Discussed is a technique for investigating the efficiency of compiled programs. Based on research that uses FORTRAN as a test subject, the method is more widely applicable. Time analyses show programmers points at which efficiencies may be increased. Also discussed are uses of the technique for comparing the efficiencies of compilers and languages, and for making performance/cost analyses. Presented are validation data for the method under several sets of conditions.

A method for the time analysis of programs

by S. L. de Freitas and P. J. Lavelle

Program efficiency depends on several factors, including the selection of a language, algorithm, structure, and programming technique suitable for each application. Furthermore, efficiency is always related to a specific hardware-software environment. The great variations among operating systems, compilers, and computer models preclude the use of a fixed set of rules for achieving program optimization.

Optimizing compilers are used to improve the efficiency and to reduce the size of the object code, at the expense of an increase in the time and storage needed for the compilation. Though the advantages of particular compilers are well known, they also contain some undesirable characteristics. Optimization is often done in a standard way for all programs, with no provision for selecting only certain parts of a program for optimization. It is difficult for the programmer to determine the nature and extent of the optimization performed, in order to improve a program at the source code level. Certain unnecessary or inefficient operations (such as data conversions caused by bad source code) are not recognized by some compilers.

This paper describes research on this problem by the authors that has led them to write a program that analyzes the output of the LIST option of the FORTRAN compiler. The experimental program inserts information based on this analysis into the compiled listing

of the program, as an aid to the programmer in determining the cost and timing of each FORTRAN statement, thereby seeking to improve program efficiency. The major advantage of our method is that it is based on the characteristics of the machine being used, and on the manner in which the FORTRAN compiler generates the object code. The result of these studies is to incorporate the nature of the data, the type of operations being performed, and all the characteristics described in the program.

A FORTRAN compiler and the IBM System/360 Model 65 computer have been used as subjects of this study. Other compilers and computers are similarly applicable subjects, as is discussed later in this paper.

the problem

Nowadays the cost-conscious computer user knows that the use of Assembler-language-level programming should be avoided as much as possible. In high-level language techniques—such as structured programming and peer reviewing—better algorithm choices are more fruitful in a great many cases than down-to-the-microsecond programming. By using improved programming techniques, however, the programmer loses contact with the machine architecture and lacks motivation for chasing the lost microsecond. General guidelines[1,2] exist, but they are simply what they are—general.

Only an in-depth analysis of the compiler-generated object code can furnish a secure set of optimization rules. Unluckily the number of possible instructions, forms, and cases is too large to cope with even for such a simple language as FORTRAN. Thus we have found it simpler to show the programmer the cost of his programming rather than to rely on incomplete optimization rules.

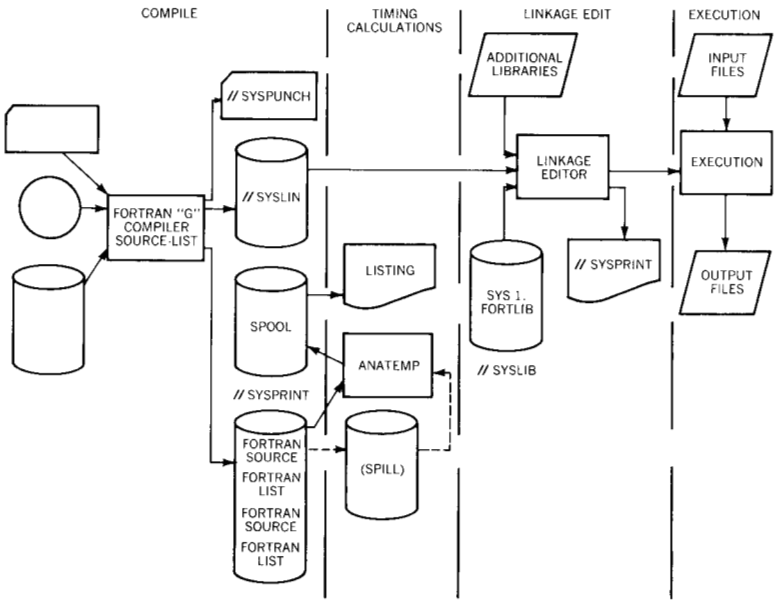
The techniques needed to introduce our compiler efficiency assessment are relatively simple under the following conditions. The compiler is modest in its objectives and does not optimize too strongly at the interstatement level. The machine architecture and instruction timings are simple. (Multiple instructions per word, high-speed buffers, cache and virtual storage, and speed-up options that hardware designers put into machines are often the nightmare of compiler designers.) Finally, the compiler source code is available and/or that development is included from the compiler design stage.

Another motivation for our research has been the difficulty of obtaining, understanding, and reliably modifying a whole real compiler in a short time. If our solution finds widespread use, compiler modification should be considered.

ANATEMP: a time analysis program

Our time analysis program, called ANATEMP, has been tested and is currently running on an IBM System/360 Model 65,[3] operating

Figure 1 General flow of a FORTRAN program using ANATEMP



under control of OS-MVT Release 21.8. ANATEMP has been applied to programs compiled by the IBM FORTRAN IV G Level 21 compiler. Figure 1 shows the flow of a FORTRAN program as it is being compiled, analyzed by ANATEMP, link-edited, and executed.

The program is compiled with the SOURCE and LIST options. The SYSPRINT output of the compilation is directed to a temporary data set on disk. Batch compilations are allowed, with each unit being analyzed separately. Note that in the SYSPRINT data set, the output from the SOURCE option for each compilation precedes the output from the LIST option for that compilation. ANATEMP then performs the following steps: (1) The source listing is read from disk into main storage. If the source contains more than eight hundred cards, the excess is saved in a spill data set on disk. When all the source data have been read and the LIST data reached, the spill data set—if used—is rewound. (2) At this point, the SOURCE data and the LIST data are analyzed together. Instructions that correspond to each executable FORTRAN statement are determined, timing calculations are performed, and the results are printed. (3) If there are additional data to be analyzed (batch compilations), return to step 1. When all data have been analyzed, ANATEMP terminates. The link-edit and execution steps are then executed normally.

analysis fields

The information provided by ANATEMP is inserted into the FORTRAN source listing of the compiler in blank fields that are not otherwise used. It is assumed that a 133-character printer is used.

An example of the information provided is shown in Figure 2. An Appendix is provided to illustrate the use of ANATEMP and to provide guidelines for source code optimizations. In Figure 2, for a main program LINK is the link time of the program with OS; for subprograms, it is the prologue time. The following is a description of the numbered fields in Figure 2.

1. *Branch flag*. B is present in this column if the statement may cause a branch, unless the branch is to an external reference.
2. *External reference*. All statements that make implicit or explicit external references are flagged with an E.
3. *Loop flag*. Statements that contain an implicit DO LOOP, or end an explicit DO LOOP contain an L in this field.
4. *Number of external references*. This field contains the number of external references made by each statement; a blank signifies no external references.
5. *Offset*. The offset field contains the location relative to the origin of the program at which each statement begins. This information is particularly useful to the programmer because it allows a quick determination of the number of bytes used to generate each statement. This is an excellent tool for debugging, since the address in the PSW at abend minus the address of the entry point often points to the statement where the abend has occurred, or, in virtual systems, where the program is paged.
6. *Minimum execution time*. When present, this field represents the minimum time necessary to execute the corresponding statement. This information is useful in analyzing compound statements (e.g., IF (A.GT.B) C = *D*), where different execution times are possible.
7. *Execution time*. This is the total time expected for execution of all instructions that compose the statement, and includes the time for the execution of in-line subprograms, if any. If the statement makes an external reference (indicated by a flag in field 2), this value excludes the execution time of the external subprogram.
8. *One iteration time*. This value, enclosed in parentheses, is the time needed to perform one iteration of a DO LOOP as defined in a DO command.
9. *DO LOOP housekeeping*. This value represents the time required to initialize a DO LOOP. It should be added to the value of field 8 to find the value of the first iteration. ANATEMP uses a stack array for DO LOOP timing calculations, since nested DO LOOPS work on a Last-In-First-Out (LIFO) basis.

The RESUME is a list of all instructions used by the program together with the number of occurrences of each and is printed at the end of the listing. The number of Defined Constants (DC) generated by the compiler is presented, and the sum of times for all

Figure 2 Demonstration of ANATEMP results

```
FORTRAN IV G LEVEL 21          MAIN          DATE = 77266     17/12/21    PAGE 0001
      1234                                                      5       6       7       8       9
                                                              LINK =  24.92

            C       INPUT/OUTPUT STATEMENTS
            C
0001                DIMENSION AAA(10)
0002          10    FORMAT (20F10.2)
0003 EL  3          WRITE(3,10) (AAA(I),I = 1,10)                 002DA8          15.33
0004 E   3          WRITE(3,10) AAA                               002DEC           4.80
            C
            C       DATA TYPES AND CONVERSIONS
            C
0005          50    I = J                                         002E0C           2.13
0006                W = K                                         002E14          13.06
0007          20    INTE4A = REAL                                 002E38          13.58
0008          30    REAL4 = INTE4B                                002E5C          13.06
0009                REAL*8 REAL8
0010          40    REAL8 = INTE4A                                002E80          13.06
0011                K1 = Q + J + E + KKK                          002EA4          40.23
0012                K222 = IFIX(Q + E) + J + KKK                  002F04          18.26
            C
            C       EXPRESSION EVALUATION
            C
0013                DIMENSION VETOR(5C), V2(50,50)
0014                DO 70 I = 1,50                                002F32           4.53
0015                VETOR(I) = 0.0                                002F42           3.33
0016                DO 70 J = 1,50                                002F4E           2.78
0017                VETOR(I) = VETOR(I) + A * B / C * V2(J,I)     002F58          20.66
0018 L        70    CONTINUE                                      002F70          17.20 (  38.79) 1.8
                                                                                        (  44.90) 3.6
0019                RESUL = A * B / C                             002FAC          13.83
0020                DO 80 I = 1,50                                002FBC           5.73
0021                VETOR(I) = 0.0                                002FDO           3.33
0022                DO 80 J = 1,50                                002FDC           2.78
0023                VETOR(I) = VETOR(I) + RESUL * V2(J,I)         002FE6           8.96
0024 L        80    CONTINUE                                      002FF6          17.20 (  27.09) 1.8
                                                                                        (  33.20) 4.8
0025                M = J + (365 - 100) / 5                       003032          18.93
0026                M1 = J + 53                                   00304E           3.53
            C
            C       SUBSCRIPT
            C
0027                DIMENSION SUB(100), SMAT(10,10)
0028                FASTER = SUB(M + 2)                           00305A           4.18
0029                SLOW = SMAT(M + 3,N)                          00306A          12.88
0030                TOTAL = 0.0                                   00308A           2.13
0031                DO 90 I = 1,100                               003092           3.33
0032                TOTAL = TOTAL + SUB(I)                        00309E           4.56
0033 L        90    CONTINUE                                      0030AA           7.90 (  13.39) 2.4
0034                TOTAL = 0.0                                   0030C6           3.33
0035                DO 110 I = 1,10                               0030D2           3.33
0036                DO 110 J = 1,10                               0030DE           3.98
0037                TOTAL = TOTAL + SMAT(I,J)                     0030EC           4.56
0038 L       110    CONTINUE                                      0030F8          15.80 (  21.29) 3.0
                                                                                        (  25.27) 2.4
            C
            C       LOGICAL IF & BRANCHING
            C
0039 B              IF(A.LT.B.OR.C.GT.F.OR.H.GE.T) GO TO 100      003130          28.09
0040 B              IF(A.LT.B) GO TO 100                          00317C           5.38
0041 B              IF(C.GT.F) GO TO 100                          00318A           5.38
0042 B              IF(H.GE.T) GO TO 100                          003198           5.38
0043 B              IF(A.GT.B) GO TO 100                          0031A6           5.38
0044 B              IF(A - B) 100, 100, 200                       0031B4           6.68
```

```
0045 B      100 GO TO (20,30,40,50), ITEM                    0031C4      10.90
0046 B      101 GO TO 200                                    0031F8       2.20
0047        200 ASSIGN 50 TO NUMBER                          0031FE       2.13
0048 B          GO TO NUMBER,(30,50,40)                      003206       2.20
0049        201 CONTINUE
          C
          C     MACHINE DEPENDENT OPTIMIZATION
          C
0050            INTEGER*2 INT2,INTX,INTY,INTZ
0051            INT2 = INTX / INTY * INTZ                    00320C      25.11
0052            INT4 = IN4X / IN4Y * IN4Z                    00322E      19.73
0053            K = MN * 2                                   003242       6.93
0054            K11 = MN1 + MN1                              00324E       3.53
0055            DIV = DIVI / 4                               00325A       9.43
0056            DV = DVI * 0.25                              003266       6.53
          C
          C     SUBPROGRAMS
          C
0057 E  1       CALL MATMPY(X,5,40,YYY,ROOT)                 003272       3.15
0058            INL = MOD(I,J)                               00327C      21.78
0059 E  1       STOP                                         003296       2.40
0060            END
```


SUBPROGRAMS CALLED

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
IBCOM=	F8	MATMPY	FC						

SCALAR MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
REAL8	120	I	128	J	12C	W	130	K	134
INTE4A	138	REAL	13C	REAL4	140	INTE4B	144	K1	148
Q	14C	E	150	KKK	154	K222	158	A	15C
B	160	C	164	RESUL	168	M	16C	M1	170
FASTER	174	SLOW	178	N	17C	TOTAL	180	F	184
H	188	T	18C	ITEM	190	NUMBER	194	INT4	198
IN4X	19C	IN4Y	1A0	IN4Z	1A4	MN	1A8	K11	1AC
MN1	1B0	DIV	1B4	DIVI	1B8	DV	1BC	DVI	1C0
X	1C4	YYY	1C8	ROOT	1CC	INL	1D0	INT2	1D4
INTX	1D6	INTY	1D8	INTZ	1DA				

ARRAY MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
AAA	1DC	VETOR	204	V2	2CC	SUB	29DC	SMAT	2B6C

FORMAT STATEMENT MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
10	2CFC								

```
*OPTIONS IN EFFECT* NOID,EBCDIC,SOURCE,LIST,NODECK,LOAD,MAP
*OPTIONS IN EFFECT* NAME = MAIN, LINECNT = 60
*STATISTICS* SOURCE STATEMENTS = 60,PROGRAM SIZE = 12964
*STATISTICS* NO DIAGNOSTICS GENERATED
```

```
FORTRAN IV G LEVEL 21          MATMPY          DATE = 77266     17/12/21     PAGE 0001

      1234                                                  5       6       7       8       9
0001              SUBROUTINE MATMPY(X,J,I,YYY,ROOT)
                                                        LINK =   76.07
0002 B            RETURN                                00016C          2.85
0003              END
```

```
FORTRAN IV G LEVEL 21          MATMPY          DATE = 77266     17/12/21     PAGE 0002
```

SCALAR MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
X	90	J	94	I	98	YYY	9C	ROOT	A0

```
*OPTIONS IN EFFECT* NOID,EBCDIC,SOURCE,LIST,NODECK,LOAD,MAP
*OPTIONS IN EFFECT* NAME = MATMPY, LINECNT = 60
*STATISTICS* SOURCE STATEMENTS = 3, PROGRAM SIZE = 372
*STATISTICS* NO DIAGNOSTICS GENERATED
*STATISTICS* NO DIAGNOSTICS THIS STEP 2
```

RESUME OF INSTRUCTIONS USED IN OBJECT CODE GENERATED BY THE COMPILER

INSTRUCTION	NUMBER TIMES USED
A	15
AD	5
AE	7
AER	1
AR	1
AW	3
BAL	8
BALR	14
BC	15
BCR	16
BXLE	8
CE	7
CR	1
D	4
DC	24
DE	3
END	2
L	108
LA	19
LCDR	5
LCR	4
LD	5
LE	23
LER	2
LH	2
LM	4
LPR	5
LR	10
LTDR	3
LTER	1
LTR	7
M	4
ME	5
MH	1
MVC	10
MVI	2

OR	2
S	2
SDR	3
SE	1
SLA	3
SLL	1
SR	4
SRDA	4
ST	27
STD	4
STE	15
STH	1
STM	4
TOTAL = 47 INSTRUCTIONS	TOTAL TIME = 598.41

statements is also given. This, of course, does not represent the expected execution time, but serves as a guideline for comparisons.

program simplification

We coded ANATEMP in FORTRAN to allow FORTRAN programmers to make modifications for various computer hardware. Two assembler subroutines were used. The first, MOVECO1, is a general-purpose subroutine to move characters from one storage location to another. The second is an interface subroutine to ADCON#, which is the FORTRAN module that is used to make conversions. Input/output operations on disk are performed sequentially, thus allowing better efficiency through automatic blocking and deblocking.

No restrictions are imposed on the programmer in using ANATEMP. When a cataloged procedure has been built (e.g., FORTTCLG, with T for timing instead of G) the programmer executes the procedure as a standard IBM procedure just as FORTGCLG would be executed.

Observations on ANATEMP

cost estimates

We evaluated the cost of using ANATEMP by comparing the CPU and execution time spent by the FORTRAN compiler and by ANATEMP for several programs of different sizes. All values were obtained from the System Management Facilities (SMF)[4] records generated by the OS operating system. The sum of these runs indicated the following results:

- Total number of input cards: 3192.
- FORTRAN source statements: 1948.
- CPU time used by the FORTRAN compiler: 95.75 seconds.
- Execution time for the FORTRAN compiler: 4.5 minutes.
- CPU time used by ANATEMP: 60.63 seconds.
- Execution time for ANATEMP: 3.5 minutes.

Table 1 Time comparisons for a program running with other programs

	Parameter	*Tests*				
		1	*2*	*3*	*4*	*5*
A	Time furnished by the system (seconds)	9.85	11.92	9.95	10.73	111.48
B	Time calculated by ANATEMP (seconds)	10.21	10.21	10.21	10.21	102.10
C	Percentage difference $\frac{A - B}{B}$	−3.52	16.74	−2.54	5.09	9.29

Thus we conclude from the cases studied that ANATEMP uses approximately 65 percent as much CPU time, and 75 percent as much execution time as the FORTRAN compiler.

accuracy of time estimates

The instruction times used are calculated from IBM formulas,[3] under certain constraints, limitations, and assumptions that arise from three different sources. With respect to compiler characteristics, certain instructions in the System/360 Model 65 set are not used in object code generated by the FORTRAN G compiler. Therefore, timings for these instructions need not be included in the program. In addition, simplifications are possible for timing calculations of the following other instructions. The Move instruction (MVC) is the only storage-to-storage (SS) instruction that is generated by the compiler. Since this instruction is used only in a standard way to perform linkage between subprograms, a fixed time can be used for the calculations. The Store Multiple (STM) and Load Multiple (LM) instruction timings depend on boundary alignment. Since they are also used only for linkage purposes by the compiler, and are aligned on a double word boundary, fixed times can be used.

Instruction times can also be affected by hardware-dependent occurrences that cannot be predicted and are not accounted for in the timing formulas.[3]

As for execution time dependence, in computing instruction times, it is assumed for all branch instructions that the branch is taken, unless the mask is zero or the second operand in a Register-to-Register (RR) instruction is zero. In particular, it is assumed that the branch is always taken for Branch on Index Low or Equal (BXLE) instructions.

After compilation, the program ANATEMP calculates the execution time for each executable FORTRAN statement, as well as oth-

Table 2 Time comparisons for a program running with no other active programs

	Parameter	Tests		
		1	*2*	*3*
A	Time furnished by the system (seconds)	9.88	9.68	97.80
B	Time calculated by ANATEMP (seconds)	10.21	10.21	102.10
C	Percentage difference $\frac{A - B}{B}$	−3.23	−5.18	−4.11

er information, using the basic times and associated formulas for each machine instruction. Several tests were made to compare the times furnished by the system and by ANATEMP. The reader should take into account the timing considerations for System/360 Model 65 as described in References 4 and 5.

In the first set of tests, summarized in Table 1, a simple program that consisted of a DO LOOP using fixed-point operations only was run concurrently with other programs. In the second set of tests, summarized in Table 2, the same program was run with no other active programs. Using the OS macroinstructions TTIMER and STIMER, the authors have developed the subroutine TIME01 to measure the CPU time elapsed between two points of a program. This subroutine was used in a third set of tests, which included fixed and floating-point operations. Table 3 shows the results of the third set of tests. Results with multiprogramming are given by tests1 and 2.

For the System/360 Model 65, as shown in Tables 1–3, the times predicted by ANATEMP represent a good approximation to the real times. We expect similar results for all computers with comparable architectures. On models in which a high-speed buffer and/or Dynamic Address Translation (DAT) are installed, the accuracy of ANATEMP may not be as good as the present results. The information that would be provided, however, would be useful since the relative values of times calculated for different commands are correct.

Concluding remarks

The research discussed in this paper can be seen as serving a number of different needs, some of which are the following:

Table 3 Time comparisons for a program running with and without multiprogramming

	Parameter					
Tests	*A* *Time furnished by the system* (seconds)	*B* *Time calculated by ANATEMP* (seconds)	*C* *Time furnished by TIME01* (seconds)	*Percentage difference* $\frac{A-B}{B}$	*Percentage difference* $\frac{C-B}{B}$	*Percentage difference* $\frac{A-C}{C}$
1	42.23	38.72	41.83	9.05	8.03	0.09
2	110.10	108.25	109.71	1.71	1.34	0.03
3	43.95	38.72	43.64	13.5	12.7	0.07
4	111.62	108.25	110.72	3.1	2.3	0.08

- It provides a tool for day-to-day programming.
- It provides a feasibility study of the value of including timing analysis facilities in production compilers.
- For a given language, timing analysis gives a comparison of the efficiencies of various compilers.
- For a given machine, timing analysis gives a comparison of the efficiencies of different languages.
- For a given problem, timing analysis can be a useful performance/cost analysis tool for various sets of languages, compilers, and machines.
- Timing analysis gives feedback for machine and compiler design.

We think that a lot of work is still to be done in this avenue of applied research. The method described in this paper can also be applied to programs written in other high-level languages, such as COBOL and PL/I. The authors believe that the type of analysis described here can also be made available as a compiler option, thus making timing analyses faster and cheaper to obtain. With additional effort, such an analysis could also detect and flag improper boundary alignment, which often reduces efficiency in System/370.

ACKNOWLEDGMENTS

The authors wish to acknowledge Messrs. D. C. Crane (IPEC-IBM) and M. P. Gonçalves (IBM-Brasil), who have contributed to the publication of this paper.

CITED REFERENCES

1. C. Larson, "The efficient use of FORTRAN," *Datamation* **17**, No. 8, 24–31 (August 1971).
2. B. Kernighan and P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill Book Company, Inc., New York, NY (1974).

3. *System/360 Model 65, Functional Characteristics*, GA22-6884-4, IBM Corporation, Data Processing Division, White Plains, New York 10604.
4. *OS SMF*, GC28-6712, IBM Corporation, Data Processing Division, White Plains, New York 10604.
5. *OS Release 21, Supervisor Services and Macro Instructions*, GC 28-6646-7, IBM Corporation, Data Processing Division, White Plains, New York 10604.

Appendix: Guidelines for source code optimizations

With results produced by ANATEMP, the FORTRAN statements that use significant amounts of time can be determined by inspection. In this way, one might spot several areas where optimization is necessary. Listed here are some examples of optimization at source level for FORTRAN programs that have been specifically compiled under FORTRAN G. These examples were possible to discover and correct using ANATEMP, and were also included by Larson[1] in describing coding techniques for FORTRAN program efficiency.

Figure 2 is used as a reference for the presentation of the following examples:

- *Input/Output statements*. Statement 0003 shows a loop flag (L), indicating that repeated calling and return of an external reference (E) is being requested. The solution is to avoid implied DO-loops in I/O statements and to attempt to minimize the number of items in I/O lists. Thus, for efficiency, statement 0004 is preferable.
- *Data types and conversions*. Perhaps the most common source of performance degradation is unnecessary internal data conversion during execution. Statements 0005 to 0012 show examples and cures.
- *Expression evaluation*. Optimization of redundant expressions, especially in loops, at source level is always recommended. In statement 0017, the expression A*B/C is always computed, although it is in fact constant. The solution is simple, as shown in statement 0023, where the redundant computation is moved out of the loop. The difference in instruction time between statement 0025 and statement 0026 shows clearly the importance of pre-evaluation of numerical expressions because these expressions are evaluated at execution time.
- *Subscripts*. As subscript computations at object time are expensive, the rule is only to use multidimensional arrays if they are essential to an algorithm. Although this may make the source coding a little more difficult to follow, the resulting improvement in performance more than compensates for the effort. Refer to ANATEMP results in statements 0027 through 0038.

Table 3 Time comparisons for a program running with and without multiprogramming

	Parameter					
Tests	*A* *Time furnished by the system* (seconds)	*B* *Time calculated by ANATEMP* (seconds)	*C* *Time furnished by TIME01* (seconds)	*Percentage difference* $\frac{A-B}{B}$	*Percentage difference* $\frac{C-B}{B}$	*Percentage difference* $\frac{A-C}{C}$
1	42.23	38.72	41.83	9.05	8.03	0.09
2	110.10	108.25	109.71	1.71	1.34	0.03
3	43.95	38.72	43.64	13.5	12.7	0.07
4	111.62	108.25	110.72	3.1	2.3	0.08

- It provides a tool for day-to-day programming.
- It provides a feasibility study of the value of including timing analysis facilities in production compilers.
- For a given language, timing analysis gives a comparison of the efficiencies of various compilers.
- For a given machine, timing analysis gives a comparison of the efficiencies of different languages.
- For a given problem, timing analysis can be a useful performance/cost analysis tool for various sets of languages, compilers, and machines.
- Timing analysis gives feedback for machine and compiler design.

We think that a lot of work is still to be done in this avenue of applied research. The method described in this paper can also be applied to programs written in other high-level languages, such as COBOL and PL/I. The authors believe that the type of analysis described here can also be made available as a compiler option, thus making timing analyses faster and cheaper to obtain. With additional effort, such an analysis could also detect and flag improper boundary alignment, which often reduces efficiency in System/370.

ACKNOWLEDGMENTS

The authors wish to acknowledge Messrs. D. C. Crane (IPEC-IBM) and M. P. Gonçalves (IBM-Brasil), who have contributed to the publication of this paper.

CITED REFERENCES

1. C. Larson, "The efficient use of FORTRAN," *Datamation* **17**, No. 8, 24–31 (August 1971).
2. B. Kernighan and P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill Book Company, Inc., New York, NY (1974).