R. MORRIS, Editor

Optimization of Expressions in Fortran

VINCENT A. BUSAM AND DONALD E. ENGLUND
Computer Sciences Corporation, El Segundo, California

A method of optimizing the computation of arithmetic and indexing expressions of a Fortran program is presented. The method is based on a linear analysis of the definition points of the variables and the branching and DO loop structure of the program.

The objectives of the processing are (1) to eliminate redundant calculations when references are made to common subexpression values, (2) to remove invariant calculations from DO loops, (3) to efficiently compute subscripts containing DO iteration variables, and (4) to provide efficient index register usage.

The method presented requires at least a three-pass compiler, the second of which is scanned backward. It has been used in the development of several FORTRAN compilers that have proved to produce excellent object code without significantly reducing the compilation speed.

KEYWORDS AND PHRASES: FORTRAN, optimization, expressions, compilers, compilation, subscripts, register allocation, DO loops, common subexpressions, invariant calculations
CR CATEGORIES: 4. 12

1. Introduction

Since the development of the first FORTRAN compiler, a major concern of compiler writers has been the production of efficient object code; in fact, the more efficient the better. However, the need for fast computation has often pushed efficient object code far out of the mind of the compiler writer. This is evidenced by the numerous one-pass compilers that have been developed recently. The need for fast compilation is often acute, especially in a university environment, where there are many programs that require very little execution time after the debugging process has been completed.

But there are also production programs, which are run many times after debugging is complete. Compilation techniques used in a compiler written for this type of program are different from the one-pass algorithms [1, 2, 3, 4, 5]. The most obvious requirement of such a compiler is that it optimize the code written by the programmer. Two of the tasks of the compiler in this area are the recognition of common subexpressions and the removal of invariant calculations from loops. Another important goal is the efficient use of registers as much as practical.

In this paper a three-pass FORTRAN compiler to perform the above-mentioned optimizations is described. The design has been used in compilers developed by Computer Sciences Corporation and has demonstrated that it can generate good object code while providing efficient compilation.

2. Pass I: Dictionary and Table Building

The first pass of the compiler transforms the FORTRAN source statements into an internal representation. During this pass, information is collected that is needed during later passes. The major tables built during this pass are:

1. Symbol dictionary
2. Statement-number dictionary
3. Constant dictionary
4. Expression dictionary
5. Encoded source program

Symbol, Statement-Number, and Constant Dictionaries. The symbol dictionary contains one entry for each variable and function in the source program. Each entry contains the attributes for the identifier. For each variable, there is a definition sublist that contains an entry for each statement in which the variable is potentially redefined. The origin of this sublist is contained in its symbol-dictionary node. The statement-number dictionary contains an entry for each referenced statement number (the statement number denoting the end of a DO is not considered a reference unless referred to by a GO TO or passed as an argument). The fall-through case for an arithmetic IF does not cause the label on the statement following the IF to be referenced. The constant dictionary contains one entry for each different constant value in the source program. Constants are entered by value rather than in the character string form. During this pass, as much constant arithmetic as possible is performed.

Expression Dictionary. Each distinct expression has an entry in the expression dictionary. Each expression is transformed into triad form, that is, operator and up to two operands. The operands allowed are pointers to dictionary entries. Whenever possible, current expressions use previous triads for part or all of the expression.

Take, for example, the following two FORTRAN statements and their corresponding expression-dictionary entries:

	Expression entry	*Operator*	*Dictionary pointers Operand 1*	*Operand 2*
I = J + K	(1)	+	J	K
	(2)	=	I	(1)
L = J + K	(3)	=	L	(1)

During the encoding of an expression, all minus signs are moved to the highest possible position in the expression. For example, the expression (−A −B) is encoded as −(A + B). For operations that are commutative, the operands are sorted and the signs changed as necessary. This allows A + B and B + A to be treated as identical expressions and, therefore, increases the likelihood of optimization. Another transformation made on the source is the explicit statement of implied expressions. For example, the statement I = X implies use of the IFIX function to convert the right-hand side of the assignment to fixed point. Therefore, when the expression is transformed into triads, the result is as if the following statement had been written:

I = IFIX (X)

(IFIX is a built-in FORTRAN function giving a fixed-point value for a floating-point argument.)

Other implied expressions are present in the use of subscripts, since code must be generated to linearize array element storage. For example, consider the array dimensioned A(20, 10). The code generated for a reference to an element of the array, say A(I, J), would be equivalent to "(address of A − 21) + (J*20 + I)". When possible, expressions involving DO variables are factored, and constant arithmetic is performed; for example, A(I, I) becomes "(address of A−21) + (21*I)". Since only the restricted forms of subscripts were allowed, factoring was performed simply by moving the induction variable to the final position in each term and factoring the induction variable from all terms with I.

The replacement of array references with their expansion in the expression dictionary provides optimized subscript calculation in the same manner as arithmetic expressions. The only implied expressions not expanded during Pass I are those associated with the setting, incrementing, and testing of index variables of DO loops and expressions involving DO variables that are apt to be computed recursively. Each DO index is treated as a unique variable with the same attributes and storage address as the index variable written by the programmer. Wherever the index variable is referenced within the loop, the reference is made to the "created variable" dictionary entry rather than the programmer-named variable. This substitution is required because of the optimization of subscripts performed in Pass II over parallel DO loops (DO loops in the same DO nest at the same level of nesting) using the same index variable. If the DO variable is used computationally, a flag associated with the DO loop is set during this pass.

An example of some of the operations and tables has been derived from the FORTRAN program on the left-hand part of Figure 1. The symbol, constant, and expression dictionaries for this program appear in Figure 2.

	FORTRAN PROGRAM	*Statement ID*	*Redefines COMMON*	*Labeled statement*	*Branch statement*	*Adjacent DO element*	*DO loop*	*DO nest*	*Parallel DO*	*Extended range DO*	*Preceding DO referenced label*	*Optimal loop candidate*
	SUBROUTINE SUB	1										
	COMMON I, J, K	2										
30	L = 0	3		0								
	DO 180 I = 1, 10	4				0	18	0	0	No	3	No
	DO 130 J = 1, 10	5				4	13	4	0	No	0	Yes
	DO 100 K = 1, 10	6				5	10	5	0	No	0	No
	GO TO 130	7			0							
80	L = L + MOD (J, 2)	8		3								
	L = L + ISUB (I)	9	0									
100	CONTINUE	10		8		6	6					
	GO TO 130	11			7							
120	GO TO 200	12		10	11							
130	CONTINUE	13		12		10	5					
	J = 0	14										
	DO 170 K = 1, L	15				13	17	4	5	No	0	Yes
	J = 1 + J + K/2	16										
170	CONTINUE	17		13		15	15					
180	CONTINUE	18		17		17	4					
	I = J	19										
200	IF (I − 200) 30, 30, 210	20		18	12							
210	RETURN	21		20								
	END	22										
Head of List			9	21	20	18						

FIG. 1. Sample program

Symbol dictionary list

Symbol	*Attributes*	*Definition list*
I	Fixed, common	19, 9, 4, 0
J	Fixed, common	16, 14, 5, 0
K	Fixed, common	15, 6, 0
L	Fixed	9, 8, 3, 0
MOD	Built-in function, fixed	0
ISUB	User function, fixed	0
I.1	Fixed, same as I in statement 4	4, 0
J.1	Fixed, same as J in statement 5	5, 0
K.1	Fixed, same as K in statement 6	6, 0
K.2	Fixed, same as K in statement 15	15, 0

Expression dictionary

ID	*Operator*	*Operand 1*	*Operand 2*
a	=	L	0
b_1	,	J	2
b_2	(	MOD	b_1
b_3	+	L	b_2
b_4	=	L	b_3
c_1	(	ISUB	I
c_2	+	L	c_1
c_3	=	L	c_2
d	=	J	0
e_1	+	1	J
e_2	/	K	2
e_3	+	e_1	e_2
e_4	=	J	e_3
f	=	I	J
g	−	I	200

Constant dictionary list

Value
1
2
10
200
0

Statement-number dictionary

30
130
200

FIG. 2. Dictionaries for sample program

Encoded Source Program. Some of the important information needed for the optimization pass is associated with the executable statement and, thus, is kept with the encoded source program. Each executable statement is assigned an ascending identification number by the compiler and is encoded in terms of dictionary-list nodes entered in the encoded source-program list. Each statement entry is in two parts: header information and the statement coding. The header part identifies the type of statement and contains the links for the nondictionary sublists formed during this pass. The statement encoding describes the various syntactic components of the statement. Figure 3 contains examples of a possible form of statement encoding.

```
Statement 3:   (1) Statement header
                   (a) Assignment-statement ID
                   (b) Sublist links
               (2) Statement encoding
                   (a) Expression a
Statement 4:   (1) Statement header
                   (a) DO-statement ID
                   (b) Sublist links
               (2) Statement encoding
                   (a) I.1   (index variable)
                   (b) 1     (starting value)
                   (c) 10    (ending value)
                   (d) 1     (increment value)
Statement 20:  (1) Statement header
                   (a) Arithmetic IF-statement ID
                   (b) Sublist links
               (2) Statement encoding
                   (a) Expression g
                   (b) 30  (negative branch)
                   (c) 30  (zero branch)
                   (d) Fall-through (positive branch)
```

FIG. 3. Examples of executable-statement encoding

Certain entries in the encoded source program are linked together to form various sublists. The important statement sublists formed are:

1. Common redefinition statements
2. Labeled statements
3. Branch statements
4. Adjacent DO-element statements

The common redefinition sublist contains one entry for each statement that invokes a subprogram or function that potentially redefines the variables in the common area. Generally, these are only the user-supplied functions and subprograms as the built-in FORTRAN functions do not use common except through their arguments. The second sublist links all statements labeled with a statement number. The third links all branch statements and statements with statement-number arguments. The last sublist links together adjacent DO elements; that is, all DOs and CONTINUEs are linked together in the order encountered in the source program. If the DO is ended with a statement other than its own unique CONTINUE, then one is supplied. Because the second pass scans the program backward, the encoded source programs and all sublists are linked in reverse order; that is, each element points to a preceding source program statement. The first five columns following the source program in Figure 1 give an example of the statement identification number assigned by the compiler and the linking of the four sublists.

Because of the importance of DO loops in determining the limits of removal of common subexpressions and in the efficient allocation of index registers, additional information is kept for each DO loop. First, the DO and its CONTINUE statement are linked to each other. Second, each DO statement points to the DO statement of the loop in which it is immediately contained. Third, each DO statement points to the DO statement immediately preceding it that is at the same DO nesting level in the current DO nest. Columns 6–8 in Figure 1 show this information for the sample program.

After the encoding scan for pass I is completed, the labeled statement chain is processed to remove from it all unreferenced statement numbers. During pass I, only a reference to a statement number results in an entry in the statement-number dictionary. All statement numbers not entered during pass I can be considered unreferenced. Also, during this processing, a pointer to the DO statement of the loop in which the statement number definition occurs and its level of nesting is entered into its dictionary entry.

The entries of the branch chain are then processed to determine their effect on the optimizations of the DO loops. As the branch chain is advanced to a new entry, the DO element chain is advanced to the DO statement entry immediately preceding the new branch entry.

Each branch entry is examined. If the branch entry is within a loop and references a statement number outside the range of the loop, the DO variable materialization flag is set, starting with the current loop, on each DO entry of the nest out of which control passes. The materialization flag is used to determine if the value of the index must be stored or may be kept in a register. If the branch entry references a statement number not at the outermost level, further processing is performed to determine whether the branch affects certain indexing optimizations (see Section 5). The loop chain is scanned, starting with the current loop and continuing until the DO of the referenced statement number is reached. Each loop encountered with a level of nesting that is one greater than that of the referenced statement number is flagged as nonoptimal. An inner loop is defined as optimal with respect to its next outer loop if the inner loop is always executed a fixed number of times for each complete execution of the outer loop of the pair. This situation exists if there are no branches out of the loop. Figure 4 indicates the branching situations which make loop B nonoptimal with respect to the next outer DO (loop A).

During this processing, every loop containing a transfer to the outermost level of the program is flagged, along with each loop containing a transfer into its range from this outermost level. Those loops having both flags are extended-range loops and have the extended-range flag set.

During this processing, each DO is also linked to the first referenced label that precedes it. If another DO or CONTINUE is between the DO under examination and the preceding referenced label, this field is set null.

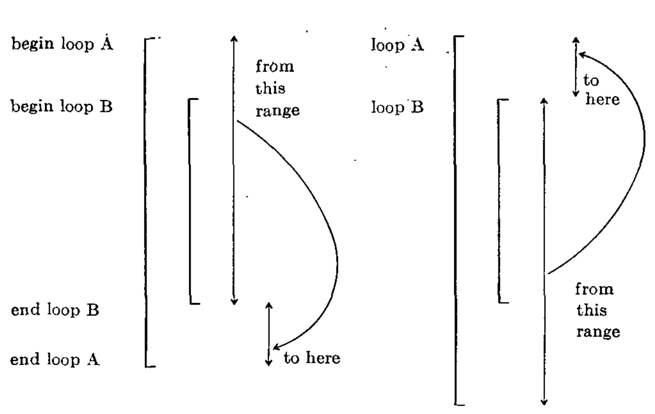

Fig. 4. Nonoptimal branching ranges

The fields for some of these items are shown in the last three columns of Figure 1.

Once all this information has been compiled, the second pass of the compiler can begin.

3. Pass II: Optimization

During pass II, the source program is scanned backward; that is, from the last statement to the first. During this pass, it is determined where each expression of the source program is to be computed. As much as possible, common subexpressions are eliminated, and invariant expressions are removed from DO loops. Each nonassignment expression of a statement is processed as encountered during this pass in order to determine the earliest point in the program flow where it can be computed.

The process of determining where in the program to compute an expression represented by an entry in the expression dictionary involves determining the range of the source program over which the operands of the expression have a constant value (ignoring redefinitions caused by branches to explicit or implied labels within this range). This process yields values called the "forward definition point" and the "backward definition point." Using these points and the DO loop structure and knowledge of referenced labels, the earliest location where it is theoretically possible to compute the expression is determined. A node is entered in the Encoded Source Program at this point. This is called the expression's "compute point." When the expression's compute point is reached during this scan, the actual best "evaluation point" for the expression is determined.

Variable Compute Points. At the start of pass II, the origin of the definition sublist for each variable contains the location of the last node in the encoded source-program list where the variable was redefined. This field is called the identifier's forward definition point. A field called the backward definition point is initialized to infinity. These two fields define the current bounds of the source program over which the variable has a common value, ignoring redefinitions caused by transfers or calls to subprograms (except where the variable is an argument) within these bounds. Whenever a statement is encountered that redefines a variable, the definition bounds are updated by setting the forward definition point to the next link in the variable's definition sublist and the backward definition point to the current statement. An example of both forward and backward definition points is given in Figure 5.

Statement ID	*Program*	*Range*	*For variable I* *Forward definition point*	*Backward definition point*
⋮		}	0	10
10	I = ⋮	}	10	20
20	I = ⋮	}	20	30
30	READ (5, 11) I ⋮	}	30	40
40	CALL FUN (I) ⋮	}	40	50
50	I =	}	50	∞
51	END			

Fig. 5. Forward and backward definition points of a variable

In order to save table space, a single pair of fields is similarly maintained for the definition bounds of all the variables in the common area. This pair is updated from the common definition sublist wherever a statement is encountered that calls a subprogram or references a function that potentially redefines common.

Information concerning bounds of loops of the current nest is kept in a pushdown list. The entire program is considered within the range of a dummy noniterative DO loop in order to provide the mechanism required to move to the program's prologue, the evaluation of expressions whose elements contain no redefinition within the program body. The encoded source-program node location containing the referenced statement number immediately preceding the current processing point is maintained and updated at each referenced label.

Expression Compute Points. The earliest point in the program at which the value of an expression may be computed is determined from the forward and backward definition points of its operands, the DO structure, and the referenced program labels. This point is referred to as the expression's compute point. At the beginning of pass II, the compute point fields in the expression dictionary are initialized to infinity.

As the encoded source-program list is scanned, the compute point of each primary expression in the source statements encountered is compared against the identification number of the statement containing the expression. If the value of the compute-point entry in the expression dictionary is greater than the current statement-identification number, this is the first time the expression has been encountered since passing another compute point for the same expression, earlier in the scanning process; therefore, the previously determined compute point is no longer valid, and a new compute point must be determined. If the compute-point field in the expression dictionary is less than the current statement-identification number, the previously established compute point has not been passed during the scan, and is therefore still valid.

The first step in ascertaining the compute point of an expression is determining the most limiting forward (lowest statement-identification number) and most limiting backward (highest statement-identification number) definition point of all of its operands. If an operand is itself an expression with an invalid compute point, this process is recursively applied to first learn its compute point. The forward and backward definition points of an identifier operand are obtained from its symbol-dictionary node. Those points are zero and infinity, respectively, for a constant, statement and format label, and built-in function operands. Those for all other operands (such as user functions) are the statement-identification numbers of the statement currently being processed. If either operand is an identifier in common, then the definition bounds of the common area are also imposed. The forward and backward definition points of an expression are illustrated in the example in Figure 6.

```
Statement
   ID      Program   Range    Definition points for I * J
   10       I =        }
   11       J =        }      Forward definition point  = 11
             :         }      Backward definition point = 25
   20       I*J        }
             :         }      Forward definition point  = 25
   25       I =        }      Backward definition point = 35
             :         }
   30       I*J
             :
   35       J =
```

FIG. 6. Forward and backward definition points of an expression

Compute Point Flow Information. The next step in determining the compute point of an expression is to determine whether the value of the expression remains fixed throughout the range of any DO loop. In order to do this, the expression's limiting forward and backward definition points are compared with the beginning and ending bounds, respectively, of each currently active DO loop in the pushdown list, starting with the innermost DO. If the definition bounds are outside the DO-loop bounds and the loop does not have an extended range, then the expression's value is independent of, and may be removed from, the loop. If the expression contains only constants and DO-iteration variables, the extended range restriction does not apply, since the DO-iteration variables may not be redefined in the extended range.

If the computation of an expression cannot be removed from any loop (see Figure 7(a)), then the expression's compute point is the maximum of (1) the limiting forward definition point of the expression and (2) the nearest preceding statement with a referenced label.

If the computation of the expression can be removed from one or more nested loops, the next step is to determine whether evaluation may be moved through any loops parallel to the outermost one. If a referenced label exists between this outermost loop and its first parallel loop (see Figure 7(b)), then the compute point is the maximum of (1) the expression's limiting forward definition point and (2) the encoded source-program node identification of that label. If no referenced label exists between the parallel loops (see Figure 7(c)), the limiting forward definition is compared with the bounds of the preceding parallel loops until one is found whose identification (1) is less than the limiting forward definition point of the expression, (2) has an extended range, or (3) is preceded

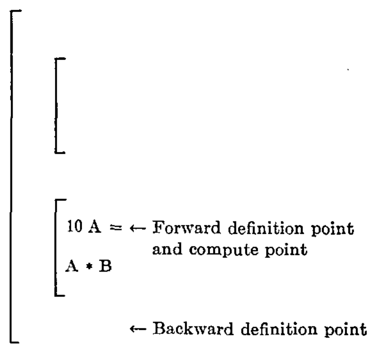
10 A =
← Forward definition point and compute point
A * B
← Backward definition point

FIG. 7(a). Example (a) of compute-point determination

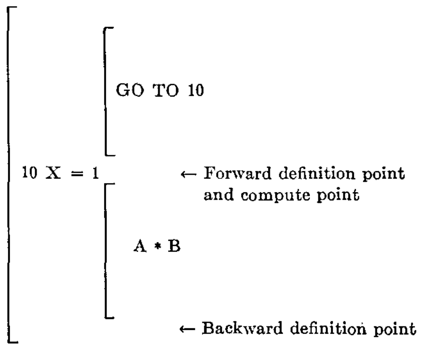
GO TO 10
10 X = 1
← Forward definition point and compute point
A * B
← Backward definition point

FIG. 7(b). Example (b) of compute-point determination

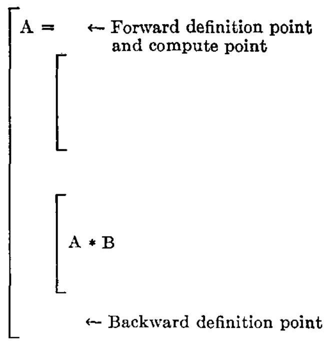
A =
← Forward definition point and compute point
A * B
← Backward definition point

FIG. 7(c). Example (c) of compute-point definition

by a referenced label. The compute point becomes either the limiting forward definition point determined for the expression or the stop condition specified above, whichever is greater.

Compute-Expression Node. Each time an expression's compute point is determined, a compute-expression node identifying the expression is inserted in the encoded source-program list after the expression's compute point. The presence of this node is used by the third pass of the compiler to determine where to generate the code for the evaluation of an expression. The expression dictionary node for the expression is initialized at this time. In addition to the fields discussed in Section 2, the expression-dictionary node contains a field for the compute point, identification of the highest and lowest statements containing a reference to it, the DO-loop level of nesting associated with its lowest reference, and a multiple-reference flag field. When initialized, the multiple-reference flag field is set to no reference, and the highest and lowest statement fields are set to zero and infinity, respectively. If the expression is a type that is not to participate in subexpression consideration (such as assignment equals, or subroutine calls) this step is skipped.

Expression References. An expression that is directly referenced in the code for the statement is a primary expression (e.g. expression in an IF statement, arguments in a CALL statement, the right-hand side of an assignment statement, or the subscript expression of the left-hand side of an assignment statement). Each primary expression of a statement is considered to be referenced directly at the point in the program where the statement appears. The operands of each primary expression and each of its contained expressions are referenced at the evaluation point of the expression in which they are immediately contained. The identification number associated with a statement is used to indicate the relative position of the reference point with respect to the other statements of the program.

An expression node update routine is used to update the fields of an expression-dictionary node at each reference to an expression in the object program. The inputs to this routine are a pointer to the expression's dictionary node and the statement-identification number at which the reference occurred. If, when entered, the highest reference field is nonzero, the multiple-reference field is set on. The highest and lowest reference fields are then updated, if necessary.

Determining Evaluation Point. The following process yields the evaluation point. If the DO-loop level of nesting associated with the lowest statement reference is greater than the current level, evaluation of the expression can be removed from at least one DO loop. The head of the outermost DO loop containing this lowest (earliest) reference is located; that is, the identification of each parallel DO statement through which the compute point was moved is compared with the lowest occurrence value until the highest DO statement less than the lowest reference is located. The compute-expression node is removed from its current location in the encoded source-program list and linked into the program list immediately preceding this outermost DO and is flagged to indicate that the expression is evaluated at this point. This is the point in the program where this expression will be evaluated. An example of this is shown in Figure 8(a). The value of its highest referenced statement is added to the compute-expression node. Now the expression node update routine is called for each of the two operands of the expression, since they will be referenced in compiling the containing expression. The reference point of these two operands is the containing expression's evaluation point, that is, the head of the loop from which it was removed.

If the computation of the expression is not removed from a loop, the multiple-reference flag is examined. If it is not set, the compute-expression node is deleted from the list. If it is set, the value of the expression's highest referenced statement field is placed on the compute-expression node and is flagged for evaluation at first occurrence of the expression. In either case, the evaluation point of the expression is at the first occurrence (that is, lowest statement field) as shown in Figure 8(b). The expression node update routine is called for the processing of its two operands to give them a reference at this evaluation point.

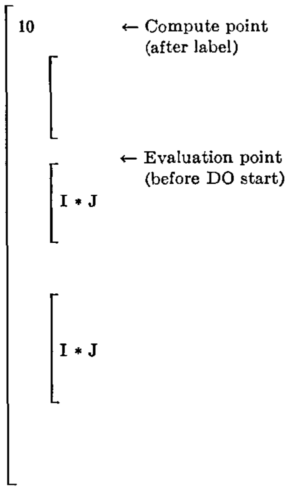
10
← Compute point (after label)
← Evaluation point (before DO start)
I * J
I * J

Fig. 8(a). Example (a) of compute-point determination

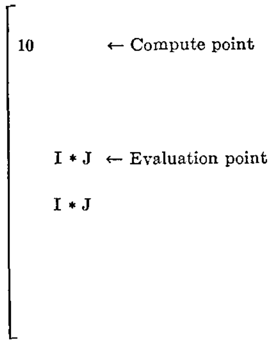

Fig. 8(b). Example (b) of evaluation-point determination

Also during this backward scan, expressions referencing DO-iteration variables are examined to determine their initializing and incrementation expressions. These expressions are added to the expression dictionary so that they may be part of this analysis. Also, the parallel DO pointers in the encoded source program list are reversed for use by the DO loop processing of pass II. After completion of pass II, the evaluation points for all expressions have been determined.

4. Pass II: Register Assignment and DO Loops

Register Assignment. In addition to the processing described above, pass II can determine register assignment optimizations on DO loops. The processing described for register assignment was developed for a multiple index register machine (for example, UNIVAC 1108) but can be extended for a machine with no distintion between index and arithmetic registers.

Each subscript expression whose value is invariant over the loop or limited by the DO variable of the loop is a potential candidate for permanent index register assignment over the range of that loop. For each loop, a list containing one entry for each such subscript expression is maintained. Each entry contains a count of the number of times the expression was referenced in the loop. This list is built during the backward scan of pass II as the indexing expressions are encountered.

When a DO statement program-list node is reached, the expressions to be permanently assigned to registers during the loop are determined. Thus, innermost loops receive assignments first. When reaching a DO statement program-list node, a variable (called N) associated with the loop is set equal to the number of index registers allocated to the indexing expressions of the inner loops of the nest. The N for an innermost DO is set to zero. For outer DO's, N is set to the maximum N for all contained loops at the next level of nesting. Expressions are assigned to registers until the number of assignments reaches the maximum number of registers available for assignment (called M). M is dependent on the number of registers in the machine and on various machine and system characteristics.

At the beginning of a DO loop it is first determined whether the number of subscript expressions removed from the loop plus the initial N for the loop is greater than M. If not, all of the subscript expressions can be permanently assigned, and the following sort step can be bypassed. If there are insufficient registers available, the list of subscript expressions removed from the loop is ordered by frequency of occurrence to allow those expressions with the most frequent occurrence to be processed and assigned first.

The entire list of subscript expressions associated with the loop is scanned to determine which of the expressions can be permanently assigned. If during examination of a subscript expression, N for the loop is currently less than M, then the subscript expression under consideration can always be given a permanent assignment over the loop in question. When this condition exists, the subscript expression is added to the list of permanent index register assignments for the loop, and the effect of this assignment on N is then determined. If N is not currently less than M during examination of a subscript expression, the subscript expression can be permanently assigned over the loop only if this assignment does not increase the value of N.

To determine the effect of an assignment on the value of N, the list of permanently assigned subscript expressions for each inner DO at the next level of this nest is examined. If the subscript expression in question is in the list for an inner DO, then assignment over the current DO will have no effect on the N for the inner DO. If, however, the expression is not in the list of an inner DO, the N for the inner DO must be increased by one if the expression is permanently assigned over the outer DO (since permanent assignment over an outer DO implies permanent assignment over an inner DO).

After determining the effect of a permanent assignment on the N's of the inner DO's, the maximum of the adjusted N's is taken. If it is not greater than M, the assignment can be made, and the N for the current loop and the N's for all inner loops at the next nest level are adjusted as required. If the maximum of the new N's is greater than M, then the expression cannot be assigned, and none of the N's is changed.

Consider, for example, the DO nest shown in Figure 9.

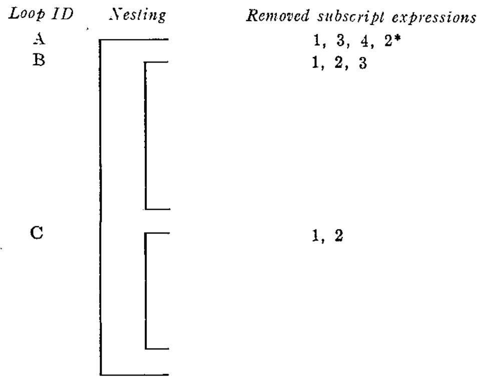

Fig. 9. DO nesting. * Indicates sorted on frequency

When reaching the DO for loop A, subscript expressions 1, 2, and 3 have been permanently assigned in loop B, and thus, N_b equals 3. N_c equals 2 since expressions 1 and 2 were permanently assigned in loop C. Assume that M equals 4. When reaching the DO for loop A, N_a is initially computed as 3. Since the number of subscript expressions (4) plus N_a is greater than M, the sort on frequency is performed. Now assignment inspection begins. Subscript expression 1 is permanently assigned over loop A without affecting N_a, N_b, or N_c. Subscript expression 3 is assigned over loop A. However, this causes N_c to be increased to 3. As it turns out, subscript expression 4 can also be permanently assigned. This assignment causes the three N's to be equal to 4. If, however, M equaled 3, then subscript expression 4 could not be assigned. Subscript expression 2 may also be assigned, even though N equals M. Each subscript expression that is given a permanent assignment and whose compute point is less than or equal to the DO-statement node of the next outer loop is given an occurrence in that next outer loop, since the expression must be loaded at the top of the inner loop if not loaded permanently over the outer loop. Thus, the subscript expression will be considered for a permanent assignment at the next outer loop of the nest, even though there were no other occurrences within the outer loop.

Optimal Subscripts. The compute point of subscripts dependent on more than one DO variable of the nest may often be adjusted to move the initialization to an outer loop.

These subscripts can be termed "optimal subscripts." A subscript is said to be optimal if it meets the following criteria:

1. The compute sequence of the subscript is at a DO-statement node.
2. The subscript contains a DO variable from an outer loop of the nest.
3. The non-DO variable elements of the subscript have a compute point outside the next outer loop.
4. The DO limits have a compute point outside the next outer DO.
5. The subscript expression can be loaded permanently over the range of the next outer loop.
6. The loop associated with the compute sequence of the subscript is optimal; that is, the loop is always executed a fixed number of times for each execution of the next outer loop of its nest.

If these criteria are met, the initial value of the subscript expression computed using the initial loop value for the DO variable can be loaded at the beginning of the outer loop. Each time through the inner loop, the value of the index expression is advanced by an increment. Each time the end of the outer loop is reached, the subscript expression must be decremented by an amount to bring it back to its original value, taking into account the effect of the new value of the DO variable of the outer loop. An example of an optimal loop appears in Figure 10. This process may be repeated through as many loops as the criteria hold.

```
    DIMENSION A(10, 10)
                                Load XR1 (index register 1) with 11
    DO 10 I = 1, 10             Loop "DO 10" definition point I
     :
    DO 20 J = 1, 10             Loop "DO 20" definition point J
    A(I,J) = 0                  (Address of A − 11) + XR1 = 0
 20 CONTINUE                    XR1 = XR1 + 10
        :
 10 CONTINUE                    XR1 = XR1 + 1 − 100
```

Fig. 10. Optimal loop expansion

When a subscripting expression with a compute point at a DO-statement node is encountered, it is scanned to determine the compute points of the non-DO-variable operands of the expression and the DO variable with the earliest definition point. The most limiting of the two values becomes the outermost point to which the subscript may be optimized. The optimal criteria are considered at each of the loops of the nest either until they are found to be violated or until the outermost point to which the subscript may be optimized is reached. Each time the criteria are met, the adjusting increment expression is determined, and, if it is not zero, an increment item is generated and linked to the encoded source program list at the associated CONTINUE-statement node. When the subscripting expression can no longer be optimized, the initial value expression of the subscript is determined by replacing the DO variables associated with each of the loops over which it was optimized with the associated initial value. An initial value item is generated and linked to the encoded source-program list at the associated DO-statement node.

Collapsing Loops. It is often determined that an inner loop can be collapsed and combined with its next outer loop. For example, this happens when the program is going column-wise through an entire array. This occurs if the A(I, J) = 0 in the program in Figure 10 is replaced by A(J, I) = 0.

The criteria that must be met to combine an inner loop with its next outer loop are:

1. There are no executable statements or code inserted by the compiler between the DO-statement nodes or the CONTINUE-statement nodes of the two loops.
2. The DO variable of neither the inner nor the outer loop is to be materialized; that is, the DO variable is not used computationally within either loop, nor as an argument, there is no branch out of the loop, and (if the DO variable is in common) there is no procedure invocation.
3. All DO-variable subscripts of the loops are optimal and contain both of the DO variables. The adjusting increment of each of these subscripts at the ends of the loops must be zero.

When these conditions are met, the inner DO is set as noniterative, causing phase III to ignore the loop when it generates code. The upper limit of the outer loop is set to the initial value of the outer loop minus one plus the product of the number of times the inner loop would have been executed, the number of executions of the outer loop, and the increment of the outer loop. The collapsing process is applied until a loop is encountered that does not satisfy the criteria.

5. Pass III: Code Generation

This last pass over the encoded source program is made forward; that is, from the beginning of the program to the end. The purpose of this pass is to form the machine-language instructions.

References to an expression which appear in statements bounded in the encoded source program list by an expression-compute node and the statement indicated in the last reference field in the compute node refer to a common value of the expression. References not within such bounds require distinct evaluations. The current status of an expression is maintained in the compute-point field of its dictionary node. When pass III begins, all of these fields are equal to zero. The compute-point field is updated from the last reference field of an expression-compute node whenever such a node for the expression is encountered in the program list during this pass.

When a reference to an expression is processed, the expression's compute-point field is compared with the identification of the statement currently being processed. If it is less, code is generated to evaluate the expression. Since this is a one-time reference to this value, it is not necessary to keep the resulting value available for later use. However, if the compute-point field is greater than or equal to the current statement identification, the expression need be evaluated only if its dictionary node is flagged, indicating this as its first reference since encountering the expression-compute node. When the expression is newly evaluated, this flag is reset and the location of the result (register and/or temporary storage) is recorded in its dictionary node; subsequent references are to this location until the statement indicated by the compute point is passed. Any change in location of the expression is reflected in its dictionary entry. Figure 11 shows an example of this sequence for a program segment.

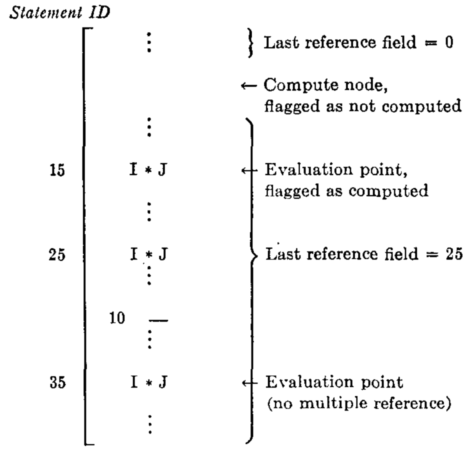
Statement ID
Last reference field = 0
Compute node, flagged as not computed
15
I * J
Evaluation point, flagged as computed
25
I * J
Last reference field = 25
10
35
I * J
Evaluation point (no multiple reference)

FIG. 11. Code generator evaluation point processing

When a DO statement node is reached, it is necessary to choose the index registers to be assigned to the subscript expressions permanently assigned over the loop. Registers chosen first are those with no worthwhile values in them. If not enough registers are available, instructions must be generated to store the results of the registers being freed before generating the instructions to load the registers. When the contents of a register are stored in memory, it is necessary to update the corresponding expression-dictionary node. Any values left in registers at the beginning must remain there throughout the entire loop; thus they can also be considered permanently assigned over the loop.

When a register is required and a free one is not available, it is desirable to know which of the registers contains the value that is referenced furthest down in the program. This register can then be freed leaving the other registers intact, since their values will be used sooner. This "next use" information for the values of the expressions in registers is obtained from a sublist that is built for each expression during pass II. There is one entry in this list for each time the value of the expression is evaluated and for which there are multiple references to that value. Each of these entries points to a sublist containing an entry for each time that value was referenced. When it becomes necessary to free a register, the proper sublist for each expression currently in a register is examined and the register whose next reference is furthest away is selected.

When a DO loop has an extended range, it is necessary that all index registers permanently assigned over the loop contain the same values on branching back into the loop as they had when the branch out was made. This is achieved by loading these registers with the values of the globally assigned expressions at each statement whose statement number is referenced from outside the loop.

6. Conclusion

There is certain information available to the programmer that is not made available to the compiler through the FORTRAN source program. This lack of information requires that the compiler always assume the worst case. For example, the process presented assumes that any subroutine call potentially redefines all of the variables in common. This severe assumption is generally not needed but is required in order to generate safe code.

Thus the method described in this paper cannot guarantee an improvement in the performance of the object code. In fact, it is possible to degrade this performance. For example, if a section of code within a loop is invariant over the loop, the method described here removes the calculation from the loop. If this expression appears in a section of the loop that is executed conditionally, it is possible that this value is not referenced during execution of the loop. In this case, a useless evaluation will have occurred. While the optimization techniques presented here are not the quintessence, several implementations of these algorithms have shown that they significantly improve the object code for almost all FORTRAN programs.

Experience has shown that some of these optimizations actually pay for themselves. This is because the time needed to perform the optimization analysis is less than would be required to perform straightforward code generation of the extra instructions produced when the optimization is not performed (for example, register assignment and instruction assembly). Furthermore, these compilers, while not achieving the compilation speeds of a one-pass processor and while requiring adjustment for differences in machine speeds, have proved to be faster than most multipass FORTRAN compilers.

Acknowledgments. The algorithms described in this paper were developed over a number of years and used in the implementation of a number of CSC FORTRAN compilers. Ideas and improvements for these algorithms were contributed by many of the Systems Programming personnel working on these projects. Major contributions to this work are acknowledged to R. Nutt, J. Erdwinn, L. Gatt, G. Rice, K. Balke, and S. Sidrane.

RECEIVED OCTOBER, 1968; REVISED JANUARY, 1969 AND JUNE, 1969

REFERENCES

1. GEAR, C. W. High speed compilation of efficient object code. *Comm. ACM 8*, 8 (Aug. 1965), 483–488.
2. NAKATO, IKUO. On compiling algorithms for arithmetic expressions. *Comm. ACM 10*, 8 (Aug. 1967), 492–494.
3. NIEVERGELT, J. On the automatic simplification of computer programs. *Comm. ACM 8*, 6 (June 1965), 366–370.
4. RYAN, J. T. A direction-independent algorithm for determining the forward and backward compute point for a term or subscript during compilation. *Comput. J. 9*, 2 (Aug. 1966), 157–160.
5. LOWRY, E., AND MEDLOCK, C. W. Object code optimization. *Comm. ACM 12*, 1 (Jan. 1969), 13–22.