Randolph G. Scarborough
Harwood G. Kolsky

Improved Optimization of FORTRAN Object Programs

For many years the FORTRAN H Extended compiler has produced highly optimized object programs for IBM System/360 and System/370 computers. A study of the object programs revealed, however, that important additional optimizations were possible, and the compiler has been enhanced accordingly. First, the range of cases handled by the optimization techniques already present in the compiler has been extended. For example, more duplicate computations are eliminated, and more invariant computations are moved from inner to outer loops. Second, several new optimizations have been added, with subscript computation and register allocation receiving particular attention. Third, certain optimization restrictions have been removed. This paper describes these improvements and reports their effects.

Introduction

The original versions of the IBM FORTRAN H compiler were written between 1963 and 1967 to support the IBM System/360 [1]. They were based on the pioneering work of Backus [2], whose group wrote the first FORTRAN compilers for the IBM 704 in 1957. At the time the FORTRAN H compiler was completed, it was recognized throughout the industry as having the most thorough analysis of source code of any high-level-language compiler available. The implementation of the object code optimization techniques in the FORTRAN H product is described by Lowry and Medlock [3]. Later the FORTRAN H compiler was extended to include new features available in System/370 and took on the name FORTRAN H Extended [4]. A library containing mathematical functions and input/output support routines was included in a companion product called the FORTRAN Mod 2 Library [5].

After many studies of FORTRAN programs in situations where the execution speed of the object program was a critical factor, we realized that the FORTRAN H Extended compiler did not always produce the expected highly optimized code. In 1976 a small study was started to determine why certain inner loops in FORTRAN programs were handled quite differently by the compiler when they were imbedded in different subroutines. The study revealed that restrictions within the compiler often impeded the optimizations the compiler was designed to perform. Some optimizations were performed incompletely or were constrained, apparently to save time during compilation, to operate only on subsets of the program. The study also revealed that several additional optimizations could improve the compiled object programs significantly.

As a result of these findings, a project to produce a new optimizer for the FORTRAN H Extended compiler was initiated. The strategy was to incorporate the existing routines which worked well, rewrite those which did not, and write new routines to perform additional optimizations. Later the effort was extended to include improvements in the FORTRAN library as well. The results were made available in September 1978 as the FORTRAN H Extended Optimization Enhancement [6].

This paper begins by describing the objectives and constraints which govern the new optimizer. For the sake of completeness, this is followed by a brief overview of the original compiler and its optimizer. The major section of the paper then describes the new optimizations and the methods used to achieve them. The paper concludes with some measures of the improvement in optimization.

Objectives and constraints

The motivation for an improved optimizer originally came from studying the computer requirements of several scientific research projects. Some of these requirements are

Copyright 1980 by International Business Machines Corporation. Copying is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract may be used without further permission in computer-based and other information-service systems. Permission to *republish* other excerpts should be obtained from the Editor.

Table 1 Instructions executed in a plasma physics program. Instruction counts are in millions.

Instruction type	*FORTRAN GI*		*H Extended*		*H Enhanced*	
	count	*percent*	*count*	*percent*	*count*	*percent*
Fixed point	70.216	83.5	7.120	38.3	1.372	11.4
Floating point	10.994	13.1	9.976	53.7	9.207	76.4
Branch, link, loop	1.456	1.7	1.435	7.7	1.435	11.9
Other instructions	1.459	1.7	.044	0.2	.044	0.4
Total instructions	84.126	100.0	18.575	100.0	12.058	100.0

beyond the scope of any available computers [7]. NASA, for example, wants a special purpose computer many times faster than any existing computer in order to simulate wind tunnel operations. Some scientific installations are running their computers at capacity, with application programs able to absorb forseeable increases in capacity. In a weather model, for example, the resolution can always become finer or more complex physical simulations can be added. Other scientific activities are producing data at such extraordinary rates that computers are straining to capture and analyze the data. A particle accelerator can produce many events for analysis during each second of operation, and the analysis of each event may require more than a second of computer time on a large processor. Satellites, such as Landsat, produce a continuous stream of images, while computers are unable to format, enhance, and extract information from the images at the rate at which they are created.

In such environments, as well as in day-to-day interactive computing, better object programs are just as valuable as faster computers. The effect of optimization on object programs can be considerable. Table 1 shows the number of instructions executed in a plasma physics program written at the IBM Palo Alto Scientific Center [8] after the program was compiled with three different compilers. FORTRAN GI is regarded as a fast compiler in an interactive environment and is not especially concerned with optimization. FORTRAN H Extended is the standard program product, and FORTRAN H Enhanced is the compiler containing the enhancements described in this paper. Moving from the essentially nonoptimizing FORTRAN GI to the best optimization level of FORTRAN H Extended reduced the number of instructions executed by 78%, and 35% of the remaining instructions were eliminated with the new compiler enhancements.

The major objective of the enhancement project was to produce better optimized object programs. A second objective was to produce a faster FORTRAN library. The library contains mathematical functions and input/output routines to support FORTRAN object programs during execution. It is not rare to find a program spending 20 or 30 percent of its time in the library (evaluating square roots, logarithms, exponentials, etc.) so that improvements here are also valuable. A third objective was to produce a faster compiler. A common criticism of optimization is that compilation takes too much time. If faster programs could be compiled in less time, then the objection, at least for users of the existing optimizer, would be eliminated.

The primary constraint on the project was our belief that the work would be useless if it were not incorporated into a product. This meant that we would have to work within the framework of the existing FORTRAN H Extended compiler and library. These jointly contain 65 806 lines of source code (excluding comments). It would have been impossible (and unproductive) for us to rewrite them entirely. We were interested only in the optimization phases of the compiler and the most critical performance kernels in the library. The remainder of the compiler and library would be accepted without change.

A user of the enhanced and standard products ideally should see no difference between them except performance. It was undesirable to require a user to rewrite his programs in order to gain improved performance. It was unacceptable to produce answers which differed from those given by the standard compiler and library. Consequently, certain optimizations which could improve the performance of object programs were not implemented. For example, the expressions A+B+C and A+C+B could not be treated as identical, even though algebraically they are equivalent, because reordering of floating-point operations may change the numerical result of the operations. Likewise, the mathematical approximations used in evaluating the FORTRAN library functions, and the order of the arithmetic operations with which the approximations were originally implemented, could not be changed. All

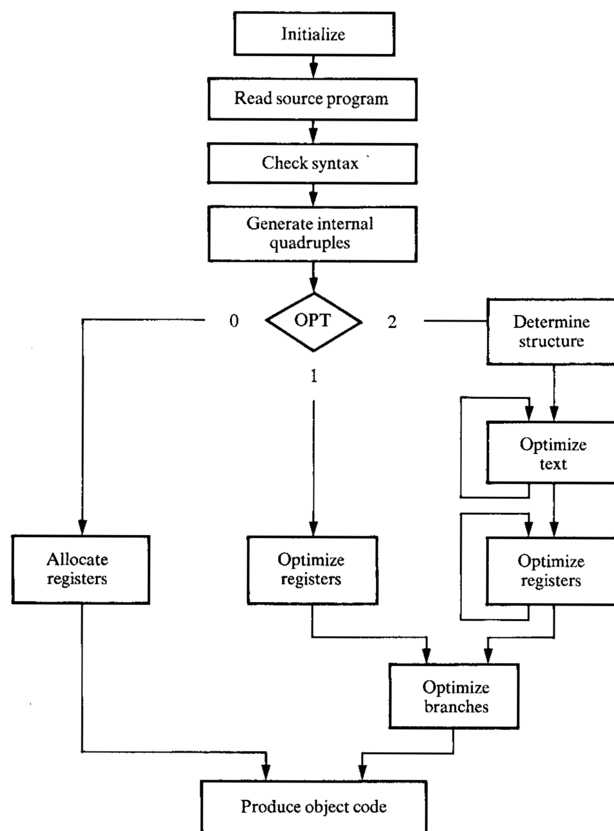

Figure 1 Processing in the FORTRAN H Extended compiler. Enhancements are made mainly in the OPT(2) path.

new optimizations, in short, were required to yield identical bit-for-bit results to the optimizations performed by the standard FORTRAN H Extended compiler and library.

FORTRAN H Extended compiler

The FORTRAN H Extended compiler was the foundation for the optimization enhancements. Figure 1 shows the general flow of processing in this compiler. The compiler is invoked by a calling program running in the VM/CMS or OS/370 environment. The compiler processes one or more FORTRAN source programs, transforming each source module into an object module containing machine instructions for execution on System/370. Upon completion, the compiler returns control to the calling program. Before the object modules are executed, they are linked together with other object modules, including routines from the FORTRAN library, to form a complete program for execution.

The compiler processes each FORTRAN source module separately. The source program is read, checked for correct syntax, and translated into an internal representation consisting of operator-operand1-operand2-operand3 quadruples (*e.g.*, C=A+B becomes +,C,A,B). Some source program constructs generate more than one quadruple. For example, subscript expressions are expanded into sets of multiplications and additions, and complex arithmetic operations generate several ordinary arithmetic operations. Temporary internal variables are created to carry the results of intermediate quadruples from their definitions to their references.

Certain simple optimizations are performed as the quadruples are constructed. For example, an integer multiplication by a constant power of two (*e.g.*, I*4 or I*16) is replaced by a left-shift operation. An exponentiation involving an integer constant power (*e.g.*, A**9) is replaced by a series of in-line multiplications. Some operations involving minus signs are converted to simpler forms; for instance, -(B-C) becomes C-B. Finally, constants employed in a subscript expression [for example, each number 7 in A(7,I-7,I+7)] are often extracted from the subscript, evaluated as constant offsets from the start of the subscripted array, and combined into an aggregate constant offset which does not require computation during execution.

At this point in the compilation the quadruples are ready for general optimization. (A user may request that optimization be bypassed or only partially executed in order to reduce compiler processing.) Optimization is performed on a loop-by-loop basis. Therefore, before any optimization procedures are executed, the structure of the source program is analyzed. The loops in the program are identified (whether written as do-loops or with if-statements), and the manner in which the loops are nested is determined.

The loops are then processed in order from the innermost loop to the outermost loop. Two passes through the program are made in this manner. The first, called text optimization, attempts to transform the quadruples into sequences which will give the same results in fewer operations. The second, called register optimization, assigns registers to the operands of the remaining quadruples and attempts to minimize the number of operand fetches and stores. A third pass, called branch optimization, determines how the object program itself will be addressed in storage and attempts to provide efficient branching from one part to another. The object program is generated after optimization simply by transcribing the information encoded in the quadruples into machine instructions.

The three optimization phases are described in the following sections.

- *Text optimization*

Text optimization is an attempt to reduce the number of arithmetic operations required to execute the FORTRAN source program. Three optimization procedures are performed: common expression elimination, backward movement, and strength reduction. These procedures analyze the quadruples and possibly replace them with quadruples which produce identical results in fewer operations.

Common expression elimination

Common expression elimination is an attempt to remove duplicate occurrences of a computation from the program. If the same expression is computed more than once, if none of the operands are changed between the separate computations, and if the first of the computations must be executed before the others are reached, then the result of the first computation is saved, the other computations are deleted, and the saved result is used in place of the results of the deleted computations. In practice most eliminated expressions involve the computation of subscripts: each occurrence of A(I,J,K) in a loop generates an extended and identical calculation.

Duplicate computations are eliminated quadruple by quadruple. Identical quadruples are located by searches backward from each quadruple through all quadruples which must be executed in the loop before the subject quadruple is reached. If there are duplicate expressions on two parallel paths, then they are eliminated only if the expression also occurs on a preceding common path.

Backward movement

Backward movement is an attempt to move invariant computations from inner loops to outer loops. If a computation is performed in a loop and if none of the operands are changed within the loop, then, since the computation always produces the same result, the computation is performed outside the loop before the loop is entered, the computation within the loop is deleted, and the result of the outer computation is used in place of the result of the inner computation. In practice most backward movements involve the computation of subscripts for those array dimensions which are invariant in inner loops.

Concurrent with backward movement, two additional optimizations are performed. First, an attempt is made to delete assignment statements. When a variable or constant is assigned to another variable, it may be possible to replace all references to the result variable with references to the variable or constant which has been assigned to it. If so, then the assignment is deleted. Second, various elementary calculations involving numeric constant operands are detected and executed. Since the operands are numeric constants, the result can be computed during compilation and the calculation can be replaced by an assignment of the numeric constant result. In practice these optimizations are performed mainly on intermediate quadruples inserted by the other optimization procedures.

Before reduction	After reduction
DO 1 I=1,N,1 1 A(9*I)=0.0	DO 1 I=9*1,9*N,9*1 1 A(I)=0.0
DO 2 I=1,N,1 2 A(9+I)=0.0	DO 2 I=9+1,9+N,+1 2 A(I)=0.0
DO 3 I=1,N,1 3 A(9−I)=0.0	DO 3 I=9−1,9−N,−1 3 A(I)=0.0

Figure 2 Examples of multiplication, addition, and subtraction strength reduction.

Strength reduction

Strength reduction is an attempt to simplify calculations which involve induction variables. Induction variables are those variables which are altered only once in a loop and which are altered at that point by being incremented or decremented by a constant within the loop. Do-loop indexes are the most common example. The value of an induction variable proceeds through an orderly sequence as the loop is executed. A constant times an induction variable, a constant plus an induction variable, and a constant minus an induction variable likewise proceed through an orderly sequence. A reference to the result of one of these functions would be just as well served by an induction variable which supplied that sequence directly. Strength reduction generates new induction variables to supply such sequences and thereby replace functions of the original induction variables. Figure 2 illustrates these reductions. In practice most strength reductions involve induction variables employed in subscript computations.

- *Register optimization*

Register optimization is an attempt to reduce the number of operand fetches and stores required to execute the FORTRAN program. Two optimization procedures are performed: local register optimization and global register optimization. These procedures analyze the quadruples remaining after text optimization and assign registers to each operand in each quadruple.

Local register optimization

Local register optimization is an attempt to keep the result of each quadruple in a register until that result is referenced in a following quadruple. A subset of the available registers is allocated for this purpose (the other registers are reserved for global register optimization and branch optimization). As each quadruple is processed, a

register is allocated to contain the result of the quadruple. If no more registers are available, then one of the variables currently in a register is displaced from its register into storage. When the last reference to the result is processed, then the register containing the result is once again made available for other quadruples.

Global register optimization
Global register optimization is an attempt to keep the variables and constants most frequently referenced in a loop in registers throughout execution of the loop. The variables and constants are sorted based on the number of times they are referenced within the loop. The registers reserved for global allocation, and any registers reserved but not required for local allocation, are then allocated to the most frequently referenced variables and constants. New quadruples are inserted to fetch the registers before loop entry and to store the registers at each loop exit. Finally, the quadruples referring to the global variables and constants are updated to reflect the global register allocations.

• *Branch optimization*
Branch optimization is an attempt to execute all branches in the program with direct branches from the source to the target. Direct branches require that the branch target be addressable with a general purpose register. As many as five registers, depending on object program size, are reserved as address registers. (These registers are not available for local or global register optimization.) The location of each branch target in the object program is determined. Branches to all branch targets which are spanned by the address registers are implemented with single branch instructions. Branches to any targets beyond the span of the address registers (this happens only in huge programs) are implemented by loading an address constant into a register and branching to the address in the register.

Improvements in optimization

As a result of examining object programs produced by the FORTRAN H Extended compiler, we found that significant improvement was possible. Our approach was pragmatic. We examined the inner loops generated by the compiler, located unnecessary instructions, determined why the instructions were generated, and developed methods to eliminate them. When no more instructions could be eliminated, even when the loop was programmed in assembly language, then the optimization was considered satisfactory.

We did not ask whether the new optimizations were perfect in theory. For one reason they were clearly improvements over the existing optimizations. For another we could devise no way, practical for implementation within the FORTRAN H Extended compiler, to improve the assembly language equivalent to the new optimizations. Further, it was true throughout the project that improvements not yet implemented would make a larger difference in performance than refinements in the optimizations which were already pragmatically complete. In any case practical problems more than theoretical problems were usually the cause of poorly optimized object programs.

The sections below describe the problems observed in the FORTRAN H Extended compiler and the optimization enhancements implemented to correct them.

• *Major optimization improvements*

Increased number of optimized variables and constants
Throughout optimization, the compiler maintains bit vectors which record where variables and constants are fetched, stored, and busy. (A variable is busy if it is referenced before it is redefined.) These vectors are present for each externally and internally labeled statement in the source program. The FORTRAN H Extended compiler generates the vectors with 127 entries, called coordinates, and allocates several global tables of this same size. Variables and constants contend based on a count of references for the first 80 coordinates. The remaining coordinates are used only for address constants and for temporaries and constants generated by the optimizer.

A variable, constant, or temporary not allocated a coordinate is not optimized. This can severely degrade the optimization of large programs, not only because individual variables are not optimized, but also because once any nonoptimizable operand is involved in an expression the expression also becomes nonoptimizable. Figures 3 and 4 show two typical effects on the object program.

Two improvements were made to remove this impediment. First, a coordinate is no longer required for constants and address constants. It is known that they are always busy and never stored, and it is not necessary to know where they are fetched in terms of labeled statements. Second, the size of the bit vectors has been made a function of the number of variables in a source program. As many as 991 variables and compiler temporaries now may be optimized (the new limit is a function of a storage allocation restriction). Very large subroutines typically require fewer than 700.

With the FORTRAN H Extended compiler, it is hard to tell if poor optimization results from a problem in an optimization routine or from the lack of an optimization

coordinate for an operand involved in the optimization. This made it impossible to evaluate accurately the effect of the original optimization procedures and the new procedures being developed. After the bit vectors were expanded, it was invariably true that poor optimization was the result of an optimizer routine.

Improved methods for computing subscripts

Many of the observed unnecessary instructions were concerned with subscripting. The real work of the FORTRAN programs under study was typically executed with expressions involving subscripted array elements; the programs had nests of loops to vary the indexes on the arrays. It sometimes seemed as if more work was done to address the array elements than was done once they were located.

The standard FORTRAN compiler evaluates subscripted array references in six steps. First, numerical constants embedded in the subscripts are extracted, evaluated, and combined into an aggregate constant subscript when the program is translated into quadruples. Second, the subscript expression remaining in each dimension is evaluated and converted to integer. Third, each of these evaluated subscripts is multiplied by the span in bytes represented by a unit subscript in the subscripted dimension. Fourth, these products are added together to produce an aggregate computed subscript. Fifth, the constants extracted from the subscript and combined to form the aggregate constant subscript are added to the aggregate computed subscript to produce the aggregate effective subscript. For an aggregate constant subscript in the range 0–4095, this addition is accomplished implicitly by encoding the constant in the displacement field of an indexed machine instruction. Sixth, the address of the array itself is added to the aggregate effective subscript to produce the address of the subscripted array element. This addition is always accomplished implicitly by using the base and index registers of an indexed machine instruction.

Two new optimization procedures were added to improve subscripting. The first, called subscript commutation, reorders the additions used in the fourth step, above, when the separately evaluated dimension subscripts are combined to form the aggregate computed subscript. Each dimension subscript is examined to determine whether it is constant or variable in the loop undergoing optimization. The additions of the subscript dimensions are then commuted so that the constant and the variable dimensions of each subscript are added separately into two separate terms, one constant and one variable. The two terms are then combined with a final addition. The evaluation of the constant term is thereafter removed from the loop by backward movement.

```
      DO 1 I=1,N
    1 Z=Z+A(I)
```

No coordinate for Z		Coordinate for Z	
LE	2,Z	AE	6,A(I)
AE	2,A(I)	BXLE	I,4,LOOP
STE	2,Z		
BXLE	I,4,LOOP		

Figure 3 Effect of optimization coordinate availability on global register optimization. Code is assembly language equivalent to the instructions generated by the compiler. Operation codes are reported exactly, but operands and registers have been given names for clarity. Only inner loop instructions are shown.

```
      DO 1 I=1,N
    1 A(I)=X*Y*B(I)
```

No coordinate for X,Y		Coordinate for X,Y	
LE	2,X	LER	2,6
ME	2,Y	ME	2,B(I)
ME	2,B(I)	STE	2,A(I)
STE	2,A(I)	BXLE	I,4,LOOP
BXLE	I,4,LOOP		

Figure 4 Effect of optimization coordinate availability on backward movement optimization.

```
      DO 1 J=1,N
    1 A(I,J,K)=0.0
```

Standard compiler		Enhanced compiler	
LR	2,IJ	STE	6,A+I+K(J)
AR	2,K	BXLE	J,10,LOOP
STE	6,A(2)		
BXLE	IJ,10,LOOP		

Figure 5 Effect of subscript commutation on an object program. The standard compiler performed strength reductions for I and J but not for K. The enhanced compiler combined I and K with the address constant for A.

Figure 5 shows an example of the improvement produced by subscript commutation.

The second of the new subscript optimization procedures, called subscript optimization, determines how the components of a subscript are combined in steps four through six, above. It attempts to eliminate two kinds of additions from the loop: the addition in step four of the final constant and variable terms produced after subscript commutation; and the addition in step five of negative and large positive aggregate constant subscripts. It also at-

tempts to minimize the number of variables required for computing subscripts within the loop so that fewer registers are occupied with subscripting.

Each subscript is decomposed to identify four subscript components. The first is the identity of the storage block containing the subscripted array: for arrays in a common block, the identity is the name of the common block; for arrays in local storage, the identity is the name of the subroutine; for arrays passed as arguments, the identity is the name of the argument. All arrays in a given storage block can be addressed with a single address constant merely by varying a displacement (described next). The name of the array is therefore ignored, and the name of the storage block containing the array is used instead. This permits all of the subscripts within a common block or within local storage to be optimized together without the impediment of the original array identifiers.

The second subscript component is a numeric displacement composed of two pieces. One is the offset of the zeroth element of the array from the start of the storage block containing the array. The other is the aggregate constant subscript extracted from the subscript when the subscript was translated into quadruples.

The third subscript component, the constant index, is that part of the subscript expression which can be identified as constant within the current loop.

The fourth subscript component, the variable index, is the remaining variable or indeterminate part of the subscript expression. For multidimensional arrays the constant index and variable index are typically the final constant and variable terms produced by subscript commutation. The four subscript components are identified as BASE, DISP, XCONST, and XVARIA in the discussion below.

When the four subscript components are combined, they yield the address of the subscripted element. The standard compiler combines them as if the expression ((XVARIA+XCONST)+DISP)+BASE had been written. Any variable and constant index terms are explicitly added (no attempt is made to separate them). If the displacement is less than zero or greater than 4095, then it is added explicitly to the index value. Finally, the remaining additions are accomplished using the three address operands of an indexed machine instruction. Two unnecessary additions, however, may have been generated in the loop.

Subscript optimization begins by reversing the order of addition. The subscript components are combined as if the expression XVARIA+(XCONST+(DISP+BASE)) had been written. If the displacement is less than zero or greater than 4095, then a new address constant incorporating the displacement is generated or, for arrays passed as arguments, the displacement and the array address are added into a new temporary outside the loop. The constant index is then added to the possibly modified address constant outside the loop. The addition of the variable index is accomplished with the index register of an indexed machine instruction. The two possible additions are eliminated by this procedure.

These changes are made only when they appear desirable. All of the subscripts in a loop are considered during this determination. The attempt is made to minimize both the number of instructions and the number of registers required for subscripting.

Three questions are asked and answered during subscript optimization. The most fundamental question is whether a variable index which is eligible for strength reduction should be optimized by strength reduction or by subscript optimization. The two procedures use different methods to eliminate the addition XVARIA+XCONST of the variable index and the constant index. Strength reduction generates a new variable equal to the sum XVARIA+XCONST for each different constant index XCONST; the new variable is used directly as a subscript. Subscript optimization generates a new temporary address constant (BASE+DISP)+XCONST for each different constant index XCONST; the unchanged variable index is used directly as a subscript. In both cases the addition XVARIA+XCONST is deleted from the loop.

This first question is answered based on the number of different constant indexes XCONST added to the variable index XVARIA. If there are more than one, then all are optimized by subscript optimization. The reason is that strength reduction produces a new induction variable, requiring initialization outside the loop and incrementation inside the loop, for each different XCONST. (The enhancements to strength reduction described later, in particular the elimination of parallel induction variables, would subsequently delete most of these induction variables by adding even more initialization outside the loop.) Subscript optimization, in contrast, removes the addition from the loop at the expense of a single addition (BASE+DISP)+XCONST outside the loop for each different XCONST and leaves the variable index XVARIA free-standing and ready for a single strength reduction. Figure 6 shows the difference in an object program. If, however, there is only one XCONST, then strength reduction is used to optimize the addition XVARIA+XCONST. This case often results in better code in outer loops because the XCONST often represents dimensions of an array which are constant in the inner loop but are variable and eligible for strength

reduction in the outer loop. This decision on where to optimize the subscript expression minimizes the number of induction variables in the loop and the initialization necessary for the loop.

The second question is whether the displacement DISP should be added explicitly to the storage block address BASE. The subscripts are sorted into groups, with all subscripts in a group having the same storage block BASE and constant index XCONST and with all subscripts in a group in ascending order based on the displacement DISP. The groups are partitioned into subgroups such that the displacements of the first and last subscripts in each subgroup differ by no more than 4095. Each subgroup can therefore be addressed by a single address constant BASE+DISP generated for the first subscript in the subgroup. This minimizes the number of address constants required to address the entire group. The pending question—whether to compute BASE+DISP explicitly—is then asked for each of these subgroup address constants. The answer is yes if the displacement of any subscript in the subgroup is less than zero or greater than 4095; otherwise, since the displacement field of an indexed machine instruction can be used in all cases, the answer is no. Explicit additions are implemented for common blocks and local storage by generating relocatable address constants during compilation (an existing address constant is used if it spans all subscripts in the subgroup) and for array arguments by generating temporaries outside of the loop during execution.

The third question is whether or not the constant index XCONST should be added to the subgroup address constant BASE+DISP outside the loop. The question is asked for each of the subgroups separately. The answer is yes if any subscript in the subgroup contains a variable index XVARIA which is not eligible for strength reduction. If any such variable index exists, then the addition of XCONST in the expression XVARIA+XCONST+(BASE+DISP) must be performed explicitly, and performing XCONST+(BASE+DISP) outside the loop is better than performing XVARIA+XCONST inside the loop. The question is not asked if XVARIA is eligible for strength reduction because, as described above, XVARIA+XCONST is eliminated by strength reduction. When the question is asked, however, if it is answered with a yes, then the decision to optimize eligible XVARIA+XCONSTs with strength reduction is revoked for the current XCONST. (There may be more than one eligible XVARIA added to this XCONST but, because of the first question asked above, this will be the only XCONST added to each of these XVARIAS.) Otherwise XCONST would be eliminated by one technique in strength reduction for the eligible XVARIAS and by a different technique in subscript optimization for the ineligible XVARIAS.

```
      DO 1 I=1,N
    1 A(I,J)=A(I,K)+A(I,L)

Strength reduction              Subscript optimization

LE     2,A(IK)                  LE     2,A+K(I)
AE     2,A(IL)                  AE     2,A+L(I)
STE    2,A(IJ)                  STE    2,A+J(I)
AR     IL,4                     BXLE   I,4,LOOP
AR     IK,4
BXLE   IJ,4,LOOP
```

Figure 6 Comparison between strength reduction and subscript optimization when more than one constant index is combined with a variable index. Both object programs were produced by the enhanced compiler. For this illustration strength reduction was not permitted to reduce the parallel induction variables.

```
      DO 1 I=1,N
    1 A(I)=B(I,J)+C(I,J,K)

Standard compiler               Enhanced compiler

LE     6,B(IJ)                  LE     6,B+J(I)
LR     2,I                      AE     6,C+J+K(I)
AR     2,JK                     STE    6,A(I)
AE     6,C(2)                   BXLE   I,4,LOOP
STE    6,A(I)
AR     IJ,4
BXLE   I,4,LOOP
```

Figure 7 Effect of subscript optimization. The standard compiler has strength-reduced the subscripts for A and B but not for C. The enhanced compiler has combined the J and K subscripts with the addresses of B and C.

This would require both kinds of initialization outside the loop and two address constants [(BASE+DISP) for strength-reduced XVARIAs and (BASE+DISP)+XCONST for subscript-optimized XVARIAs] within the loop. In order to eliminate this duplication, the decision is made to optimize all of the XVARIA+XCONST additions for the current XCONST with subscript optimization. The decision is also made effective for any previously processed subgroups.

As a result of this optimization, the new compiler is less sensitive to the number of dimensions in an array when it computes subscripts. Provided that the number of bytes between successively fetched elements is the same, a subscript computed for a vector can be used to address an array, and vice versa. Figure 7 shows the improvement this can produce.

Improvements and extensions to strength reduction

Many of the observed unnecessary instructions were concerned with strength reduction, particularly in support of subscripting. The unnecessary instructions were attribut-

```
      DO 1 I=1,N
    1 A(8*(I+4))=0.0
```

Standard compiler		Enhanced compiler	
LR	2,I	STE	6,A(I)
AR	2,X4	BXLE	I,N,LOOP
SLL	2,3		
SLL	2,2		
STE	6,A(2)		
BXLE	I,N,LOOP		

Figure 8 Effect of continuing strength reduction until no more reductions can be performed. The standard compiler, initially finding no multiplication strength reductions, has prematurely terminated addition strength reduction. The enhanced compiler has combined I and 4 and executed two subsequent multiplication reductions.

```
      DO 1 I=1,NM1
    1 A(N-I)=A(N-I)*A(N-I+1)
```

Standard compiler		Enhanced compiler	
LR	2,N	LE	6,A+0(I)
SR	2,I	ME	6,A+4(I)
SLL	2,2	STE	6,A+0(I)
LE	6,A+0(2)	BXH	I,4,LOOP
ME	6,A+4(2)		
STE	6,A+0(2)		
BXLE	I,4,LOOP		

Figure 9 Effect of improved strength reduction on an effectively backwards do-loop.

able in part to problems in the original implementation and subsequent maintenance and in part to additional reductions which were possible but not programmed.

The standard compiler is prepared to reduce multiplications, additions, and sometimes subtractions as illustrated in Fig. 2. The figure shows how the reductions are performed in terms of FORTRAN source language variables. Usually, however, the reductions are performed for compiler temporary variables created to calculate subscripts. For example, a subscript A(I,J) on a 10-by-10 array A, where I is an induction variable and J is a constant, results in two reductions: the multiplication 4*I to compute the byte offset corresponding to the subscript I, and the addition (4*I)+(40*J) to compute the byte offset for the entire subscript.

The standard compiler performs multiplication reductions first and addition and subtraction reductions second. It does not return to multiplication once the others have been started. Consequently reduction may be incomplete. In the loop

```
  DO 1 I=1,N
1 A(I+J)=0.0
```

I+J is reduced, but the implicit multiplication by 4 which generates the byte subscript on the array A is not reduced. This problem has been corrected: reduction now cycles until no more reductions can be found. Figure 8 gives an example.

In the standard compiler a number of specific reductions have been disabled. The disabled cases are distinguished by the nature of the operands (variables, constants, compiler temporaries). This was done apparently to compensate for errors in optimization. Rather than correct an error in the implementation, the case occasioning the error was disabled. Almost all instances of subtraction reduction disappeared because of this maintenance. It is interesting to note something written by Lowry and Medlock [3] in 1967: "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." The problem has been corrected—for now.

With these corrections and certain similar extensions the compiler quite reliably reduces eligible computations. One result is that backwards do-loops now generate optimized code. Figure 9 gives an example.

One extension in particular is helpful in making the reductions collapse as illustrated. When a source language variable is assigned the result of a reduction, the enhanced compiler attempts to employ that variable directly as the new induction variable. For example, in the loop

```
  DO 1 I=1,N
  IJ=I+J
  IK=I+K
  IL=I+L
1 A(IJ)=A(IK)+A(IL)
```

IJ, IK, and IL are used directly as induction variables when the I+J, I+K, and I+L additions are reduced. The standard compiler instead obtains a compiler temporary variable to implement each reduction; the temporary is assigned into the source language variable. The enhanced technique makes further computations with the source language variable eligible for reduction (because it is now an induction variable), while the standard technique leaves them irreducible. This change can greatly improve programs in which the programmer has "simplified" subscripting by using vectors in place of arrays (and using vector subscript expressions equivalent to the array subscript expressions) or in which the programmer has performed common expression elimination by hand (techniques very often seen especially in older FORTRAN programs). Figure 10 gives an example.

An attempt is made to reduce the number of duplicate induction variables required to support inner loops on parallel execution paths. When an outer loop has more than one inner loop, these inner loops are separately optimized. If the induction parameters for the inner loops are identical and if the inner loops have no common preceding loop with these same parameters, then the duplicates are not detected by common expression elimination. For a normal expression this usually causes no penalty in execution: only one instance of the expression will be executed since the duplicates are on parallel paths. Strength reduction, however, generates initializations outside the loop and incrementations inside the loop which are executed regardless of which parallel path is followed. A strength reduction to support one of the paths therefore penalizes all of the other paths. Consequently, each new induction variable created to replace an old induction is used immediately to replace any duplicates of that old induction on any parallel paths.

An attempt is made to reduce the number of parallel induction variables generated for the current loop. Parallel induction variables are those which are incremented by the same number in the same place. They maintain a constant interval between their values. For example, the loop

```
DO  1  I=1,N
1 A(I+J)=A(I+K)+A(I+L)
```

will contain three parallel induction variables after the additions and multiplications have been reduced. If one of the parallel induction variables is used solely for subscripting, then usually it may be eliminated by one of the others. Since the effective address of a subscripted array element includes the sum of the subscript variable and the address constant, a variable may be replaced by another variable in a subscript if the address constant is modified by the difference between the two variables. This difference is constant for parallel induction variables. A new temporary address constant, equal to the old address constant less the distance between the two induction variables, is obtained for each array subscripted by the original induction variable, and the subscript is replaced with the parallel induction variable. Figure 11 shows an example.

Integration of local and global register optimization

The standard compiler performs two register optimizations, local and global, in separate passes over the program, as previously indicated. Local register optimization attempts to keep the result of each quadruple in a register until the result is referenced in a following quadruple. The operation performed in that quadruple is executed if possible in a register already containing an operand. Sequences of computations thereby become optimized into a single register, as each intermediate result becomes an operand of a following quadruple. Global register optimization, however, subsequently assigns specific registers to the most frequently referenced variables and constants in each loop. When these global variables appear in local sequences, then the local sequences may be disrupted. Operands which were optimized into the sequence now may have to be accessed in registers outside the sequence.

A new optimization procedure, called global register remapping, attempts to recapture the integrity of the local optimization after global optimization has been performed. It revises those optimized sequences which contain global variable definitions and references to use ei-

```
      DO 1 I=1,N
      I2=I+2
      I5=I+5
    1 A(I2)=A(I5)
```

Standard compiler		Enhanced compiler	
LR	I2,I	LE	6,A+I2(I)
AR	I2,X2	STE	6,A+I5(I)
LR	I5,I	BXLE	I,4,LOOP
AR	I5,X5		
LR	3,I2		
SLL	3,2		
LE	6,A(3)		
LR	2,I5		
SLL	2,2		
STE	6,A(2)		
BXLE	I,4,LOOP		

Figure 10 Effect of including source language variables in strength reduction.

```
      DO 1 I=1,N
    1 A(I+J)=A(I+K)+A(I+L)
```

Standard compiler		Enhanced compiler	
LR	2,I	LE	6,A+K−J(IJ)
AR	2,K	AE	6,A+L−J(IJ)
SLL	2,2	STE	6,A(IJ)
LE	6,A(2)	BXLE	IJ,4,LOOP
LR	2,I		
AR	2,L		
SLL	2,2		
AE	6,A(2)		
LR	2,I		
AR	2,J		
SLL	2,2		
STE	6,A(2)		
BXLE	I,4,LOOP		

Figure 11 Effect of eliminating parallel induction variables in strength reduction.

After local optimization		After global optimization		After definition remapping type 1		After definition remapping type 2	
LE	2,A	LE	2,A	LE	2,A	LE	6,A
AE	2,B	AE	2,B	AE	2,B	AE	6,B
AE	2,C	AE	2,C	AE	2,C	AE	6,C
STE	2,X†	LER	6,2††				

†perhaps
††always

Figure 12 Effect of definition register remapping on the object program for X=A+B+C when X is assigned a global register. Store of X will be generated in local register optimization only if it is necessary.

After local optimization		After global optimization		After reference remapping	
LE	2,X	LE	2,6	AE	6,B
AE	2,B	AE	2,B	AE	6,C
AE	2,C	AE	2,C	STE	6,A
STE	2,A	STE	2,A		

Figure 13 Effect of reference register remapping on the object program for A=X+B+C when X is assigned a global register.

```
  DO 1 I=1,N
  Q=A(I)*B(I)
1 D(I)=C(I)*Q
```

Standard compiler		Enhanced compiler	
LE	2,A(I)	LE	0,A(I)
LER	6,2	ME	0,B(I)
ME	6,B(I)	ME	0,C(I)
LE	2,C(I)	STE	0,D(I)
MER	2,6	BXLE	I,4,LOOP
STE	2,D(I)		
BXLE	I,4,LOOP		

Figure 14 Effect of global definition and reference register remapping. The standard compiler preserves the value of the variable Q while the enhanced compiler recognizes that the variable is effectively a temporary.

ther the global registers or the local registers, but not both, depending on whether the value of the global variable must be preserved. Definitions and references are treated separately.

Definition register remapping is performed when a global variable is defined by a locally allocated program sequence. It attempts to compute the global variable directly into the register in which it will be retained in order to avoid a load-register instruction following the computation. Figure 12 contains an example.

Definition remapping is accomplished in one of two ways. First, when local allocation has determined that it is not necessary to store the result into the variable (because all references preceding the next definition have been locally optimized into registers), the result is retained in its local register. If it was not necessary to place the result into storage, then it is not necessary to preserve the result in the global register. Second, when local allocation has determined that it is necessary to store the result into the variable, the sequence which computes the variable is moved from the local into the global register. The quadruples are searched backward to find the beginning of the sequence—a quadruple which computes the local register but in which neither source operand is resident in the register before computation. If an exceptional condition is detected, then the sequence (including the load from the local into the global register) is retained in the local register. The most common exceptional condition is a reference to the global variable itself during the computation (the computation cannot be moved into the global register because the computation would destroy the variable before the reference). If no exceptional conditions prohibit remapping, then the sequence is moved into the global register.

Reference register remapping is performed when a global variable is referenced in a locally allocated program sequence. It attempts to perform the computation which references the global variable in the register containing the global variable in order to avoid a load-register instruction preceding the computation. Figure 13 contains an example.

Reference remapping is not always possible and is often not necessary. It is not necessary when the global variable is being retained in the locally allocated register as a result of the first kind of definition remapping. It is not necessary when the result of the quadruple referencing the global variable has been locally optimized into a register different from that containing the global variable. It is not possible if it is not permissible to destroy the value of the global variable because it will be referenced before it is redefined. When reference remapping is performed, the quadruples are searched forward to find the beginning of the next sequence using the local register—a quadruple which computes the local register but in which neither source operand is resident in the register before computation. The quadruples are then searched backward to find the beginning of the current sequence. The sequence is then moved from the local into the global register.

Figure 14 gives an example showing both definition and reference remapping.

Using linkage registers for other purposes

After all of the register optimization procedures have been executed, certain general purpose registers may be

completely unused even though there remain variables and constants which could be maintained in registers. There are two reasons for this. First, under System/370 conventions, registers 14, 15, 0, and 1 are used as subroutine linkage registers. The FORTRAN compiler reserves them for this purpose, using them also as temporary registers for address constants, subscripts, and computations which were not optimized into registers by the register optimization routines. In loops which do not call subroutines, these registers may be unused. Second, registers allocated during local optimization may be freed of all references as a result of global optimization and global remapping. When a computation which was assigned registers by local allocation involves variables which later are assigned registers by global allocation, it is possible that global remapping may move the entire computation from the local registers into the global registers. If all references to a register are remapped, then the register becomes unused.

A new optimization procedure, called global register scavenging, attempts to recover these unused registers. Global register scavenging allocates any unused general registers to those address constants which are not already allocated registers. (We did not have time to make the registers available for more general optimization.) Quadruples referencing the constants are changed to reflect the global register allocations. This eliminates a register load for each reference to the selected constants.

Removal of unnecessary global register initializations
Global register optimization, once it has decided which variables and constants will be allocated global registers, inserts quadruples to initialize the registers with the selected variables and constants. These quadruples are inserted automatically at two locations. First, they are inserted outside the loop for each global variable which is busy on entrance into the loop. Second, they are inserted within the loop at each exit from any nested inner loop for each global register used differently in that inner loop. The first initialize the register; the second reinitialize the register after its destruction in an inner loop. In both cases, however, it is possible that the variable in the register is never referenced before the register is initialized, as part of the processing performed by the current loop in preparation for an inner loop, with a different variable which is a global variable in a global register for that nested inner loop.

Unnecessary initializations are most likely to arise as an effect of two common programming structures. First, in a set of nested loops the first statement in each nested loop may be the initialization for the next nested inner loop. The initialization usually consists of loading the global registers for the inner loop. The global registers prepared for the loop at each level are destroyed immediately when that loop prepares the global registers for the next inner loop. Second, in a loop which contains a sequence of several inner loops, the statement following each inner loop may be the initialization for the next inner loop. The global registers reinitialized for the current loop on exit from the preceding inner loop are destroyed immediately when the current loop initializes the global registers for the succeeding inner loop.

A new optimization procedure, called global register purging, attempts to locate and eliminate such unreferenced initializations.

Reduced number of registers reserved for branching
The standard compiler reserves from two to five general purpose registers for addressing the code and data in an object program. Registers reserved for addressing the program cannot be used for optimizing the program. There are only 16 general purpose registers. Four of these are reserved as linkage registers (although they may be recovered, as described above, by global register scavenging). Reserving five of the remaining 12 leaves only seven registers for optimization.

A new optimization procedure, called section-oriented branch optimization, implements branches with direct branch instructions without requiring that the entire object program be addressable with a fixed group of address registers. Two or three registers, depending on program size, are reserved as program address registers. If the entire program can be spanned by the reserved address registers, then all branches are implemented immediately with direct branches. Otherwise two registers are reserved to address the front end of the program (this is necessary because of the internal structure of the object program). Branches to this part of the object program are implemented with direct branches. Branches to the remaining part of the object program are implemented with section-oriented branch optimization. That remaining part of the object program is divided into sections each of which is separately addressable with a single address register. The third reserved address register is used to hold the address of the currently executing section. The initial statement in each section is modified to load the section address register with the address of the section. Branches into a section are implemented with a load of the section address for the section containing the branch target followed by a direct branch to the branch target. Branches within a section, however, which are far more common than branches between sections, are implemented simply with direct branches.

The gain in register optimization should exceed the cost of the section address loading. A very large subroutine

```
      DO 1 I=1,N
      ZR=ZR+AR(I)*BR(I)-AI(I)*BI(I)
    1 ZI=ZI+AR(I)*BI(I)+AI(I)*BR(I)
```

Standard compiler		Enhanced compiler	
LE	4,AR(I)	LE	0,AR(I)
STE	4,TEMP1	ME	0,BR(I)
LE	2,BR(I)	AER	4,0
STE	2,TEMP4	LE	2,AI(I)
MER	4,2	ME	2,BI(I)
AE	4,ZR	SER	4,2
LE	2,AI(I)	LE	2,AR(I)
STE	2,TEMP3	ME	2,BI(I)
LE	6,BI(I)	AER	2,6
STE	6,TEMP2	LE	6,AI(I)
LER	0,2	ME	6,BR(I)
MER	0,6	AER	6,2
SER	4,0	BXLE	1,4,LOOP
STE	4,ZR		
LE	4,TEMP1		
MER	4,6		
AE	4,ZI		
ME	2,TEMP4		
AER	2,4		
STE	2,ZI		
BXLE	1,4,LOOP		

Figure 15 Effect of not using temporaries to eliminate duplicate array elements. The standard compiler generates temporaries for each of the duplicate array references and maintains ZR and ZI in storage. The enhanced compiler retains ZR and ZI in registers and generates no stores within the loop.

(1321 executable source cards) from a weather model was compiled both with and without section-oriented branch optimization (all other optimizations were performed). Section-oriented branching reduced the number of general register fetches and stores in the program from 1976 to 1707 (a 15% reduction) and the number of branch target address constants from 219 (for the branch targets not spanned by the five reserved registers) to 6 (one for each of six sections).

- *Minor optimization improvements*

A number of refinements were implemented in the existing optimization procedures during the project. These refinements were triggered by observed problems in the object programs. The problems were solved with simple changes (sometimes requiring extensive rewriting) to the existing optimization procedures.

Improvements in subscript computation

Numeric constants employed in subscript expressions are now extracted into an aggregate constant subscript for arrays with variable dimensions as well as for arrays with constant dimensions. Only the latter are eligible in the standard compiler. Constants are extracted in leading constant dimensions and in the initial variable dimension. With the dimension declarations A(N), B(M,N), and C(7,M,N), for example, all constants except those in the rightmost dimensions (the N dimensions) of B and C are eligible for extraction.

Improvements in common expression elimination

In common expression elimination, in order to reduce compiler processing, searches for duplicate computations were arbitrarily restricted. Only the preceding 25 externally or internally labeled statements on the main line execution path were searched to locate a duplicate for a quadruple. This limit has been removed.

A problem involving duplicate references to a subscripted array element was corrected. The compiler obtains an internal temporary variable to pass the result of a duplicate computation between its initial and its subsequent occurrences. The subscripting of an array involves at least two quadruples and is treated as an expression subject to elimination so that larger expressions such as 2*A(I) can be detected and eliminated. When the subscript itself is the only common expression, the optimization causes the array element to be fetched into a temporary and the temporary to be referenced in place of the original array element. This is an improvement if the temporary is globally allocated a register because a fetch will have been saved for each reference to the array element. Usually, however, registers are scarce and the temporary must be stored. Therefore this optimization often results in decreased performance. Consequently the enhanced compiler does not allocate a temporary to eliminate a duplicate array subscript. Figure 15 shows an example of the difference this can make.

Improvements in backward movement

The standard compiler does not compute negative constant numbers. Instead, when a negative constant is needed, it generates a positive constant and compiles the instructions necessary to load it and complement it during execution. This has been changed so that the negative value is computed during compilation.

Improvements in local register optimization

When no register is available for the result of a quadruple, then a variable must be displaced from a register into storage. The standard compiler chooses the variable already in a register which is least heavily referenced in the local group of source statements undergoing optimization. This can severely degrade optimization in some cases. The enhanced compiler chooses instead the variable whose definition is farthest backward from the location where the register is required. This choice makes a register available for other optimizations for the longest possible time.

When a variable is displaced from a register into storage, quadruples already optimized on the assumption that

it is in a register may no longer be correctly optimized. The standard compiler ignores this. The enhanced compiler reprocesses the quadruples with the knowledge that the displaced variable must be fetched from storage.

All four floating-point registers, rather than just three, are eligible for optimization.

Improvements in global register optimization
When a variable is computed immediately preceding an inner loop and is allocated a global register for that inner loop, an attempt is made to compute the variable directly into its global register for the inner loop.

An attempt is made to allocate, as a global register for a variable or constant in an outer loop, that register which has been most frequently allocated to it in any nested inner loops.

A better estimate is made of the utility of having each variable and constant in a global register. The standard compiler basically counts each reference to a variable or constant equally. The enhanced compiler counts 0, 1, or 2 depending on whether the reference is already a register reference as a result of local allocation, whether the reference would be changed from a storage reference to a register reference if the variable were in a global register, or whether an entire machine instruction would disappear if the variable were in a global register.

A better estimate is made of the utility of reserving registers for branch-on-index instructions. Branch-on-index instructions increment a variable by a constant, compare the variable to a limit, and branch if the variable is less than or equal to (BXLE) or higher than (BXH) the limit. This is exactly what is required to implement a do-loop. A branch-on-index instruction requires three registers but replaces three (add, compare, branch) or five (load, add, store, compare, branch) instructions. The standard compiler allocates registers to variables strictly in order of utility. If the three components of the branch-on-index instruction are in registers after all registers are allocated, then the branch-on-index instruction is generated. The enhanced compiler instead tentatively reserves the three registers for the instruction. When a variable requires a register, when no other registers are available, and when any components of the branch-on-index instruction have not yet been chosen because their utilities are individually less than that of the variable under consideration, then a choice is made: if the aggregate utility of the remaining branch components, augmented by an increment to represent the improvement effected by the branch-on-index instruction itself, exceeds that of the variables which would be allocated registers based solely on individual utility, then the registers are preserved for the branch-on-index instruction. Otherwise they are broken apart and given to the other variables.

Table 2 Instructions compiled for the innermost loops of subroutine HARM. Only instructions within the loops, not instructions initializing the loops, are included.

Do-loop label	*Standard compiler*	*Enhanced compiler*	*Ratio Enh/Std*
DO 19	15	8	0.53
DO 50	20	13	0.65
DO 80	73	50	0.68
DO 85	88	62	0.70
DO 220	106	79	0.75
DO 850	38	34	0.89
DO 892	7	4	0.57
DO 940	20	9	0.45
DO 970	9	4	0.44

Measurements of improved optimization

The improvement in performance resulting from the improvement in optimization varies widely. There are at least two reasons for this. First, the improvement in optimization can vary within each program and from program to program. A demonstration is given in Table 2. The table shows the number of instructions compiled by the old and new compilers for each innermost do-loop of subroutine HARM (the fast Fourier transform program in the IBM Scientific Subroutine Package). The percentage of instructions eliminated by the enhanced optimization ranges from 11 to 56 percent. Other programs show other distributions. In some loops compiled by the standard compiler no instructions can be eliminated, so no improvement is seen.

Second, the improvement in performance varies for a given program from machine to machine. Table 1, which compared the instructions executed in a plasma physics program compiled by three compilers, reveals why. Improvement in optimization not only reduces the number of instructions executed, it also changes the nature of the instructions executed. While the number of fixed-point instructions diminishes rapidly with the improvements in optimization, the number of floating-point instructions remains relatively constant. The optimizers are purging the subscripting and loop-control operations supporting the floating-point computations which perform the real work of the program. Therefore the improvement in performance is most apparent on machines whose floating-point operation times are relatively fast with respect to their

Table 3 Improvement in execution CPU time produced by the enhanced optimizer on different processors. The table entries are the execution times for the programs compiled under FORTRAN H Extended at its highest optimization level, OPT(2), divided by the execution times for the programs compiled under FORTRAN H Enhanced at its highest optimization level, OPT(3).

Program description	*Model 145*	*Model 158*	*Model 168*	*Model 195*
Evaluation of E	1.02	1.02	1.12	1.08
Evaluation of PI	1.02	1.02	1.12	1.08
Eigenvalues and eigenvectors	1.08	1.13	1.14	1.48
Discrete Fourier transform	1.13	1.19	1.19	1.16
Directed graph analysis	1.01	1.05	0.99	1.03
Polynomial synthetic division	1.04	1.05	1.11	1.15
Least-squares solution	1.12	1.36	1.16	1.59
Least-squares solution	1.13	1.33	1.16	1.61
Matrix inversion	1.14	1.31	1.23	1.39
Integration of equations	1.05	1.07	1.11	1.21
Polynomial least-squares	1.05	1.10	1.08	1.28
Integer sort	1.06	1.08	1.06	1.08
Plasma physics experiment	1.17	1.19	1.29	
Weather model		1.14	1.32	1.50

Table 4 Improvement in compilation and execution CPU time produced by the enhanced optimizer on customer programs. Ratios are FORTRAN H Extended OPT(2) times divided by FORTRAN H Enhanced OPT(3) times. All runs were on Model 168s.

	Compile ratio	*Execute ratio*
Customer "A"		
Job 1 (data analysis)	1.42	1.09
Job 2 (8 subroutines)	1.52	1.12
Job 3 (33 subroutines)	1.21	1.12
Job 4 (heavy trig functions)	1.12	1.27
Job 5 (partly under H Extended)	1.36	1.24
Job 6 (macro preprocessor)	1.14	1.36
Customer "B"		
Job 1 (geometry fitting)	1.60	1.15
Job 2 (Monte Carlo)	1.64	1.09
Job 3 (geometry fitting)	0.94	1.05
Job 4 (matrix inversion)	1.80	1.56
Customer "C"		
Job 1 (37 subroutines)	1.48	1.46
Customer "D"		
Job 1	1.27	1.64
Customer "E"		
Job 1	1.48	1.21

fixed-point operation times. For the plasma physics program, with its 35% reduction in instructions executed, the improvements were 17%, 19%, and 29% on System/370 Models 145, 158, and 168 (with high-speed multiply), respectively.

Programs representing a variety of scientific and mathematical computations were timed on several processors. These timings were made under the VM/370 operating system while the machines were under no more than a moderate load, except for the Model 195 timings which were made under OS/360 on a heavily loaded machine. These programs make little use of the FORTRAN library and the original library was used in all cases. Table 3 summarizes the improvement in performance. Information received about the performance of the improved optimization (both compiler and library) on customer programs is reported in Table 4.

Optimization of the FORTRAN library

The goal of the enhanced library is to reduce the number of instructions which must be executed to evaluate a mathematical function or to convert a data item between external and internal representations. The most commonly used mathematical functions and the formatted input and output conversion routines have been optimized. Internal linkage instructions not necessary to the mathematical computation or data item conversion have been removed, and instructions within the computation or conversion have been refined so that the same results are produced by fewer operations.

The effect of the enhancements on the mathematical library is indicated by Table 5. The table shows the number of instructions required to execute various common mathematical functions. The count includes only instructions in the library functions; instructions (usually three) in the calling program are excluded.

The effect of the enhancements on the input-output library is indicated by Table 6. The table shows the number of instructions required to execute various common data

element conversions. The counts record the processing for a single array element in an array read or write [for example A(3) in 'WRITE (6,9) (A(I),I=1,9)' or 'WRITE (6,9) A'].

Faster compiler processing

The third goal of the project was to make the compiler itself faster. Once all the new optimizations were complete, the compiler was monitored as it compiled a group of programs. Each spot consuming one percent or more of the compilation time was reviewed.

Many of these were small loops which could be written better for speed. In other spots simple equivalent procedures could save time. These latter typically involved the creating, searching, or sorting of linked lists.

Huge improvements were possible in several places. For example, over one-third of the compilation time in programs with large numbers of labeled statements was spent in a single compiler procedure. This procedure is given a list, the forward connection table, which records which statements are reached from each statement in the program. It inverts the list to produce another list, the backward connection table, which records which statements branch to each statement in the program. The original algorithm searches the forward connection table once for each labeled statement in the program. The new algorithm searches it four times.

Similarly, much time was spent determining where each variable was busy in the program. This was done by tracing the flow of the program for each variable eligible for optimization. The compiler now processes 32 variables (convenient because of the logical register operations) at a time.

Improved algorithms also were written for analyzing the structure of the program and selecting each loop for processing.

As a result of these changes, the new compiler usually is faster than the old even though it does more optimization. Table 7 compares the compilation times for a batch of programs. Note that the compiler usually is faster than FORTRAN G1 as well.

Concluding remarks

The new optimizations and the other enhancements discussed in this paper were developed as part of a pragmatic approach directed toward a pragmatic goal—to compile perfect inner loops. This approach has been very successful in increasing the performance of application programs. It has produced a faster compiler, a faster library, and faster object programs.

Table 5 Comparison of standard and enhanced mathematical libraries. Table entries are the number of instructions required in the library to execute the indicated function when X = 1.23456789 and Y = (1.23456789, 1.23456789) (counts for other arguments may vary slightly). Overhead to invoke the function is excluded.

Function	*Single precision*		*Double precision*	
	Standard	*Enhanced*	*Standard*	*Enhanced*
SIN(X)	47	31	54	38
COS(X)	48	32	54	37
EXP(X)	53	45	65	47
LOG(X)	49	35	55	41
SQRT(X)	36	27	42	33
X**X	131	81	151	89
Y*Y complex	29	13	29	13
Y/Y complex	46	25	46	25

Table 6 Comparison of standard and enhanced data element conversion. Table entries are the number of instructions required in the library to process one array data element under the indicated format (counts may vary slightly with data element values). Processing for the initiation of format conversion and the execution of the input/output transfer is excluded.

Format	*Input conversion*		*Output conversion*	
	Standard	*Enhanced*	*Standard*	*Enhanced*
A8	45	15	44	13
I8	220	34	130	28
F8.3 (real*4 data)	305	110	246	116
F8.3 (real*8 data)	316	132	259	135

Table 7 Comparison of compilation CPU time for three compilers. Times are in seconds. A single program containing 20 187 cards (14 716 excluding comments) which implement 101 subroutines was compiled. Times were taken on a Model 145 under VM/370 with compiler parameters appropriate for interactive computing.

Optimization	*FORTRAN G1*	*H Extended*	*H Enhanced*
OPT(0)	655	354	210
OPT(1)		529	272
OPT(2)		732	370
OPT(3)			402

The development of these improvements was aided by a powerful interactive programming environment: VM/370 and its Conversational Monitor System (CMS).

The compiler optimization enhancements are written primarily in FORTRAN. The compiler currently contains 27 415 FORTRAN and 16 271 assembler lines of code, excluding comments. All of the new optimization logic is written in FORTRAN; assembler is used only for bit manipulation. Overall 9661 lines of FORTRAN source code but only 1483 lines (most outside the optimizer) of assembly source code are new or modified programming. (The FORTRAN language implemented in the compiler is augmented by some in-line functions for shifting, masking, and bit manipulation and by a STRUCTURE statement which, like an assembly DSECT or a PL/I BASED structure, provides overlays on storage.) The compiler which is shipped as a product has been compiled under itself at its highest level of optimization.

Cited and general references

1. *IBM System/360 and System/370 FORTRAN IV Language*, Order No. GC28-6515, available through the local IBM branch office.
2. J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, "The FORTRAN automatic system," *Proc. Western Joint Computer Conf.* **11**, 188–198 (1957). Reprinted in *Programming Systems and Languages*, S. Rosen, Ed., McGraw-Hill Book Co., Inc., New York, 1967, pp. 29–47.
3. E. S. Lowry and C. W. Medlock, "Object Code Optimization," *Commun. ACM* **12**, 13–22 (1969).
4. *IBM System/360 OS FORTRAN IV (H Extended) Compiler Programmer's Guide*, Order No. SC28-6852; *IBM FORTRAN IV (H Extended) Compiler and FORTRAN Library (Mod II) for OS and VM/370 (CMS) Installation Reference Material*, Order No. SC28-6861; available through the local IBM branch office.
5. *IBM System/360 FORTRAN IV Library Mathematical and Service Subprograms*, Order No. GC28-6816, available through the local IBM branch office.
6. *FORTRAN H Extended Optimization Enhancement IUP No. 5796-PKR*, Order No. SH20-2100 (Sept. 1978), available through the local IBM branch office.
7. P. Lykos and J. White, "An Assessment of Future Computer System Needs for Large-Scale Computation," *Tech. Memo. 78613*, NASA, Washington, DC 20546, 1980.
8. J. Gazdag, "Numerical Solution of the Vlasov Equation with the Accurate Space Derivative Method," *J. Comput. Phys.* **19**, 77–89 (1975).
9. A. V. Aho and J. D. Ullman, "The Theory of Parsing, Translation, and Compiling," Prentice-Hall, Inc., Englewood Cliffs, NJ, Vol. I, 1972, Vol. II, 1973.
10. F. E. Allen and J. Cocke, "A catalogue of optimizing transformations," *Design and Optimization of Compilers*, R. Rustin, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972, pp. 1–30.
11. F. E. Allen, "Program optimization," *Annual Review in Automatic Programming*, Vol. 5, Pergamon Press, Inc., Elmsford, NY, 1969, pp. 239–307.
12. T. E. Cheatham, Jr., "The Theory and Construction of Compilers," Computer Associates, Wakefield, MA, 1967.
13. J. Cocke and J. T. Schwartz, "Programming Languages and Their Compilers: Preliminary Notes," Courant Institute of Mathematical Sciences, New York, 1970.
14. R. W. Floyd, "The syntax of programming languages—a survey," *IEEE Trans. Electron. Computers* **EC-13**, 346–353 (1964). Reprinted in *Programming Systems and Languages*, S. Rosen, Ed., McGraw-Hill Book Co., Inc., New York, 1967.
15. D. Gries, *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., New York, 1971.
16. D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1968.
17. P. M. Lewis II, D. J. Rosenkrantz, and R. E. Stearns, *Compiler Design Theory*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1976.

Received April 3, 1980; revised June 17, 1980

The authors are located at the IBM Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.