An Empirical Study of FORTRAN Programs†

DONALD E. KNUTH

Computer Science Department, Stanford University, Stanford, California 94305

SUMMARY

A sample of programs, written in FORTRAN by a wide variety of people for a wide variety of applications, was chosen 'at random' in an attempt to discover quantitatively 'what programmers really do'. Statistical results of this survey are presented here, together with some of their apparent implications for future work in compiler design. The principal conclusion which may be drawn is the importance of a program 'profile', namely a table of frequency counts which record how often each statement is performed in a typical run; there are strong indications that profile-keeping should become a standard practice in all computer systems, for casual users as well as system programmers. This paper is the report of a three month study undertaken by the author and about a dozen students and representatives of the software industry during the summer of 1970. It is hoped that a reader who studies this report will obtain a fairly clear conception of how FORTRAN is being used, and what compilers can do about it.

KEY WORDS   FORTRAN   Optimization   Efficiency   Compiler

1. INTRODUCTION

Designers of compilers and instructors of computer science usually have comparatively little information about the way in which programming languages are actually used by typical programmers. We think we know what programmers generally do but our notions are rarely based on a representative sample of the programs which are actually being run on computers. Since compiler writers must prepare a system capable of translating a language in all its generality, it is easy to fall into the trap of assuming that complicated constructions are the norm when in fact they are infrequently used. There has been a long history of optimizing the wrong things, using elaborate mechanisms to produce beautiful code in cases that hardly ever arise in practice, while doing nothing about certain frequently occurring situations. For example, the present author once found great significance in the fact that a certain complicated method was able to translate the statement

$$C[I \times N + J] := ((A + X) \times Y) + 2 \cdot 768 + ((L - M) \times (-K))/Z$$

into only nineteen machine instructions compared to the twenty-one instructions obtained by a previously published method due to Galler and co-workers (see Knuth[1]). The fact that arithmetic expressions usually have an average length of only two operands, in practice, would have been a great shock to the author at that time!

There has been widespread realization that more data about language use is needed; we cannot really compare two different compiler algorithms until we understand the input data

---

they deal with. Of course, the great difficulty is that there is no such thing as a 'typical programmer'; there is a tremendous variation among programs written by different people with different backgrounds and sympathies and indeed there is considerable variation even in different programs written by the same person. Therefore we cannot trust any measurements to be very accurate, although we can measure the degree of variation in an attempt to determine how significant it is. Not all properties of programs can be reduced to simple statistics; it is necessary to study selected programs in detail in order to appreciate their characteristics more clearly. For a survey of early work on performance measurement and evaluation, see Calingaert[2] and Cerf.[3]

During the summer of 1970, the author worked together with several other people in order to explore the nature of actual programs and the corresponding implications both for software design and for computer science education. Our results are by no means a definitive analysis of programming behaviour; our goal was to explore the various possibilities, as a group, in order to set the stage for subsequent individual research rather than to go off in all directions at once. Each week the entire group had an eight-hour meeting in order to discuss what had been learned during the previous week, hoping that by combining our differing points of view we might arrive at something reasonably close to Truth.

A first idea for obtaining 'typical' programs was to go to Stanford's Computation Center and rummage in the waste-baskets and the recycling bins. This gave results but showed immediately what should have been obvious: waste-baskets usually receive undebugged programs. Furthermore, it seems likely that compilers usually are confronted with undebugged programs, too; so it was necessary for us to choose whether we wanted to study the distributions of syntax errors, etc. or to concentrate on working programs. Some excellent analyses of common errors have already been made (see Freeman, Moulton and Muller)[4, 5] and one of our main goals was to study the effects of various types of optimization; so we decided to restrict ourselves to programs which actually run to completion.

The waste-basket method turned up some interesting programs but it was not really satisfactory. If we wanted to automate the process, extensive typing from the listings would have been necessary; so we tried another tack. Our next method of obtaining programs was to post a man by the card reader at various times; he would ask for permission to copy decks on to a special file. Fifteen programs, totalling about 5000 cards, were obtained in this way; but the job was very time-consuming since it was necessary to explain the objectives of our project each time and to ask embarrassing questions about the status of people's programs.

The next approach was to probe randomly among the semi-protected files stored on disks looking for source text; this was successful, resulting in thirty-three programs, totalling about 20,000 cards. We added nine programs from the CSD subroutine library and three programs from the 'Scientific Subroutine Package', and some production programs from the Stanford Linear Accelerator Center. A few classical benchmark programs (nuclear codes, weather codes and aerospace calculations) were also contributed by IBM representatives and to top things off we threw in some programs of personal interest to members of the group.

This procedure gave us a quite varied collection of programs: some large, some small; some sophisticated, some crude; some important, some trivial; some for production, some for play; some numerical, some combinatorial.

It is well known that different programming languages evolve different styles of programming, so our study was necessarily language-dependent. For example, one would expect that expressions in APL programs tend to be longer than in FORTRAN programs. But virtually all of the programs obtained by our sampling procedure were written in

FORTRAN (this was the first surprise of the summer), so our main efforts were directed toward the study of FORTRAN programs.†

Was this sample representative? Perhaps the users of Stanford's computers are more sophisticated than the general programmers to be found elsewhere; after all we have such a splendid Computer Science Department! But it is doubtful whether our Department had any effect on these programs, because for one thing we do not teach FORTRAN; it was distressing to see what little impact our courses seem to be having, since virtually all of the programs we saw were apparently written by people who had learned programming elsewhere. Furthermore, the general style of programming that we found showed very little evidence of 'sophistication'; if it was better than average, the average is too horrible to contemplate! (This remark is not intended as an insult to Stanford's programmers; after all we were invading their privacy, and they would probably have written the programs differently if they had known the code was to be scrutinized by self-appointed experts like ourselves. Our purposes were purely scientific in an attempt to find out how things are, without moralizing or judging people's competence. The point is that the Stanford sample seems to be reasonably typical of what might be found elsewhere.) Another reason for believing that our sample was reasonably good is that the programs varied from text-editing and discrete calculations to number-crunching; they were by no means from a homogeneous class of applications. On the other hand, we do have some definite evidence of differences between the Stanford sample and another sample of over 400 programs written at Lockheed (see section 2 of this paper).

The programs obtained by this sampling procedure were analysed in various ways. First we performed a *static* analysis, simply counting the number of occurrences of easily recognizable syntactic constructions. Statistics of this kind are relevant to the speed of compilation. The results of this static analysis are presented in section 2. Secondly, we selected about 25 programs at random and subjected them to a *dynamic* analysis, taking into account the frequency with which each construction actually occurs during one run of the program; statistics of this kind are presented in section 3. We also considered the 'inner loops' of 17 programs, translating them by hand into machine language using various styles of optimization in an attempt to weigh the utility of various local and global optimization strategies; results of this study are presented in section 4. Section 5 of this paper summarizes the principal conclusions we reached, and lists several areas which appear to be promising for future study.

2. STATIC STATISTICS

We examined a large number of FORTRAN programs to see how frequently certain constructions are used in practice. Over 250,000 cards (representing 440 programs) were analysed by Mr. Maybury at the computer centre of Lockheed Missiles and Space Corporation in Sunnyvale.

Table I shows the distribution of statement types. A 'typical Lockheed program' consists of 120 comment cards, plus 178 assignment statements, 63·5 IF's, 56 GO TO's, 34 CALL's,

---

† By contacting known users of ALGOL, it was possible to collect a fairly representative sample of ALGOL-W programs as well. The analysis of these programs is still incomplete; preliminary indications are that the increased flexibility of data types in ALGOL-W makes for much more variety in the nature of inner loops than was observed in FORTRAN, and that the improved control structures make GO TO's and labels considerably less frequent. A comprehensive analysis of ALGOL 60 programs has recently been completed by B. Wichmann.[6]

We analysed one PL/I program by hand. COBOL is not used at Stanford's Computation Center, and we have no idea what typical COBOL programs are like.

21 CONTINUE's, 18 WRITE's, 18 FORMAT's, 17 DO's, 72 miscellaneous other statements and 31 continuation cards (mostly involving COMMON or DATA). Essentially the same over-all distribution of statement types was obtained when individual groups of

Table I. Distribution of statement types

| | Lockheed | | Stanford | |
	Number	Per cent†	Number	Per cent†
Assignment	78,435	41·0	4,869	51·0
IF	27,967‡	14·5‡	816‡	8·5‡
GO TO	24,942	13·0	777	8·0
CALL	15,125	8·0	339	4·0
CONTINUE	9,165	5·0	309	3·0
WRITE	7,795	4·0	508	5·0
FORMAT	7,685	4·0	380	4·0
DO	7,476	4·0	457	5·0
DATA	4,468	2·0	28	0·3
RETURN	3,639	2·0	186	2·0
DIMENSION	3,492	2·0	141	1·5
COMMON	2,908	1·5	263	3·0
END	2,565	1·0	121	1·0
BUFFER	2,501	1·0	0	
SUBROUTINE	2,001	1·0	93	1·0
REWIND	1,724	1·0	6	
EQUIVALENCE	1,382	0·7	113	1·0
ENDFILE	765	0·4	2	
INTEGER	657	0·3	34	0·3
READ	586	0·3	92	1·0
ENCODE	583	0·3	0	
DECODE	557	0·3	0	
PRINT	345	0·2	5	
ENTRY	279	0·1	15	0·2
STOP	190	0·1	11	0·1
LOGICAL	170	0·1	9	0·1
REAL	147	0·1	3	
IDENT	106	0·1	0	
DOUBLE	3		99	1·0
OVERLAY	82		0	
PAUSE	57		6	0·1
ASSIGN	57		4	
PUNCH	52		5	0·1
EXTERNAL	23		1	
IMPLICIT	0		16	1·5
COMPLEX	6		0	
NAMELIST	5		0	
BLOCKDATA	1		2	
INPUT	0		0	
OUTPUT	0		0	
COMMENT	52,924	(28)	1,090	(11)
CONTINUATION	13,709	(7)	636	(7)

† Per cent of total number of statements excluding comments and continuation cards.

‡ The construction 'IF ( ) statement' counts as an IF as well as a statement, so the total is more than 100 per cent.

about thirty programs were tested, so these statistics tended to be rather stable. We forgot to test how many statements had non-blank labels.

The same test was run on a much smaller but still rather large collection of programs from our 'Stanford sample' (about 11,000 cards). Unfortunately the corresponding percentages shown in Table I do *not* agree very well with the Lockheed sample; Stanfordites definitely use more assignments and less IF's and GO's than Lockheedians. A superficial examination of the programs suggests that Lockheed programmers are perhaps more careful to check for erroneous conditions in their data. Note also that 2·7 times as many comments appear on the Lockheed programs, indicating somewhat more regimentation. The professional programmers at Lockheed have a distinctly different style from Stanford's casual coders.

The 7,933 DO loops were further investigated to determine their length and depth of nesting; about 95 per cent of the DO statements used the default increment of one. Most DO loops were quite short involving only one or two statements:

Length	1	2	3	4	5	>5
Number	3,046	1,467	758	576	1,043	1,043
Per cent	39·0	18·5	9·5	7·0	13·0	13·0

The depth of DO nesting was subject to considerable variation; the following totals were obtained:

Depth	1	2	3	4	5	>5
Number	4,211	1,853	1,194	437	118	120
Per cent	53·5	23·0	15·0	5·5	1·5	1·5

Of the 28,783 IF statements scanned, 8,858 (30 per cent) were of the 'old style' IF (...) n_1, n_2, n_3 or IF (...) n_1, n_2 while the other 19,925 (70 per cent) had the form IF (...) statement; 14,258 (71 per cent) of the latter were IF (...) GO TO'. (These count also as GO TO statements.) Only 1,107 of the 25,719 GO TO statements were computed (switch) GO's.

An average of about 48 blank columns was found per non-comment card. A compiler's lexical scanner should therefore include a high-speed skip over blanks.

Assignment statements were analysed in some detail. There were 83,304 assignment statements in all; and 56,751 (68 per cent) of them were trivial replacements of the form $A = B$ where no arithmetic operations are present!† The remaining assignments included 10,418 of the form $A = A$ op α, i.e. the first operand on the right is the same as the variable on the left. An attempt was made to rate the complexity of an assignment statement, counting one point for each + or − sign, five for each * and eight for each /; the distribution was

Complexity	0	1	2	3	4	5	6	7	8	9
Number	56,751	14,645	1,124	106	267	2,436	1,988	562	2,359	552
Per cent	68·0	17·5	1·3	0·1	0·3	3·0	2·0	0·6	3·0	0·6

Occurrences of operators and constants were also tallied:

Operator	+	−	*	/	**	=	standard function	constant
Occurrences	17,973	10,298	12,348	4,739	1,108	90,257	3,994	49,386

---

† In the Stanford sample the corresponding figures were 2,379 out of 4,869 (49 per cent); this was another example of a Lockheed versus Stanford discrepancy.

It is rather surprising to note that 7,200 (40 per cent) of the additions had the form $\alpha + 1$; 349 (3 per cent) of the multiplications had the form $\alpha * 2$; 180 (4 per cent) of the divisions had the form $\alpha / 2$; 427 (39 per cent) of the exponentiations had the form $\alpha ** 2$. (We forgot to count the fairly common occurrences of $2 * \alpha$, $2. * \alpha$, $\alpha * 2.$, $\alpha / 2.$, $2.0 * \alpha$, etc.)

The program analysed indices, although it was unable to distinguish subscripted variables from calls on programmer-defined functions. Of the 166,599 appearances of variables, 97,051 (58 per cent) were unindexed, 50,979 (30·5 per cent) had one index, 16,181 (9·5 per cent) had two, 2,008 (1 per cent) had three and 380 (0·2 per cent) had four.

Another type of 'static' test on the nature of FORTRAN programs was also made, in an attempt to discover the complexity of *control flow* in the programs. Cocke's 'interval reduction' scheme[7] was applied to fifty randomly selected FORTRAN programs and subroutines, and in every case the flow graph was reduced to a single vertex after six or less transformations. The average number of transformations required per program was only 2·75.

The obvious conclusion to draw from all these figures is that compilers spend most of their time doing surprisingly simple things.

3. DYNAMIC STATISTICS

The static counts tabulated above are relevant to the speed of compilation, but they do not really have a strong connection with the speed of object program execution. We need to give more weight to statements that are executed more frequently.

Two different approaches to dynamic program analysis were explored in the course of our study, the method of frequency counts or program profiles and the method of program status sampling. The former method inserts counters at appropriate places of the program in order to determine the number of times each statement was actually performed; the latter method makes use of an independent system program which interrupts the object program periodically and notes where it is currently executing instructions.

Frequency counts were commonly studied in the early days of computers (see von Neumann and Goldstine)[8] and they are now experiencing a long-overdue revival. We made use of a program called FORDAP, which had been developed previously in connection with some research on compilation; FORDAP takes a FORTRAN program as input, and outputs an equivalent program which also maintains frequency counts and writes them on to a file. When the latter program is compiled and run, its output will include a listing of the executable statements together with their frequency counts. See Figure 1, which illustrates the output corresponding to a short program, using an extension of FORDAP which includes a rough estimate of the relative cost of each statement, Ingalls.[9] The principles of preparing such a routine were independently developed at UCLA by Russell;[10] Russell's efforts were primarily directed towards a study of potential parallelism in programs, but he also included some serial analyses of large scale routines which exhibit the same phenomena observed in our own studies.

Frequency counts add an important new dimension to the FORTRAN programs; indeed, it is difficult to express in words just how tremendously 'eye-opening' they are. Even the small example in Figure 1 has a surprise (the frequency counts reveal that about half the running time is spent in the subroutine linkage of the FUN function). After studying dozens of FORDAPed programs, and after experiencing the reactions of programmers who see the frequency counts of their own programs, our group came to the almost unanimous conclusion that all software systems should provide frequency counts to all programmers, unless specifically told not to do so.

```
    EXECUTABLE STATEMENTS                                    EXECUTIONS    COST

10   READ (5,1) XC,YC,H,JNT,IENT                                 2          100
     IF (IENT) 20,40,20                                          2            4
20   WRITE (6,2) H,XO,YO                                         1           50
     CALL RK2 (FUN,H,XO,YO,JNT,IENT,A)                           1            2
     STEP=FLCAT(JNT)*H                                           1           11
     X=XO                                                        1            1
     DO 30 I=1,IENT                                              1            2
     X=X+STEP                                                   30           60
30   WRITE (6,3) X,A(I)                                         30         1530
     GO TO 10                                                    1            1
40   STOP                                                        1            1
     END                                                         1            1



     FUNCTION FUN(X,Y)                                        1200        18000
     FUN =1./X                                                1200         9600
     RETURN                                                   1200         6000
     END



     SUBROUTINE RK2(FUN,H,XI,YI,K,N,VEC)                        1           15
     H2=H/2.                                                    1            8
     Y=YI                                                       1            1
     X=XI                                                       1            1
     DO 2 I=1,N                                                 1            2
     DO 1 J=1,K                                                30           60
     T1=H*FUN(X,Y)                                            300         2700
     T2=H*FUN(X+H2,Y+T1/2.)                                   300         5400
     T3=H*FUN(X+H2,Y+T2/2.)                                   300         5400
     T4=H*FUN(X+H,Y+T3)                                       300         3300
     Y= Y+(T1+2.*T2+2.*T3+T4)/6.                              300         6900
1    X=X+H                                                    300          900
2    VEC(I)=Y                                                  30           90
     RETURN                                                     1            5
     END
```

Figure 1. The profile of a short program

The advantages of frequency counts in debugging have been exploited by Satterthwaite[11] in his extensions to Stanford's ALGOL-W compiler. They can be used to govern selective tracing and to locate untested portions of a program. Once the program has been debugged, its frequency counts show where the 'bottlenecks' are, and this information often suggests improvements to the algorithm and/or data structures. For example, we applied FORDAP to *itself*, since it was written in FORTRAN, and we immediately found that it was spending about half of its time in two loops that could be greatly simplified; this made it possible to double the speed of FORDAP, in less than an hour's work, without even looking at the rest of the program. (See example 2 in section 4.) The same thing happened many times with other programs.

Thus our experience has suggested that frequency counts are so important they deserve a special name; let us call the collection of frequency counts the *profile* of a program.

Programs typically have a very jagged profile, with a few sharp peaks. As a very rough approximation, it appears that the nth most important statement of a program from the standpoint of execution time accounts for about $(\alpha-1)\alpha^{-n}$ of the running time, for some α and for small n. We also found that less than 4 per cent of a program generally accounts for more than half of its running time. This has important consequences, since it means that programmers can make substantial improvements in their own routines by being careful in just a few places; and optimizing compilers can be made to run much faster since they need not study the whole program with the same amount of concentration.

Table II shows how the relative frequency of statement types changes when the counts are dynamic instead of static; this table was compiled from the results of twenty-four FORDAP runs, with the statistics for each program weighted equally. We did not have time to break down these statistics further (to discover, for example, the distribution of operators, etc.), except in one respect: 45 per cent of the assignment statements were simply replacements (of the form $A = B$ where B is a simple variable or constant), when counting statically, but this dropped to 35 per cent when counting dynamically. In other words, replacements tend to occur more often outside of loops (in initialization sections, etc.).

Table II. Distribution of executable statements

	Static	(per cent)	Dynamic
Assignment	51		67
IF	10		11
GO TO	9		9
DO	9		3
CALL	5		3
WRITE	5		1
CONTINUE	4		7
RETURN	4		3
READ	2		0
STOP	1		0

The other approach to dynamic statistics gathering, based on program status sampling, tends to be less precise but more realistic, in the sense that it shows how much time is actually spent in system subroutines. We used and extended a routine called PROGTIME[12] which was originally developed by Johnston and Johnson to run on System/360 under MVT. PROGTIME spawns the user program as a subtask, then samples its status word at regular intervals, rejecting the datum if the program was dormant since its last interruption. An example of the resulting 'histogram' output appears in Figure 2; it is possible (although not especially convenient) to relate this to the FORTRAN source text.

In general, the results obtained from PROGTIME runs were essentially what we would have expected from the FORDAP produced profiles, except for the influence of input/output editing times. The results of FORDAP would have led us to believe that the code between relative locations 015928 and 015A28 in Figure 2 would consume most of the running time, but in fact FORDAP does not give the true over-all picture since 70 per cent of the time was actually spent in those beloved system subroutines IHCECOMH and IHCFCVTH (relative locations 016A88–019080). Roughly half of the programs we studied involved substantial amounts of input/output editing time, and this led us to believe that considerable gains in efficiency would be achieved if the compilers would do the editing in-line wherever possible. It was easy to match up the formats with the quantities to be edited, in every case we looked at. However, we did not have time to study the problem further to investigate just how much of an improvement in performance could be expected from in-line editing. Clearly the general problem of editing deserves further attention, since it seems to use up more than 25 per cent of the running time of FORTRAN programs in spite of the extremely infrequent occurrence of actual input/output statements reflected in Table II.

Due to the random nature of the sampling process, two PROGTIMEs of the same program will not give identical results. It is possible to get accurate frequency counts and accurate running times by using the technique of 'jump tracing' (see Gaines).[13] A jump trace routine scans a program down to the next branch instruction and executes the intervening code at machine speed; when a branch occurs the location transferred to is written on to a file. Subsequent processing of the file makes it possible to infer the frequency counts. The jump trace approach does not require auxiliary memory for counters and it can be used with arbitrary machine language programs. Unfortunately we did not have time to develop such a routine for Stanford's computers during the limited time in which our study was performed.

4. THE INNER LOOPS

We selected seventeen programs at random for closer scrutiny; this section contains a summary of the main features of these programs. (It is worth emphasizing that we did not modify the programs nor did we discard programs that did not produce results in accordance with our preconceived ideas; we analysed every routine we met whether we liked it or not. The result is hopefully a good indication of typical FORTRAN programming practice and we believe that a reader who scans these programs will obtain a fairly clear conception of how FORTRAN is being used.) First the program profile was found by running it with FORDAP and PROGTIME. (This caused the chief limitation on our selection, for we were unable to study programs for which input data was on inaccessible tapes or otherwise unavailable.) In each case a glance at the profile reduced the program to a comparatively small piece of code which represented the majority of the execution time exclusive of input/output statements. These 'inner loops' of the programs are presented here; the names of identifiers have been changed in order to give some anonymity, but no other changes have been made.

In each case we hand-translated the inner loop into System/360 machine language using five different styles of 'optimization':

Level 0

Straight code generation according to classical one-pass compilation techniques.

Level 1

Like level 0 but using local optimizations based on a good knowledge of the machine; common subexpressions were eliminated and register contents were remembered across statements if no labels intervene, etc., and the index of a DO was kept in a register but no optimizations requiring global flow analysis were made.

Level 2

'Machine-independent' optimizations based on global flow analysis, including constant folding, invariant expression removal, strength reduction, test replacement and load-store motion (cf. Allen).[14]

Level 3

Like level 2 plus machine-dependent optimizations based on the 360, such as the use of BXLE, LA, and the possibilities afforded by double indexing.

Level 4

The 'best conceivable' code that would be discovered by any compiler imaginable. Anything goes here except a change in the algorithm or its data structures.

DONALD E. KNUTH

```
RUNNING PROFILE FOR ENTITY   2        MAIN    1  0001E478

                    0 PERCENT   2 PERCENT   .4 PERCENT   6 PERCENT   8 PERCENT
                    (    0)     (  132)     (  264)      (  395)     (  527)     NUMBER
ABS.    REL.        -------------------------------------------------------------
C3E640  0C0000      *                                                              1
C4BCCC  0156A8      *****************                                             53
C4BD20  0156C8      ************************                                     102
04BC40  0156E8      **********************                                       86
C4BCC0  015768      **************************                                  102
C4BDE0  015788      *****************************************                   158
C4BEC0  0157A8      *************************************************           179
C4BEE0  015888      ***                                                         15
C4BFC0  0158A8      ****************                                            83
C4BF20  0158C9      ************                                                 50
C4BF40  0158E8      *                                                            1
C4BF80  015926      **********************************************************  311
C4BFA0  015948      *************************************************           259
04BFC0  015968      **********************************                         173
C4CCE0  015428      *                                                            1
C4DCE0  016A88      *********************************                           156
C4D1E0  016B88      *                                                            1
C4D220  016BC8      *                                                            7
C4D2C0  016C68      *                                                            7
C4D2E0  016C88      *                                                            2
C4D3C0  016CA8      *                                                            2
C4D320  016CC8      *                                                            1
C4C3A0  016D48      *                                                            1
C4D4C0  016E68      *                                                            2
C4D4E0  016E88      *                                                            2
C4D5C0  016EA8      *                                                            3
C4D520  016EC8      *                                                            4
C4D5E0  016F08
C4D5A0  016F48
C4D5C0  016F68      ********************                                        1-7
C4D5E0  016F88      ***************************************************        501
C4D6C0  016FA8      *************************************************          527
C4D6E0  017028      ********************                                       202
```
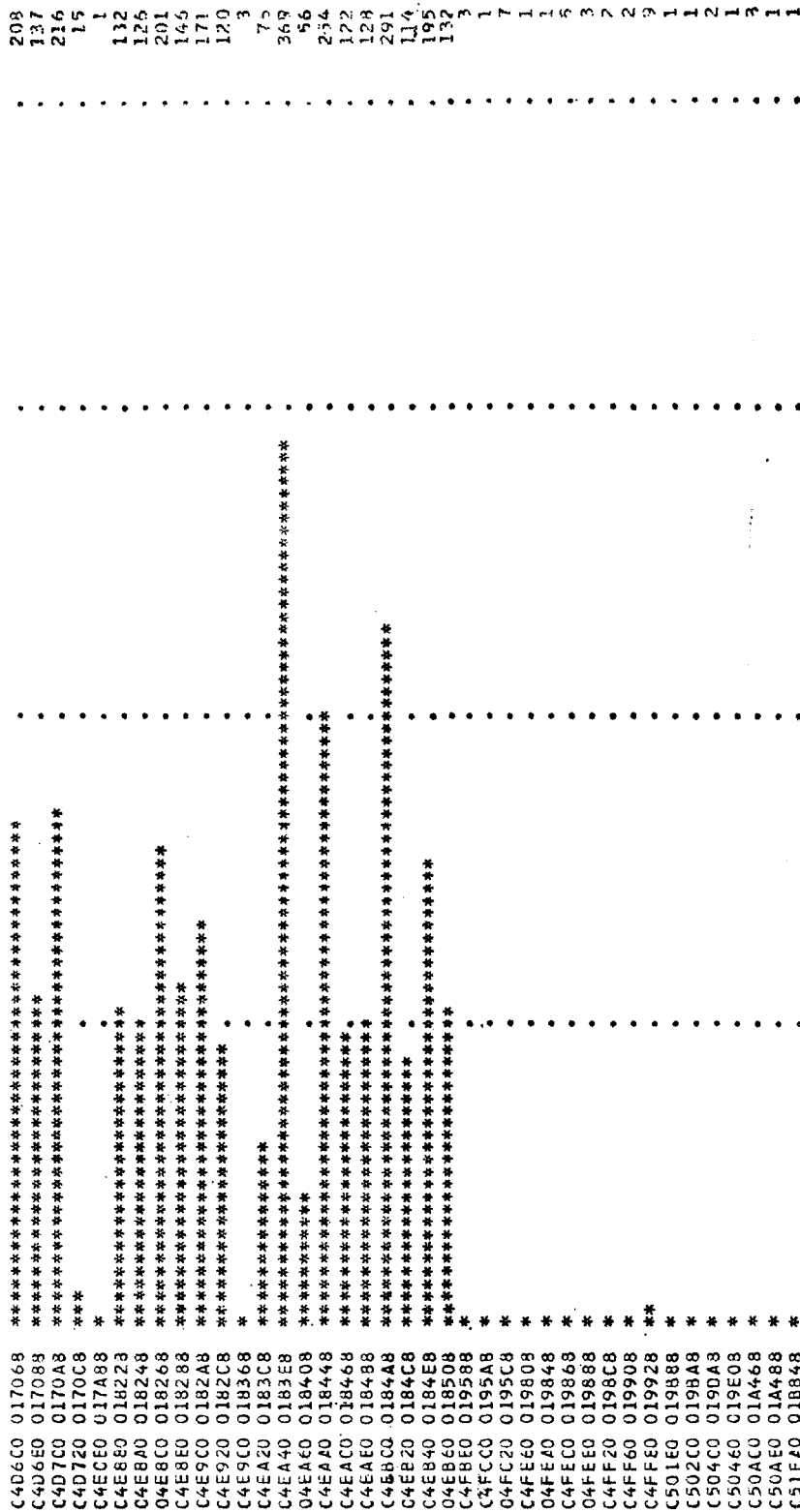
continued

Figure 2. Histogram corresponding to a PROGTIME run

These styles of optimization are not extremely well defined, but in each case we produced the finest code we could think of consistent with that level. (In nearly every case this was noticeably better than the optimizations produced by the existing FORTRAN compilers; FORTRAN H OPT 02 would presumably be able to reach level 3 if it were carefully tuned.) Level 4 represents the ultimate achievable, by comparison with what is realized by current techniques, in an attempt to assess whether or not an additional effort would be worth while.

These styles of optimization can best be appreciated by studying Example 1 for which our machine language coding appears in the Appendix to this paper. It is appropriate to restrict our attention solely to the inner loop, since the profiles show that the effect of optimization on this small part of the code is very indicative of the total effect of optimization on the program as a whole.

In order to compare one strategy to another, we decided to estimate the quality of each program by hand instead of actually running them with a timer, as in Wichmann.[15] We weighted the instructions in a crude but not atypical manner as follows: Each instruction costs one unit, plus one if it fetches or stores an operand from memory or if it is a branch that is taken, plus a penalty for specific slower opcodes:

Floating add/subtract	add 1	
Multiply	add 5	
Divide	add 8	
Multiply double	add 13	
Shift	add 1	
Load multiple	add $\frac{1}{2}n$	(n registers loaded)
Store multiple	add $\frac{1}{3}n$	(n registers stored)

This evaluation corresponds roughly to 1 unit per 0·7 microseconds on our model 67 computer. Other machine organizations ('pipelining', etc.) would, of course, behave somewhat differently, but the above weights should give some insight. We also assumed the following additional costs for the time spent in library subroutines, cf. Reference 16.

SQRT	85
SIN, COS	110
ALOG	120
ERF	130
Complex multiply	60
Real ** Integer	75

Example 1

The first program we studied involved 140 executable statements, but the following five represented nearly half of the running time:

```
      DO 2 J = 1, N
      T = ABS(A(I, J))
      IF (T − S) 2, 2, 1
    1 S = T
    2 CONTINUE
```

Statement 1 was executed about half as often as the others in the loop. The programs in the Appendix have a 'score' of

$$37 \cdot 5, \quad 28 \cdot 5, \quad 14 \cdot 0, \quad 8 \cdot 0, \quad 7 \cdot 0$$

for levels 0, 1, 2, 3, 4 respectively.

The same program also included another time-consuming loop,

$$\text{DO } 3 \text{ J} = 1, \text{N}$$

$$3 \quad A(I, J) = A(I, J)*B$$

for which the respective scores are

$$51 \cdot 0, \quad 29.0, \quad 17 \cdot 0, \quad 12 \cdot 0, \quad 11 \cdot 0$$

In this case, level 0 is penalized for calculating the subscript twice.

Example 2

(This came from the original FORDAP program itself.) Although there were 455 executable statements, over half of the program time was spent executing two loops like this:

$$\text{DO } 1 \text{ J} = 38,53$$

$$\text{IF (K(I).EQ.L(J)) GO TO 3}$$

$$1 \quad \text{CONTINUE}$$

$$2 \quad \ldots$$

The five styles of translation give respective scores of

$$23 \cdot 0, \quad 19 \cdot 0, \quad 9 \cdot 0, \quad 5 \cdot 0, \quad 3 \cdot 5$$

Level 4's score of 3·5 is obtained in an interesting way which applies to several other loops we had examined earlier in the summer; we call it the technique of *combining tests*. The array element L(54) is set equal to K(I), so that the loop involves only one test; then after reaching L3, if J = 54 we go back to L2. The code is

```
Q1   LA 3, 8(0, 3)
     C  4, 0(0, 3)
     BER 5        (Register 5 contains A(L3))
     C  4, 4(0, 3)
     BNE Q1
L3   ...
```

If necessary, L(54) could be restored.

Of course, in this particular case the loop is executed only sixteen times, and so it could be completely unrolled into 32 instructions

```
C  4, L(38)
BER 5
C  4, L(39)
BER 5
  ⋮
C  4, L(53)
BER 5
```

reducing the score to 3·0. But in actual fact the L table was loaded in a DATA statement and it contained a list of special character codes; a more appropriate program would replace the entire DO loop by a single test

$$\text{IF (LT(K(I))) 1, 3, 1}$$

for a suitable table LT, thereby saving over half the execution time of the program. (Furthermore, the environment of the above DO loop was

$$\text{DO 2 I} = 7, 72$$

so that any assembly language programmer would have reduced the whole business to a single 'translate and test'.)

Example 3

```
        DOUBLE A, B, D
        DO 1 K = 1, N
        A = T(I − K, 1 + K)
        B = T(I − K, J + K)
    1   D = D − A*B
```

(This is one of the few times we observed double precision being used, although the numerical analysis professors in our department strongly recommend against the short precision operators of the 360; it serves as another indication that our department seems to have little impact on the users of our computer!) The scores for this loop are

$$89·0, \quad 67·0, \quad 38·0, \quad 13·0, \quad 0·12$$

here level 2 suffers from some clumsiness in the indexing and a lack of knowledge that an ME instruction could be used instead of MD.

Example 4

Here the inner loop is longer and involves a subroutine call. The following code accounted for 70 per cent of the running time; the entire program had 214 executable statements.

```
        DO 1 K = M, 20
        CALL RAND(R)
        IF (R .GT. .81) N(K) = 1
    1   CONTINUE
    ...  ...
        SUBROUTINE RAND(R)
        J = I*65539
        IF (J) 1, 2, 2
    1   J = J + 2147483647 + 1
    2   R = J
        R = R*.4656613E − 9
        I = J
        K = K + 1
        RETURN
        END
```

(Here we have a notoriously bad random number generator which the programmer must have got out of an obsolete reference book; it is *another* example of our failure to educate the community.) Conversion from integer to real is assumed to be done by the sequence

$$\begin{array}{lll} \text{X} & \text{r,} & = \text{XL4'80000000'} \\ \text{ST} & \text{r, SPEC} + 4 \\ \text{LD} & 0, \text{SPEC} \\ \text{AD} & 0, \text{SPEC1} \end{array}$$

for suitable contents of SPEC and SPEC1. By further adjusting these constants the multiplication by $0.4656613E - 9 \approx 2^{-31}$ could be avoided; but this observation was felt to be beyond the scope of level 4 optimization, although it would occur naturally to any programmer using assembly language.

The most interesting thing here, however, is the effect of subroutine linkage, since the long prologue and epilogue significantly increase the time of the inner loop. The timings for levels 0–3 assume standard OS subroutine conventions, although levels 2 and 3 are able to shorten the prologue and epilogue somewhat because of their knowledge of program flow. For level 4, the subroutine was 'opened' and placed in the loop without any linkage; hence the sequence of scores,

$$119.9, \quad 105.1, \quad 81.4, \quad 76.2, \quad 27.2$$

Without subscripting there is comparatively little difference between levels 0 and 3; this implies that optimization probably has more pay off for FORTRAN than we would find for languages with more flexible data structures.

It would be interesting to know just how many hours each day are spent in prologues and epilogues establishing linkage conventions.

Example 5

The next inner loop is representative of several programs which had to be seen to be believed.

```
      DO 1 K = 1, N
      M = (J - 1)*10 + K - 1
      IF (M.EQ.0) M = 1001
      C1 = C1 + A1(M)*(B1**(K - 1))*(B2**(J - 1))
      C2 = C2 + A2(M)*(B1**(K - 1))*(B2**(J - 1))
      IF ((K - 1).EQ.0) T = 0.0
      IF ((K - 1).GE.1) T = A1(M)*(K - 1)*(B1**(K - 2))*(B2**(J - 1))
      C3 = C3 + T
      IF ((K - 1).EQ.0) T = 0.0
      IF ((K - 1).GE.1) T = A2(M)*(K - 1)*(B1**(K - 2))*(B2**(J - 1))
      C4 = C4 + T
      IF ((J - 1).EQ.0) T = 0.0
      IF ((J - 1).GE.1) T = A1(M)*(B1**(K - 1))*(J - 1)*(B2**(J - 2))
      C5 = C5 + T
      IF ((J - 1).EQ.0) T = 0.0
      IF ((J - 1).GE.1) T = A2(M)*(B1**(K - 1))*(J - 1)*(B2**(J - 2))
      C6 = C6 + T
    1 CONTINUE
```

After staring at this for several minutes our group decided it did not deserve to be optimized. But after two weeks' rest we looked at it again and found interesting applications of 'strength reduction', both for the exponentiations and for the conversion of K to real. (The latter applies only in level 4, which knows that K does not get too large.) The scores were

$$1367 \cdot 0, \quad 545 \cdot 0, \quad 159 \cdot 0, \quad 145 \cdot 0, \quad 104 \cdot 0$$

Level 1 optimization finds common subexpressions and level 2 finds the reductions in strength. Level 4 removes nearly all the IF tests and rearranges the code so that C1 and C2 are updated last; thus only B1**(K−1) is necessary, not both it and B1**(K−2).

Example 6

In this case the inner loop involves subroutine calls instead of a DO loop:

	SUBROUTINE S(A, B, X)	9
	DIMENSION A(2), B(2)	9
	X = 0	9
	Y = (B(2) − A(2))*12 + B(1) − A(1)	9
	IF (Y.LT.0) GO TO 1	9
	X = Y	5
1	RETURN	9
	END	9
	SUBROUTINE W(A, B, C, D, X)	4
	DIMENSION A(2), B(2), C(2), D(2), U(2), V(2)	4
	X = 0	4
	CALL S(A, D, X)	4
	IF (X.EQ.0) GO TO 3	4
	CALL S(C, B, X)	2
	IF (X.EQ.0) GO TO 3	2
	CALL S(C, A, X)	1
	U(1) = A(1)	1
	U(2) = A(2)	1
	IF (X.NE.0) GO TO 1	1
	U(1) = C(1)	0
	U(2) = C(2)	0
1	CONTINUE	1
	CALL S(B, D, X)	1
	V(1) = B(1)	1
	V(2) = B(2)	1
	IF (X.NE.0) GO TO 2	1
	V(1) = D(1)	0
	V(2) = D(2)	0
2	CALL S(U, V, X)	1
3	CONTINUE	4
	RETURN	4
	END	4

The numbers at the right of this code show the approximate relative frequency of occurrence of each statement; calls on this subroutine accounted for 60 per cent of the execution time of the program. The scores for various optimization styles are

$$1545 \cdot 5, \quad 1037 \cdot 5, \quad 753 \cdot 3, \quad 736 \cdot 3, \quad 289 \cdot 0$$

Here 270 of the 1545·5 units for level 0 are due to repeated conversions of the constant 0 from integer to real. Levels 2 and 3 move the first statement 'X = 0' out of the main loop, performing it only if 'Y.LT.0'. The big improvement in level 4 comes from inserting the code for subroutine S in line and making the corresponding simplifications. Statements like $U(1) = A(1)$, $U(2) = A(2)$ become simply a change in base register. Perhaps further reductions would be possible if the context of subroutine W were examined, since if we denote $12*A(1)+A(2)$ by a, $12*B(1)+B(2)$ by b, etc., the subroutine computes $\max [0, \min (b, d) - \max (a, c)]$.

Example 7

In this program virtually all of the time exclusive of input/output editing was spent in the two loops

```
      DO 1 I = 1, N
      A = X**2 + Y**2 - 2.*X*Y*C(I)
      B = SQRT(A)
      K = 100.*B + 1.5
    1 D(I) = S(I)*T(K)
      Q = D(1) - D(N)
      DO 2 I = 2, M, 2
    2 Q = Q + 4.*D(I) + 2.*D(I + 1)
```

where array D was not used subsequently. The scores are

$$744 \cdot 0, \quad 387 \cdot 0, \quad 316 \cdot 0, \quad 292 \cdot 0, \quad 256 \cdot 0$$

Here level 1 computes X**2 by 'MER 0, 0' instead of a subroutine call and it computes 2.*D(I + 1) by 'AER 0, 0' instead of multiplying. Level 4 combines the two DO loops into one and eliminates array D entirely. (Such savings in storage space were present in quite a few programs we looked at; some matrices could be reduced to vectors, and some vectors could be reduced to scalars, due to the nature of the calculations. A quantitative estimate of how much space could be saved by such optimization would be interesting.)

Example 8

Ninety per cent of the running time of this program was spent in the following subroutine

```
      SUBROUTINE COMPUTE
      COMMON ...
      COMPLEX Y(10), Z(10)
      R = REAL(Y(N))
      P = SIN(R)
      Q = COS(R)
      S = C*6.*(P/3. - Q*Q*P)
      T = 1.414214*P*P*Q*C*6.
```

$U = T/2.$
$V = -2.*C*6.*(P/3. - Q*Q*P/2.)$
$Z(1) = (0., -1.)*(S*Y(1) + T*Y(2))$
$Z(2) = (0., -1.)*(U*Y(1) + V*Y(2))$
RETURN
END

This was the only example of complex arithmetic that we observed in our study. The scores

$$841 \cdot 5, \quad 735 \cdot 5, \quad 336 \cdot 0, \quad 336 \cdot 0, \quad 249 \cdot 0$$

reflect the fact that levels 0 and 1 make six calls on the complex-multiply subroutine, while levels 2 and 3 expand complex multiplication into a sequence of real operations (with obvious simplifications). Level 4 in this analysis makes free use of the distributive law, e.g. $S = C*P*(2. - 6.*Q*Q)$, although this may not be numerically justified. Furthermore level 4 assumes the existence of a single 'SINCOS(R)' subroutine that computes both the sine and cosine of its argument in 165 units of time; programmers who calculate the sine of an angle usually want to know its cosine too and vice versa, and it is possible to calculate both in somewhat less time than would be required to compute them individually.

Example 9

A program with 245 executable statements spent 70 per cent of its time in

```
      DO 2 K = 1, M
      DO 2 J = 1, M
      X = 0.
      Y = 0.
      DO 1 I = 1, M
      N = (J + J + (I - 1)*M2)
      B = A(K, I)
      X = X + B*Z(N)
   1  Y = Y + B*Z(N - 1)
      DY(L) = W*X
      DY(L + 1) = -W*Y
   2  L = L + 2
```

when M was only 3. Scores (for the innermost I loop only) are

$$84 \cdot 0, \quad 69 \cdot 0, \quad 30 \cdot 0, \quad 24 \cdot 0, \quad 24 \cdot 0$$

reflecting the fact that level 4 cannot do anything for this case.

Example 10

In this excerpt from a contour plot routine, the CALL is only done rarely:

```
      DO 1 I = L, M
   1  IF (X(I - 1, J) .LT.Q .AND. X(I, J) .GE. Q) CALL S(A1, A2, A3, A4, 7, A5)
```

The scores, assuming that X(I) .LT.Q about half the time, are

$$40 \cdot 0, \quad 31 \cdot 5, \quad 14 \cdot 5, \quad 7 \cdot 5, \quad 5 \cdot 0$$

Level 3 keeps Q in a register, while level 2 does not. Level 4 is especially interesting since it avoids testing $X(I-1, J)$ .LT.Q in those cases where it is known to be true from the previous loop. We had noticed similar situations in other routines.

Example 11

This 'fast Fourier transform' example shows that inner loops are not always signalled by the word 'DO'.

```
1   K = K+1
    A1 = A(K)*C(J)+A1
    B1 = B(K)*C(J)+B1
    K = K+1
    A2 = A(K)*S(J)+A2
    B2 = B(K)*S(J)+B2
    J = J+I
    IF (J.GT.M) J = J−M
    IF (K.LT.M) GO TO 1
```

The scores are

$$118 \cdot 0, \quad 91 \cdot 0, \quad 60 \cdot 0, \quad 54 \cdot 0, \quad 50 \cdot 0$$

Level 4 is able to omit the second '$K = K+1$', and to use a BXLE for '$J = J+I$'.

Example 12

Unfortunately an inner loop is not always as short as we had hoped. This rather long program (1,300 executable statements) spent about half of its time in the following rather horrible loop:

```
DO 3 I = 1, M
JO = J1
IF (JO.EQ.0) JO = J2
J1 = J1+1
J3 = J3+1
J4 = J4+1
IF (J4.EQ.(L(J−1)+1)) J4 = 1
J5 = J1+1
IF (J5.EQ.(J2+1)) J5 = 1
U1 = U(J1, K1, K2)
V1 = V(J1, K1, K2)
W1 = W(J1, K1, K2)
P(J1) = .25*(Q1(I)*(V1+V(J3, K3, K2))*(W1+W(J3, K3, K2))
        +Q2(I)*(V1+V(J3+1, K3, K2))*(W1+W(J3+1, K3, K2))
        −Q3(I)*(V1+V(J4, K4, K2))*(W1+W(J4, K4, K2))
        +D*((U1+U(J5, K1, K2))*(W1+W(J5, K1, K2))
          −(U1+U(J0, K1, K2))*(W1+W(J0, K1, K2))))
        +R1(J1, K1)*R2(K2)*(S(J1, K2+1)*(W1+W(J1, K1, K2+1))
          −S(J1, K2)*(W1+W(J1, K1, K2−1)))
```

```
      IF (I.EQ.1) GO TO 1
      J6 = J4 − 1
      IF (J6.EQ.0) J6 = L(J − 1)
      P(J1) = P(J1) − .25*Q4(I)*(V1 + V(J6, K4, K2))*(W1 + W(J6, K4, K2))
      GO TO 3
    1 IF (M.EQ.1) GO TO 2
      P(J1) = P(J1) + .25*Q4(I)*(V1 + V(J3 − 1, K3, K2))*(W1 + W(J3 − 1, K3, K2))
      GO TO 3
    2 P(J1) = P(J1) + .25*Q4(I)*(V1 + V(J2 + 4, K3, K2))*(W1 + W(J2 + 4, K3, K2))
    3 CONTINUE
```

Here levels 2 and 3 have just enough registers to maintain all the necessary indices; the scores are

$$792 \cdot 0, \quad 368 \cdot 0, \quad 242 \cdot 0, \quad 238 \cdot 0, \quad 207 \cdot 0$$

Level 4 observes that J6 can more easily be computed by 'J6 = J4' before J4 is changed, and the Q4(I) terms are included as if they were conditional expressions within the big formula for P(J1).

Example 13

Here is a standard 'binary search' loop:

```
          I = 0
          K = N + 1
        1 J = (I + K)/2
          IF (J.EQ.I) GO TO 5
          IF (X(J) − X0) 2, 4, 3
        2 I = J
          GO TO 1
        3 K = J
          GO TO 1
        4 ...
        5 ...
```

The scores

$$38 \cdot 5, \quad 33 \cdot 0, \quad 27 \cdot 0, \quad 21 \cdot 0, \quad 10 \cdot 0$$

for the inner loop are of interest primarily because level 4 was able to beat level 3 by a larger factor than in any other example (except where subroutines were expanded in-line). The coding for level 4 in this case consisted of six packets of eight lines each, one for each permutation of the three registers α, β and γ:

```
      L1αβγ  LA    γ, 0(α, β)
             SRL   γ, 1
             NR    γ, 8
             CR    γ, α
```

$$\text{BE} \quad \text{L5}\alpha$$
$$\text{CE} \quad 0, \text{X}(\gamma)$$
$$\text{BL} \quad \text{L1}\gamma\beta\alpha$$
$$\text{BE} \quad \text{L4}\gamma$$
$$\text{L1}\alpha\gamma\beta \quad \dots$$

Here 4I, 4J and 4K are respectively assumed to be in registers α, γ and β; register 8 contains -4. Division by 2 can be reduced to a shift since it is possible to prove that I, J and K are non-negative. Half of the 'CR γ, α; BE L5α' could have been removed if X(0) were somehow set to '$-\infty$'; this would save another 10 per cent.

Actually the binary search was not the inner loop in the program we analysed although the programmer (one of our group) had originally thought it would be! The frequency counts showed that his program was actually spending most of its time *moving* entries in the X table to keep it in order when new elements were inserted. This was one of many cases we observed where a knowledge of frequency counts immediately suggested vital improvements by directing the programmer's attention to the real bottlenecks in his program. Changing to a hash-coding scheme made this particular program run about twice as fast.

Examples 14–17

From this point on the programs we looked at began to seem rather repetitious. We worked out four more examples, summarized here with their scores:

```
    DO 1 I = 1, N
    C = C/D*R
    R = R+1.
1   D = D-1.
```
$[45{\cdot}0, 42{\cdot}0, 27{\cdot}0, 21{\cdot}0, 20{\cdot}0]$

```
    DO 1 J = I, N
    H(I, J) = H(I, J) + S(I)*S(J)/D1 - S(K+I)*S(K+J)/D2
1   H(J, I) = H(I, J)
```
$[136{\cdot}0, 103{\cdot}0, 58{\cdot}0, 49{\cdot}0, 41{\cdot}5]$

```
    REAL FUNCTION F(X)
    Y = X*.7071068
    IF (Y.LT.0.0) GO TO 1
    F = 0.5*(1.0+ERF(Y))  ⎫
    RETURN                ⎬ low frequency
1   F = 1.0-0.5*(1.0+ERF(-Y))
    RETURN
    END
```
$[219{\cdot}5, 208{\cdot}5, 191{\cdot}3, 191{\cdot}3, 151{\cdot}0]$

```
    DO 1 I = 1, N
1   A = A+B(I)+C(K, I)
```
$[41{\cdot}0, 31{\cdot}0, 14{\cdot}0, 9{\cdot}0, 8{\cdot}0]$

(The latter example is the loop from 015928 to 015A28 in Figure 2.)

Cursory examination of other programs led us to believe that the above seventeen examples are fairly representative of the programs now being written in FORTRAN, and that they indicate the approximate effects achievable with different styles of optimization (on our computer). Only one of the other programs we looked at showed essentially different characteristics, and this one was truly remarkable; it contained over 700 lines of straight calculation (see the excerpts in Figure 3) involving no loops, IF's or GO's! This must be some sort of record for the length of program text without intervening labelled statements, and we did not believe it could possibly be considered typical.

```
U23 =-ES12T*SETN + ES12B*SEBN                                         264.
U24 =-ES22T*SETN + ES22B*SEBN                                         265.
U3C = ES66T*SETN +. ES66B*SEBN                                        266.
U31 =-ES66T*SETN + ES66B*SEBN                                         267.
V3T =-2.*((ES11T+M*ES12T)*SXT+(M*ES22T+ES12T)*SYT)*C2XC2Y             268.
1       -2.*DSQRT(M)*TT*ES66T*S2XS2Y                                  269.
V3B =-2.*((ES11B+M*ES12B)*SXB+(M*ES22B+ES12B)*SYB)*C2XC2Y             27C.
1       -2.*DSQRT(M)*TB*ES66B*S2XS2Y                                  271.
V4T =-8.*((ES11T+M*ES12T)*SXT+(M*ES22T+ES12T)*SYT)*C4XC4Y             272.
1       -8.*DSQRT(M)*TT*ES66T*S4XS4Y                                  273.
V4B =-8.*((ES11B+M*ES12B)*SXB+(M*ES22B+ES12B)*SYB)*C4XC4Y             274.
1       -8.*DSQRT(M)*TB*ES66B*S4XS4Y                                  275.
V5T =-2.*((9.*ES11T+M*ES12T)*SXT+(M*ES22T+9.*ES12T)*SYT)*C2XC6Y       276.
1       -6.*DSQRT(M)*TT*ES66T*S2XS6Y                                  277.


A(3) = -A11*ML2*2.*XI1 - 4.*A22*ML2*4.*XI2 - A13*ML2*2.*XI1           604.
1      +T1*64.*XI3                                                    605.
2      -TML2C*(A11 + 2.*A22 + A13)                                    606.
3      -4.*FY*SB02*ML2                                                607.
A(4) = -ML2*(2.*XI1*(XI2+XI3)-BETA*XI1/LSC )-A11*Q2S                  608.
1      -TML2C*(XI1/4. +XI2 + XI3)                                     609.
2      +(HX*(K11B+M*K12BX)*SA11+M*HY*(M*K22B+K12BY)*SB11              61C.
3          +HXY*M*K66B*SC11)/2.                                       611.
4      +Y3                                                            612.
A(5) = -ML2*XI3*16. - A22*Q2S*16.                                     613.
1      = TML20*2.*XI3                                                 614.
2      +Y4                                                            615.


B(14,16)= +Y1315                                                     959.
B(14,17)= +Y1316                                                     960.
B(15,14)= B(14,15)                                                   961.
B(16,14)= B(14,16)                                                   962.
B(17,14)= B(14,17)                                                   963.
B(15,1) = 0.                                                         964.
B(15,2) = 0.                                                         965.
B(15,3) = -4.*ML2*HY                                                 966.
B(15,15) = -HY*HY*MSQ*D11B/(2.*DB) + Y1414                           967.
B(15,16)= Y1415                                                     968.
B(15,17)= Y1416                                                     969.
B(16,15)= B(15,16)                                                   970.
B(17,15)= B(15,17)                                                   971.
B(16,1)= +ML2*HXY                                                    972.
B(16,2) = C.                                                         973.
B(16,3) = C.                                                         974.
B(16,16) = -HXY*HXY*M/(4.*C66B) + Y1515                              975.
```

Figure 3. Excerpts from a remarkable program

All but one of the DO loops in the above examples apparently have variable bounds, but in fact the compiler could deduce that the bounds are actually constant in most cases. For instance in Example 17, N is set equal to 805 at the beginning of the program and never changed thereafter.

Table III summarizes the score ratios obtained in three examples; 0/1 denotes the ratio of the score for level 0 to the score for level 1, etc.

It may be objected that measurement of the effects of optimization is impossible since programmers tend to change the style of their FORTRAN programs when they know what kind of optimizations are being done for them. However, the programs we examined showed no evidence that the programmers had any idea what the compiler does, except perhaps the knowledge that '1' is or is not converted to '1·0' at compile time when appropriate. Therefore we expect that such feedback effects are very limited.

Table III. Execution speed ratios with various types of optimization

Example	0/1	0/2	0/3	0/4	1/4	2/4	3/4
1a	1·3	2·7	4·7	5·4	4·1	2·0	1·1
1b	1·8	3·0	4·3	4·8	2·7	1·5	1·1
2	1·2	2·6	4·6	6·6	5·4	2·6	1·4
3	1·3	2·3	6·8	7·4	5·6	3·2	1·1
4	1·1	1·5	1·6	4·4	3·9	3·0	2·8
5	2·5	9·0	9·4	13·1	5·2	1·5	1·4
6	1·5	2·0	2·1	5·4	3·6	2·6	2·5
7	1·9	2·4	2·5	2·9	1·5	1·2	1·1
8	1·1	2·5	2·5	3·4	3·0	1·3	1·3
9	1·2	2·8	3·5	3·5	2·9	1·3	1·0
10	1·3	2·8	5·3	8·0	6·3	2·9	1·5
11	1·3	2·0	2·2	2·4	1·8	1·2	1·1
12	2·2	3·3	3·3	3·8	1·8	1·1	1·1
13	1·2	1·4	1·8	3·9	3·3	2·7	2·1
14	1·1	1·6	2·1	2·3	2·1	1·4	1·1
15	1·3	2·3	2·8	3·3	2·5	1·4	1·1
16	1·1	1·1	1·1	1·5	1·4	1·3	1·3
17	1·3	2·9	4·6	5·1	3·9	1·8	1·1

Note that level 3 and level 4 programs ran four or more times as fast as level 0 programs, in about half of the cases. Level 3 was not too far from level 4 except in Examples 4 and 6 where short subroutine code was expanded in line; by incorporating this technique and the idea of replicating short loops, level 3 would come very close indeed to the 'ultimate' performance of level 4 optimization. (Before conducting this study, the author had expected a much greater difference between levels 3 and 4 and had been experimenting with some more elaborate schemes for optimization, capable of coming close to the level 4 code in the binary search example above. But the sample programs seem to show that existing optimization techniques are good enough, on our computer at least.)

5. SUMMARY AND CONCLUSIONS

Compiler writers should be familiar with the nature of programs their compiler will have to handle. Besides constructing 'best cases' and 'worst cases' it is a good idea to have some conception of 'average cases'. We hope that the data presented in this paper will help to give a reasonably balanced impression of the programs actually being written today.

Of course every individual program is atypical in some sense, yet our study showed that a small number of basic patterns account for most of the programming constructions in use. Perhaps these programs can be used to make a more realistic comparison of compiler and

machine speeds than is obtained with the 'GAMM test'.[17] See also P. Bryant's comparison of FORTRAN compilers summarized in Reference 7, pp. 764–767.

Our sample may not be correct, and so we hope people in other parts of the world will conduct similar experiments in order to see if independent studies yield comparable results.

While gathering these statistics we became convinced that a comparatively simple change to the present method of program preparation can make substantial improvements in the efficiency of computer usage. The program profiles (i.e. collections of frequency counts) which we used in our analyses turned out to be so helpful that we believe profiles should be made available routinely to all programmers by all of the principal software systems.

The 'ideal system of the future' will keep profiles associated with source programs, using the frequency counts in virtually all phases of a program's life. During the debugging stage the profiles can be quite useful, e.g. for selective tracing; statements with zero frequency indicate untested sections of the program. After the program has been debugged it may already have served its purpose, but if it is to be a frequently used program the high counts in its profile often suggest basic improvements that can be made. An optimizing compiler can also make very effective use of the profile, since it often suffices to do time-consuming optimization on only one-tenth or one-twentieth of a program. The profile can also be used effectively in storage management schemes.

In early days of computing, machine time was king, and people worked hard to get extremely efficient programs. Eventually machines got larger and faster, and the pay off for writing fast programs was measured in minutes or seconds instead of hours. Moreover, in considering the total cost of computing, people began to observe that program development and maintenance costs often overshadowed the actual cost of running the programs. Therefore most of the emphasis in software development has been in making programs easier to write, easier to understand and easier to change. There is no doubt that this emphasis has reduced total system costs in many installations, but there is also little doubt that the corresponding lack of emphasis on efficient code has resulted in systems which can be greatly improved, and it seems to be time to right the balance. Frequency counts give an important dimension to programs, showing programmers how to make their routines more efficient with comparatively little effort. A recent study[18] showed that this approach led to an elevenfold increase in a particular compiler's speed. It appears useful to develop interactive systems which tell the programmer the most costly parts of his program and which give him positive reinforcement for his improvements so that he might actually *enjoy* making the changes! For most of the examples studied in the previous section we found that it was possible for a programmer to obtain noticeably better performance by making straightforward modifications to the inner loop of his FORTRAN source language program.

In the above remarks we have implicitly assumed that the design of compilers should be strongly influenced by what *programmers* want to do. An alternate point of view is that programmers should be strongly influenced by what their *compilers* do; a compiler writer in his infinite wisdom may in fact know what is really good for the programmer and would like to steer him towards a proper course. This viewpoint has some merit, although it has often been carried to extremes in which programmers have to work harder and make unnatural constructions just so the compiler writer has an easier job. When weighted frequency counts are supplied to a programmer, it will become clear to him just which aspects of a language the implementor has chosen to treat most efficiently; the reporting of this information seems to be the best way to exert a positive influence on the users of a language.

The results of our study suggest several avenues for further research. For example, additional static and dynamic statistics should be gathered which are more meaningful with

respect to local optimizations. A more sophisticated study of these statistics would also be desirable.

Our survey seems to have given a reasonably clear picture of FORTRAN as it is now used. Other languages should be studied in a similar way, so that software designers can conceptualize the notion of 'typical' programs in COBOL, ALGOL, PL/I, LISP, APL, SNOBOL, etc.

We found that well-done optimization leads to at least a four- or five-fold increase in program speed (exclusive of input/output editing) over straight translation in about half of the programs we analysed. This figure is based on a computer such as the 360/67 at Stanford, and it may prove to be somewhat different on other configurations; it would be interesting to see how much different the results would be if the seventeen examples were worked out carefully for other types of computers. Furthermore, a study of the performance gain which would be achieved by in-line format editing is definitely called for.

As we discussed the example programs we saw many occasions where it is natural for compiler optimization to be done interactively. The programmer could perhaps be asked in Example 11 whether or not J will be non-negative and less than 2^{24} throughout the loop (so that $J = J + 1$ can be done with a 'load address' instruction); in Example 8 he might be asked whether the distributive law could be used on his formulae; in Example 7 he might be asked if $X**2 + Y**2$ can ever overflow (if not, this calculation may be taken out of the loop); and so on.

As the reader can see, there is considerable work yet to be done on empirical studies of programming, much more than we could achieve in one summer.

APPENDIX

Examples of hand translation

The following code was produced from

```
      DO 2 J = 1, N
      T = ABS(A(I, J))
      IF (T − S) 2, 2, 1
    1 S = T
    2 CONTINUE
```

using the various styles of hand translation described in the fourth section. Only the inner loop is shown, not the initialization.

Level 0

			Cost
Q1	ST	5, J	2
	L	3, J	2
	M	2, = A(AROWS)	7
	A	3, I	2
	SLL	3, 2	2
	LE	0, A(3)	2
	LPER	0, 0	1
	STE	0, T	2
	LE	0, T	2
	SE	0, S	3
	BNH	L2	1·5
	B	L1	2 × 0·5
L1	LE	0, T	2 × 0·5
	STE	0, S	2 × 0·5
L2	L	5, J	2
	A	5, = F'1'	2
	C	5, N	2
	BNH	Q1	2

A 'dedicated' use of registers, and a straightforward statement-by-statement approach, are typical of level 0.

Level 1

			Cost
Q1	ST	5, J	2
	LA	3, AROWS	1
	MR	2, 5	6
	A	3, I	2
	SLL	3, 2	2
	LE	0, A(3)	2
	LPER	0, 0	1
	STE	0, T	2
	CE	0, S	2
	BNH	L2	1·5
L1	LE	0, T	2 × 0·5
	STE	0, S	2 × 0·5
L2	LA	5, 1(0, 5)	1
	C	5, N	2
	BNH	Q1	2

Note the uses of LA and CE, the knowledge of register contents, and the removal of the redundant branch. The redundant LE in location L1 is still present because the occurrence of a label potentially destroys the register contents.

Level 2

			Cost
Q1	LE	0, 0(0, 3)	2
	LPER	0, 0	1
	LER	4, 0	1
	SER	4, 2	2
	BNH	L2	1·5
L1	LER	2, 0	1 × 0·5
L2	A	3, = A(AROWS*4)	2
	C	3, SPEC	2
	BNH	Q1	2

Here SPEC contains the precomputed address of A(I, N); S is maintained in floating register 2.

Level 3

			Cost
Q1	LE	0, 0(0, 3)	2
	LPER	0, 0	1
	CER	0, 2	1
	BNHR	2	1·5
L1	LER	2, 0	1 × 0·5
L2	BXLE	3, 4, Q1	2

Here register 2 is preloaded with the address of L2 (for a microscopic improvement), and registers 4 and 5 are preloaded with appropriate values governing the BXLE.

Level 4

			Cost
Q1	LE	0, 0(0, 3)	2 × 0·5
	LPER	0, 0	1 × 0·5
	CER	0, 2	1 × 0·5
	BNHR	2	1·5 × 0·5
L1.1	LER	2, 0	1 × 0·25
L2.1	LE	0, 4(0, 3)	2 × 0·5
	LPER	0, 0	1 × 0·5
	CER	0, 2	1 × 0·5
	BNHR	6	1·5 × 0·5
L1.1	LER	2, 0	1 × 0·25
L1.2	BXLE	3, 4, Q1	2 × 0·5

Since the loop program is so short it has been duplicated, saving half of the BXLE's, when proper initialization and termination routines are appended. (The code would have been written

Q1	LE	0, 0(0, 3)
	LPER	0, 0
	CER	0, 2
	BHR	2
L2.1	LE	0, 4(0, 3)
	LPER	0, 0
	CER	0, 2
	BHR	6
L2.2	BXLE	3, 4, Q1
	...	
L1.1	LER	2, 0
	B	L2.1
L1.2	LER	2, 0
	B	L2.2

if the frequency counts of this program would have given less weight to statement 1.)

Note that the FORTRAN convention of storing arrays by columns would make these loops rather inefficient in a paging environment; a compiler should make appropriate changes to the storage mapping function for arrays in such a case.

REFERENCES

1. D. E. Knuth, 'RUNCIBLE—algebraic translation on a limited computer', *Communs Ass. comput. Mach.* **2**, 18–21 (1959).
2. P. Calingaert, 'System performance evaluation: survey and appraisal', *Communs Ass. comput. Mach.* **10**, 12–18 (1967).
3. V. G. Cerf, *Measurement of Recursive Programs*, Ph.D. Thesis, School of Engineering and Applied Science, University of California, Los Angeles, California, Report 70-43, 1970.
4. D. N. Freeman, 'Error correction in CORC, The Cornell computing language', *AFIPS Fall jt comput. Conf.* **26**, 15–34 (1964).
5. P. G. Moulton and M. E. Muller, 'DITRAN—a compiler emphasizing diagnostics', *Communs Ass. comput. Mach.* **10**, 45–52 (1967).
6. B. A. Wichmann, *Some Statistics from ALGOL Programs*, National Physical Laboratory, Central Computer Unit Report, 1970.
7. J. Cocke and J. T. Schwartz, *Programming Language and their Compilers*, 2nd edn., Courant Institute of Mathematical Science, New York University, 1970.
8. J. von Neumann and H. H. Goldstine, *Planning and Coding of Problems for an Electronic Computing Instrument*, Institute for Advanced Study, Princetown, New Jersey (3 vols.), 1947–1948. Reprinted in von Neumann's *Collected Works* (Ed. A. H. Taub), Vol. 5, Pergamon, Oxford, 1963, pp. 80–235.
9. D. Ingalls, 'FETE—a FORTRAN execution time estimator (in preparation).
10. E. C. Russell, Jnr., *Automatic Program Analysis*, Ph.D. Thesis, School of Engineering and Applied Science, University of California, Report 69-12, 1969.
11. E. Satterthwaite, *Source Language Debugging Tools*, Ph.D. Thesis, Stanford University (in preparation).
12. T. Y. Johnston and R. H. Johnson, *Program Performance Measurement*, SLAC User Note 33, Revision 1, Stanford University, California, 1970.
13. R. S. Gaines, *The Debugging of Computer Programs*, Ph.D. Thesis, Princetown University, 1969, Chap. 3.
14. F. E. Allen, 'Program optimization', *A. Rev. autom. Progrmg.* **5**, 239–307 (1969).
15. B. A. Wichmann, *A Comparison of ALGOL 60 Executive Speeds*, National Physical Laboratory, Central Computer Unit Report 3, 1969.

16. IBM System/360 FORTRAN IV Library: Mathematical and Service Subprograms, File number S360-25, Form C28-6818-0, Table 13, Model 65, 1969.
17. E. Schmid, 'Rechenzeitenvergleich bei Digitalrechnern, *Computg.* **5**, 163–177 (1970).
18. S. C. Darden and S. B. Heller, 'Streamline your software development', *Comput. Decisions*, **2**, 29–33 (1970).
19. D. E. Knuth, *Fundamental Algorithms*, *The Art of Computer Programming*, Vol. 1, Addison–Wesley, Reading, Massachusetts, 1968.

CORRIGENDA

Owing to the U.K. postal strike it was not possible to include the following corrections in the text as published.

Page 116, line 37, change 'the running time' to 'the (level 0) running time'.

Page 118, in Example 4 after 'SUBROUTINE RAND(R)' insert the line 'COMMON I,K'.

Page 119, after line 11, insert the new sentence (no paragraph): 'Indeed, a careful FORTRAN programmer would never convert to floating point at all in the above use of RAND, he would simply compare J to the constant $\cdot 81 \times 2^{31}$.'

Page 120, line before Example 6, change 'is necessary' to 'is a necessary auxiliary variable'.

Page 126, line 6, before 'This must be' insert the new sentence: '(Idiosyncrasies of spacing indicate that it was generated by hand, not by program.)'