FORTRAN OPTIMIZATIONS AT THE
SOURCE CODE LEVEL

THEESIS

Presented to the Graduate Council of the
North Texas State University in Partial
Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Willie D. Barber, Jr.
Denton, Texas
August, 1977

This paper discusses FORTRAN optimizations that the user can perform manually at the source code level to improve object code performance. It makes use of descriptive examples within the text of the paper for explanatory purposes. The paper defines key areas in writing a FORTRAN program and recommends ways to improve efficiency in these areas.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF ILLUSTRATIONS</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1. Significance of the Problem</td>
<td></td>
</tr>
<tr>
<td>2. The Human Factor</td>
<td></td>
</tr>
<tr>
<td>II. EFFICIENT USE OF DECLARATIVES</td>
<td>12</td>
</tr>
<tr>
<td>1. Efficient Subscripting</td>
<td></td>
</tr>
<tr>
<td>2. Data Type and Conversion</td>
<td></td>
</tr>
<tr>
<td>3. Efficient Initialization</td>
<td></td>
</tr>
<tr>
<td>III. EFFICIENT USE OF EXECUTABLES</td>
<td>32</td>
</tr>
<tr>
<td>1. External Subprograms</td>
<td></td>
</tr>
<tr>
<td>2. Loop Optimization</td>
<td></td>
</tr>
<tr>
<td>3. Expression Evaluation</td>
<td></td>
</tr>
<tr>
<td>IV. EFFICIENT INPUT/OUTPUT OPERATIONS</td>
<td>54</td>
</tr>
<tr>
<td>1. Unformatted I/O</td>
<td></td>
</tr>
<tr>
<td>2. List Equivalencing</td>
<td></td>
</tr>
<tr>
<td>3. Blocking Records</td>
<td></td>
</tr>
<tr>
<td>V. CONCLUSION</td>
<td>64</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>68</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Array storage</td>
<td>19</td>
</tr>
<tr>
<td>2.</td>
<td>The mapping of a two-dimensional array</td>
<td>20</td>
</tr>
<tr>
<td>3.</td>
<td>Arrays using the same subscripts</td>
<td>23</td>
</tr>
<tr>
<td>4.</td>
<td>Efficient use of array with same subscripts</td>
<td>24</td>
</tr>
<tr>
<td>5.</td>
<td>Equivalence arrays</td>
<td>31</td>
</tr>
<tr>
<td>6.</td>
<td>Reduction in parameter list</td>
<td>41</td>
</tr>
<tr>
<td>7.</td>
<td>Reduction in parameter list</td>
<td>42</td>
</tr>
<tr>
<td>8.</td>
<td>Record marks</td>
<td>62</td>
</tr>
<tr>
<td>9.</td>
<td>Record blocking</td>
<td>63</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

The key to optimization in FORTRAN is to understand the problem to be solved and to find the best possible method to solve that problem. Coding the best method without attempting to be efficient will almost always be better than writing the most efficient code using a poor method. In this paper, the best algorithm will not be discussed. It will be assumed that the programmer has formed and translated the best method into FORTRAN and wishes to improve the efficiency of the program by optimizing the code itself. An assumption is also made that the programmer does not want to use assembly language. It cannot be run on other manufacturers' machines and is hard to write and debug for large programs. The programmer must evaluate the program and discern its applicability for optimization. For example, a short program, run a few times would not prove a worthy candidate for the use of optimization.

In order to be able to optimize a FORTRAN program, the programmer must do the following. First, find the most repeated code: (e.g., the innermost loops or subprograms), and do the best possible FORTRAN programming there (4, p. 1). The rest of the program usually may be given less attention because most of the computer time spent on most programs is
in this repeated code (5, p. 1). The other area of optimization on most programs is in the input/output operations of the programs.

There are several ways of finding these areas to be optimized in the program. Finding the innermost loop is often quite easy (5, p. 1). Sometimes this can be done by just a glance at the program by drawing in the loop level, or the programmer may use some utility program to identify that area of the program where most of the time is spent. For example, Control Data Corporation has written a routine called "SPY" (1) that will return the areas where most of the time is spent in executing a program. In some cases, the loops are obscured by the fact that a library routine may contain them. This is particularly evident when the user supplies a FUNCTION or SUBROUTINE for integration or differential equation solving routines. There may be millions of function references or subroutine calls from such routines so the user must carefully optimize his supplied subprogram to minimize computer time.

There are two types of FORTRAN optimization: general types which apply to all compilers, and individual types which are tailored to suit a particular compiler or computer.

Therefore, some of the examples that are given may be good for one compiler or computer but very poor for another compiler or computer.
Significance of the Problem

The FORTRAN user is concerned, basically, with four distinct factors, all of which influence turnaround time and job cost: compiler size, compiler speed, compiler generated object program size, and compiler generated object program speed.

The size and speed of the compiler are of interest to the FORTRAN user during the development and debugging stages of the program. If the compiler is relatively large, a FORTRAN job might often find itself at the bottom of the scheduling queue competing for core space. On the other hand, the space requirement of any language translator is often, but not always, an indicator of its speed. Some language translators have faster compilation speed in return for larger core requirements. For many compilers, the space requirement is a function of the size and complexity of the source program being compiled.

While both the size and the speed of the FORTRAN compiler are important factors during the development cycle of a program, the user typically has little or no control over these factors. Many installations have only one FORTRAN compiler available, and, in developing a production program, the user would be ill-advised to code in such a way as to minimize compilation time or compile-time core space requirements. However, in installations where more than one compiler is available, it would be appropriate to compare their
attributes and select a "development compiler" which would minimize program development costs. Many compilers provide the user with certain compile-time options, each of which has a direct effect on compiler performance and job cost. Again, it would be appropriate to study these options and set defaults. The user may prepare a list to recommend certain options which minimize compilation time for programs in the development stage (4, p. 2).

When a FORTRAN program reaches the production stage, the user is concerned only with the performance of the compiler-produced object code. The compiler or compilation options selected to generate the production object code should be the one which most highly optimizes the object code for the target machine. Compilation time and compiler core space requirements for this process are no longer an issue because this is presumably the final compilation.

Most language translators have no compile-time options which relate to object code optimization. This is not to say that any such compiler does not optimize object code, but rather that the compiler unconditionally applies a fixed set of optimizations to all programs. It is conceivable that such compilers might produce more highly optimized object code than do compilers providing explicit optimization options.

In the absence of a definitive document describing precisely all optimizations performed by a given compiler,
it is extremely difficult to determine the conditions under which optimizations are done. It is even more difficult to determine the degree to which global optimizations are usually constrained by available core storage during compilation, the amount of which is often too small to accommodate an entire subprogram (4).

The number and type of optimizations performed by compilers are many and varied. Basically, optimizations fall into two main categories:

1. machine independent optimizations, and
2. machine dependent optimizations.

In each category there are optimizations for speed and optimizations for size of the object program. The optimizations for speed vs. size are almost always in conflict with one another.

A few compilers in existence at the present time have an optimization feature that will in some cases make an object program execute more efficiently. Some manufacturers have gone so far as to have several levels of optimization for a particular compiler (2, p. 2). Each level will improve the efficiency of object program execution.

Most programmers who have used optimization features in compiling programs have found that the optimization features can cause erroneous results in their expected output. These problems have occurred because the program was changed as it was optimized. There have been many reported problems in this area (2, p. 31).
Another significant point is that even the most sophisticated optimizing compiler has no way of knowing which sections of the program being compiled will be executed most frequently. If the compiler assigns equal weight to all sections, optimizations for infrequent paths often tend to interfere with optimizations for the more frequent paths.

It is important that the FORTRAN user become familiar with certain coding techniques which tend to result in fairly good object code regardless of the compiler. The remainder of this paper addresses itself to FORTRAN optimizations that the user can perform manually at the source code level to improve object code performance. Certain optimizations for speed will increase the size of the object program, and certain optimizations for size will increase execution time. It must be the user's decision as to whether any optimization will have any real payoff, based on the execution frequency of the code in question.

The use of these techniques clearly does not eliminate the need for an optimizing compiler because

1. Many of the machine-dependent optimizations (such as register assignment) cannot be anticipated or guided by the user's source program and

2. Even the most highly optimized source program can be enhanced by an optimizing compiler (4, p. 2).

In fact, a few of the source code optimizations listed below may actually be detrimental in that they may obscure
the intent of the program to certain compilers, but this will be mentioned in each case.

There are debates as to whether there is a need for optimization by the programmer at the source code level. These debates have generated interest and have resulted in further research in the area. There are many reasons why this topic could have been chosen, only a few have been discussed here.

A FORTRAN programmer may improve his efficiency in programming by making use of the concepts and techniques presented in this paper.

The Human Factor

The "goodness" of a program cannot be measured only in narrow technical terms (3). Although good programming practice includes minimizing the storage space required, reducing the execution time, and creating a modular well-documented program, a broader view is necessary. Programmers must go beyond purely technical details and consider the impact of their work. The most compelling design goal is to produce a program that performs its required task with maximum benefit and minimum disadvantage.

Programs should be designed and written so that a single error will not have a catastrophic effect on the entire system. Furthermore, software must be designed so that it is easy for programmers to make changes and "forgiving" of operator errors.
However, problems of poorly designed programs have been all too common in the past and continue to plague the industry.

The software failure situation stems from the fact that programmers tend to isolate themselves from the purpose for which the software is being created in the first place. Programming is a type of art form or craft in which different people will perform the same job in different ways, and quality and operating efficiency of the job may vary (6).

The problem results, in part, from the fuzzy thinking of many programmers who believe the software they create is an extension of themselves, much like the artist who views his painting as an extension of his creative ability. While creating a painting, however, the artist gives little thought to who will buy it, where it will be displayed, or whether it will fit properly with paintings of other artists in some grand exhibit.

This attitude may be understandable in fine arts, but has no place in the software business. The programmer does not work alone. There is always a customer or end user who is paying for the project and who wants and deserves a product that can be used effectively in the customer environment.

The point that what programmers produce belongs to the customer seems difficult for some programmers to grasp because of their ego involvement in the development of the programs.
The following guidelines should be helpful for programmers who want to produce software that is efficient and useful for the customer (6).

1. Keep the end user in mind throughout the entire software development process.

2. Be sure you know not only what you are going to code but why, and know this before you start. In knowing why you write a program, you get a better understanding of what the end user intends to do with your software.

3. Do not rush into coding prematurely. The more time you spend in analysis and design, the fewer errors you will have to correct during checkout.

4. Design the program in anticipation of changes to come. You will have less difficulty in accommodating changes when they arrive. You may be assured that some changes will have to be made.

5. Do not consider your job complete until the program documentation has been written and read by another programmer so that he can understand what you developed and, if necessary, he can modify it without undue difficulty.

6. Do not be afraid to admit errors or to ask for help. If there are problems they will be found sooner or later. It is best to find them as soon as possible.
7. Even though documentation is required to be prepared in accordance with certain standards, the outcome will not be useful unless you fully understand what the program is for and how it will be used.

In conclusion, a programmer should get his ego satisfaction from a satisfied customer. The important and long-term rewards go to those who put a smile on the customer's face and keep it there.

The purpose of the following chapters is to describe how FORTRAN compilers deal with source code constructions written by the user. The intent is not to describe in detail how each compiler works or to provide lists of specific coding rules because compilers are different, and each compiler would then require separate treatment. The goals of the following chapters are to provide the user with a feel for those things in FORTRAN source coding which usually have a direct effect on the object code produced by various compilers.

Unless otherwise specified in the following chapters, all examples will use standard FORTRAN naming conventions, (i.e., an INTEGER identifier begins with the letters I, J, K, L, M, or N and everything else is real).
CHAPTER BIBLIOGRAPHY


CHAPTER II

EFFICIENT USE OF DECLARATIVES

In this chapter major concepts of efficient subscripting of arrays will be discussed. There will be a description of internal data conversions and how minimizing the number of data conversions will improve program performance. The final topic in this chapter will discuss the use of initialization statements. It will be shown how a programmer can use the BLOCK DATA and DATA statements to improve efficiency in program execution.

Efficient Subscripting

The data aggregates in FORTRAN are subscripted array variables, vectors and matrices. Access to an element of an array is obtained by specifying subscripts. In a one-dimensional array, for example,

```
DIMENSION A(10)
```

the third element is located by coding A(3), the Nth element in A(N). In two-dimensional arrays, such as

```
DIMENSION B(10, 10)
```

the element in the Ith row and Jth column is located by coding B(I,J). Since most computers are designed with a
linear arrangements of addressable storage locations, multi-
dimensional arrays must be mapped onto a linear ordering (3). This is especially true for IBM 360 and UNIVAC 1100 series computers. Thus the physical arrangement of data in the computer's storage is usually different from the logical structure of the aggregate. It is necessary to compute the address of a particular element from the subscripts given. The mappings which transform subscripts into machine addresses are called subscripting functions; these are inserted in the object code by the compiler. The frequent evaluation of subscripting functions contributes to program inefficiency, since each time reference is made to an element of array, a physical address must be calculated.

Subscript computations at object time are especially expensive for multidimensional arrays. It is always best to apply the rule "never use a vector when scalars will do, and never use an N-dimensional array when an array of N-1 dimensions will do."

If arrays are necessary, it is best to use constant subscripts whenever possible. If a constant subscript is used with a vector, for example $X(7)$, the resulting object code will resemble a reference to a scalar. This is true when using IBM 360, UNIVAC 1100 series computers. In general, a constant subscript in any dimension of an array reference will have the effect of reducing the rank of the array by one at compile time.
If multidimensional arrays are essential to an algorithm, the complexity of references may sometimes be reduced. Consider the following example:

```
DIMENSION X(10, 20, 8), Z(30, 30)

DO 5 I = 1, 10
DO 5 J = 1, 20
DO 5 K = 1, 8
  5 X(I, J, K) = 0

DO 75 I = 1, 30
  75 Z(I, I) = 1.0
```

The above could be changed so that an improvement in both core space and execution time would be realized.

The following code is functionally equivalent to the source code given in the previous example.
DIMENSION X (10, 20, 8), Z(30, 30)
DIMENSION XXX (1600), ZZ(900)
EQUIVALENCE (XXX(1), X(1, 1, 1))

DO 5 I = 1, 1600
5 XXX(I) = 0

DO 75 I = 1, 900, 31
75 Z(I) = 1.0

Changes like the above sometimes make the source coding more difficult to follow, but the performance improvement is often worth the inconvenience. If the program needs clarification, use more comments to explain what has been done.

If a specific element of an array is to be accessed many times, it is more efficient to transfer the contents of that array location to a scalar variable. This avoids the recalculation of the address each time the value is required.

This technique can frequently save large amounts of execution time. In general, the greater the dimensionality of the array, the more complex the subscription function will be. Therefore, if the programmer is using some cross section of an array frequently (such as the diagonal of a square
matrix or one plane of a three-dimensional array), it may be more efficient to move that set of values into an array of reduced degree. Most FORTRAN compilers map two-dimensional arrays by going down the columns and across the rows from left to right (4). Thus, an array of three rows and three columns would be as shown in Figure 1, page 17.

Array location (1, 1) occupies the first position, (2, 1) occupies the second position, (3, 1) occupies the third position, (1, 2) occupies the fourth position, and this will continue until array location (3, 3) occupies the ninth position. The complete map of the array is shown in Figure 2, page 18. The general rule for mapping a two-dimensional array onto linear storage is

Linear Position of array location \((I, J) = (J-1)\times NROWS + I\)

where \(NROWS\) is the number of rows in the DIMENSION statement for ARRAY. Every time an element is accessed from a two-dimensional array, the calculation for finding the linear position is performed. This calculation requires an addition, a multiplication, and a subtraction. In some cases it may be possible to reduce the amount of subscript calculation. If the same array element is needed in several places, replace the subscripted value with a scalar variable to which the value of the element has been assigned. In the statements

\[
\begin{align*}
I &= K \\
J &= K + 1 \\
X &= B(I, J)\times A
\end{align*}
\]
### 2-Dimensional Array

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(1,1)$</td>
<td>$A(1,2)$</td>
<td>$A(1,3)$</td>
</tr>
<tr>
<td>$A(2,1)$</td>
<td>$A(2,2)$</td>
<td>$A(2,3)$</td>
</tr>
<tr>
<td>$A(3,1)$</td>
<td>$A(3,2)$</td>
<td>$A(3,3)$</td>
</tr>
</tbody>
</table>

### Location Array Element

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(1,1)$</td>
<td>$A(2,1)$</td>
<td>$A(3,1)$</td>
<td>$A(1,2)$</td>
<td>$A(2,2)$</td>
<td>$A(3,2)$</td>
<td>$A(1,3)$</td>
<td>$A(2,3)$</td>
<td>$A(3,3)$</td>
</tr>
</tbody>
</table>

Fig. 1--Array storage
Fig. 2--The mapping of a two-dimensional array
\[ Y = B(I, J) \times C \]
\[ Z = B(I, J) \times D \]

the amount of subscripting may be reduced by coding

\[
\begin{align*}
I &= K \\
J &= K + 1 \\
TEMP &= B(I, J) \\
X &= TEMP \times A \\
Y &= TEMP \times B \\
Z &= TEMP \times D
\end{align*}
\]

As with reducible expressions, some compilers will not re-subscript identical variables provided that the statements are immediately juxtaposed. The reason is that when new expressions are computed, they replace available values in the computer's registers. However, with most compilers, the statements

\[
\text{DIMENSION } A(5, 5), B(5, 5)
\]

\[ X = A(I, J) + B(I, J) \]

will require two subscript evaluations.

A substantial portion of program execution time may be spent in the calculation of the subscripting function. The elimination of some of this computation can substantially reduce execution time. As mentioned previously, if the same subscripted element is referenced several times, some improvement may result from replacing the subscripted variable with a temporary variable assigned the value of the subscripted element. If different arrays use the same subscripts,
further improvement can be obtained by performing the subscripting function in the program. Programs which involve much subscripting such as Gauss-Jordan eliminations, matrix inversions, and eigenvalue calculations can benefit from this rule.

In the following example the location to be filled is read as I, J, and K; array B receives the larger of X and Y; and array C the smaller of X and Y. (See Figure 3, p. 21) This program could be rewritten with a single explicit calculation of the array subscripts. (See Figure 4, p. 22)

Data Type and Conversion

One of the most overlooked areas of source code optimization is minimizing the number of internal data conversions during execution. Strict FORTRAN compilers, in a way, force the user to think about conversions by not permitting mixed mode expressions. However, because mixed mode expressions are permitted by some compilers, it is easy to write programs which compile without error and always produce correct output but which require many unnecessary and transparent data conversions during execution. In some compilers, such conversions result in calls to FORTRAN library subroutines; in others, in-line code performs the conversions (2, 3).

The mode of an arithmetic statement must be integer or floating point. The variable on the left-hand side of the
DIMENSION A(8, 6, 40), B(8, 6, 40), C(8, 6, 40)

DO 7 N = 1, 1920
READ (5, 100) I, J, K, X, Y

100 FORMAT (3I2, 2F10.2)
A(I, J, K) = X + Y
IF (X.GT.Y) GO TO 5
B(I, J, K) = Y
C(I, J, K) = X
GO TO 7

5 B(I, J, K) = X
C(I, J, K) = Y

7 CONTINUE

Fig. 3--Arrays using the same subscripts
DIMENSION A(8, 6, 40), B(8, 6, 40), C(8, 6, 40)
DIMENSION AA(1920), BB(1920), CC(1920)
EQUIVALENCE (A(i, 1, 1), AA(i)), (B(1, 1, 1), BB(1)),
            (C(1, 1, 1), CC(1))
DO 7 N = 1, 1920
    READ (5, 100) I, J, K, X, Y
100 FORMAT (312, 2F10.2)
C--------------------------------------------------------------------------
C COMPUTE MSUB AS THE LINEAR SUBSCRIPT WHICH EQUALS THE
C SUBSCRIPT (I, J, K). THAT IS:
C A(I, J, K) IS AA(MSUB)
C B(I, J, K) IS BB(MSUB)
C C(I, J, K) IS CC(MSUB)
C--------------------------------------------------------------------------
MSUB = (K-1)* 48 + (J-1)* 8 + I
AA(MSUB) = X + Y
IF(X.GT.Y) GO TO 5
BB(MSUB) = Y
CC(MSUB) = X
GO TO 7
5 BB(MSUB) = X
    CC(MSUB) = Y
7 CONTINUE

Fig. 4--Efficient use of array with same subscripts
equal sign in an arithmetic statement need not be of the same mode as the expression on the right-hand side. If the variable on the left is an integer quantity and the expression on the right is floating point, the expression will first be evaluated in floating point, the portion following the decimal point will be dropped, and the remainder will be converted to an integer quantity. If the variable on the left is floating point and the expression on the right is integer, the latter will be evaluated as an integer expression, and the result will be converted to floating point (3).

Consider the following example:

\[ X = I \times Z - (I-1)/T + I \times 6 \]

Two conversions from INTEGER to REAL are required and some compilers would produce three. The expression \((I-1)\), because of the FORTRAN rules for expression evaluation, must be evaluated in integer mode and the result converted to real. The appearance of the variable \(I\) in two other places would cause either one or two additional conversions, depending on the "intelligence" of the compiler.

To reduce the number of conversions in the previous example to one, the following method could be used:

\[ V = I \]
\[ X = V \times (Z + 6.0) - (V-1.0)/T \]

Where \(I\) is converted to REAL by the assignment statement \(V=I\). Collecting the first and last terms in the original example and factoring also saves execution time by reducing
the number of multiplies. The following is another example which illustrates that the most elegant algorithm may not be the most efficient.

```fortran
DO 4 J=1, 1000
  4  X(J) = J
```

The above example will result in 1,000 conversions of J from INTEGER to REAL at execution time. The following code, while more lengthy in its source form, will execute faster and, for most compilers, take less core space because no conversions are required:

```fortran
Z=0.0
DO 4 J=1,1000
  Z=Z+1.0
  4  X(J)=Z
```

Conversions from INTEGER to REAL and vice versa are usually more costly than conversions from real to DOUBLE PRECISION. Conversions from DOUBLE PRECISION to REAL are usually free, as are conversions within the INTEGER category (i.e., INTEGER * 2 to INTEGER * 4 and vice versa).

It would be wise for the users of a FORTRAN compiler to determine how the compiler handles constants with respect to data type. It would be easier for a compiler to ignore the context in which a constant is used and simply preserve the type specified by the user rather than to perform a compile-time conversion of the constant to the optimal type for
execution (8). For example:

\[ \text{IF}(X.EQ.17) K = 2.0 \]

A "quick" compiler might emit object code which contains conversions of 17 from INTEGER to REAL and 2.0 from REAL to INTEGER. If such is the case, the user should always attempt to match the constant type with the type of usage, which, for the above example, would become \( \text{IF}(X.EQ.17) K = 2 \).

Efficient Initialization

The initialization statements BLOCK DATA and DATA may be used to advantage when available. Since these initialization statements preset values of previously allocated variables; they cause no increase in storage requirements. Using initialization statements benefits the program in two ways.

1. The instructions and constants required for initializing the variables will be eliminated from the program, reducing the size of the object program.

2. The execution time of the program will be reduced because of the elimination of those statements (4).

Because all storage allocation in FORTRAN is "static," the data initialization statements cannot be used to initialize variables in subroutines which will be called more than once. This characteristic can be used to advantage when setting "first time" switches and defining default values for variables.
The data initialization statements are particularly useful when arrays are being initialized. The next example shows how an initializing DO loop may be replaced by a DATA statement. Note that since no statement number precedes the DO statement, it will be executed only once. The DO is, therefore, a candidate for replacement by the DATA statement.

\[
\text{DIMENSION COST (20)}
\]
\[
\text{DO 20 I=1, 20}
\]
\[
\text{20 COST(I)=0.0}
\]

This coding may be replaced by

\[
\text{DIMENSION COST (20)}
\]
\[
\text{DATA COST/20 * 0.0/}
\]

The DATA statement may be used to prestore constants rather than using executable statements for initialization. This is particularly important for subprograms which are executed many times since both execution time and storage are saved. As was stated earlier, executable statements must be used for initialization if certain variables or arrays must be initialized every time the subprogram is entered. Consider the following example:

\[
\text{SUBROUTINE JOE (X, Y, A, B, N)}
\]
\[
\text{DIMENSION A(N), B(N), TEMP(100), KEY(6)}
\]
\[
\text{DO 1 I=1, 100}
\]
\[
\text{1 TEMP(I)=0.0}
\]
\[
\text{PI=3.14159265358979}
\]
\[
\text{DO 2 I=1,4}
\]
Here, array TEMP must be initialized during every call of JOE but PI and array KEY need only be initialized once so an optimized version is as follows:

```
SUBROUTINE JOE (S,Y,A,B,N)
DIMENSION A(N), TEMP(100), KEY(6)
DATA PI/3.14159265358979/, KEY/4*3,7,9.
DO 1 I=1, 100
  1 TEMP(I)=0.0
```

The most obvious area for reducing the storage required is in the organization of data stored in arrays. A useful technique is to force two or more data arrays to share the same space in storage. For example, if \( A \), a 500 element one-dimensional array, is used in one part of a program and \( B \), another array of 500 element (or less) is needed in another portion of the program, they can be forced to share the same storage locations by including an `EQUIVALENCE` statement after the `DIMENSION` statements. For example,
DIMENSION A(500), B(500)
EQUIVALENCE (A(1), B(1))

which causes A(1) and B(1), A(2) and B(2), ... to share the same locations. Care should be taken to ensure that the two arrays are never used at the same time.

Arrays of different dimensions may be EQUIVALENCE'd. If a two-dimensional array were used in the first half of a program and a one-dimensional array were needed for an independent section of the program, the following statements might be used:

DIMENSION A(50, 50), B(2500)
EQUIVALENCE (A(1,1), B(1))

The A array and the B array share the same locations. The use of EQUIVALENCE eliminates wasted space. When one array space is no longer needed, it can be used by another array. (See Figure 5, page 29.)

Occasionally, it may be useful to have two copies of the same data in storage - one copy INTEGER and the other REAL. If sufficient storage is not available, one copy can be stored and the mode can be changed in place when required. Assume the array of 1600 elements is referred to as I when INTEGER and AI when REAL. Then it is possible to write:

DIMENSION I(1000), AI(1000)
EQUIVALENCE (I(1), AI(1))

causign the two arrays to be in the same storage locations. If the data is to be converted from INTEGER to REAL, the
<table>
<thead>
<tr>
<th>A(1,1)</th>
<th>A(2,1)</th>
<th>A(3,1)</th>
<th>A(4,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(2)</td>
<td>B(3)</td>
<td>B(4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(1,2)</th>
<th>A(2,2)</th>
<th>A(3,2)</th>
<th>A(4,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(51)</td>
<td>B(52)</td>
<td>B(53)</td>
<td>B(54)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(1,50)</th>
<th>A(2,50)</th>
<th>A(3,50)</th>
<th>A(48,50)</th>
<th>A(49,50)</th>
<th>A(50,50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2451)</td>
<td>B(2452)</td>
<td>B(2453)</td>
<td>B(2498)</td>
<td>B(2499)</td>
<td>B(2500)</td>
</tr>
</tbody>
</table>

Fig. 5--Equivalence arrays
following statements are executed.

DO 50 J=1,1000

50 AI(J)=I(J)

To convert back to integer,

DO 60 J=1,1000

60 I(J)= AI(J).

Of course this conversion process is time consuming, but only one copy of the data is kept in storage. This process saves space at the expense of time.
CHAPTER BIBLIOGRAPHY


CHAPTER III

EFFICIENT USE OF EXECUTABLES

A program that is written today will more than likely have in one form or another SUBROUTINES, EXPRESSIONS, and DO LOOPS as part of its structure. These three things will be executed many times during program execution. In this chapter, emphasis will be placed on optimizing the use of these three integral parts of a computer program.

We need to know the cost in computer time of invoking external subprograms. We need to know how to minimize the amount of execution time spent in evaluation of an expression. In FORTRAN the direct techniques for interation is the DO statement. Since a large proportion of a program's time is usually spent in loops, the loop should be optimized.

External Subprograms

It is always costly in terms of execution speed to invoke external subprograms. There are two different forms of external subprograms, open and closed (2, p. 20).

An open subprogram is one which is inserted into the calling program's code every time it is invoked. Assembler language programmers frequently use open subprograms called MACRO-instructions. Open subprograms are inserted by the FORTRAN compiler for performing tasks which require only a
few instructions. Functions such as FLOAT, IFIX, and ABS are frequently compiled as open subprograms.

A closed subprogram is one which is independent of the main routine. Closed subprograms are used more frequently than open subprograms. FORTRAN subroutines and FORTRAN functions are closed subprograms. No matter how many times it is invoked, only one copy of a closed program is inserted into the program. For this reason, closed subprograms are more efficient than open subprograms in their use of processor storage. However, the code required to link a program to a subroutine and to pass to it the calling parameters imposes both a computational and a storage penalty. Thus, care should be taken to insure efficient subprogram linkage. In this discussion the word linkage and load mean the same thing. IBM and UNIVAC use the word "linkage" whereas CDC and HONEYWELL use the word "load."

One of the best ways to reduce subprogram linkage overhead is to eliminate unnecessary and redundant subroutine calls. Subprograms may be invoked either explicitly or implicitly (2, p. 28). Subroutines are explicitly invoked by means of a CALL statement, and functions are explicitly invoked by means of a function reference in an arithmetic or logical expression. Implicit invocation of a subprogram occurs when the compiler is required to insert relatively complex code into the object program. Implicit subprogram calls are generated for such things as mode conversions,
subscripting, some types of exponentiation, and input/output statements. Indeed, some compilers with limited storage generate implicit calls for almost every type of statement (2, p. 23).

In the next chapter, this paper will deal with INPUT/OUTPUT statements. INPUT/OUTPUT statements always result in calls to FORTRAN library routines, so the user should minimize the number of INPUT/OUTPUT statements.

Like all other subprogram calls, a reference to an external user supplied SUBROUTINE or FUNCTION is expensive because of the linkage overhead, which is a function of the number of parameters and their usage. There is always a constant amount of linkage overhead present even if the subprogram has no explicit parameters. There is an additional burden placed on the object code of the calling program as a result of a SUBROUTINE CALL, because the compiler must emit code which assumes that every variable in every COMMON block specified by the calling program must be available to, and may be changed by, the called subprogram (1). In some cases, the effect of this assumption on register load/store activity could produce more overhead in the calling program than in the linkage itself.

Many compilers, because of hardware architecture, produce more efficient object code for programs containing a minimum number of COMMON blocks. This is because the placement of COMMON blocks is not a function of the compiler but
of the loader, and each COMMON block requires separate addressability which, for a program containing a large number of COMMON blocks, interferes with optimum register assignment. This is true for UNIVAC 1100 series computer but is not the case for CDC 6600. For machines with an elaborate indirect addressing capability, it would be possible for a compiler to emit code which is not complicated by this addressability problem. However, for most machines, it is probably best for the user to combine all COMMON blocks into one.

As far as explicit subprogram parameters are concerned, it is generally best to minimize the number of parameters. Until recently, almost all FORTRAN compilers treated all parameters as "call-by-reference" which means that the called program requires the external address of the parameter at every reference (7, p. 6). For a called program containing many references to external parameters, a considerable amount of overhead is introduced. For such programs, it is usually best to move all scalar parameters to local variables at the outset in the called program and then use the local copy of the parameter at all subsequent references, minimizing the number of explicit references to scalar parameters. For example,

```fortran
SUBROUTINE X(A,B,C,D,E,F,G)
DIMENSION C(10,4), F(400), T(50)
```
Y = A
Z = B
W = D
S = G

RETURN
END

The previous example assumes that A, D, and G are input-only parameters, that E is output-only, and that B is both input and output. The arrays C and F could be either or both. Regarding the scalars A, B, D, E and G, note that the minimum number of references is made to each, with the body of the subroutine using their local equivalents, Y, Z, W, V, and S, respectively.

It is usually not worthwhile to move a scalar parameter to a local variable if the parameter is referenced only once or twice. It is never worthwhile to move array parameters to local arrays in the called program, especially for large arrays, because the overhead of moving them and the extra storage required probably far offset any performance gain that could be realized by so doing (8).

In order to reduce the number of implicit function calls, the programmer must be aware of situations in which they are likely to be included in the object code. Visual
inspection of an object listing (if available) or a "load map" may be useful here.

One situation in which implicit function calls are generated is mode conversion in an arithmetic or logical expression. The expression

\[ X = A + I \]

will result in the following object code:

1. Convert the value of I to floating point, and store in some temporary location.
2. Add the value of A to the converted value of I.
3. Store the result in \( X \).

Some programmers explicitly recognize the fact that a conversion will be performed by writing the equivalent statement:

\[ X = A + \text{FLOAT}(I) \]

However, the code generated is the same in either case. Mode conversions may be reduced by treating them as reducible expressions and assigning the conversions of temporary variables. For example, the statements

\[ X_1 = A + I \]
\[ X_2 = A + I \]

are more efficiently written as

\[ AI = I \]
\[ X_1 = A + AI \]
\[ X_2 = B + AI \]

It is important that the programmer be alert to situations in
which an implicit function is likely to be invoked. Sometimes a simple, innocent-looking modification to a program may allow an implicit call to sneak in. The statement

\[ X = A \times 2 \]

is quite straightforward but the similar statement

\[ X = A \times 2, \]

will cause an implicit function call in most systems. The first statement will usually be compiled into a multiplication, whereas the second will be evaluated by logarithms.

In Figure 6, p. 39, a program is given with multiple calls to a subprogram. The example assumes that the variables LENGTH, MIN, MAX are unchanging.

In Figure 7, p. 40, a rewrite of the example is given implementing a reduction in the parameter list. This example is true for certain computers.

The CDC 6600 FORTRAN compiler will accept the reduction in parameter list given in Figure 2 and the generated object code will be executed more efficiently. On the other hand, the UNIVAC 1108 FORTRAN compiler will accept the reduction in parameter list given in Figure 2 but the generated object code will not be executed. There is a difference in the way addresses are passed to subprograms by different computers, and the programmer must be aware of these differences in order to program them efficiently (10).
PROGRAM FIX (INPUT, OUTPUT,...

CALL ITEM (J, K, LENGTH, MIN, MAX)

CALL ITEM (M, N, LENGTH, MIN, MAX)

CALL ITEM (IVAR, JVAR, LENGTH, MIN, MAX)

END

SUBROUTINE ITEM (I, J, K, L, M)

RETURN
END

Assume the variables LENGTH, MIN, MAX are unchanging.

Fig. 6--Reduction in Parameter List
PROGRAM FIX (INPUT, OUTPUT,...
  .
  .
CALL ITEM (J, K, LENGTH, MIN, MAX)
  .
  .
CALL ITEM (M, N)
  .
  .
CALL ITEM (IVAR, JVAR)
  .
  .
END

SUBROUTINE ITEM (I, J, K, L, M)
  .
  .
RETURN
END

LENGTH, MIN, MAX could also possibly be placed in common.

Fig. 7-- Reduction in Parameter List
Loop Optimization

In FORTRAN the most direct technique for iteration is the DO statement. Since a large proportion of the program's time is usually spent in loops, the loops should be optimized. Computer time should not be wasted by careless programming of loops. Operations which can be performed before entering a loop should be removed from the scope of the loop; redundant calculations should be avoided.

Expressions in DO loops that involve variables which are independent of the loop variable should be evaluated outside the loop and, if necessary, stored in a temporary location. This technique avoids unnecessary and repetitious calculation. In the next example the expression \((X^2 + 3.0X + 2.0)\) is independent of the loop variable \(J\), but will be calculated fifty times. The expression can be calculated once outside the loop and stored in location QUAD:

\[
X = 3.4 \\
DO 9 J = 1, 50 \\
9 \ Y = Y + A(J) * (X^2 + 3.0X + 2.0)
\]

should be replaced by

\[
X = 3.4 \\
QUAD = X^2 + 3.0X + 2.0 \\
DO 9 J = 1, 50 \\
9 \ Y = Y + A(J) * QUAD
\]

In the following program the expression \((3.0X \times \text{SIN}(X))\) may be removed from the loop. As a result, the expression
needs to be evaluated only once, instead of one hundred times. Notice also that the SIN function, which is reducible, is invoked only once:

\[
X = .4739 \\
DO 7 I = 1, 100 \\
7 A(I) = 3.0*X*SIN(X)
\]

should be replaced by

\[
X = .4739 \\
Y = 3.0*X*SIN(X) \\
DO 7 I = 1, 100 \\
7 A(I) = Y
\]

Expressions involving subscripted variables in which subscripts are independent of the DO loop should also precede the loop. This avoids the calculation of a subscripting function each time the expression evaluated:

\[
I = K+L \\
DO 6 J = 1, 30 \\
6 A(J) = B(J)*C(I)
\]

should be replaced by

\[
I = K+L \\
CI = C(I) \\
DO 6 J = 1, 30 \\
6 A(J) = B(J) * CI
\]

If two adjacent DO loops have the same limits on the loop variable, it may be possible to combine the two loops. This operation not only reduces the size of the program, but can also substantially reduce the execution time.
In the following example, two six hundred word arrays are set to zero by two separate DO loops:

```
DO 20 I = 1, 600
20 A(I)=0.0
DO 30 I = 1, 600
30 B(I)=0.0
```

However, only one loop is necessary:

```
DO 40 I = 1, 600
A(I) = 0.0
B(I) = 0.0
```

This technique, known as loop jamming, reduces overhead in the loops by 50 per cent (2).

Every iteration of a DO loop requires incrementing and testing of the loop variable. This overhead may be reduced at the expense of additional instructions by the technique of "unrolling" the loop (2).

In the program segment

```
DO 7 I = 1, 4
7 A(I) = 0.0
```

run time can be reduced by coding

```
A(1) = 0.0
A(2) = 0.0
A(3) = 0.0
A(4) = 0.0
```

If the number of iterations of a loop is large, a complete unrolling is impractical. The overhead may be halved
by a partial unrolling as follows:

\[
\text{DO } 7 \ I = 1, 1000 \\
7 \ A(I) = 0.0
\]

may be replaced by

\[
\text{DO } 7 \ I = 1, 500 \\
A(I) = 0.0 \\
7 \ A(I+500) = 0.0
\]

or by

\[
\text{DO } 7 \ I = 1, 999, 2 \\
A(I) = 0.0 \\
7 \ A(I+1) = 0.0
\]

If a programmer is concerned with reducing execution time, a four way unrolling is possible.

\[
\text{DO } 7 \ I = 1, 997, 4 \\
A(I) = 0.0 \\
A(I+1) = 0.0 \\
A(I+2) = 0.0 \\
7 \ A(I+3) = 0.0
\]

Of course, eight way, ten way, or hundred way unrollings are possible, but the programmer must decide how many unrollings are necessary.

If a test inside a DO loop is not influenced by any of the variables in the loop then it may be possible to remove the test from the loop. For example, in the code:
DO 50 K = 1, 1000
IF(T.GT. 0.0) GO TO 40
A(K) = B(K) + C(K)
GO TO 50
40 A(K) = B(K) - C(K)
50 CONTINUE

the variable T is unaffected by the loop, but the IF statement is executed 1000 times. The program could be restructured to decrease the execution time by coding

IF(T.GT. 0.0) GO TO 40
DO 30 K = 1, 1000
30 A(K) = B(K) + C(K)
GO TO 60
40 DO 50 K = 1, 1000
50 A(K) = B(K) - C(K)
60 CONTINUE

Although this second program is more complicated, the IF statement is executed only once.

Many computer decisions can be avoided during searches to find a value if the code is properly written. Thus, to find which element in an array is the same as a given value, one can write

DO 10 I = 1, N
IF(XA.EQ.A(I)) GO TO 20
10 CONTINUE
I = N+1
20 .....
If I is N + 1 at statement 20, XA is not in the array. This requires two decisions per pass through the loop: one for the IF and one for the DO answering the question "is I equal to N?" A better method requiring only one decision per loop is

\[ A(N + 1) = XA \]
\[ I = 0 \]
\[ 10 \ I = I + 1 \]
\[ \text{IF}(XA .NE. A(I)) \text{ GO TO 10} \]

Unless the machine fails, the loop is guaranteed to find the value in array A, which is equal to XA, since the last element A(N + 1) is set to XA initially. This is a case where the normal FORTRAN DO statement is not always best.

In nested loops, attempt to make the innermost loop have operands that are in adjacent words whenever possible. For example,

```
DIMENSION A(16, 16), B(16, 16)
DO 10 I = 1, 16
  DO 10 J = 1, 16
    10 A(I, J) = A(I, J) + B(I, J)*T/FLOAT(J)
```

should be re-written by reversing the order of the I and J loops as

```
DIMENSION A(16, 16), B(16, 16)
DO 10 J = 1, 16
  DO 10 I = 1, 16
    10 A(I,J) = A(I,J) + B(I,J)* T/FLOAT(J)
```
since the first form uses elements that are sixteen words apart while the second form uses adjacent elements. This prevents possible memory bank conflicts in some machines with interleaved memory banks and simplifies subscript incrementing in many compilers. On the CDC 6600 computer such a change can make an enormous difference in efficiency. However, the same change on the UNIVAC 1108 would result in no gain in efficiency. The UNIVAC 1108 does not have interleaved memory banks.

Expression Evaluation

One of the most frequent causes of computational inefficiency is the calculation of unnecessary expressions. Some compilers are able to detect and correct these faults at the expense of increased compile time. Since one of the advantages of "high level" languages is computer independence, one should not rely upon the existence of a sophisticated compiler to optimize programs; especially since the optimizing compiler may not always be available. Of course, a "smart" compiler should be employed when available, since even the most sophisticated program may benefit from compile-time optimization (7, p. 4).

It is often difficult to determine the extent and nature of the optimization provided by a given compiler. Compilers such as WATFOR and IBM OS/360 FORTRAN G provide very little optimization; however, the IBM OS/360 FORTRAN H performs extensive optimization. The CDC SCOPE 3.4 RUN2P3 Compiler has
very little optimization, whereas the CDC SCOPE 3.4 FTN Compiler has three different levels of optimization. As a rule, it is best to eliminate redundant computations when doing so provides a reasonable return for the effort expended.

The user should always attempt, in the source program, to minimize the number of execution-time evaluations in an expression. Most compilers, including those that were discussed in the preceding paragraph, are able to do this automatically for the user for common subexpressions within a single statement only.

For example:

\[ X(J*I) = ((A*B)/Y + T/(A*B)) + Z(J*I) \]

Most compilers would recognize that \((A*B)\) and \((J*I)\) are both used twice and would produce object code which evaluates each of these expressions only once \((7)\).

Consider the following example:

\[ Z = ((A+(1.0/X))/(4-(A+1.0/X))) \]

The expressions \((1.0/X)\) and \((A+(1.0/X))\) would be evaluated only once. For compilers which recognize these common subexpressions within a single statement, two conditions are sometimes placed on the source code so that such subexpressions are detectable.

1. A subexpression is a candidate for detection as common only if it is parenthesized. Both examples that have been given have parentheses surrounding the common subexpressions. If, in the first example, the first occurrence of the
expression \((A*B)\) had not been parenthesized, \(A*B\) would not have been recognized as being common and would have been evaluated twice. Note that subscript expressions are always candidates for this optimization because subscripts are always enclosed in parentheses (a comma separating subscripts for multidimensional arrays is an exception but can be handled internally by the compiler with no problems).

2. Two parenthesized subexpressions are recognized as being common only if they are identical. It is not sufficient that they be algebraically equivalent--they must be identical. For example, \((A*B)\) would not be recognized as being identical to \((B*A)\), although they are equivalent expressions. The definition of "identical" depends on the compiler--some do a character-for-character comparison of the source code (which would mean that \(1.0\) is different from \(1.00\)) and others perform the comparison after encoding the expression internally (where equivalent constants would be considered identical).

In most compilers, within a single statement, multiple references with identical arguments to the same function will usually result in only one evaluation of the function, regardless of whether or not the references are parenthesized.

The recognition of common subexpressions within a single statement is fairly commonplace among FORTRAN compilers. However, only a few compilers are capable of detecting and eliminating redundant evaluations of expressions on a more global basis (i.e., across many statements). However, in the
absence of information about any specific compiler, it is always possible for the user to write a source program which will minimize the number of evaluations of an expression, regardless of the compiler being used.

Even though a certain expression may appear only once in the source program, the flow of the program may necessitate its evaluation many times. Consider the following example:

```
DO 5 I = 1,100
Q(I) = 0.0
DO 5 J = 1,100
5 Q(I) = Q(I)+A*B*X(J,I)
```

The subexpression A*B, although constant throughout both DO-loops, might be evaluated 10,000 times depending on the compiler. Also, the effective address of Q(I) might be computed 10,100 times even though it is constant throughout the inner loop on J. To eliminate the possibility of these redundant evaluations, the above code could be changed to:

```
AB = A*B
DO 5 I = 1,100
S = 0.0
DO 4 J = 1,100
4 S = S+X(J,I)
5 Q(I) = AB*S
```

Again, an optimizing compiler capable of recognizing common subexpressions on a global basis could have produced,
for the first example, object code resembling the latter example with two exceptions:

1. The multiplication by AB would appear in the inner loop if the compiler is incapable of algebraically factoring out the constant multiplier AB.

2. A new variable S would not be introduced, but the effective address of Q(I) would be computed only once (in the outer loop). In this regard, it might be detrimental to the object code to introduce the variable S in the source program (5).

There are a few miscellaneous optimizations related to the evaluation of expressions which are machine dependent. The optimizations listed below are for the CDC 6600 and UNIVAC 1108 machines (10).

1. Since an add instruction is usually much faster than a multiply, it is best to replace a "multiply by 2" with an add:

   \[(2.0 \times X) \text{ would become } (X+X)\]

   a. This change should not be made for "preferred subscripts";

   b. This change should not be made for binary machine if the compiler is capable of reducing the strength of the multiply to a shift instruction.

2. Since a multiply instruction is usually faster than a divide, it is best to use multiplies whenever possible:

   \[(X/4.0) \text{ would become } (X\times0.25)\]
3. For exponentiation using small constant integer exponents, it is usually best to replace the operation with a series of multiplies:

\[ x^{**4} \text{ would become } (x*x*x*x) \]

A few compilers may be able to perform all the above optimizations automatically for the user. But, in the absence of information about specific compilers, other than the two mentioned above, the user should make these optimizations manually.
CHAPTER BIBLIOGRAPHY


CHAPTER IV

EFFICIENT INPUT/OUTPUT OPERATIONS

Optimization of input or output is not as easy or "obvious" as is optimization of a program. There are many different types of devices connected to computers having varying speeds and capabilities. However, there are some general principles that can be stated which apply to most equipment. Before those principles are given, there are several things the user must consider.

One of the first considerations is how the data will be used. It can be for internal use: copied to and from a computer device because there is not enough room in memory, or for external use: by humans or by other computers such as punched cards, line printer output, or magnetic tapes to be transmitted elsewhere. Internal usage suggests no need to translate the binary values, while external usage will always require translation from binary to something else which in most cases result in characters.

A second consideration is what kind of devices can be used. Devices are usually used for internal purposes only, or for external purposes only, except magnetic tape which can be used for both. Thus Large Core Memory (LCM), disks or drums, or magnetic tape are used for internal purposes while tape, line printers, card readers, card punches, plotters or
time sharing terminals are used to interface with humans or other machines. The distinction is that internal-purpose devices are fast and the other devices are (usually) slower.

Another consideration is the various types of input/output: access, word type, packing, and character set. Internal-purpose devices (except tape) are capable of sequential access or random access. By random access is meant that data can be conveniently retrieved in a different order from the one in which it was written. A number of devices are capable of both machine binary data or character coded data. In some cases small groups of data can be packed into larger groups. Where coded data or other machines are involved, translation between different character codes or word representation may be needed.

Now that these considerations have been discussed, the following sections will give specific information on efficient input/output operations.

Unformatted I/O

The FORMAT statement indicates the conversions required between the external and internal representation of data. The character form of the output sent to the printer is very different from the internal form required for computation. To perform these conversions it is necessary to invoke subroutines at execution time. (3) Because these operations often use a surprising amount of compute time, it is good
practice to use unformatted I/O statements when data is written out to be reread later by the same or another program. Information to be printed must, of course, be sent to the printer in properly converted form.

To convert the following formatted WRITE statement

```
WRITE (8,100) NPART, NWHSE, (NSTOCK (I), I=1, 10)
100 FORMAT (I10, I8, 10 (I6))
```

to an unformatted WRITE statement, simply delete the FORMAT number and statement:

```
WRITE (8) NPART, NWHSE, (NSTOCK (I), I=1, 10)
```

The CDC 6000/7000/cyber computer is the point of reference in the following paragraph. (1, 5)

If the data consists of small integers, binary does not pack very efficiently. It takes 60 bits to represent each value whereas it may take as few as 6 bits to represent a one-digit value in character coded form. However, because of the relative inefficiency of character code translation, it would be better in this case for the user to pack or unpack several values into one machine binary word. For example, to pack 3 one-digit values into one word,

\[ J = I(1) + 10 \ast I(2) + 100 \ast I(3) \]

could be used. To unpack these same values later,

\[ I(1) = \text{MOD} (J, 10) \]
\[ I(2) = \text{MOD} (J/10, 10) \]
\[ I(3) = \text{MOD} (J/100, 10) \]
could be used. Note that this slows down the computer's access to the data but speeds up the I/O (5, p. 309).

List Equivalencing

Use a single data array in I/O statements where possible because it minimizes the number of input/output subroutine calls. This would be done via EQUIVALENCE of variables or smaller arrays to a large array, if necessary. With CDC FTN compiler (and many other compilers), an implied DO loop with unit increment is treated as a single array. Thus the code:

```
DIMENSION A (10)
WRITE (3) A
WRITE (3) (A (I), I=1,10)
```

is the same for both WRITE statements. However, if the increment is not one or if the compiler being used does not optimize unit increments, the implied - DO will not be as efficient as transferring a whole array. This is because each element in the implied DO will require an I/O subroutine call for each pass through the loop versus one call for a whole array. (7)

Another example in making use of the EQUIVALENCE statement in I/O processing is as follows:

A program outputting five variables such as:

```
WRITE (6,100) HOURS, WAGE, FICA, FEDTAX, STTAX
```

will be more efficient to write
DIMENSION DUMMY (5)
EQUIVALENCE (HOURS, DUMMY (1)), (WAGE, DUMMY (2)),
.     (FEDTAX, DUMMY (3)), (FICA, DUMMY (4)),
.     (STTAX, DUMMY (5))
WRITE (6,100) DUMMY

This procedure would save four subroutine calls. This is a significant number if the WRITE statement is executed frequently.

Blocking Records

Reading or writing a record is far slower than computing. Many computers are capable of performing tens of thousands of arithmetic operations in the time it takes to perform a single I/O operation. The disparity in time between the two types of operations exists because I/O operations are generally mechanical whereas arithmetic operations are electronic. In general, the process of accessing a record which requires starting a tape moving or positioning a disk arm consumes the most time. The actual data transfer of the record is rapid. By following the two rules given below a programmer may save large amounts of time by reducing the number of I/O requests.

1. For recording media such as cards and print lines whose size is fixed, a programmer should pack as much data per record as is consistent with legibility.

2. For other recording media such as tapes and disks, a programmer should write the largest record possible.
Operating systems provide a service to the programmer known as record blocking. Suppose a programmer were writing card image records to a tape:

```
DIMENSION NCARD (80)
WRITE (10,100) NCARD
100 FORMAT (80A1)
```

With most operating systems, it is possible to specify, on a control card, that the records are to be blocked. Blocking reduces data transmission time by increasing the density of data on the storage medium (usually tape or disk). On tape or disk, successive records are often separated by unused - and hence wasted - space. The amount of space wasted can be as great as 0.75 inches (on 7-track magnetic tapes). (A record is the data transmitted by a single read or write operation.) It is often possible to pack the records one after another, separated only by an end-of-record flag of some sort, thus reclaiming most of the lost space. See Figure 8, p. 60 for a look at different types of record marks.

Blocking must be accompanied by buffering. Blocking requires buffering because a whole block of data must be transmitted between the device and the computer, even if only a few words are needed. The buffer is used for the assembly and dis-assembly of the block (5, 8).

The data in the NCARD array is a single logical record. The illustration given in Figure 9, p. 61 shows a tape written in unblocked format, that is, with one logical record per block.
ZERO-BYTE AS RECORD MARK

| DATA | ZB | DATA | ZB | ZB |

Each record is terminated by a zero-byte (12 bits of zeroes). Three records are shown; the third is empty.

CONTROL WORD WITH RECORD LENGTH

| 7  | 7 WORDS | 3  | 3 WDS | 0  | 9  | 9 WORDS |

Each record is preceded by a control word giving the length of the record. Four records are shown; the third is empty.

Fig. 8--Record Marks
Fig. 9--Record Blocking

80 character blocks

3200 character blocks
Each contains 40 records
The 0.6 inch blank spaces between the blocks are called inter-block gaps. Notice that most of the tape is blank. With a blocking factor of 40, a larger fraction of the tape contains useful information and there is less wasted space. As a result of blocking, there are fewer tape accesses, less tape motion and faster execution. It is important to stress that blocking has no effect on program code. The operating system does all the work of blocking when writing and de-blocking when reading in the FORTRAN program. However, the larger the blocks used, the more core storage the operating system will use for buffers.
CHAPTER BIBLIOGRAPHY


CHAPTER V

CONCLUSION

Many of the recommendations set forth in this paper do not conform to today's concepts found in structured programming techniques. It is not the intent of the author to persuade programmers against using structured programming techniques in optimizing FORTRAN programs.

It is now recognized that high level language programs, e.g., ALGOL, PL/1, and COBOL, have potential for being efficient with the structured program approach. This is not necessarily true in writing a FORTRAN program. FORTRAN seems to be the one major high-level programming language that is not suited to the structured programming concept. There are three major reasons for this.

1. There is no concept of block structure in the FORTRAN language, as it is presently constituted. That is, it is not possible to group several statements together and treat them as if they were one statement. This immediately forces the programmer to "jump" around groups of statements.

2. There is no nested IF-THEN-ELSE statement in FORTRAN.

3. The DO statement in FORTRAN is not as powerful as the analogous form in COBOL, ALGOL, and PL/1.
Because of the lack of the above attributes in FORTRAN, it is almost impossible to do good structured programming in the language. However, it is the opinion of this author that a programmer should structure his programs as much as possible for readability and good programming style.

In developing a FORTRAN program the use of good programming style is important. However, with today’s competition among computer services, etc., and providing these services in a cost conscious environment, efficiency in programming may be the difference in the making of a successful business. This simply means that the programmer who has good programming style, and can write efficient programs, has the best chance of being successful.

It is a point of this author not to disagree with the structural programming approach, because the objectives of structured programming differ from the objectives of efficiency in programming. Structured programming objectives are as follows:

a. an increase in programmer productivity;
b. reliability of the produce code;
c. maintainability of the written code.

Whereas the objectives of efficient programming are to produce the most efficient object code that is possible for increase performance.

The recommendations that were discussed in this paper were proposed for those programmers who are truly concerned
about object code performance, and the things those programmers could do at the source code level to enhance the performance of the object code.

Many of the techniques discussed in this paper have been experienced by this author. A general purpose structural analysis program executing on a CDC 6600 computer was not competitive with the same general purpose structural analysis program executing on a comparable system. It was the task of this author to make the structural analysis program competitive on the CDC 6600.

First, the program was examined for repeated code. This was done by examining the source listing. However, there was not much that could be found by this method. The next step was to execute a program written by Control Data Corporation called "SPY" which would monitor the program and print out a report of CPU times by ascending core addresses. The core address intervals to be used could be determined by the programmer. By reading the printed core load map, a programmer is able to determine in what routines most of the CPU time has been spent. It was found that the program was spending most its time performing I/O commands. Immediately the I/O routines were modified to do random I/O rather than sequential and a 40 per cent in reduction of computer time was realized.

There are other cases in which these techniques were used and the results were good. One in particular was also
done by this author. The application was an oil exploration technique making use of well log data. The routine discovered by the "SPY" program was a cross correlation routine. The routine was modified employing techniques used in this paper and a 15 per cent reduction in computer time was achieved.

It is known that FORTRAN compilers that are written today are much more sophisticated, and many of them now have optimization features built in. However, because of the many variations in compilers on many different machines, it is to the advantage of the programmer to do as much optimizing of the FORTRAN program at the source code level that is possible. The probability of a program being executed on several different computers today is very high.

It has also been found that by selecting optimization features on some compilers have caused erroneous results and even program aborts during execution of the program. This is still another reason for a programmer to perform source code optimizations and not depend on the compiler to do optimizing for him.

It is the hope of the author of this paper that a programmer, after reading this paper, will be knowledgeable about the things that he can do at the source code level that will enhance object code performance.
BIBLIOGRAPHY

Books


Kreitzberg, Charles B. and Ben Shneiderman, The Element of FORTRAN Style: Techniques for Effective Programming,


Articles


