23-73
LA-5219-MS

LA. 230

CDC 7600
FORTRAN Optimizing Techniques

los alamos
scientific laboratory
of the University of California
LOS ALAMOS, NEW MEXICO 87544

UNITED STATES
ATOMIC ENERGY COMMISSION
CONTRACT W-7405-ENG. 36

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

Printed in the United States of America. Available from
National Technical Information Service
U. S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22151
Price: Printed Copy ~~\$2.00~~; Microfiche \$0.95

21.00

los alamos
scientific laboratory
of the University of California
LOS ALAMOS, NEW MEXICO 87544

LA-5219-MS

UC-32

ISSUED: June 1973

CDC 7600

FORTRAN Optimizing Techniques

by

Frank McGirt
Karl J. Melendez
Lawrence Rudzinski

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

CONTENTS

Abstract	1
I. INTRODUCTION	1
II. PROGRAMMING TECHNIQUES	2
A. Subscripted Variables	2
B. Branching Statements	2
C. DO Loops	3
D. Arithmetic Operations	4
E. Subroutines	5
F. Bank Conflicts	5
G. Miscellaneous	6
III. PROGRAMMING AND OPTIMIZATION TOOLS	7
IV. CONCLUSION	7
References	7

CDC 7600 FORTRAN OPTIMIZING TECHNIQUES

by

Frank McGirt
Karl J. Melendez
Lawrence Rudsinski

ABSTRACT

This paper describes operations and techniques that the FORTRAN programmer can perform manually at the source-code level to improve object-code performance.

I. INTRODUCTION

The problem of FORTRAN program optimization has often been attacked by providing an optimizing capability in the FORTRAN compiler.¹⁻⁶ This approach has several advantages. The programmer is able to use very complicated optimizing strategies simply by specifying a control-card parameter. The source code is left in a readable form, and in-code documentation such as comments is unaffected. In addition, a compiler with optimizing capability which can produce object code that is competitive in performance with hand code makes high-level languages acceptable for many purposes that still depend on hand code. An example of such a benefit is operating-system implementation and maintenance. The use of an optimal high-level language for system writing reduces the initial programming effort and makes the job of maintaining and moving such systems to new machines much more straightforward.

There are also disadvantages to optimizing compilers. Even the most sophisticated compiler cannot decide which sections of the program being compiled will be executed most frequently. If equal weight is assigned to all sections, optimization for infrequently used paths may interfere with

optimization for those paths used more frequently. Optimizing compilers are often expensive to develop and maintain, and almost always expensive to use. If a computer installation job mix consists primarily of short jobs that are run only a few times and then discarded, the cost of an optimizing compiler would far outweigh its usefulness. Optimizing compilers sometimes attempt to optimize with respect to the speed of object-code execution and also the size of the object code produced. Optimization for speed and size are usually in basic conflict with each other. Unless adequate documentation describing precisely all optimization done by the compiler is available, it is extremely difficult to determine the conditions under which optimization is done. It is also very important that the FORTRAN programmer learn programming techniques that result in good object code, regardless of the compiler used.

At present there is no formal optimizing compiler available for use at the Los Alamos Scientific Laboratory (LASL). Also, the most heavily used compilers at LASL execute on machines other than those for which they were originally developed. One would therefore expect that there are programming techniques that the FORTRAN

programmer could use which would lead to more efficient object-code execution. To check this hypothesis we conducted a survey of user jobs. This survey randomly sampled jobs submitted to the CDC 7600 computers and determined whether execution efficiency could be increased. Efficiency gains were achieved in three rather simple ways that limited the effort expended on each program. These three methods were:

1. Improvement of FORTRAN coding techniques,
2. Correction of deficiencies in system routines, and
3. Improvement of numerical algorithms.

This paper is concerned with the results obtained using Method 1. We describe techniques and operations that the FORTRAN programmer can perform manually at the source-code level to improve object-code performance. It is still the user's responsibility to determine whether these optimization techniques are of real benefit, based on execution frequency of the code in question.

II. PROGRAMMING TECHNIQUES

Techniques that the FORTRAN programmer can use at the source-code level to improve the object-code execution efficiency are presented. Examples with timings of operations are included when appropriate.

Most of the techniques are aimed at eliminating extraneous memory fetches. A quantity is fetched from memory whenever it appears on the right-hand side of an assignment statement, or in the logical or arithmetic expression of an IF statement. If quantities can be made to remain in registers until needed for computation, time-consuming memory fetches can be avoided.

A. Subscripted Variables

Whenever a subscripted quantity such as $A(I,J)$ or $A(I,J,K)$ is referenced in a FORTRAN program, much subscript computation

occurs. This is because a multiple-dimensional array is stored as a one-dimensional array. For some applications, one can eliminate expensive subscript computations by using identities to transform multiple-dimensional arrays into one-dimensional arrays. For example, if array A is dimensioned $L \times M$, then $A(I,J)$ is equivalent to $A(I+L*(J-1))$. If array A is dimensioned $L \times M \times N$, then $A(I,J,K)$ is equivalent to $A(I+L*(J-1)+L*M*(K-1))$. An example of a possible use of the transformation is as follows:

EXAMPLE 1

Old Method	New Method
DIMENSION A(5,1000,2)	DIMENSION A(5,1000,2)
DO 10 I=1,1000	IASUB=K*5000*(J-1)
10 A(K,I,J) = X(I)*Y(I)	DO 10 I=1,1000
	A(IASUB)=X(I)*Y(I)
	IASUB=IASUB+5
	10 CONTINUE

The new method is twice as fast as the old for at least two reasons. Expensive subscript computation is done once outside the DO loop instead of within it, and memory fetches of each index I,J,K within the loop are replaced by a single fetch of IASUB.

B. Branching Statements

Generally it is more efficient to structure FORTRAN programs so that code is executed in a straight-line manner. Straight-line flow avoids jumps that are costly in FORTRAN because such jumps are almost always out of stack. (The stack is a buffer containing 12 words of instructions. Each instruction is available much sooner than if it had to be read directly from core. See Ref. 7 for more information.)

There are two basic types of FORTRAN branching statements, the Arithmetic IF statement,

```
IF(X) 5,10,5
```

and the Logical IF statement

```
IF(X.EQ.0.) GO TO 10.
```

Given a Logical IF statement, it is generally about 30% faster to fall through, (fail the test) then to branch out. Therefore, program flow should be arranged so that the most frequently executed path is to fall through the test. In the example below, the New Method is faster, assuming that X is rarely equal to zero.

EXAMPLE 2
(X rarely equal to zero)

Old Method	New Method
IF(X.NE.0) GO TO 13	IF(X.EQ.0) GO TO 100
GK=Y	X=GH+GY
X=GK+GT	Y=X
10 T=Y+X	130 T=Y+X
STOP	STOP
13 X=GH+GY	100 GK=Y
Y=X	X=GK+GT
GO TO 10	GO TO 130

If there is a choice between using a Logical IF statement and using an Arithmetic IF statement for which two of the three statement numbers are identical, always use the Logical IF statement. The Logical IF Statement is faster to execute, reduces the number of statement numbers required, and contributes to a more readable program. The example below illustrates a use of an Arithmetic IF statement for which Logical IF statements can be substituted efficiently.

EXAMPLE 3

Old Method	New Method
IF(I) 10,5,5	IF(I.GE.0) X=X+1.0
5 X=X+1.0	Y=Y+1.0
10 Y=Y+1.0	

Arithmetic IF statements should be used instead of Logical IF statements when the three possible branches actually are three distinct statement numbers.

C. DO Loops

DO loops are probably used inefficiently more often than any other FORTRAN statement. This may be because most FORTRAN programmers do not realize the amount

and complexity of the object code that the FORTRAN compiler must generate for each DO statement encountered. Variables are set up for the DO-loop index, DO-loop increment, and DO-loop limit. Because these quantities are seldom kept in registers, expensive stores and fetches must be executed when the increment is added to the index. Tests, made to check if the index has exceeded the limit, are also expensive. Therefore even simple DO-loop constructs are expensive to use, and the FORTRAN programmer must carefully ensure that additional inefficiencies are not added needlessly. Several techniques can be used to improve DO-loop performance. In the following example, a constant is removed from the DO loop and calculated once instead of being recalculated each time the loop is executed.

EXAMPLE 4

Old Method	New Method
DO 10 I=1,1000	C=3.5/10.99+8.0
10 X(I)=(3.5/10.99+8.0)*Y(I)	DO 10 I=1,1000
	10 X(I)=C*Y(I)

Often a DO-loop with a fixed small number of iterations (≤ 10) can be replaced by a straight-line section of code.

EXAMPLE 5

Old Method	New Method
DO 10 I=1,6	A(1) = B(1)*X
10 A(I)=B(I)*X	A(2) = B(2)*X
	A(3) = B(3)*X
	A(4) = B(4)*X
	A(5) = B(5)*X
	A(6) = B(6)*X

It is advantageous for the programmer to unwind DO-loops manually to gain execution speed. The DO-loop closing code of the FORTRAN compiler is fairly lengthy, and in the following example the frequency of execution of such code is halved. Note, however, that the new method will require more memory to execute than the old method because extra instructions have been added. If these extra instructions force the loop not to be contained within the instruction

stack, the new method may not be more efficient than the old.

EXAMPLE 6

Old Method	New Method
DO 10 I=1,501	DO 10 I=1,500,2
A(I)=Y+B(I)	A(I)=Y+B(I)
10 B(I)=X+C(I)	B(I)=X+C(I)
	A(I+1)=Y+B(I+1)
	10 B(I+1)=X+C(I+1)
	A(501)=Y+B(501)
	B(501)=X+C(501)

Often expensive arithmetic operations such as exponentiations can be changed to multiplies in a DO-loop. In the following example, the exponentiation of X is changed to a multiply, and the multiply of J is changed to an addition.

EXAMPLE 7

Old Method	New Method
DO 10 I=1,100	XI = X
Y(I)=X**I+Z	JI = J
10 K(I)=J*I+L	DO 10 I=1,100
	Y(I)=XI+Z
	XI = XI*X
	K(I)=JI+L
	10 JI = JI+J

Again, the new method will require more memory to execute than the old.

D. Arithmetic Operations

Several programming techniques may be used to improve the efficiency of arithmetic operations, especially if one realizes that certain arithmetic operations are faster than others.

On the CDC 7600, integer addition and subtraction are faster than floating-point addition and subtraction, because the integer hardware units are faster than the corresponding floating-point units. Integer multiplication and division are slower than floating-point multiplication and division on the 7600, because there are no hardware units for integer multiplication or division. Integers must therefore be converted to floating-point numbers for multiplication or division, and then the results must be converted back to integers. Double-precision

operations are slow because they are not fully hardware operations; they should be avoided unless absolutely necessary. Elementary arithmetic operations and execution times are listed in Table I. When these arithmetic operations appear in more complicated FORTRAN expressions, the execution times given for single operations may not be valid.

Note that floating-point additions, subtractions, and multiplications, and integer multiplications require about the same unit of time. Integer additions and subtractions take slightly less than this unit of time. Single-precision divides take 2 to 3 units of time; double-precision multiplications, additions, and subtractions take about 3 units; and double-precision divides take about 8 units.

TABLE I

7600 ELEMENTARY FORTRAN ARITHMETIC OPERATIONS WITH EXECUTION TIMES

Operation	7600 Clock Periods
Integer Addition (I=M+N)	12
Integer Subtraction (I=M-N)	12
Floating-Point Addition (A=B+C)	16
Floating-Point Subtraction (A=B-C)	16
Integer Multiplication (I=M*N)	18
Integer Division (I=M/N)	39
Floating-Point Multiplication (A=B*C)	15
Floating-Point Division (A=B/C)	30
Double-Precision Addition (A=B+C)	48
Double-Precision Subtraction (A=B-C)	48
Double-Precision Multiplication (A=B*C)	49
Double-Precision Division (A=B/C)	145

Note that floating-point divides are quite slow. Often, they can be eliminated by using the following identities.

$$X = A/B + C/B \quad \equiv \quad X = (A+C)/B$$

$$X = (A/B) * (C/D) \quad \equiv \quad X = (A*C)/(B*D)$$

The use of parentheses can appreciably affect the time required to evaluate arithmetic expressions. Generally, parenthetical operations are initiated before

nonparenthetical operations, and those matched parentheses groups whose left parenthesis is farthest to the right are done first. For example $A*B*(C/D)$ is faster than $(A*B*C)/D$, because in the former case the divide is being done while A and B are fetched and multiplied. In the latter case, the divide is done last and the result is stored after a full 20-clock-cycle wait for the divide to complete.

Often one can combine FORTRAN logical and arithmetic operations to improve object code performance.

EXAMPLE 8
($M \geq 0$)

Old Method	New Method
I = MOD(M,2)	I = M.AND.1B
I = MOD(M,4)	I = M.AND.3B
I = MOD(M,8)	I = M.AND.7B

and so on, for powers of two.

In the above example, the new method executes twice as fast as the old for each expression.

It is better to use as few temporary variables as possible in arithmetic computations. Whenever a variable appears on the left-hand side of an assignment statement, it must be stored in memory. Variables on the right-hand side of an assignment statement almost always must be fetched from memory. Memory fetches require about 8 clock cycles on the 7600 before the fetched quantity is ready for use, and therefore they can be quite expensive. This is a very important programming technique which if used carefully can lead to substantial savings.

In the following example, the new method takes about 25% less time to execute than the old.

EXAMPLE 9

Old Method	New Method
C1 = A*B	X = A*B+C*(D+E)
C2 = D+E	
C3 = C*C2	
X = C1+C3	

E. Subroutines

The FORTRAN compiler will sometimes produce more efficient object code if short program sections are made into separate subroutines with short calling sequences. This is because the compiler will tie up B-registers so that computations, which can be done most efficiently by using B-registers, are not allowed to use them. Putting the program section into a subroutine forces the compiler to free the B-registers which become available for computation by the subroutines. This technique should be used with care, and it may be helpful only in exceptional cases. Subprogram calls are generally expensive.

Rather than using long subroutine calling sequences, one should pass the required arguments in COMMON blocks. If a subroutine requires more than six arguments, the compiler must generate the object code to set up a table containing the addresses of the subroutine arguments. This setup code is executed before each call to the subroutine. Whenever an argument (in excess of six) is used by a subroutine, one memory fetch is required to get the argument address and a second is required to get the value of the argument. Use of subroutine calling sequences with more than six arguments can, therefore, be very expensive if the subroutine is called often; if the 7th, 8th, ---, etc. arguments are used frequently within the subroutine; or if some combination of both these conditions is present.

The best procedure is to use subroutine arguments as flags to control program flow within the subroutine and to pass data to and from the subroutine via COMMON blocks. However, this procedure may damage the generality of the subprogram.

F. Bank Conflicts

On the CDC 7600, Small Core Memory (SCM) is organized into $65K_{10}$, 60-bit words, in 32 banks of 2048 words each. The banks are logically independent, and consecutive memory addresses go to different banks.

If FORTRAN arrays are dimensioned in certain ways, memory-bank conflicts are almost certain to occur during variable referencing and they will considerably impair object-code performance.

For example, if a two-dimensional array whose first dimension is a multiple of 32 is used in a loop so that the second dimension is singly incremented each time the loop is executed, the array values will be fetched from the same bank. Since the same bank can be accessed only once every 10 clock cycles, significant wait times between memory fetches may be introduced.

EXAMPLE 10

Old Method	New Method
DIMENSION A(32,100)	DIMENSION A(32,100)
DO 10 I=1,32	DO 10 J=1,100
DO 10 J=1,100	DO 10 I=1,32
10 A(I,J) = 0.	10 A(I,J) = 0.

In the above example for the old method with I fixed, all A(I,J) values are in the same bank; therefore, a bank conflict is created when memory is referenced. In the new method for any fixed J, all A(I,J) values are in different banks, so possible bank conflicts are eliminated. An alternative solution to the old method is to avoid multiples of 32 for the first dimension of Array A. It is always good practice to make the inner loop on the first index, because expensive calculations required to map two-dimensional arrays into one-dimensional arrays for memory reference are minimized. (See Sec. A)

G. Miscellaneous

1. The FORTRAN compiler on the CDC 7600 has an optimization feature that may be specified by using a control-card parameter. Use of this feature may save up to 10% in execution time for some codes.

2. It is good FORTRAN programming practice to eliminate statement numbers wherever possible. Statement numbers cause delays because object code is forced to start on word boundaries (a force upper

occurs) whenever statement numbers are encountered. This is because the FORTRAN compiler expects to have to reference the code tagged by the statement number. In addition, the FORTRAN compiler can't remember anything across statement labels, so variables fetched just before a statement label may be refetched just after the label. The utility program TIDY⁸ may be used to eliminate unreferenced statement numbers.

3. Several FORTRAN support routines, such as EXP(X), SIN(X), COS(X), and SQRT(X), are available in versions that are considerably faster but less accurate than the standard system library versions.

4. Variables in subprograms can be repetitively initialized more efficiently by using DATA statements such as

```
DATA IA(1),IA(2)/5,8/
```

than by using assignment statements such as

```
IA(1) = 5
IA(2) = 8 .
```

This is because the DATA statement is non-executable, so variables listed in DATA statements are initialized by the compiler during the compiling process. Assignment statements perform variable initialization during program execution.

5. Use of multiple-replacement statements to declare equality of variables can be expensive. Multiple-replacement statements should be replaced by several single-replacement statements.

Example 11

Old Method	New Method
A = B = C = D = 0.	A = 0.
	B = 0.
	C = 0.
	D = 0.

In the above example, the new method executes twice as fast as the old.

6. For efficient object-code performance, do not use the \$ to separate FORTRAN

source statements. When multiple FORTRAN statements separated by \$ are encountered, the compiler begins a one-to-one translation of source statements to object code without regard to object-code efficiency. The utility program TIDY will unpack FORTRAN programs to a single source statement per line.

III. PROGRAMMING AND OPTIMIZATION TOOLS

Several tools developed by the LASL Computing Applications and Services Group are of interest to the FORTRAN programmer. One such tool is STAT⁸ which uses a hardware-interrupt feature of the CDC 7600 to allow the user to monitor the Program Address Register (P-register) during program execution. From the STAT output, a user can determine the most heavily used areas of his code which can sometimes be recoded with large gains in object-code efficiency.

A second program is REGREF⁸ which is useful for examining object code. Delays resulting from conflicts of registers, functional units, and the instruction stack are given.

Finally, a timing routine TFN⁹ which gives the execution time of a given function or subroutine is available. The time returned by TFN contains all delays including bank conflicts and branch-instruction delays and is accurate to within a 7600 clock period. The time to transfer to and from the routine is included, but the time to set up the arguments for the function or subroutine being timed is not included.

IV. CONCLUSION

Operations and techniques that can be used to improve object-code performance

have been described. In lieu of a FORTRAN compiler with optimizing capability, the programmer must perform these operations manually at the source-code level.

Timings of examples have been included as guidelines for judging the effects of using the operations and techniques. It is, however, the programmer's responsibility to determine whether the optimization techniques are of real benefit, based on the execution frequency of the actual code in question.

REFERENCES

1. Edward S. Cowrey and C. W. Medlock, "Object Code Optimization," CACM XII, 13-22 (1969).
2. Vincent A. Busam and Donald E. England, "Optimization of Expressions in FORTRAN," CACM XII, 666-674 (1969).
3. Sam F. Mendicino and Richard G. Zwakenberg, "A FORTRAN Code Optimizer for the CDC 6600," Lawrence Radiation Laboratory report, UCRL-14162 (April 1965).
4. Sam F. Mendicino, "The LRLTRAN Compiler," CACM XI, 747-755 (1968).
5. Dennis J. Frailey, "A Study of Code Optimization Using a General Purpose Optimizer," Ph.D. Thesis, Department of Computer Science, Purdue University (1971).
6. Richard G. Zwakenberg, "CDC 6600/7600 Optimization," ACM SIGPLAN Notices, Vol. V., 86-100 (July, 1970).
7. Control Data 7600 Computer System Reference Manual, Control Data Corp., Arden Hills, Minn. (1970).
8. "Program Library, Abstracts and Usages," Programmer's Information Manual, Vol. 2, Los Alamos Scientific Laboratory (1972).
9. L. Wayne Fullerton, Los Alamos Scientific Laboratory, unpublished report.