The IF conversion phase of PFC attempts to eliminate all goto statements in the program. The execution order of the original program is maintained by computing a logical condition for each action statement. This condition is called a guard.

Definition: The guard for an action or conditional action statement is a Boolean expression which represents the conditions under which the statement is executed. That is, when control reaches the statement, the original statement is executed if and only if its guard evaluates to true. □

The original program is transformed by replacing simple action statements with conditional action statements of the form:

IF (guard) statement

IF statements (other than conditional branches) can be replaced by IF statements in which the guard is conjoined to the original condition. If the guard of a statement is identically **true**, it can be written without the IF qualifier.

For the purpose of analysis, branches can be categorized into three types:

(1) exit branch: a branch that terminates one or more loops, as in

```
      DO 100 I = 1, 100
         IF (ABS(A(I)-B(I)) .LE. DEL) GOTO 200
         ...
  100 CONTINUE
      ...
  200 CONTINUE
```

(2) forward branch: a branch whose target occurs after the branch but at the same loop nesting level. Note that since branches into the range of a DO loop are not permitted, a branch to a label after the branch must be either a forward branch (if the label is at the same nesting level) or an exit branch (if the label is outside the loop in which the branch occurs).

```
        DO 100 I = 1, 10
           IF (A(I).EQ.0.0) GOTO 100
           B(I) = B(I) / A(I)
 100    CONTINUE
```

(3) backward branch: an branch to a statement occurring lexically before the branch but at the same nesting level, as in

```
  10    I = I + 1
        A(I) = A(I) + B(I)
        IF (I .LE. 100) GOTO 10
```

In accordance with this classification, IF conversion uses two different transformations to eliminate branches within the program.

(1) Branch relocation moves branches out of loops until the branch and its target are nested in

the same number of DO loops. This 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. Backward branches are left in place.

The following sections present these two techniques in more detail.

3. Exit branches

Exit branches differ from other branches in that exit branches affect the execution of statements both before and after the branch. That is, since a branch out of a DO loop terminates execution of the loop, it affects all the statements in the loop. Consider the following example:

```
          DO 100 I = 1,100
             S1
             IF (X(I)) GOTO 200
             S2
100       CONTINUE
          S3
200       S4
```

Once the jump is taken, the DO loop is terminated and neither statement S_1 nor S_2 will be executed thereafter. If the DO loop were not present, producing

```
          S1
          IF (X(I)) GOTO 200
          S2
          S3
200       S4
```

statement S_1 is completely unaffected by the branch. Thus, exit branches are more complicated than forward branches, since eliminating them requires modification of the guards of all statements within the loop exited.

If all exit branches can somehow be converted into forward or backward branches, then the problem of IF conversion becomes much simpler. In other words, if PFC can relocate the branches so that every branch is nested in exactly the same DO loops as its target, branch removal will then eliminate these branches naturally with forward branches.

The basic procedure used in branch relocation and elimination is the computation of a Boolean guard expression for each statement. This guard evaluates to true if and only if the statement would be executed in the original program. By converting the guard to a logical expression in Fortran and using it as a condition in an IF clause, PFC can then test for vectorization using only data dependence.

Guards in PFC are based on a system of formal logic. The atoms of this logical system are predicates expressing conditions that may hold at various points in the program. For example, one possible predicate is p = "A(I).LT.0 was true on the most recent execution of statement 300." If "A(I).LT.0" is the condition for a jump past statement 350, the predicate p should certainly be part of the guard for that statement. The operations which may be applied to predicates are conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$). Hence a guard might be the conjunction of several predicates, e.g.

$$p_1 \wedge p_2 \wedge \neg p_3$$

In order to separate the issue of correctness from the issue of simplification, we will distinguish between the logic used to represent guards internally and their actual appearance in the output language. In our logic, we can compute a provably correct guard for a particular statement; however, this does not imply that we can find, in a reasonable time, the most concise Fortran representation for that guard.

The duality of the logic of conditions and their external representation is mirrored by our implementation. We represent the guards internally in a form quite different from the external representation (see Section 7). Notationally, we will use the function μ to map the internal representation of conditions to a realization in the language being generated. An internal condition may have many external representations; we therefore assume that μ will choose one that is suitably concise. For example, μ might employ the Quine-McCluskey prime implicant simplification procedure to generate a simple external representation of a given internal guard [Quin 52, McCl 56]. The use of μ allows us to delay consideration of any simplification issues until Section 6.

Returning to branch relocation, movement of an exit branch out of a loop requires that the execution of each statement in the loop be guarded by an expression which will be true in the modified program only while the branch has not been taken in the original program. More generally, each statement will be guarded by an expression which is the conjunction of exit flags, denoted ex_i, where an exit flag is a Boolean variable associated with a particular branch in the original program. The exit flag ex_i is defined to be true at a statement if the branch associated with the flag would not have been taken before control reached the statement in the original program. In order to compute a realization for ex_i, we will introduce a corresponding logical variable EXi

into the program. EXi will be used to capture the condition controlling the loop exit each time that condition is evaluated, so that $\mu(ex_i)$ = EXi. We will use the convention that lower case variables represent conditions and upper case variables represent their realizations as Fortran logical variables.

In the case of branches out of a single loop, there is one exit flag for each exit branch. Upon entry to the loop, all exit flags are true, since the loop has not yet been exited. Each exit branch of the form

```
IF (P) GOTO S1
```

within the loop is associated replaced by an assignment of the form

```
EXi = .NOT. P
```

which captures the condition ex_i = "the exit branch would not have been taken at its most recent execution." A new branch of the form

```
IF (.NOT. EXi) GOTO S1
```

is generated immediately following the loop to simulate the effect of the branch in the loop. Note that this branch will be taken only if the exit branch would have been taken in the original program. Finally, the guards of all statements within the loop (including the newly generated assignment) are modified by conjoining each exit flag for that loop:

$$ex_1 \wedge ex_2 \wedge \ldots \wedge ex_n.$$

The overall effect is to arrange the modified program so that an exit flag is set to false whenever the corresponding exit branch in the original program would have been taken. Thus, once an exit flag becomes false, no other statement in the loop will be executed, even though the DO statement will continue to run iterations.

Here is the previous example after relocation:

```
        EX1 = .TRUE.
        DO 100 I = 1,100
           IF (EX1) S1
           IF (EX1) EX1 = .NOT. X(I)
           IF (EX1) S2
100     CONTINUE
        IF (.NOT. EX1) GO TO 200
        S3
200     S4
```

This method is easily extended to multiple loops by treating a branch out of more than one loop as a branch out of the outermost loop. Consider the following more complicated example:

```
        DO 200 I =1,100
50         S1
           DO 100 J=1,100
              S2
              IF X(I,J) GO TO 300
              S3
              IF Y(I,J) GO TO 50
              S4
100        CONTINUE
           S5
200     CONTINUE
300     S6
```

After the branch relocation, this code becomes

```
        EX1 = .TRUE.
        DO 200 I =1,100
50         IF (EX1) S1
           IF (EX1) EX2 = .TRUE.
           DO 100 J=1,100
              IF (EX1 .AND. EX2) S2
              IF (EX1 .AND. EX2) EX1 = .NOT. X(I,J)
              IF (EX1 .AND. EX2) S3
              IF (EX1 .AND. EX2) EX2 = .NOT. Y(I,J)
              IF (EX1 .AND. EX2) S4
100        CONTINUE
           IF (EX1 .AND. .NOT. EX2) GOTO 50
           IF (EX1) S5
200     CONTINUE
        IF (.NOT. EX1) GOTO 300
300     S6
```

This transformation is effected by applying the simple method to the first jump with respect to the outer loop and the second jump with respect to the inner loop. Note that the exit flags are mutually exclusive; that is, once any exit flag is set to false (indicating that an exit branch has been taken), no other exit flag in any loop that the corresponding jump would have left can be set to false. Hence, if a loop is implicitly terminated by an exit branch, that branch can be identified by scanning the exit flags for the one which is false.

The algorithm for branch relocation is given in Figure 1. The guard on every statement other than an IF is initially true. The algorithm proceeds by computing the loop guard for this loop, applying itself recursively to nested DO loops (which computes guards for the statements in those loops) then conjoining the loop guard for the current loop to the guard of every statement under its control.

After the procedure is called on every DO statement at the outmost level, no exit branches will remain in the program. To demonstrate the correctness of branch relocation, we must show two things:

(1) the algorithm removes all exit branches, and

(2) the modified version performs exactly the same computation as the original.

The first point follows rather trivially from statement S_1 of the algorithm. The body of loop

```
procedure relocate_branches (x);
   /* x is the DO statement for the loop        */
   /* loop_guard will be the conjunction of     */
   /*   all exit flags for the loop             */
   loop_guard = true;
S1:for each exit branch IF (P) GOTO S1
         that exits the loop headed by x do
   begin
      create a new unique exit flag exi
         with realization EXi;
      insert the assignment "EXi = .TRUE."
         prior to x;
      loop_guard = loop_guard ∧ exi;
      insert the branch "IF (.NOT. EXi) GOTO S1"
         after the loop;
S2:   replace the exit branch by
         the assignment "EXi = .NOT. P"
   end
   for each DO statement y contained in x do
       relocate_branches (y);
S3:for each non-DO statement y contained in x do
       guard(y) = guard(y) ∧ loop_guard;
end relocate_branches;
```

Figure 1: **Branch Relocation**

S_1 converts a particular exit branch to an assignment. Since no new exit branches are created by the procedure (the generated branches must be at the same level as their targets), and since S_1 is executed for each exit branch in a loop, the modified code will contain no exit branches.

The second point follows from two observations about the transformations being applied.

(1) The only difference between action statements in the original program and the modified program is that all exit flags for loops in which the statement is contained are conjoined to its guard.

(2) Each exit branch is replaced by an assignment statement that sets the corresponding exit flag to false if the condition controlling the branch is true - in other words, if the branch would have been taken the exit flag becomes false.

One important concern about correctness is that the transformation might have introduced side effects that would not have occurred in the original program. A possible source of such side effects is the computation of guard values. The branch removal algorithm is very careful to compute branch conditions at the point where they would have taken place in the original program and save them in logical variables. The computation of guards then amounts to evaluating logical expressions in these logical variables, thereby avoiding the problems of side effects.

Since all branches out of the loop have been eliminated, every DO loop in the modified program, once entered, will run its course - even though some exit flag is false and no real computation is being done. This is an essential part of the transformation, but it may have the unfortunate effect of unexpectedly long running times when the purpose of the DO loop iteration is to provide a bound large enough to insure that the loop would be terminated by a branch on detection of a special condition. Hopefully, the speedup gained from vectorization will more than offset this inefficiency.

Branch relocation is an elegant prepass to branch removal for many reasons. First, it makes no distinction between backward branches and forward branches. Second, it allows the identification of branches and targets, thus providing information necessary for branch removal.

4. Forward Branches

The simplest type of control dependence results from forward branches. Since the execution of the statements between the branch and its target clearly depend on the value of the variables in the branch expression, IF conversion must determine guards that correctly reflect this dependence Once the guards are in place, the jump is unnecessary and is removed. The process of eliminating forward branches is known as forward branch removal.

Fundamental to all phases of branch removal is the idea of a current condition, which is simply a logical expression (guard) reflecting the conditions under which the statement presently under consideration will be executed. As branch removal moves from statement to statement in the program, it conjoins or disjoins Boolean variables with the current condition to generate the guard for the next statement. These Boolean variables represent facts about the forward branches of the program (such as whether or not they would be taken).

A forward branch affects control flow at two locations: at the branch, where control flow can diverge from ordinary sequential flow; and at the target label, where the split rejoins sequential flow. Thus, the current condition (or cc) must be modified at these points to remove forward branches.

(1) At the branch: In the absence of other control flow changes, the statement immediately

following a forward branch is executed only when control flow reaches the branch and the branch is not taken. Thus, if the current condition at the forward branch is cc_1 and the predicate controlling the branch is p, the guard for the following statements will be $cc_1 \wedge \neg p$.

(2) At the target: Similarly, control flow can reach the target of the branch either sequentially from the previous statement or via the branch itself. Under the previous assumptions, if the guard on the statement prior to the target is cc_2, the guard on the target should be $cc_2 \vee (cc_1 \wedge p)$. In the absence of other changes in control flow (so that $cc_2 = cc_1 \wedge \neg p$), the guard on the target statement is $(cc_1 \wedge \neg p) \vee (cc_1 \wedge p)$ which simplifies to cc_1. In other words, if control flow reaches the branch, control flow will reach the target regardless of which execution path is taken.

An example should make these ideas clearer.

```
          DO 100 I = 1, 100
             IF (A(I).GT.10) GO TO 60
S1              A(I) = A(I) + 10
                IF (B(I).GT.10) GO TO 80
S2                 B(I) = B(I) + 10
S3     60       A(I) = B(I) + A(1)
S4     80 B(I) = A(I) - 5
      100 CONTINUE
```

We introduce two Boolean variables br_1 and br_2 to capture the two branch conditions in the loop. Such variables are called branch flags. The branch flag br_1 is defined to be true if and only if "A(I).GT.10" evaluates to true in the first IF statement. We use the Fortran logical variables BR1 and BR2 to capture the values of br_1 and br_2, so BR1 = $\mu(br_1)$ and BR2 = $\mu(br_2)$. In the program text this is accomplished by inserting the assignments:

```
BR1 = A(I).GT.10
BR2 = B(I).GT.10
```

in place of the two IF statements. By using logical variables to capture the values of conditions at the original point of evaluation, PFC ensures that later assignments in the program cannot accidentally change the conditions controlling statements.

Following the conventions for forward branch removal described above, we find that the statements in the loop are controlled by the following conditions.

statement	controlling condition
S_1	$\neg br_1$
S_2	$\neg br_1 \wedge \neg br_2$
S_3	$br_1 \vee (\neg br_1 \wedge \neg br_2)$
S_4	$br_1 \vee (\neg br_1 \wedge br_2) \vee (\neg br_1 \wedge \neg br_2)$

In order to prevent the proliferation of long expressions involving logical variables like BR1 and BR2, the IF conversion procedure must be able to recognize identities and simplify logical expressions. For example, it should surely recognize that the condition controlling S_4 is always true. Thus, simplification is an important aspect of IF conversion. With simplification, the IF conversion procedure in PFC would convert the example loop above into the following.

```
         DO 100 I = 1, 100
            BR1 = A(I).GT.10
S1          IF (.NOT. BR1)       A(I) = A(I) + 10
            IF (.NOT. BR1)       BR2 = B(I).GT.10
S2          IF (.NOT. BR1 .AND. .NOT. BR2)
       X                         B(I) = B(I) + 10
S3          IF (BR1 .OR. .NOT. BR2)
       X                         A(I) = B(I) + A(I)
S4          B(I) = A(I) + 5
     100 CONTINUE
```

Note that the condition controlling S_3 is different from what one would initially expect. When we first ran this example on a prototype PFC system that used the Quine-McCluskey prime implicant simplifier [Quin 52, McCl 56], we thought the simplifier was incorrect. After some thought however, we realized that the simplifier had indeed produced a correct (and simpler) version of this condition.

Figure 2 outlines the algorithm used to eliminate forward branches. The procedure forward_convert is called on each statement in the original code. cc_0 is initialized to TRUE before the first call, and is then reset by each succeeding call. The algorithm assumes the existence of a set of queues (in the array predicate_list) and basic queue primitives. Note that only forward branches are converted; therefore all the expressions to be disjoined at a target must be in its predicate list at the time the guard for that target is created.

5. Backward Branches

While branch removal can eliminate forward branches quite handily, it cannot remove the last type of control dependence - backward branches. In fact, backward branches cannot be directly eliminated from a program, because a backward branch creates an implicit loop. A looping construct cannot be simulated with guarded statements; thus backward branches cannot be directly eliminated.

```
procedure forward_convert (x, cc0)
    returns condition;

    /* x is the statement under consideration   */
    /* cc0 is the condition prior to x,         */
    /* cc1 will be the condition guarding x     */

    /* predicate_list(x) is a queue of all     */
    /*    predicates that must be disjoined    */
    /*    at x because of branches to x.       */

    cc1 ← cc0;

    while not_empty ( predicate_list(x) ) do
    begin
       p <- get_from_queue ( predicate_list(x) );
       cc1 ← cc1 ∨ p
    end

    case statement_type(x) in

       /* IF (P) GOTO y (forward to same level) */
       begin
          create a new branch flag bri
             with realization BRi;
          replace x  with "IF (μ(cc1)) BRi = P;";
          add_to_queue (predicate_list(y),
                        cc1 ∧ bri);
          cc1 ← cc1 ∧ ¬br1
       end

       /* GOTO y  (forward to same level)      */
       begin
          add_to_queue (predicate_list(y), cc1 );
          cc1 ← false;
          delete statement x
       end

       /* All other type statements */
       begin
          guard x by cc1
       end

    esac;

    return (cc1)
end forward_convert;
```

Figure 2. **Forward Branch Removal.**

Backward branches create more problems than just implicit loops, however. Forward branch removal in the presence of backward branches cannot be handled by the algorithm in Figure 2, because of code like the following:

```
          IF (X) GO TO 200
          ...
100       S1
          ...
200       S2
          ...
          IF (Y) GO TO 100
```

Forward branch removal as illustrated in Figure 2 would set the guard for S_1 to $\neg X$. This guard is incorrect because it would prevent S_1 from being executed when X is true and the backward branch to 100 is taken.

One possible approach to IF conversion that avoids the complications of backward branches is to isolate these branches, leaving the code under their control (known as an implicitly iterative region) untouched. Of course, this approach inhibits removal of any forward branches into an implicitly iterative region.

This limitation seems quite severe, so we must consider the problem more carefully. A guard for S_1 must reflect two alternatives:

(1) S_1 is executed on the first pass through the code only if X is false.

(2) S_1 is always executed any time that backward branch is taken.

These alternatives suggest a generalized approach: one set of conditions is used to guard the first pass through an implicitly iterative region and a different set is used to guard subsequent passes. These guard conditions can be established by using a Boolean variable which is false on the first pass through the region and true whenever the backward branch has been taken. In other words, a branch back flag bb (with realization BB) will denote the fact that the backward jump has been taken.

Applying this idea to the previous example would produce:

Statement	Guard
BR1 = X	**true**
...	$\neg br_1$
BB1 = .FALSE.	**true**
100 S_1	$\neg br_1 \vee (br_1 \wedge bb_1)$
...	$\neg br_1 \vee (br_1 \wedge bb_1)$
200 S_2	**true**
...	
IF (Y) THEN	
BB1 = .TRUE.	
GOTO 100	
ENDIF	

One noteworthy point is that BB1 is set to true only if a branch back occurs.

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

(1) Fall through: control can fall through from the statement before y. The condition under which this path is taken is completely encoded by the current condition on exit from the predecessor.

(2) Backward branch: control can enter the implicitly iterative region by a branch with branch flag br_i and branch backward (flag bb_j) to y. The condition under which this

can happen is $br_i \wedge bb_j$. Since bb_j is set to true when the branch occurs, it incorporates the condition that the backward branch was reached from the target of the forward branch and the backward branch condition was true.

Hence, the guard at the target of the backward branch is

$$cc_y \vee (br_i \wedge bb_j)$$

If there is more than one jump into the iterative region, the second term should be the disjunction of the each branch condition conjoined with bb_j.

The condition generated at the target must also be slightly modified. Consider the following example.

```
      IF (X) GO TO 200
100   S1
      GO TO 300
200   S2
      IF (Y) GO TO 100
300 S3
```

The correct guard for S_2 must be $br_1 \wedge \neg bb_1$, since S_2 is executed if and only if the forward branch to 200 was taken and the backwards branch has not been taken. In order to remove the branch preceding S_2, the term $\neg bb_1$ must be 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 iteration of these regions. The previous example after complete branch removal becomes

```
      BR1 = X
100   IF (.NOT.BR1 .OR. BB1.AND.BR1) S1
      /* GO TO 300 has been eliminated */
200   IF (.NOT. BB1 .AND. BR1) S2
      IF (.NOT. BB1 .AND. BR1 .AND. Y) THEN
         BB1 = .TRUE.
         GO TO 100
      ENDIF
300 S3
```

At S_3, the current condition of $\neg bb_1 \wedge br_1$ is disjoined with the target condition $\neg br_1 \vee bb_1 \wedge br_1$. The result after simplification is true which mirrors the fact that S_3 should always be executed.

Figure 3 contains the general branch removal algorithm which incorporates these observations. The only major modification to the algorithm in Figure 2 is the check, encapsulated in process_branch (Figure 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.

```
procedure remove_branches (x, cc0)
   returns condition;

   /* x is the statement under consideration.  */
   /* cc0 is the current condition prior to x  */
   /* cc1 is the current condition after x     */

   cc1 ← cc0;
   while not_empty ( predicate_list(x) ) do
   begin
      p ← get_from_queue ( predicate_list (x) );
      cc1 ← cc1 ∨ p
   end
   case statement_type(x) in

      /* IF (P) GOTO y (forward to same level) */
      begin
         create a new logical guard bri
            with realization BRi;
         replace x with  "IF (μ(cc1)) BRi = P";
         process_branch (x, y, cc1 ∧ bri);
         cc1 ← cc1 ∧ ¬bri
      end

      /* GOTO y  (forward to same level      ) */
      begin
         process_branch (x, y, cc1);
         cc1 ← false;
         delete statement x
      end

      /* IF (P) GOTO y (backward to same level)*/
      begin
         let bbj be branch back flag associated
            with this branch (realization: BBj);
         insert "BBj = .FALSE." before y;
         let TPk be a new temporary variable;
         replace x with the statements
            "TPk = μ(cc1)"
            "IF (TPk) TPk = P"
            "IF (TPk) BBj = .TRUE."
            "IF (TPk) GO TO y"
      end;

      /* All other statements */
      begin
         guard (x) ← guard (x) ∧ cc1
      end
   esac;
   return (cc1)
end remove branches;
```

Figure 3: **Complete branch removal**

The branch removal procedure used in PFC has several advantages. First, no special cases are needed for backward branches unless there is a branch into the region under the control of that backward branch. Without the presence of another branch, the branch back flag never enters the current condition. Second, the branch back flag simplifies out of the current condition after the target of the last forward branch into the implicitly iterative region. This simplification reflects the fact that the condition for execution of all statements after the last possible external entry to the backwards branch should be

independent of any specific iteration of the backwards branch. Most important, however, is the ability of the algorithm to handle any pathological combination of backwards branches with minimal effort.

6. Boolean Simplification

In developing the conceptual basis for IF conversion, we have purposely attempted to factor out issues of representation. We have referred to two representations, one internal and one external, for the conditions constructed by IF conversion. The basic method insures that the guards attached to the output program will be correct, but we need some mechanism to insure that the output program will be clean and readable. In other words, we need to find external representations for the conditions which are as simple as possible.

The simplification function is built into the operator μ which maps internal representations to external representations. Internally, the guards are maintained in a form suitable for quickly performing the fundamental operations of branch removal -- creating a new branch flag and merging two conditions at a label. The actual simplification is performed by applying a version of the Quine-McCluskey prime implicant simplifier [Quin 52, McCl 56].

6.1. Preliminaries

We begin with a bit of notation for the ensuing discussion. In a Boolean formula, variables and negations of variables will be referred to as literals. A conjunction of literals is known as a fundamental formula if no variable appears in it twice. Any alternation of fundamental formulas is a (disjunctive) normal formula and the fundamental formulas of which it is an alternation are called terms.

Let $\underline{V}$ be a set of variables. We denote by $nf(\underline{V})$ the set of all normal formulas over $\underline{V}$. A fundamental formula t is a minterm over $\underline{V}$ if each variable in $\underline{V}$ occurs in it exactly once. If there are n variables in $\underline{V}$, there are 2^n minterms, since each minterm can contain either a variable or its negation.

Every Boolean formula can be written as the alternation of minterms; we refer to this representation as the canonical expansion or canonical disjunctive normal form. The Quine-McCluskey procedure simplifies Boolean formulas by reducing them to canonical disjunctive normal form and then finding a minimal set of prime implicants for the set of minterms. A fundamental formula ϕ is prime implicant of a formula $\underline{\Phi}$ if $\phi \supset \underline{\Phi}$ and there exists no shorter conjunction of a subset of the literals in ϕ that also implies $\underline{\Phi}$.

Hence the Quine-McCluskey procedure contains three phases:

(1) Reduction to of the formula $\underline{\Phi}$ to canonical form.

(2) Construction of the set $\underline{P}$ of all prime implicants for the formula. If $\vee\underline{P}$ is the alternation of all members of $\underline{P}$, then $\vee\underline{P} \equiv \underline{\Phi}$.

(3) Selection of the shortest set $\underline{S} \subset \underline{P}$ such that $\vee\underline{P} \equiv \underline{\Phi}$.

Phase 3 is of combinatorial complexity in the number of prime implicants, but since the best simplification is not strictly necessary, a good heuristic to select $\underline{S}$ is acceptable. Phase 2 can be implemented in time proportional to $n^{1.58}m$ where n is the number of minterms and m is the number of variables used in $\underline{\Phi}$ [AlKW 82]. However, the method requires $O(3^m)$ storage, so it is impractical for m larger than eight or nine. However, there exist slightly slower methods which have much smaller storage requirements. McCluskey's original technique is one such [McCl 56]. Phase 1 is also potentially exponential since a few short formulas in m variables can give rise to 2^m minterms.

6.2. Simplification in PFC

In PFC, we avoid phase 1 of the Quine-McCluskey procedure by internally maintaining the guards as a set of minterms over the set of branch flags active at the time the guard is created. This representation allows us to take advantage of the observation that conditions are modified during branch removal in only two ways:

(1) At a forward branch a new branch flag is created and two new conditions are formed from it by conjoining it and its negation to the current condition at the branch. Internally, this result can be effected by conjoining the new flag and its negation to every minterm in the current condition collection. The ones with the negation comprise the current condition for the next statement while the ones with the unnegated flag comprise the condition attached to the branch.

(2) At a target some collection of conditions must be disjoined. This disjunction is handled by extending the minterms to be over the same set of variables, then simply taking the union of all minterms in the various collections.

Conditions in PFC are actually represented by two parts. The first part (the branch flag list) is a list of branch flags present in the condition, maintained in the order that the forward branches they represent were encountered. The second part is a set of minterms. The disjunction of these minterms represents the actual condition.

A simple example should clarify the method used. Consider the following code:

```
          IF (X) GOTO 300
          ...
          IF (Y) GOTO 100
          ...
          IF (Z) GOTO 200
          ...
    100   CONTINUE
          ...
    200   CONTINUE
          ...
    300   CONTINUE
```

As each of the branches are passed, the current condition is conjoined with the branch flags to produce a single minterm $\neg br_1 \wedge \neg br_2 \wedge \neg br_3$ as the current condition after all branches. The expression to be disjoined at statement 100 is $\neg br_1 \wedge br_2$. Since the current condition includes br_3, which is not in the target condition, we expand the target condition by rewriting it as the disjunction of two minterms: $(\neg br_1 \wedge br_2 \wedge br_3) \vee (\neg br_1 \wedge br_2 \wedge \neg br_3)$. When this expression is disjoined with the current condition, no simplification can be performed (other than reversing the transformation made in the target condition). Thus the current condition after statement 100 is

$$(\neg br_1 \wedge \neg br_2 \wedge \neg br_3) \vee (\neg br_1 \wedge br_2 \wedge br_3) \vee (\neg br_1 \wedge br_2 \wedge \neg br_3)$$

At statement 200, the expression $br_1 \wedge br_2 \wedge br_3$ is disjoined with the current condition, giving:

$$(\neg br_1 \wedge \neg br_2 \wedge \neg br_3) \vee (\neg br_1 \wedge br_2 \wedge br_3) \vee (\neg br_1 \wedge br_2 \wedge \neg br_3) \vee (\neg br_1 \wedge \neg br_2 \wedge br_3)$$

The first and last minterms simplify to $\neg br_1 \wedge \neg br_2$. The second and third minterms simplify to $\neg br_1 \wedge br_2$. These minterms combine to produce $\neg br_1$. Finally, at statement 300, the flag $\neg br_1$ is simplified out, resetting the current condition to true.

This example leads to several new observations.

(1) Once a branch flag is simplified out of the current condition, it never reenters the condition. The disappearance of a flag implies that all possible execution paths since the branch associated with the flag have merged together. Whether or not the branch was taken will have no effect on the execution of subsequent statements.

(2) The order in which branch flags may be simplified out of the condition is exactly the reverse of the order in which the branch flags are introduced. The previous example demonstrates this point clearly, since br_3 must be removed from the current condition before br_2 can be removed.

The proof of these statements is straightforward but not obvious. In the interest of space, we will omit it here. The interested reader is referred to a technical report on simplification in PFC [AlKW 82].

The minterm representation for guards can be exponentially larger than the shortest representation, as our earlier discussions indicate. However, this growth occurs only when the last branch jumps around a section of code containing the targets of all previous branches. For local, structured branches, branch flags simplify out very shortly after entry. Since the growth of minterms can be exponential in the worst case, regardless of the representation, we chose this method in order to optimize the time required to simplify structured code. Note that simplification with this representation merely involves testing the set of minterms to see if each element $\ldots \wedge br_n$ has a partner $\ldots \wedge \neg br_n$. If so, br_n may be removed from the minterm, and its predecessor checked for the same condition; otherwise, the condition is in simplest terms. By carefully ordering the minterms as they are added to the condition, we can insure that simplification is acceptably efficient.

```
procedure process_branch (x, y, br);
    /* x is the branch                              */
    /* y is the target                              */
    /* br is the condition on the branch            */
    stmt_guard ← true;
    for each implicitly iterative region
                    that x jumps into      do
    begin
       let bb_j be the branch back flag
          controlling the region;
       let x_j be the target of
          the backward branch;
       add_to_queue (predicate_list(x_j), br∧bb_j);
       stmt_guard ← stmt_guard ∧ ¬bb_j
    end
    add_to_queue (predicate_list(y),
                  br ∧ stmt_guard);
end process_branch;
```

Figure 4. **Forward Branch Processing**

Afterwards the simplified condition may be generated using phases 2 and 3 of the Quine-McCluskey procedure. These phases are required only when the actual current condition is altered. This scheme can also be expanded to handle backwards branch flags by adding the backward branch flag to the current condition's branch flag list and expanding when the first branch into the implicitly iterative region is encountered.

7. Implementation

IF conversion in PFC is performed in three separate passes over the program. The first pass analyzes the branches in the code, marking backward branches and exit branches. Next branch relocation is performed, followed by branch removal. These passes are basically as described above, although the algorithms differ slightly in order to promote efficiency and simpler conditions. Simplification using the abstract representation described previously is performed only during branch removal. Afterwards, the guards are converted to the same intermediate form as all other expressions. A final pass over the program unlocks backward branches and converts them to WHILE loops. At this point, all branches have been removed from the program.

Figure 5 briefly outlines the structure of PFC. Prior to IF conversion, PFC normalizes DO loops and analyzes the program to uncover its basic block structure [Kenn 81]. DO loop normalization modifies all loops to run from one to some upper bound by increments of one. In doing so, new loop induction variables are generated which allow easy identification of the loop controlled. Additionally, the nesting level of every statement is noted, thereby allowing easy determination of jumps out of loops.

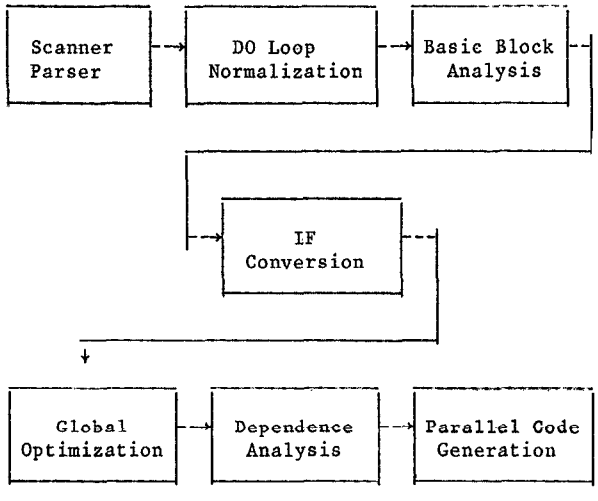

Figure 5: **Structure of PFC**

Basic block analysis is not important to IF conversion directly, but it is vital to the global optimization phase following IF conversion. Note that by explicitly guarding every statement in the program, IF conversion greatly increases the number of basic blocks in a program. Specifically, every guard and every statement has become a block to itself. If these blocks were used in global optimization, the analysis would be horrendously slow, and in many cases, far less accurate than possible. However, by making use of the fact that IF conversion does not change the execution order of the program, we are able to use the basic blocks constructed before IF conversion to successfully optimize the program as it exists after IF conversion. In addition, the use of basic blocks can facilitate the incorporation of unvectorized IF statements into block IF constructs.

After IF conversion, PFC applies global optimization transformations to the program. These transformations include dead code elimination, constant propagation, and induction variable substitution. In addition to replacing implicit induction variables with functions of the true induction variables, induction variable substitution propagates certain expressions forward within loops (within the limits set by the basic block analysis). This propagation will replace flags that are constant within a loop by the actual expression assigned to the flag. This transformation is advantageous for two reasons. First, scalars inside DO loops either inhibit or greatly increase the cost of vectorization. Second, the resulting code is much closer in appearance to the original code, making the transformations easier to understand.

Another important transformation performed by PFC is scalar expansion, which is part of a recurrence breaking phase during parallel code generation. As described earlier, the use of scalar flags inside DO loops can cause scalar dependences, thereby inhibiting vectorization. Scalar expansion will replace scalar variables by equivalent array variables, thereby breaking some of the dependences. There are three distinct advantages to using scalar flags in IF conversion, rather than using logical arrays directly. First, this approach allows IF conversion to focus strictly on the problem of converting control

dependences -- it need not be concerned with the tedious details of converting scalars to arrays. Second, this approach guarantees that arrays are not created unless some vectorization is gained by the approach. Strictly expanding every scalar can greatly increase the amount of storage required by a program, without necessarily permitting any vectorization at all. Scalar expansion, however, will not expand a scalar unless some vectorization results. Third, arrays created in IF conversion are tested for dependence in the dependence analysis phase of PFC. Unfortunately, dependence testing is not exact in PFC; PFC may determine that two array references are dependent when in actuality they are independent. In particular, the expansion of exit flags gives rise to false recurrences when tested by the dependence analysis phase. Scalar expansion is sophisticated enough to recognize that these dependences are false, however, and ignores them, thereby permitting more vectorization.

8. Conclusions

IF conversion has proved to be an extremely valuable transformation in PFC because it permits vectorization of sections of code that PFC must otherwise leave untouched. The present implementation of IF conversion is complete as described here, with two exceptions. First, simplification is not yet completed. As a result, IF conversion can only be run on short examples, since the current condition tends to rapidly become unwieldy. Second, we have only briefly explored the possibilities of converting guards in unvectorized code to block IF constructs.

IF conversion has implications far beyond the applications to vectorization. By converting control dependences to data dependences, IF conversion is useful in such applications as data flow languages, code structuring, and goto elimination. More generally, it demonstrates in a practical program transformation system that any branching construct can be successfully converted to a structured construct. This result, though well known [BohJ 66, Hare 80], is intellectually pleasing as well as practically useful.

References

[AllK 82] J.R. Allen and K. Kennedy, "PFC: a program to convert Fortran to parallel form," Report MASC TR 82-6, Department of Mathematical Sciences, Rice University, Houston, Texas, March, 1982.

[AlKW 82] J.R. Allen, K. Kennedy, and J. Warren, "Simplification of Boolean formulas in PFC," Dept. Mathematical Sciences, Rice University, Houston, Texas, November 1982.

[ANSI 81] American National Standards Institute, Inc., "Proposals approved for Fortran 8x," X3J3/S6.80 (preliminary document), November 30, 1981.

[Bane 76] U. Banerjee, "Data dependence in ordinary programs," Report 76-837, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, November 1976.

[BohJ 66] C. Bohm and G. Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules," Comm ACM 9, 5, May 1966.

[Burr 77] Burroughs Corporations, "Implementation of FORTRAN," Burroughs Scientific Processor brochure, 1977.

[GibK 81] C. Gibbons, and K. Kennedy, "Simplification of functions," Rice Technical Report 476-029-10, Rice University, January 1981.

[Hare 80] D. Harel, "On folk theorems," Comm ACM 23, 5, July 1980, 379-389.

[Kenn 80] K. Kennedy, "Automatic translation of Fortran programs to vector form," Rice Technical Report 476-029-4, Rice University, October 1980.

[Kuck 77] D.J. Kuck, "A survey of parallel machine organization and programming," Computing Surveys 9, 1, March 1977, 29-59.

[KKLP 81] D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, "Compiler transformation of dependence graphs," Conf. Record of the Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, Va., January 1981.

[KKLW 80] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," Proc. IEEE Computer Society Fourth International Computer Software and Applications Conf., IEEE, Chicago, October 1980.

[McCl 56] E.J. McCluskey, "Minimization of Boolean functions," Bell System Tech. J. 35, 5, November 1956, 1417-1444.

[Quin 52] W.V. Quine, "The problem of simplifying truth functions," Am. Math. Monthly 59, 8, October 1952, 521-531.

[Towl 76] R.A. Towle, "Control and data dependence for program transformations," Ph.D. Dissertation, Report 76-788, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, March 1976.

[Wolf 78] M.J. Wolfe, "Techniques for improving the inherent parallelism in programs," Report 78-929, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, July 1978.