Programming Techniques

R. M. McCLURE, Editor

Object Code Optimization

EDWARD S. LOWRY AND C. W. MEDLOCK*
International Business Machines Corp., Poughkeepsie, N. Y.

Methods of analyzing the control flow and data flow of programs during compilation are applied to transforming the program to improve object time efficiency. Dominance relationships, indicating which statements are necessarily executed before others, are used to do global common expression elimination and loop identification. Implementation of these and other optimizations in OS/360 FORTRAN H are described.

KEY WORDS AND PHRASES: compilers, data flow analysis, dominance, efficiency, FORTRAN, graph theory, loop structure, machine instructions, object code, optimization, redundancy elimination, register assignment, System/360
CR CATEGORIES: 4.12, 5.24, 5.32

Introduction

The implementation of object code optimization techniques in the OS/360 FORTRAN H compiler is presented and the potential extensions of these techniques are discussed.

The compilation process basically converts programs from a form which is flexible to a form which is efficient in a given computing environment. Compiler writers are challenged on the one hand by increasingly complex hardware and on the other by the fact that much of the complexity and rigidity of large, costly programs results from conscious efforts to build in efficiency. Since the methods of analyzing and transforming programs used in FORTRAN H are relatively unspecialized, it is hoped that they will help form a basis for solving the many remaining problems.

A major novel technique used is the computation of "dominance" relationships indicating which statements are necessarily executed before others. This computation facilitates the elimination of common expressions across the whole program and the identification of the loop structure (not depending on DO statements). No distinction is made between address calculations resulting from subscripts and other expressions. Another important technique is the tracing of data flow for unsubscripted variables.

The FORTRAN H compiler performs the most thorough analysis of source code and produces the most efficient object code of any compiler presently known to the writers. For small loops of a few statements, it very often produces perfect code. The efficiency is limited mainly by the (rather unnecessary) restrictions on the types of data organization that can be described in FORTRAN and the inability to combine subprograms in a single compilation. Of course the optimization does not compensate for inefficient source coding except in minor ways. The methods will apply to almost any other procedural language, but they are not effective for interpretive systems or for object code that relies heavily on precoded library subroutines.

Most optimization can be optionally bypassed to gain a 40 percent reduction in compilation time. However, this expands the object code by about 25 percent, and execution times are increased threefold.

Compiler Organization

The optimization within the FORTRAN H compiler proceeds as follows. In a pass through text the arithmetic translator converts FORTRAN expressions into three-address text and, at the same time, builds up "connection lists" which describe the possible forward paths through the program. The connection lists are then sorted to give the flow backward as well as forward.

From the connection lists the dominance relations are computed which provide the information from which loops are identified along with their initialization blocks. A scan is then performed for most unsubscripted variables to determine where in the program the variable is "busy," i.e. at what points it contains a value that will be subsequently fetched.

The text is then examined for possible optimizations starting with the innermost loops and working outward until the whole program is covered. First, common expressions are eliminated from each loop. Then loop-independent expressions are moved out of the loop. Some statements of the form $A = B$ are eliminated by replacing references to A with B. Induction variables are then identified and multiplications of them are reduced to additions by the introduction of new induction variables. An attempt is then made to reorder some computations to form additional constant expressions that may be moved out of the loop.

When all loops have been processed by the "machine-independent optimizer" or text optimization phase, a similar pass is made through the loops doing register

* Present address: Sun Oil Co., DX Division, Tulsa, Oklahoma

allocation. For each loop some registers are assigned for very local usage and other available registers are assigned each to a single variable across the loop so that no memory references to that variable occur in the loop. Status bits are set to indicate which variables, bases, and index quantities must be loaded into registers or stored for each text entry.

On the basis of the status bit settings, the amount of code that will be generated for each text entry can be calculated; hence, the branch addresses can then be computed. Finally, the object code is generated from the text. The text retains the same format throughout optimization except that register numbers and status bits are set during register allocation.

Division of the Program into Blocks

In the arithmetic translator phase the program is broken into computational "blocks" whose relationship may be represented by a directed graph (see Figure 1) that illustrates the flow of control through the program.

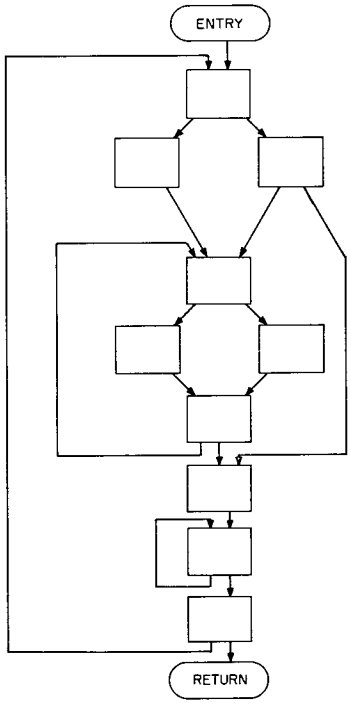

FIG. 1. Block structure

Each block consists of a sequence of statements, only the first of which may be branched to, and only the last of which contains a branch. Logical IF statements may produce more than one block. If the statement following the logical expression is not a GO TO, it will form a separate block. If the logical expression contains .AND. or .OR. operators, the statement will generally be broken down into a set of equivalent statements that do not contain such operators. This transformation considerably accelerates the execution of logical IF statements by avoiding evaluation of many logical expressions and reducing the explicit use of logical variables [1]. For example,

IF (*A.LT.B*.OR.*C.GT.D*) GO TO 10

is equivalent to

IF (*A.LT.B*.) GO TO 10
IF (*C.GT.D*) GO TO 10.

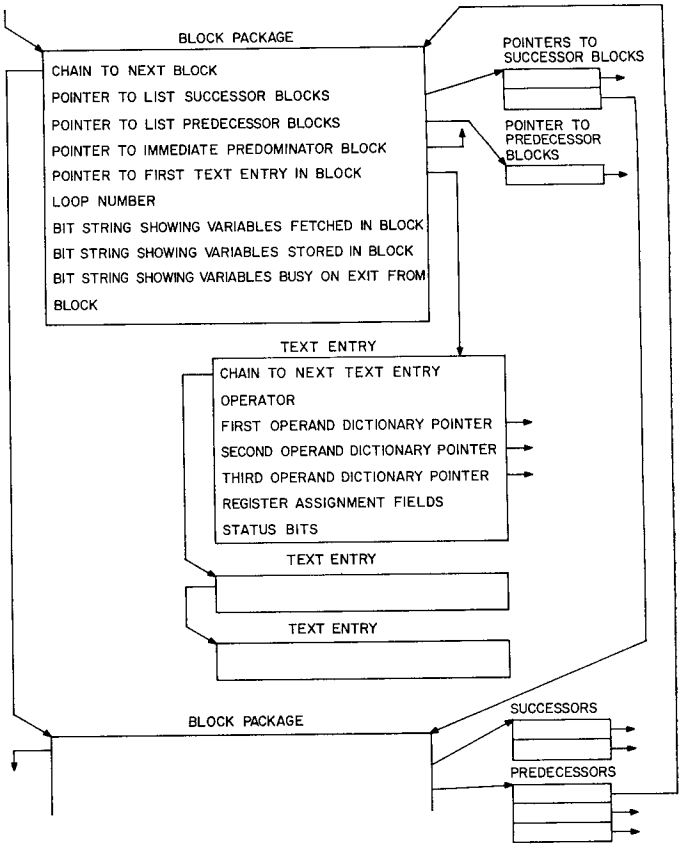

FIG. 2. Organization of block packages, text entries and connection lists.

Internally, a structure of fields called the block package (see Figure 2) is associated with each block. One field points to a list of all blocks that could be executed immediately after the given block. Such blocks are called "successors" of a given block. The lists of successors are constructed during the arithmetic translator phase, and they must be complete. Thus errors could be introduced into the program if the list of statement numbers in an assigned GO TO statement is not correctly provided. After the lists of successors are completed, they are sorted to provide lists of predecessors for each block.

"Program entry blocks" are distinguished by their lack of predecessors, and "return blocks" are distinguished by their lack of successors.

Three bit strings are also included in each block package. Each string is 128 bits long and describes the status of variables in the block. The first string indicates which variables or constants are fetched in the block. The second string tells which variables are stored in the block. The third string, which is set during the data flow analysis, indicates which unsubscripted variables are busy on exit

from the block. These strings constitute a significant space overhead, but they save much scanning.

Dominance Relations

The idea of dominance relations between the blocks of a program was suggested by Prosser [2] and refined by Medlock. We say that a block I "predominates" a block J

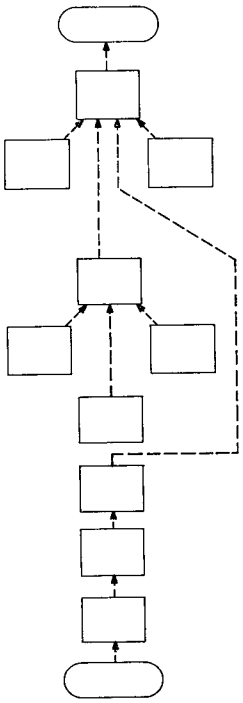

FIG. 3. Dominance relations. Each block points to its immediate predominator.

if every path along a sequence of successors from a program entry block to J always passes through I. Conversely we may define a post-dominance relationship, but it is less useful.

The relation is transitive: If I predominates J and J predominates K, then I predominates K. Further, if block K is predominated by both blocks I and J, then we can show that either I predominates J or vice versa. We may conclude that if a block K is predominated by several blocks, then one of them, J, is predominated by all the other predominators of K. We call this block the "immediate predominator" of K.

All the predominance relations in a program may then be summarized by indicating in each block package the immediate predominator for that block (if there is any). The set of all predominators of a block is given by scanning along the chain of immediate predominators. The dominance relations between the blocks of Figure 1 are illustrated in the tree-like pattern of Figure 3.

To compute the immediate predominator for a block K, we may first lay out some arbitrary nonlooping path from a program entry block to K. The path contains all predominators of K and the one closest to K on the path is its immediate predominator. We then remove from the path the block nearest to K if we find a chain of predecessors from K to a program entry block or a more remote block on the path where the chain does not go through the nearest block. The closest block remaining on the path after repeatedly removing blocks in this way is the immediate predominator.

Loop Analysis

It is very desirable to move computations and storage references from frequently executed blocks to infrequently executed ones whenever possible. In practice, this means moving loop-independent computations out of loops and assigning registers to variables across loops. In the absence of more complete information, we assume that depth of loop nesting and frequency of execution are related. Our procedure recognizes only those loops that are entered at one point; but the others pose no serious problem since, in principle, they can be reduced to single entry loops by replicating part of the program, and, in practice, they do not occur often. The optimization makes no distinction between DO loops and equivalent IF loops. The loop analysis described here is a slight improvement on the one actually used in the compiler.

If a loop is entered at only one point, then that loop entry block must predominate all members of the loop. We may scan for all blocks J which are branched to from some block predominated by J (or which is J itself), and we flag those blocks as loop entry blocks (see Figure 4). Usually a loop entry block is immediately preceded by an initialization block for the loop. If the immediate predominator of a loop entry block has just one successor,

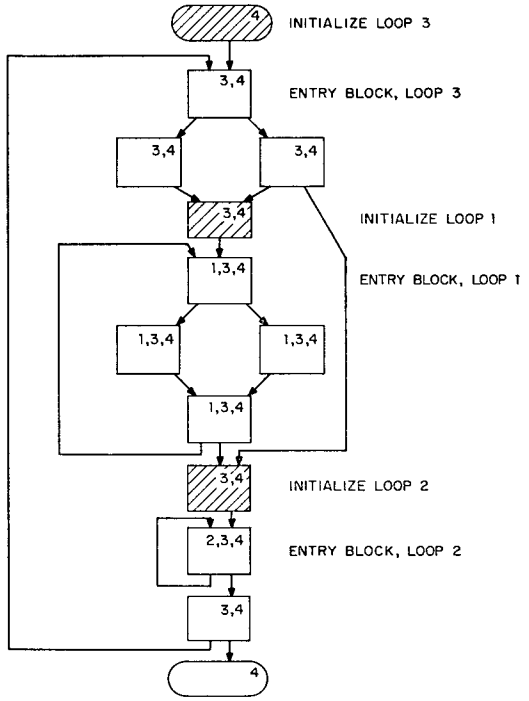

FIG. 4. Loop structure: Membership in four loops is indicated. (Initialization blocks are shaded and the initialization block for loop number 1 has been inserted.).

then it is a satisfactory initialization block. Otherwise, a block is inserted to branch or fall through to the loop entry block. Some of the blocks that are predecessors of the loop entry block are then changed so they branch or fall through to the inserted initialization block instead of the loop entry block. The blocks so changed are those that are not predominated by the loop entry block (nor are they the loop entry block itself).

We may then associate with each block K an "initialization block" for the most deeply nested loop which contains K. To do this, we scan the chain of immediate predominators of K until we find a loop entry block such that there is a forward path from K to the loop entry block that does not include the immediately preceding initialization block. If such an initialization block is found, it then becomes the initialization block for K.

If a block has no initialization block, we assign it a nesting depth of zero. If the initialization block for a block K has a nesting depth N, then block K has nesting depth $N + 1$. Nesting depths are used only in the assignment of loop numbers, a discussion of which follows.

Blocks are assigned loop numbers so that all blocks with a common initialization block are assigned the same number, and the members of a loop have a lower number than their initialization block (which is not considered a member of the loop).

In performing text optimization and register assignment, we start with the lowest loop number (an innermost loop) working from inner to outer loops until the whole program has been covered. The last "loop" processed is not a normal loop since some blocks in it do not lie on a closed path. When we finish working on a loop, all its member blocks are marked as completed and each is assigned the same loop number as its initialization block. This has the effect of identifying the blocks of the inner loop as a properly nested part of the next outer loop.

Note that the blocks belonging to a loop may not correspond exactly to the range of an associated DO. A branch out of the range of a DO will cause other statements (the extended range of the DO) to be added to the loop if there is a branch back into the DO range. Both an unconditional GO TO which branches out of the range of a DO without returning and possibly some preceding statements will be excluded from the loop.

Data Flow Analysis

We say that a variable is "busy" at a point in the program if at that point it contains data that will subsequently be fetched. An unsubscripted variable is not busy immediately before a store into it because any data it contains cannot be subsequently fetched as it will be erased by the store operation (see Figure 5).

Information about where a variable is busy is useful in the following ways:

1. When a variable is assigned to a register across a loop, it must generally be loaded on entry to the loop and stored on exit from it. However, if it is known that the variable is not busy at a point crossing the loop boundary, the corresponding load or store may be omitted. Further, if a variable is altered within a block but is not busy on exit from the block, it is usually not necessary to return the variable to storage. Within such a block only register references to the variable need occur.

2. An induction variable may sometimes be eliminated from a loop if it is not busy on exit from the loop.

3. We may sometimes move the store of a variable into the initialization block of a loop if its not busy-on-entry to the loop.

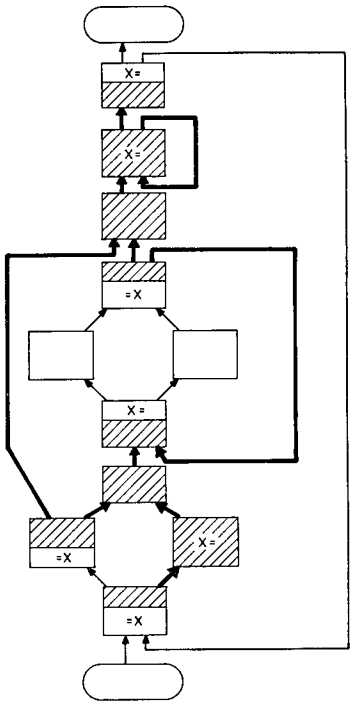

FIG. 5. Data flow. The shaded areas and heavy lines indicate where variable X is busy.

4. Statements of the form $X = Y$ may sometimes be eliminated. If X is stored into just once and Y is not stored into when X is busy, then all references to X can be replaced by Y. Here X is said to be "subsumed" by Y.

5. If two variables are not busy at any common point, then they may be combined into a single variable. This facilitates register allocation and saves space for one of the variables.

The information on where variables are busy is used in FORTRAN H in each of the above ways except the last. The combining of external variables to save space has been done by Yershov [3].

In the arithmetic translation phase the most frequently referenced variables, constants, arrays, and address constants are each assigned a number from 2 to 127 to identify their respective positions in the bit strings of the block package. Other variables and constants get no attention during optimization.

If two unsubscripted variables of the same type and length are equivalenced, we treat them as a single variable. Otherwise, equivalenced variables are excluded from the set included in the bit strings because of the danger of

changing the program logic during optimization. If a program adheres to USASI Standard FORTRAN, this problem will not arise; but in practice it does. An additional level of optimization that optimizes equivalenced variables for those programs conforming to the standard could be introduced.

Arrays and constants are assumed to be busy everywhere in the program. For each unsubscripted variable with a position in the bit strings, a scan is made to determine which blocks it is busy on exit from. Appropriate bits are set in the busy-on-exit strings for each block. Using the fetch and store bit strings, the scan is made starting at each fetch and working backward along all paths until stopped by stores. Initially the busy-on-exit strings are used to indicate whether the store or fetch precedes when both occur in a block. All COMMON data is assumed to be both fetched and stored at a CALL, a nonlibrary function reference, or a return block. If the scan reaches an entry block to a subprogram, it continues at all return blocks so that local variables can be remembered from one execution of the subprogram to the next.

Common Expression Elimination

The first of the global optimizations performed on a given loop is common expression elimination (see Figure 6). Common expressions have usually been combined into a single calculation only when they appear in a single statement. In the FORTRAN H compiler, widely separated common expressions are found and combined. This means that the scans must be performed quickly, and that we must check for statements that may change the operand values of the expression between the two appearances of the expression.

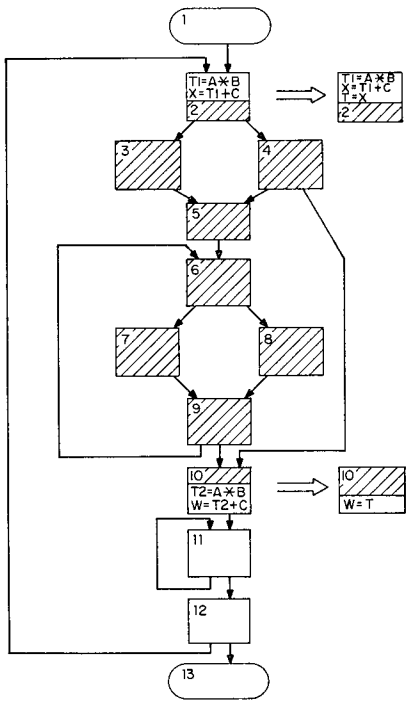

FIG. 6. Common expression elimination. Blocks 2 and 10 are shown before and after the elimination. Shaded areas indicate the locations of potentially interfering stores.

The process is particularly important for higher-dimensional subscript expressions in FORTRAN since these occur repeatedly and generate extensive address arithmetic. There is no convenient way to eliminate such common expressions at the source language level.

At any point we search only for matching expressions consisting of two operands and an operator, but repeated application may combine much larger expressions. An expression such as $A * B$ may be replaced by the result of an earlier computation of $A * B$ or $B * A$ provided that the earlier occurrence predominates it and neither A nor B can be altered between the two evaluations of the expression.

The search for an expression matching the expression $A * B$ which occurs in a block K is then limited to text items preceding it in both block K and other blocks in the same loop that predominate K, but are not in inner loops that were processed earlier. For the following reason there is no need to search outside the loop containing K: If there is any possibility of replacing $A * B$ by the result of an earlier computation done outside the loop, it is also possible to move the expression outside the loop into the initialization block. The movement outside the loop will be done later, and the common expression that remains will be found when processing the outer loop.

The scanning for common expressions could grow in proportion to the square of the program size; so to prevent very long scans only the first ten predominating blocks are examined. Only blocks whose bit strings indicate that both A and B are fetched are considered. A hash-table technique could be used to shorten the scan. The hash index would be formed from the two operand pointers and the operator.

Every expression in block K is first compared with every other expression in block K. Then every expression in block K is compared with expressions in the immediate predominator of K, and then its immediate predominator, and so on. At each stage the "store" bit strings for all blocks between K and the predominating block are OR'ed together to easily determine which variables might have been changed.

To identify the set of "interfering blocks" between K and its immediate predominator, we search backward from block K marking blocks along all paths without passing through the immediate predominator. When looking for matches in the second predominator of K, we search backward from the immediate (or first) predominator and mark additional blocks without passing through the second predominator, and so on.

To make rapid scans backward from a block along all paths, an auxiliary, working vector may be used in which a list of all blocks reached is developed. Two pointers to the list are maintained between which one can locate at any time the set of all blocks already reached by the scan, but whose predecessors have not yet been examined.

If we find a store into A or B, either above the expression

$A * B$ in block K or in an interfering block at any stage, we may flag as unmovable the text item containing $A * B$. We omit it from consideration when examining subsequent predominators since the same interfering stores will prevent its elimination.

When two suitable occurrences of $A * B$ are found for which no interfering stores occur, the action taken depends on the result operand. Suppose the earlier occurrence has the form $P = A * B$ and the later occurrence has the form $Q = A * B$. If P is not stored into at any interfering point, we replace the second text item by $Q = P$. If P is altered in between, we create a temporary variable T, insert $T = P$ immediately after the earlier text item, and replace the second item by $Q = T$. Furthermore if Q is a generated temporary (which must be stored only once), we immediately eliminate the statement $Q = P$ or $Q = T$ and replace fetches of Q by P or T, respectively. If the original sequences were $T1 = A * B$, $X = T1 + C$ in the predominating block, and $T2 = A * B$, $W = T2 + C$ in block K, we would then replace the second sequence by $W = T1 + C$ in block K and thus set up another common expression $(T1 + C)$ for elimination (see Figure 6).

Proliferation of generated temporaries that are busy across block boundaries should be avoided since it is generally necessary to allocate space for them. In the above example, $T1$ becomes busy between two blocks as a result of the first expression elimination, but is then eliminated from the later block by the second. When a temporary becomes busy across a block boundary, we should refrain from allocating space until we are sure it will not return to its local status. When space is allocated, it is best to use any existing variable of the same type we can find which is not busy in the region spanned.

Eliminating common expressions of the form $B(I)$ poses a problem. It may be necessary to store $B(I)$ into a temporary location and fetch the temporary later in place of $B(I)$. Fetching the temporary later may not be significantly faster than fetching $B(I)$, but we pay the penalty of storing into the temporary anyway. On the other hand if we fail to eliminate such a common expression, we will fail to recognize any larger common expression of which $B(I)$ may be a part.

Backward Movement

An attempt is made to move all loop-independent computations out of loops. An expression such as $A * B$ may be moved from a loop to the corresponding initialization block if neither A nor B is stored in the loop. We may OR together the store bit strings for all blocks in the loop to identify variables that are unchanged in the loop.

Suppose A and B are constant in the loop. A text item of the form $X = A * B$ may be moved to the end of the initialization block if X is not stored elsewhere in the loop and is not busy on exit from the initialization block. If such a movement is made X is identified as constant in the loop, thus permitting other movements.

If the whole text item cannot be moved, we may create a temporary T and insert $T = A * B$ into the initialization block and replace the original text item with $X = T$. It is desirable to perform common expression elimination before backward movement since otherwise several temporaries could be generated when moving different occurrences of a single constant expression out of the loop. In general, absolute constants and variables that are constant within the loop are treated in the same way except that expressions involving only absolute constants will be directly evaluated rather than moved out of the loop.

In an expression such as $A * K1 * K2$, where $K1$ and $K2$ are constant in the loop, the constant expression $K1 * K2$ would not be so readily detected since the internal text would have the form $T1 = A * K1$, $T2 = T1 * K2$. To recognize constant parts, such expressions must be reordered.

To determine when reordering is appropriate, we may scan forward through the text of a block examining arithmetic text items involving a constant interacting with a variable. A text item of the form $T = V + K$, where K is a loop constant, is said to have a result that includes one additive constant unless V is the result of an earlier text item in the block, in which case it includes one more additive constant than the earlier text item. Similarly, a text item of the form $T = V * K$ is said to have a result including one multiplicative constant or one more than an earlier text item whose resultant is V. For this purpose subtraction is regarded as additive and floating-point division as multiplicative. When a text item has a result that includes two additive or two multiplicative constants, we reorder.

Thus the final result of an expression such as

$$(K1 + A)/K2 - K3$$

would be regarded as including one multiplicative and two additive constants. It would then be expanded and reordered as though its form were $A/K2 + (K1/K2 - K3)$. The expression $K1/K2 - K3$ would then be recognized as a constant that can be moved out of the loop; so one addition would be saved in the loop. Since the reordering of floating-point computations may yield different results, the reordering of expressions is invoked only when a separate optimization level is specified.

Absolute constants occurring in subscript expressions are normally absorbed into the referencing instruction address. However, an additive portion of the address arithmetic expression resulting from a subscript may be variable, but constant in the loop. Rather than adding such a quantity into the index value within the loop, we may add it into the instruction address (or base register on System/360) outside the loop. In FORTRAN H this is done when a subscript operator is encountered whose index value includes an additive constant.

The movement of some loop-independent expressions to the initialization block may result in error conditions. For example, a square root function may be evaluated within a loop only when the argument is positive. If we move it to the initialization block without determining the sign of the argument an error condition will arise when the argu-

ment is negative. Overflows and division by zero may also arise in this way. In FORTRAN H functions like sine, exponentiation, and division may be moved, although square root, arcsine, and log may not. The former are less likely to cause trouble, but occasionally error messages will result. Any loop-independent expression may be moved from the loop entry block without danger since no testing can precede it. One approach to solving the problem is simply to ignore the interruptions and error conditions when a high level of optimization is specified; another is to avoid moving an expression past tests involving the operands of the expression or to move such tests along with the expression.

Induction Variable Optimization

An "induction variable" is one that is altered within a loop only by adding to it a constant or a variable that is constant in the loop. Such variables are frequently multiplied by constants (or variables constant in the loop) as in subscript evaluation, and such multiplications can be effectively reduced to additions by introducing new induction variables. The only novelty in this treatment of induction variable optimization is the use of "busy" information.

If I is an induction variable, we may replace an expression such as $I * K1$ by a variable $I2$ in the following way (see Figure 7). At the bottom of the initialization block insert $I2 = I * K1$. At each point (usually just one) where I is incremented by a statement such as $I = I + K2$, insert $I2 = I2 + K3$ where $K3 = K1 * K2$ has been computed as an absolute constant or inserted into the initialization block. The expression $I * K1$ is then equivalently replaced by $I2$, and the multiplication within the loop has been effectively replaced by one addition (sometimes more). In principle, exponentiation of the form $K ** I$ could be reduced to multiplication in an analogous manner. However, there is at present no provision made for such a process in FORTRAN H.

If an expression such as $I + K4$ occurs (in a context other than the incrementing of I) we may replace it by ntroducing a new induction variable $I3$. We initialize $I3$ iy inserting $I3 = I + K4$ into the initialization block and bncrement it in the same way that I is incremented in the ioop. Since this transformation just trades one addition for another, it is not useful except when it clears the way for eliminating the original induction variable.

If uses of an induction variable I are replaced by the above transformations, it may be possible to eliminate all references to I within the loop. If I is not busy on exit from the loop and the only fetches of it in the loop occur when I is incremented or compared with a loop constant, and, in addition, some other induction variable such as $I2$ has been introduced by transforming I as above, then I may be eliminated. We replace I in any comparison operation by the new induction variable $I2$ performing the lsame transformation on the loop constant to which I was compared as was performed on I to initialize $I2$. Thus a comparison such as $I \leq K5$ would be replaced by $I2 \leq K6$,

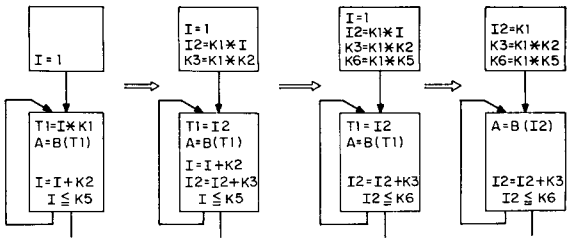

FIG. 7. Induction variable optimization. The first transformation trades an addition for a multiplication. The second eliminates the original induction variable. The third shows further improvements from subsumption.

where $K6 = K5 * K1$ is either computed, or inserted into the initialization block. After such a replacement, the only use of I is to increment itself, and that may be deleted.

Subsumption

A variable X is said to be "subsumed" by a variable Y if references to X are replaced by references to Y. Text items of the form $X = Y$ may sometimes be eliminated when X is subsumed by Y.

In FORTRAN H, subsumption is limited to a single block in which X is either stored into again later in the block or is not busy on exit from the block. All fetches of X are replaced by those of Y if they occur after the $X = Y$ text item but before any store of X. If a store into Y occurs before such a fetch of X but after the $X = Y$ text item, then the subsumption is not possible.

More generally, two variables may be merged if at all points where both are busy they have the same value.

In induction variable optimization we often replace a text item of the form $T = I * K$ by $T = I2$, and we may immediately eliminate the text item by subsuming T.

Register Allocation

Effective register assignment is very important on a System/360 computer, where many registers are available and base registers must be loaded prior to storage references. The most important technique used by FORTRAN H is the rather simple one of noting which variables, constants, and base addresses are referenced most frequently within a loop and assigning many of them to individual registers across the loop.

Before such global assignments are made across the entire loop, two passes are made across each block in a loop assigning generated temporaries and other variables which are both stored and fetched within a block. The first pass goes forward through the block building up tables used during the subsequent backward pass, the one which actually assigns registers. It is more convenient to make the pass backward through the block in assigning even-odd register pairs, matching result operands to the same register as a fetched operand in a text item, and deciding which variables should not be assigned. If very much deeply parenthesized computation occurs in the loop, the four floating-point registers will tend to be used up during this local register assignment procedure. This is of no consequence, however, since globally assigning floating-

MINUS: Used for All Subtract Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	XXXXXXXX00000000
2	LH	R2,D(0,B2)	0000111100000000
3	LH	R1,D(X,B2)	1100000000000000
4	L	B3,D(0,BD)	XX00XX00XX00XX00
5	LCR	R3,R3	0010001000000010
6	LR	R1,R2	0000110100001101
7	LH	R3,D(0,B3)	0100010001000100
8	LCR	R1,R3	0001000000000000
9	SH	R1,D(X,B3)	1000100010001000
10	SR	R1,R3	0100010101110101
11	AH	R3,D(X,B2)	0010000000000000
12	AH	R1,D(X,B2)	0001000000000000
13	AR	R3,R2	0000001000000010
14	L	B1,D(0,BD)	XXXXXXXXXXXXXXXX
15	STH	R1,D(0,B1)	XXXXXXXXXXXXXXXX

FIG. 8. Coding skeleton and bit strings for the subtraction operator.

point variables across a loop is only a minor benefit except in small loops, in which case the registers will be available anyway. A variable locally assigned in a block is generally not stored unless it is busy on exit from the block.

After the local assignment, the most active acceptable variables are selected for global assignment across the loop. Loads and stores of globally assigned variables are inserted at entry and exit to the loop when the variable is busy across the loop boundary. Constants in the loop need not be stored on exit.

All registers needed by an inner loop are assigned for its use. This may leave no unused registers available for global assignment to outer loops. To prevent this, an effort is made to extend global assignments in inner loops to apply to outer loops as well. Because of the importance of having base addresses loaded, provision is also made for assigning registers across outer loops to constants (including base addresses) even when the register is differently assigned to an inner loop. Such registers must be restored on exit from the inner loop.

For large loops it is desirable to assign some variables to registers across block boundaries without extending the assignment across the whole loop. For this reason local assignment is performed by working on small groups of blocks. A suitable group should be entered at just one point, i.e. all predecessors of a member of the group (except the first member) should also be members of the group.

Rather than perform a load of a subscripted variable followed by a register-register (RR) instruction, in System/360 it is preferable to combine these into a register-storage (RX) instruction. To accomplish this it is desirable to reorder the text so that a subscript text item is usually followed immediately by the operation that uses its result.

One register is reserved and permanently loaded for branching, and as many as three others may be preempted if the program exceeds the 4K bytes addressable by one base register.

The above discussion only sketches the scope of the existing register allocation scheme. This scheme is very successful for loops that are not too large, but to be fully competitive with hand coding in large loops, a more refined iterative procedure is needed.

Code Generation

For each text item the code generated will depend on what operands are in registers, what operands must be preserved in registers, whether a result is to be stored, and what base addresses must be loaded to reference storage. During register allocation, these conditions are expressed in eight status bits for each text item: bits 1 and 2 give the status of the first fetched operand; bits 3 and 4 give the status of the second fetched operand; bits 5, 6, and 7 indicate whether base registers must be loaded for each of the three operands; and bit 8 indicates whether the result operand is to be stored or not.

For a fetched operand, the four settings of the two bits have the following meanings:

00—The operand must be fetched from storage and is not retained in registers.

01—The operand must be fetched from storage and retained in a register after the operation.

10—The operand is available in a register whose contents are erased since the result operand is formed in the same register.

11—The operand is available in a register and it must be retained in that register after the operation.

To provide a compact representation of the best possible code for each of the possible status settings, a "coding skeleton" is provided for each operator. The status bits are used to select a bit string which in turn selects from the skeleton those instructions that are actually generated for the given text item. The example in Figure 8 gives the coding skeleton and bit strings for the subtraction operator. The fifteen instructions include, in the order of generation, all those that might ever be generated from a subtraction text item. The instructions are expressed in terms of half-word integer subtraction, but modification to handle full-word and floating-point subtraction is straightforward. The bit string that selects instructions from the skeleton is formed by taking the first four bits describing the fetched operand status and using them to select one of the sixteen columns illustrated. The remaining four status bits are then inserted into the column (replacing *X*'s) to indicate needed base-register loading and the storing of the result operand.

The register allocation and code generation procedures are facilitated by the parallelism between RR and RX instructions on System/360.

Additional Optimizations

For completeness, a number of other kinds of optimization are listed here [4–7]. Only the first three are currently used in FORTRAN H.

1. Use of in-line coding for many mathematical function subprograms.

2. Use of multiplication for exponentiation to an integer. A sequence of square and multiply operations may be used based on the bit pattern of the exponent.

3. If one operand is a power of two, a number of improvements may be possible: Integer multiplication or division may become a shift, multiplication by 2.0 becomes an addition, and multiplication by one or addition of zero disappear.

4. Simplification of logical expressions using DeMorgan's theorem.

5. Conversion of floating-point division by a constant to a (faster) multiplication.

6. Common factor analysis. $A * B + A * C$ becomes $A * (B + C)$.

8. Combination of moves of adjacent fields into a single instruction.

9. Unpacking of certain types of data and repacking it after it has been referenced a number of times rather than operating on it repeatedly in an inconvenient form.

10. Reordering of operations within a block to minimize the number of registers needed or to maximize the opportunities for parallel execution.

11. Reordering of sequences of conditional branches (using the Huffman code principle) to minimize the average time through them. (This depends on frequency information for the different branches.)

12. Elimination of unnecessary unconditional branches by reordering the code. An unconditional branch from within a loop to the loop entry block can often be effectively moved out of the loop by moving the block that ends in the unconditional branch so that it precedes the loop entry block. The unconditional branch is then deleted from that block, and one is inserted into the initialization block to branch to the loop entry block.

Edited Source Listing

In FORTRAN H a documentation aid based on dominance relations is provided in the form of an edited source listing (see Figure 9). The entries to and exits from loops are indicated by the loop numbers in the left-hand margin. Each line of code is indented to illustrate the control flow more clearly. The basic rule for indenting is that a statement S_1 is indented one more than the statement S_2 immediately predominating it unless S_1 is the last statement for which S_2 is an immediate predominator. If S_1 is the last (or only) statement with S_2 as its immediate predominator, it is indented the same as S_2. The value of this system is subjective and not well illustrated in the necessarily short program shown. However, it does illustrate application of the analysis to a very different problem.

```
            C  *   PRIME NUMBER PROBLEM
      100   WRITE(6,8)
        8   FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
           119X,1H1/19X,1H2/19X,1H3)
          101 I=5
(002      103 IF(I.GT.1000) GOTO 7
              A=I
      102     A=SQRT(A)
      103     JBAR=1000
          104     DO 1 K=3,J,2
(001      105     L=I/K
          106     IF(L*K-I)1,2,4
            1         CONTINUE
(001)         C
      107     WRITE (6,5) I
            5   FORMAT(I20)
            2       I=I+2
                    GOTO 3
 002)               C
            4   WRITE (6,9)
            9   FORMAT (14H PROGRAM ERROR)
            7 WRITE (6,6)
            6 FORMAT (31H THIS IS THE END OF THE PROGRAM)
          100 STOP
              END
```

FIG. 9. Edited source listing with loop boundaries and indenting based on dominance.

A Language Suggestion

If the dominance relations and loop structure of a program are analyzed on every compilation, we can use that analysis to give useful meaning to some novel source language expressions. One type of expression could have a form such as (expr1. AT. 20, expr2. AT. 30, expr3). The occurrence of such an expression causes a new variable to be introduced—call it T. At the statement with label 20, the statement T = expr1 is effectively inserted. Similarly, T = expr2 is inserted at statement 30. The statement T = expr3 is inserted at the bottom of the closest block that predominates both the expression and statements 20 and 30, and is at the same (or shallower) depth of nesting as the expression. The expression itself is then replaced by the generated variable T.

The last part of the expression (expr3) then forms an initialization of a value that is subsequently modified, depending on the flow through the program, and finally used. Such expressions could eliminate many housekeeping variables whose meaning is not apparent without searching out all places where they are set.

Compiler Development

The compiler was itself written in FORTRAN and bootstrapped three times. The first time was to convert from running on the IBM 7094 to System/360—an arduous procedure. The second time was to optimize itself, which reduced the compiler size from about 550K to about 400K bytes. It was then rewritten to take advantage of language extensions permitting efficient bit manipulation and referencing of structured data. After the third bootstrapping compilation time was reduced by about 35 percent, and its capacity was nearly doubled; it is now about 700 statements using a 256K byte storage, which is the smallest storage the compiler will operate in.

All dictionary and text are retained in main storage during compilation. This had a psychological advantage during development of the optimization techniques, but it now appears unnecessary. The techniques can be applied using a smaller storage area where the only text in core at any moment is the loop being optimized. There is reason to doubt that the optimization features could have been included in the compiler and debugged in a reasonable amount of time if it had not been written in a higher level language. During debugging there was a tendency for some optimization features to become disabled. This disability often went unnoticed since the test cases still ran correctly.

In building an optimizer there are many temptations to waste effort on insignificant problems.

Conclusion

A number of questions remain which cannot be fully answered at present. Which methods are the most fruitful? Can they be adapted to much smaller, faster compilers? What should have been done differently? Can less general methods be nearly as effective? Are the methods applicable to other than FORTRAN coding?

The following opinions are offered. Probably the global common expression elimination and register allocation were the most important techniques but taken as a whole many minor ones were significant. The flow analysis contributed substantially to the effectiveness of most optimizations. A significantly smaller compiler could probably have the same abilities. One reason is that the optimization (as applied during boot strapping emphasizes speed rather than space). The main defects in the present design are probably the slowness of scanning for common expressions and the excessively cautious handling of equivalenced and complex variables. The general methods used seem to provide a valuable (but probably not very wide) margin of efficiency over more specialized approaches to optimization of scientific programs. The practical effect has been to show that very good code can be compiled for System/360.

The writers are inclined to see the technical value of thorough flow analyses of programs more in terms of the potential extensions discussed below. Three desirable developments seem easily attainable.

(1) It should be possible to code almost any program at a level substantially higher than assembler language without penalty in object code efficiency. The main exception would be when no computer of sufficient capacity is available for optimizing compilation. This is based on the observation that almost any deficiency in the object code of FORTRAN H could be overcome by judiciously tinkering within the established analytic framework.

(2) Optimizing compilers could be economically constructed for several source languages on a single machine type by preprocessing each into a machine-dependent intermediate text which would then be compiled using a common optimizer. This is based on the observation that the optimization procedures in the FORTRAN H compiler were almost completely independent of the features of the source language. The slowness of such compilations would not be a serious limitation when high speed nonoptimizing compilers are also available.

(3) The analyses required to optimize may be extended to detect irregularities in the control flow and data flow of programs. In any sizable program there are a number of categories of data that may interact only in restricted ways, e.g. a temperature may not be meaningfully added to an acceleration, or two address quantities may not be multiplied together. Given some declarations defining such categories and their interactions, the data flow analysis could check large programs for consistency, e.g. before a field in a record is used it may be logically necessary to check a control field to interpret it or before using a pointer in a chained list it may be logically necessary to check for an end of chain. With the help of suitable declarations we can make sure that such checking is actually included in the program. In effect we may go beyond checking for syntactic correctness of the source program to assure that the processing does not violate "syntactic relations" between the data being processed.

Less immediate applications of program analysis will involve extensive restructuring of programs by automatic or semiautomatic means. Very flexible programs can be written using list processing languages, e.g. LISP, SNOBOL, COMIT but there is no practical automatic way to transform such programs to use compact data organizations and avoid redundant scanning. Efficient programs tend to be inherently inflexible since equivalent data is often carried in more than one form for rapid use. This necessitates carefully coordinated changes. The temporaries used in the FORTRAN H compiler are a restricted method of introducing equivalent forms of data. Automatic design of intermediate tables is a logical extension which would greatly enhance machine independence. These considerations suggest that progress toward flexibility in programming depends more on compilation techniques than language improvements.

Other challenges in this area are the adaptation of programs to hierarchies of storage speed and to parallel CPU's. In these problems and in the reduction of redundant scanning the main objective is to reduce randomness of data referencing, or more specifically, to process the data in focus as much as possible before moving on to other data. This suggests that a unified assault on these problems is appropriate.

The writers feel that the area of analysis and transformation of programs is extremely fruitful for programming research. The present economic value of the next few steps is unmistakable, and the first serious technical stumbling blocks have yet to be encountered.

Acknowledgments. Many people contributed to the code optimization of the FORTRAN H compiler. Thanks are due particularly to: G. Lomax, T. C. Schwarz, D. H. Fredricksen, R. K. Stevens, D. Widrig, E. W. Filteau, R. W. Holliday, and J. C. Laffan.

RECEIVED NOVEMBER, 1967; REVISED MARCH, 1968

REFERENCES

1. HUSKEY, H. D., AND WATTENBURG, W. H. Compiling techniques for Boolean expressions and conditional statements in ALGOL 60. *Comm. ACM 4*, 1 (Jan. 1961), 70–75.
2. PROSSER, R. T. Applications of Boolean matrices to the analysis of flow diagrams. Proc. Eastern Joint Comput. Conf., Dec. 1959, Spartan Books, New York, pp. 133–138.
3. YERSHOV, A. P. ALPHA—an automatic programming system of high efficiency. *J. ACM 13*, 1 (Jan. 1966), 17–24.
4. NIEVERGELT, J. On the automatic simplification of computer programs. *Comm. ACM 8*, 6 (June 1965) 366–370.
5. GEAR, C. W. High speed compilation of efficient object code. *Comm. ACM 8*, 8 (Aug. 1965) 483–488.
6. ALLARD, R. W., WOLF, K. A., AND ZEMLIN, R. A. Some effects of the 6600 computer on language structures. *Comm. ACM 7*, 2 (Feb. 1964), 112–119.
7. ALLEN, F. E. Program optimization. In *Annual Review in Automatic Programming, Vol. 5*, Pergamon, New York (in press).