SOME STATISTICS FROM ALGOL PROGRAMS

B. A. WICHMANN
National Physical Laboratory, Teddington, Middlesex, England

CCU REPORT NO.11, August 1970

Abstract

A substantial amount of statistical information is given of Algol programs run at the National Physical Laboratory. This involved the monitoring of 155 million instructions. Information from this can be used by compiler-writers in deciding which features of a compiler are worth optimising. Statistics are also given of the occurrence of basic symbols in source text and of instructions in compiled programs.

From the execution statistics, an Algol mix has been devised. This has been run on 18 Algol systems giving a clear indication of the weaknesses and merits of each one, as well as a comparison of their execution speeds.

Contents

1 Introduction

A large part of the statistics gathered and analysed in this report are dependent on the Algol compiler described in Randell and Russell's book "ALGOL 60 Implementation" [1] . Fortunately this book contains sufficient detail for those without access to a KDF9 to make their own analysis of the statistics given in this report.

Three general methods have been used to gather statistics from the KDF9 Whetstone Algol system. Firstly, the interpreter has been modified to obtain dynamic counts of each interpretive instruction. Secondly a static count of these instructions has been made. Thirdly, the Algol source text, stored as files on the systems disc can be scanned and analysed in various ways.

This work has been done over a period of nearly two years. Much help has been given by various people including A.L. Hillman, R. Green, and M. Parsons. Additional statistical information has also been provided by Dr. C. Phelps of Oxford University which is analysed in section 2. Many helpful comments principally by R. Scowen and M. Woodger have resulted in amendments to the work. Also additional timing data have been sent to the author since the publication of [3] which appear in 5.4.

2 Whetstone Algol dynamic analysis

2.1 Introduction

The reader is expected to be familiar in outline with Randell and Russell's book ALGOL 60 Implementation [1]. References to the book use the section heading preceded by RR, e.g.:- RR 2.6.2. The interpretive instructions are expanded with the abbreviation left in capital letters, for instance Make Storage Function, which is written as MSF in [1] and in parts of this report where the context is clear.

Execution of the object program operations is implemented with one switch on the current operation number. This part of the interpreter can be modified to add up the number of times each type of operation has been executed. The interpreter is extremely slow to execute — typical instruction times are 300 to 800 microseconds. For this reason adding 40 microseconds to accumulate the dynamic counts does not degrade the system significantly. Consequently this modification can easily be used on all programs submitted to the Whetstone system. This avoids the sampling involved in some methods.

At first, the modified interpreter output the counts of each program to paper tape. Almost all programs were monitored in three days separated by a fortnight in order to capture different programs. A similar

modification was done at Oxford University by Dr. C. Phelps. Later, the modification was repeated but the counts were accumulated automatically on the disc. Because of this, it was possible to monitor programs continuous for over a month. A total of nearly a thousand programs were monitored although a substantial number of reruns means that only about two hundred of these were distinct.

In interpreting these results, one must bear in mind the environment within which these figures were obtained. The National Physical Laboratory is a research establishment in which in general, the scientists do their own program writing and debugging. Program development takes a large share of the available computer time, almost all of which is in ALGOL. The Whetstone Algol system is used only for program development and production runs use the Kidsgrove compiler [2]. It is thought that a fairly high proportion of the users have little programming experience, and that this is reflected in the programs they write.

Since Whetstone ALGOL is used for program development only, production programs are likely to show different properties. Some of these can be guessed at. For instance, an operator like Make Storage Function will probably only be executed a few times per program regardless of the amount of computing time used. One would imagine that the use of the standard input/output routines would not increase linearly with the amount of computer time used.

2.2 Outline of the statistics

The complete table of statistics on the interpreted instruction is given in Appendix A. For the purpose of the calculation of variance, the figures have been divided into seven groups as follows:

- 1) Three days separated by a fortnight at NPL, collected in October and November 1960
- 2) Figures collected Oxford University during April 1969 by Dr. C. Phelps.
- 3) to 7) Figures collected automatically at NPL during February March and April 1970. These have been divided into groups representing about 7 to 10 days use of the Whetstone Algol system.

Sometimes it is appropriate to give all seven figures, in which case it is written as “total (f1, f2, f3, f4, f5, f6, f7)”. Where appropriate, figures are round to three places. Usually one is interested in the frequency of execution of one of the interpreted instructions in which case the average of the frequency from the seven samples is given. The variance of the seven frequencies as a percentage is then given in brackets as an indication of the reliability of the information. For instance, the frequency of execution of Block Entry is 13 400 (26) operations per million. As expected, the more frequent operations have a smaller variance (typically 10 per cent) as opposed to the less frequent ones with a variance of often 100 per cent. In general, the number of programs in the seven samples is an indication of the comprehensiveness of the data. In the case of the first sample, rather more care was taken over its collection whereas the other ones are merely all programs submitted over a particular period of time. The last four samples are in date order and so, because of the gradually changing population of programs being run, the correlation between samples might be expected to correspond to the difference of the sample numbers. A good check on the statistics would be to see if the Oxford University figures are significantly different from those at NPL.

The number of programs executed was 949 (93, 120, 122, 349, 71, 121, 120). The number of elementary operations in each set was 155 (12, 6.3, 16.3, 67, 12, 25.4, 15.8) millions. Hence the average number of operations in each program was 152 000 (165 000, 63 500, 133 000, 192 000, 170 000, 210 000, 132 000). The average time taken to execute one instruction was 664 (738, 761, 598, 593, 616, 655, 687) microseconds. The percentage of programs to terminate without error was 54 (58, 55, 51, 55, 41, 49, 63).

2.3 Analysis of the dynamic counts

The figures give very accurate details of some aspects of the use of ALGOL. For instance, the number of times each standard function was used is known. But in other cases the operation counts give little information. For example, type conversion is not explicit so it is not known how often a floating point number has to be fixed to store it in core. Also, the operations INDA and INDR do not give the dimensions of the arrays. So, in order to proceed further with some analysis it is necessary to make guesses about some of the missing information.

Operator	*Frequency*	*Variance*
$\times$	36 800	(9)
$+$	31 000	(14)
$-$	26 300	(11)
$/$	11 000	(27)
$=$	6 980	(26)
NEG	5 230	(21)
$>$	3 600	(24)
$\uparrow$	3 530	(59)
$<$	3 290	(38)
$\neq$	2 230	(56)
$\vee$	2 180	(32)
$\leq$	1 030	(70)
$\geq$	914	(86)
$\wedge$	884	(74)
$\div$	481	(70)
$\neg$	115	(107)
$\equiv$	1	(241)
$\supset$	0	(infinite)

Table 1: Frequency of execution of the Algol operators

Operator	*Frequency*	*Variance*
Take Integer Constant 1	43 200	(4)
Take Integer Constant	22 500	(33)
Take Integer Constant 0	6 630	(25)
Take Real Constant	5 450	(44)
Take Boolean Constant False	124	(68)
Take Boolean Constant True	78	(126)

Table 2: The constant operations

2.3.1 Operations

The frequency of arithmetic and boolean operations can be found directly from corresponding elementary operation frequencies and are listed in Table 1.

Unfortunately the type of the operands is not known, for instance $\times$ is executed for real or integer operands or indeed a mixture in which case the integer operand is converted to real. A rough guess is attempted on the ratio of real to integer working based upon other operations (see section 2.3.9 and 2.4).

The fact that multiply appears higher in the table than add does not mean that the machine instructions for multiplication are likely to be used more often than those for addition. In fact, addition occurs more frequently than multiplication because if its use in address calculation (INDex Address, INDex Result) and storage allocation (Make Storage Function, Procedure Entry, Block Entry, Call Block etc). The possibility of calculating the frequency of execution of various machine instructions in Algol compiled code is examined in detail in section 5.4.2.

2.3.2 Use of constants

Whetstone Algol inserts constants in the code when required, so it is not possible to find out how much storage would be saved if each different constant appeared only one.

The frequency of the relevant operations is as in Table 2.

In analysing these, allowance must be made for the fact that constants as actual parameters to proce-

dures do not give rise to these operations. The use of special operation-codes for 0 and 1 is clearly fully justified, but it is thought that a large proportion of the integer constants not 0 or 1 are in fact small integers.

2.3.3 Labels and Switches

Compared with implicit transfers, labels are very rarely used.

The use of switches was only about five per cent of the **goto**'s: Frequency of **goto**'s = 2 010 (51) from GoTo Accumulator Frequency of the use of switches = 94 (105) from Take Switch Address.

The average value of the switch index can be seen from the ratio (Decrement Switch Index/Take Switch Address) which is about fifteen (with a large variation).

One can also tell how frequently labels are passed in a program since the operation TRACE is generated for each call of a procedure (Procedure Entry) or passing a label.

Frequency of passing a label = 3 370 (44)

Percentage of **goto**'s/passing a label is 58 with a standard deviation of 8 per cent from the seven samples (which is surprisingly constant in view of the wide variation of the frequencies).

2.3.4 For loop code

The **for** loop control code is rather complex, so a careful reading of RR2.6 is advisable.

On initialising the **for** loop, Unconditional Jump, Call Function Zero and For Block Entry are executed. For each arithmetic element two LINK and one For Arithmetic is executed together with For Return on completion of the controlled statement. For each **while** element three LINK's one For While and then For Return is executed. The **step until** elements is more complex in that four LINK's FORS1 and For Return are used first time round the loop and four LINK's, FORS2, and For Return subsequently.

The figures reveal as expected that the **step until** element is much more frequently used than the others. Other facts can also be deduced.

Frequency of entry to **for** loops For Block Entry = 2700 (24)

Frequency of exits from **for** loops by exhaustion (For Statement End) = 2510 (27)

Percentage of **for** loops from which an abnormal exit has been made is 3.6 (106)

It is not possible to find out how many arithmetic elements are in a **for** loop, or how many times **while** is obeyed, but with the **step until** this is possible.

Average number of times round **step until** for loop (FORS1/FORS2) 7.9 (15) This is a number which one would expect to increase substantially for production runs as opposed to the program testing which was monitored.

2.3.5 Procedure entry code

Because the correspondence between actual and formal parameters is checked at run time (RR 2.5.6), an analysis of parameters is possible. However the parameter operations are not executed directly, so an analysis is only possible of the formal parameters by inspecting the frequency of the check operations.

The table 3 gives a list of all the check operations and their frequency. Also tabulated is the "average" number of parameters of the check-type, that is, frequency/procedure entry count.

The sum of the check operations gives the average number of parameters to procedures, which is 1.88 (19).

Procedures without parameters These are called by Call Function Zero and Call Formal Function Zero instead of Call Function and Call Formal Function. Unfortunately CFZ is also used in **for** statements, but the use of CFZ and and CFFZ to call functions without parameters can be found from (PE-CF-CFF). This has a mean of 788 (104) operations per million. The percentage of calls of procedures without parameters out of all procedures averages at 4.33 with a variance of 108.

Operation	*Meaning*	*Frequency*	*Variance*	*Frequency/ Proc. Entry*	*Variance (Freq/PE)*
Procedure Entry		20 100	26		
Check and Store Real	real by value	18 100	26	.922	26
Check and Store Integer	integer by value	13 100	26	.664	17
Check Arithmetic	real/integer by name	2 480	79	.140	89
Check STring	string	1 790	76	.089	54
Check Array Real	real array by name	656	34	.036	45
Check Array Integer	integer array by name	471	180	.018	160
Check and Store Boolean	boolean by value	113	93	.006	very large
Check Label	label by name	59	91	.003	very large
Copy Real Formal Array	real array by value	43	85	.002	very large
Check Boolean	boolean by name	24	170	.001	very large
Check PRocedure	procedure	9	153	very small	
Check Function Real	real procedure	3	169	very small	
Check and Store Label	label by value	2	208	very small	
Copy Integer Formal Array	integer array by value	0	223	very small	
Check Function Boolean	boolean procedure	not used			
Check Function Integer	integer procedure	…			
Check Switch	switch	…			
Check Array Boolean	boolean array by name	…			
Copy Boolean Formal	boolean array by value	…			

Table 3: Procedure Entry and Check Operations

Code procedures The standard functions are not dealt with as ordinary code procedures and are considered in the next section. Although the user can write code procedures, the most frequently used ones are almost certainly “write (integer, integer, real)”, “read (integer)”, “write text (integer, string)”, where all the parameters (except the string) are by value.

At Oxford University, the Whetstone Algol system is somewhat different so that these input-output procedures are treated as standard functions. In this case, it is possible to determine the percentage of procedure calls which are for standard input-output routines. This is 59 per cent.

A type procedure is entered by the interpretive code R DOWN, I DOWN or B DOWN as appropriate, but ordinary procedures are entered by the use of R DOWN.

The percentage of these operations to the total use of procedures is

- R DOWN 40.7 (38)
- I DOWN 4.2 (large)
- B DOWN 0 (very large)

(The R DOWN value from Oxford has been corrected to do this calculation).

Standard functions The standard functions are dealt with as ordinary procedures except that the body of the procedure is a single syllable operation code unique to that function. Hence the use of each of the standard functions can be determined from the tables and is repeated in Table 4.

The total for alI the standard functions 8 718 (56) operations per million.

As a percentage of the use of procedures, the used standard functions is 41.8 per cent with a variance of 37.

From the last section, the total use of code procedures and standard functions is 86.8 per cent of all procedures (variance only 9). This ratio can also be determined from (RETURN-CBL)/PE, which gives the fraction of pure Algol procedures (assuming procedures were left by the ordinary mechanism (not by goto etc)).

Operator	*Frequency*	*Variance*
SQRT	1 750	(49)
COS	1 490	(101)
ABS	1 390	(40)
SIN	1 020	(108)
ENTIER	909	(122)
EXP	831	(92)
LN	644	(83)
ARCTAN	591	(149)
SIGN	82	(82)

Table 4: The standard functions

Operator	*Frequency*	*Variance*
Take Formal Address	24 400	(39)
Take Formal Real	2 690	(40)
Take Formal Address Integer	1 890	(42)
Take Formal Integer	1 820	(109)
Take Formal Address Real	1 700	(93)
Take Formal Boolean	85	(143)
Take Formal Label	0	(151)

Table 5: The Take Formal operations

2.3.6 Use of formals

Within the procedure, the use of formals specified by name can be found from inspecting the frequency of the relevant Take Formal operations (RR 2.5.6), see Table 5. The use of formals specified by value cannot be separated from the use of non-formals.

It is possible to see how many times on average each formal is accessed. The operations TFI, TFAI, TFR and TFAR correspond to a formal which has an appropriate Check Arithmetic operation. Hence the average number of references to arithmetic call by name parameters is 4.46 (49).

The repeated use of arrays called by name can be found from the ratio TFA/(CAR + CAI + CAB) = 29.4 (52).

(TFA is in fact used in assignments to formal Booleans, and also in the use of formal switches, but their frequency is so low that this can fairly safely be ignored.)

2.3.7 Expressions as actual parameters

The calling sequence with an expression as a parameter consists of a subroutine starting with Block Entry and ending with End Implicit Subroutine (RR 2.5.4.3). Although Block Entry occurs on entry to blocks (RR 2.2.3) and also with End Implicit Subroutine in switch declarations (RR 2.4.2) these occurrences are negligible in comparison to their use in expression parameters. The total number of parameters which could have had expressions as parameters is known from the relevant check operations, namely CSI, CSR, CA, CL, CSB, CSL and CB. However in the case of CA, CL and CB the expression could have been evaluated more than once, and so the counts of the relevant take formal operations must be added instead (TFI, TFR, TFB, TFL, TFAI, TFAR). (The use of TFA in the assignment to formal booleans is again ignored.)

Hence the percentage (on the basis of execution) of expression parameters is 35.2 (16).

2.3.8 Unconditional Jumps

The Unconditional Jump code appears in about 7 different contexts as follows:

Context	*Frequency*	*Variance*
1	197	(51)
2	98	(51)
3	2 067	(34)
4	0	(-)
5	94	(105)
6	19 528	(29)
7	2 594	(23)
TOTAL	24 578	(20)

Table 6: Unconditional jumps by context

1. After Call BLock (RR 2.2.7) and so has exactly the same count as CBL
2. After Block Entry (RR 2.2.8) in order to jump round procedure bodies. For two or more consecutive procedures there is only the one jump, so allow about half the CBL count for this.
3. Jump round **else** in conditional expressions or conditional statements (RR 2.1.5 and 2.1.6). If the conditional statements always had and **else** one would expect a count of about half of If False Jump for this.
4. Jump round switch declarations (RR 2.4.2). One would expect count for this to be much less than CBL and so negligible.
5. Jump to end of switch declaration after evaluating a switch designator (RR 2.4.2). Same frequency as Take Switch Address.
6. Jump round actual operations (RR 2.5.3) Exactly the same as the Call Function plus Call Formal Function count.
7. Jump round address calculation of control variable in **for** loop (RR 2.6.1). This is the same as the count for Call Function Zero in this context.

But since Call Function Zero can arise in the call of procedures without parameters, this must be subtracted (see 3.5.1).

Since only the jumps of type 3 cannot be estimated with any accuracy, the total jump count can be used to find their frequency, completing the table 6.

The ratio of 3 to If False Jump is 13.3% (variance 21). Since this is significantly less than half, it appears that conditional statements are more common than expressions and that the **else** clause is often not present (or the condition is usually false, which seems unlikely).

Note that a large number of transfers in fact implicit, such as RETURN, and For Return, and, of course, the evaluation of expression parameters which starts with BE and ends with EIS.

2.3.9 Analysis of arithmetic variable store and fetch

One would clearly like to know the proportion of real to integer fetchs and stores and also the proportion of array element accessing to simple variables. Unfortunately the operation codes do not give these figures directly, for instance INDex Address and INDex Result do not give the dimension of the arrays accessed. So a very approximate analysis has been done.

Since the boolean variables (as opposed to relational operators) are used very rarely, only integer and real quantities are considered. More significantly the use of formals in the operation Take Formal Address is omitted although an allowance has been made for this by adjusting the Take Real Address and Take Integer Address figures for array accessing. The fact that INDex Result is used in switches is also ignored.

The **for** loop control code is dealt with rather differently from other statements, so the effect this has on the counts should be removed. Making the assumption that most of the **for** loops are of the form:

Use	*Proportion as percentage*	*Variance*
Integer	57.0	7
Real	43.0	9
Fetch	77.3	2
Store	22.7	6
1) Integers as subscripts	24.5	6
2) Integer in control loop	16.0	17
3) Integer array fetch	2.8	20
4) Integer array store	1.8	29
5) Real array fetch	9.2	9
6) Real array store	5.6	13
7) Integer fetch excluding 1 and 2	9.4	24
8) Integer array store excluding 2	2.3	18
9) Real fetch	20.7	17
10) Real store	7.6	16

Table 7: Variable usage

for i := 1 **step** 1 **until** n **do**

we have that TIA is used twice and TIR once for every FORS2 operation. Reducing these counts by the appropriate amount we have (TIA) modified = 16 800, (TIR) modified = 111 000.

If one further assumes that subscript expressions involve only one simple integer variable, and that the average array dimension is 1.26 (see section 4.1.2) then := (INDA + INDR) $\times$ 1.26.

The residue of the TIR is thus 80 400.

So a large proportion of integer work is involved in fetching subscripts.

The operations TIA and TRA are used to fetch array words, and TFA is used exclusively for this (again ignoring formal booleans etc.). Since INDA or INDR is executed for each array word fetch, one can reduce TIA and TRA by proportion to allow for this.

(TIA) corrected = 7 630, (TRA) corrected = 24 800.

These figures should therefore represent the approximate frequency of storing to simple integer and real variables.

These results can be summarised in the following way.

Total use of simple and array variables = 329 000 operations/million.

By making various assumptions about the use of variables table 7 can be constructed.

3 Whetstone Algol static analysis

3.1 Introduction

The Whetstone Algol system in use at this laboratory is the one working under the PROMPT operating system provided by ICL. The compiler reads the Algol text off the disc and compiles the program in its own core area. After successful compilation, three items are written back to the disc. First there is the Whetstone Algol interpretive code as described in [1] secondly there is a reference table giving the correspondence between syllable numbers in the interpretive code and the Algol text (it is used to give the positions of run-time failure), and lastly there is the machine-code resulting from the compilation of code procedures.

The compiler is not load and go, so there is a separate running phase in which a controller program reads the relevant blocks from the disc and interprets the Whetstone Algol code. In fact, any program can read this information off the disc so it is possible to write an Algol program to “anticompile” the interpretive code of any program.

Operation	*Address field*
Unconditional Jump	points to Call Function
30	
‘string’	
Parameter STring	points to ‘string’
Parameter Integer Constant	points to ‘30’
Call Function	

Table 8: Example of procedure call

3.2 The anticompiler

This program makes a single pass through the interpretive code, printing out the code in a binary format together with the instruction in the textual form for instance, Take Integer Result. The program will anti-compile several programs one after the other, printing out the total number of operations for each program and finally the total for all the programs. The anticompilation itself is not as straightforward as might be hoped. Each elementary operation is of the form

<function><operand>

where the function is 8 bits and the operand a multiple of 8 bits which is determined by the function. So the decoding table used by the anticompiler consists of the size of the operand and text for each function. The complication comes with procedure parameters where explicit constants and strings are stored with the code and so must be avoided. This can be done by inspecting the relevant parameter operations which precede the Call Function or Call Formal Function operation.

Example

Algol: write text (30, ‘string’) produces the interpretive code in table 8.

So the action of the anticompiler is as follows. Each operations is decoded in a simple manner until an Unconditional Jump operation is met. Simple decoding continues unless this points to a Call Function or Call Formal Function operation. In the case of a function call a backward scan is made of the parameter operations (the number of parameters is in the operand part of the call function instruction). When the parameter operation is a Parameter Boolean Constant, Parameter Real Constant, or Parameter Integer Constant the six syllables of the constant are skipped Over. If the operation is Parameter STring, then a skip is made to the next whole word, the string is printed in Algol basic symbol form, and another skip to the next whole word is made to get the anticompiler back in step. The most complex case is when the parameter operation is Parameter SubRoutine. Then a sequence beginning with Block Entry and ending with End Implicit Subroutine must be decoded. Since this can include a function call, various variables used in decoding the first function call must be stacked, and then unstacked by the End Implicit Subroutine. Note that End Implicit Subroutine can occur in switches and in this case no unstacking must be done (the stack depth is then zero).

3.3 Use of the anticompiler

40 programs were analysed using the anticompiler, but only the total number of operations for each program was printed. The anticompiler worked at about 120 elementary operations per second or about one third the speed of the Whetstone Algol compiler. This meant that this static analysis was substantially more expensive in computer time than the dynamic counts used in section 2. For this reason only 40 programs were taken.

The 40 programs had the following mean values: Program Text size 6.05 disc blocks or about 3890 48 bit words, Size of machine code 1097 words, Size of interpretive code 1031 words, Number of operations 1934, Number of syllables for each operation 3.3.

The main output was the frequency per thousand of the elementary operations (printed in Appendix A).

3.4 Remarks on the operation counts

Although only 40 programs were examined, the static counts are not so very unreliable. The reason for this is that there is no equivalent of a small "compute loop" to have a large effect on the counts which happens with the dynamic counts.

Some of the counts are best expressed in so many operations per program. For instance, the average program contains

- 2.5 Blocks (from Call BLock)
- 30 code procedures (from R DOWN + B DOWN + I DOWN)
- 2 standard functions (from abs, sqrt etc)
- 36 procedures (from Procedure Entry)

The parameter operations do appear in the static counts but not in the dynamic one since they are not executed directly.

Although the static and dynamic counts do not come from the same series of programs, the ratio between the two does show the expected behaviour. An obvious example is FORS1 and FORS2. These necessarily have the same static count, but the dynamic count for FORS2 is naturally very much larger than that for FORS1. So, because of the for loop mechanism in FORS1, For Block Entry, and For Statement End have a higher static count to dynamic count ratio. Similarly because of the procedure mechanism in Unconditional Jump, Call Function and REJECT have a high ratio. The ratio gives a compiler writer an indication of those parts of compiled code output on which he should concentrate in reducing code size rather than execution time.

4 Statistics from the Algol source text

On KDF9, program text is ordinarily kept on the disc in an internal Algol basic symbol code. For this purpose, the 116 Algol basic symbols are extended by the editing characters space, tab and newline and also some additional compounds symbols needed for the system namely **KDF9**, **ALGOL**, **EXIT**, **segment**, **library**, $\rightarrow$ and $\Rightarrow$. The symbols **KDF9** and **ALGOL** are used to bracket the body of procedures written in machine-code. The character $\rightarrow$ terminates each text file, while the others are used for other parts of the system which is not relevant to the present note.

Appendix B contains a table of the frequency of the KDF9 Algol basic symbols obtained from 200 programs and a total of 955 599 symbols. The program ignored symbols appearing between **KDF9** and **ALGOL** and between **comment** and ;. The other two types of comment were not ignored (**end** comment and parameter comments).

A further program was written to analysis the length of identifiers and digit sequences. The length of identifiers is distributed roughly on a negative exponential so that the probability of having an identifier of length n is half that of length $n-1$. The digit sequences were similarly distributed but with a local maximum around the machine accuracy of 11 decimal places.

4.1 Analysis of particular basic symbols

A further text analysis program was written to examine the use of particular 'rare' symbols. This worked merely by printing out every line containing that symbol. This program was used to examine for loops, array bounds, and the exponential and integer divide operators.

4.1.1 The for symbol

A total of 824 **for** loops were examined from 37 programs. The results are best summarized by means of two tables, 9 and 10.

	Number	*Percentage*
One **step - until** element	780	94.7
list of expressions	26	3.2
Containing **while**	7	.8
Other forms	11	1.3

Table 9: Overall structure of for loop

Type	*Number*	*Percentage*
(A) <constant> **step** 1 **until** <sv>	414	51.6
(B) <sv> 1 **until** <sv>	69	8.6
(C) <other> **step** 1 **until** <sv>	69	8.6
(D) <constant> **step** 1 **until** <other>	66	8.2
(E) <constant> **step** 1 **until** <constant>	55	6.8
(F) other loops with **step** 1	9	1.1
(A) to (F) that is, all **step** 1 loops	682	84.9
(G) <sv> **step** -1 **until** <constant>	20	2.5
(H) <other> **step** -1 **until** <constant>	34	4.2
(I) Other forms with **step** -1	36	4.5
(G)+(H)+(I) that is, all **step** -1 loops	90	11.2
(J) Other forms with **step** <constant>	8	1.0
(K) Other forms without constant step	22	2.7

Table 10: Classification of **step - until** elements

Since the analysis program only printed out one line of text, in some cases all the symbols between **for** and **do** were not output. This happened only with lists of expressions which were too long for a line, in which case it was assumed that the list continued without a **step - until** or while element.

For similar reasons, the types of variables used in the **for** loops are not known although the simple variables are thought to be integers except in two or three cases.

The only type of loop which is worth while analysing further is the **step - until** element. The 803 **step - until** elements were classified into groups by the complexity of the three expressions. The three types of expressions considered are

1. <constant> = explicit constant, possibly with a sign
2. <sv> = simple variable
3. <other> = expression but not the last two

The 803 **step - until** elements can be classified in table 10.

4.1.2 Use of the : symbol

Lines containing this symbol were printed out to determine values of the array bound pairs in array declarations.

A total of 272 bound pairs were found from 47 programs. Of these 137 had both bounds constant (possibly signed). Thee lower bounds were as follows:

- lower bound = 0, 57 times
- lower bound = 1, 208 times
- other constant 6 times (in fact, always equal to -1),

Number in list	*Total count*
1	127
2	31
3	24
4	10
5	4
6	5
7	4
8	6
9	1
13	1

Table 11: Bound-pair counts

- other once.

The dimension distribution was as follows:

- 161 one dimension
- 54 two dimensions
- 1 three dimensions and no others

So the average array dimension was 1.26.
The distribution of the number of arrays declared with one bound-pair list was as in Table 11.

4.1.3 The ↑ operator

A total of 524 ↑ symbols were found in 76 programs. 416 of these symbols were not (apparently) in comment or after the final end of the program (a system feature). These operators were used as follows:

- ↑ 2: 269 occurrences
- ↑ 3: 21 occurrences
- ↑ 4: 3 occurrences
- ↑ 5: 7 occurrences
- other 101 occurrences

Unfortunately it is not possible to tell with the 101 other cases the type of the exponent i.e. integer or real or that of the base.

4.1.4 The ÷ operator

This was comparatively rare compared with the last three symbols. Only 63 occurrences were found in 146 programs. These were divided up as follows:

- division by constant: 29,
- test to see if variable is divisible by a constant, for instance $i = i \div 4 \times 4$: 25,
- general: 9.

5 Weighting factors for the simple statements

5.1 Introduction

In a previous report [3], the author could not assign weighting factors to the simple statements used in a timing comparison. Hence it was not possible to give an overall figure of performance, since the relative significance of each of the statements was not known.

The counts presented in section 2 do allow one to estimate weights for these statements. So this section gives an example of how the operation counts can be used. As in the rest of this paper, the weights are calculated seven times for each set of counts. As before, the results are usually given in the form of an average and variance as a percentage of the seven figures.

Each of the simple statements can be thought of as a number of elementary operations. Clearly one wishes to find weights for each of the statements so that the weighted sum of all the elementary operations is as close as possible to the counts given in Appendix A. The degree of match should be measured in terms of computing time rather than operation numbers. Although this could be considered as a simple minimising problem, the weights can be estimated more easily, since many of the operation codes only occur in one or two statements.

It must be emphasised that, due to the very wide natural variation in Algol programs, the weights cannot be relied upon with any accuracy (as is indicated by the variance).

5.2 Operation codes for the simple timing statements

A list of the operation codes corresponding to the simple statements follows. The operation in brackets represents executed code placed apart from the open code of the statement itself.

1. $x := 1.0$, code: TRA ‘x’, TRC ‘1.0’, ST.
2. $x := 1$, code: TRA ‘x’, TIC 1, ST.
3. $x := y$, code: TRA ‘x’, TRR ‘y’, ST.
4. $x := y + z$, code: TRA ‘x’, TRR ‘y’, TRR ‘z’, +, ST.
5. $x := y \times z$, code: TRA ‘x’, TRR ‘y’, TRR ‘z’,$\times$, ST.
6. $x := y/z$, code: TRA ‘x’, TRR ‘y’, TRR ‘z’, /, ST.
7. $k := 1$, code: TIA ‘k’, TIC 1, ST.
8. $k := 1.0$, code: TIA ‘k’, TRC ‘1.0’, ST.
9. $k := l + m$, TIA ‘k’, TIR ‘l’, TIR ‘m’, +, ST.
10. $k := l \times m$, code: TIA ‘k’, TIR ‘l’, TIR ‘m’, $\times$, ST.
11. $k := l \div m$, code: TIA ‘k’, TIR ‘l’, TIR ‘m’, $\div$, ST.
12. $k := l$, code: TIA ‘k’, TIR ‘l’, ST.
13. $x := l$, code: TRA ‘x’, TIR ‘l’, ST.
14. $l := y$, code: TIA ‘l’, TRR ‘y’, ST.
15. $x := y \uparrow 2$, code: TRA ‘x’, TRR ‘y’, TIC ‘2’,$\uparrow$, ST.
16. $x := y \uparrow 3$, code: TRA ‘x’, TRR ‘y’, TIC ‘3’,$\uparrow$, ST.
17. $x := y \uparrow z$, code: TRA ‘x’, TRR ‘y’, TRR ‘z’,$\uparrow$, ST.
18. $e1[1] := 1$, code: TIA ‘e1’, TIC 1, INDA, TIC 1, ST.

19. $e2[1,1] := 1$, code: TIA ‘e2’, TIC 1, TIC 1, INDA, TIC 1 , ST.

20. $e3[1,1,1] := 1$, code: TIA ‘e3’, TIC1, TIC1, TIC1, INDA, TIC 1, ST.

21. $l := e1[1]$, code: TIA ‘l’, TIA ‘e1’, TIC1, INDR, ST.

22. **begin real** a; **end**, code: CBL, UJ, BE, RETURN.

23. **begin array** a[1:1]; **end**, code: CBL, UJ, BE, TIC1, TIC1, MSF, RETURN.

24. **begin array** a[1:500]; **end**, code: CBL, UJ, BE, TIC1, TIC ‘500’, MSF, RETURN.

25. **begin array** a[1:1, 1:1]; **end**, code: CBL, UJ, BE, TIC1, TIC1, TIC1, TIC1, MSF, RETURN.

26. **begin array** a[1:1, 1:1, 1:1]; **end**, code: CBL, UJ, BE, TIC1, TIC1, TIC1, TIC1, TIC1, TIC1, MSF, RETURN.

27. **begin goto** abcd; abcd: **end**, code: TL ‘abcd’, GTA, TRACE ‘abcd’.

28. **begin switch** ss := pq; **goto** ss[1]; pq: **end**, code: CBL, UJ, BE, UJ, BE, DSI, TL ‘pq’, UJ, ESL, EIS, TSA ‘ss’, TIC 1, INDR, GTA, TRACE ‘pq’, RETURN.

29. $x := sin(y)$, code: TRA ‘x’, UJ, PR ‘y’, CF ‘sin’. (TRACE ‘sin’, PE, CSR, ‘sin’.)

30. cos instead of sin.

31. abs instead of sin.

32. exp instead of sin.

33. ln instead of sin.

34. $sqrt$ instead of sin.

35. $arctan$ instead of sin.

36. $sign$ instead of sin.

37. $entier$ instead of sin.

38. $p0$, code: CFZ ‘p0’, REJECT.
    (TRACE ‘p0’, PE, RETURN).

39. $p1(x)$, code: UJ, PR ‘x’, CF ‘p1’, REJECT.
    (TRACE ‘p1’, PE, CSR, RETURN).

40. $p2(x, y)$, code: UJ, PR ‘x’, PR ‘y’, CF ‘p2’, REJECT.
    (TRACE ‘p2’, PE, CSR, CSR, RETURN).

41. $p3(x, y, z)$, code: UJ, PR ‘x’, PR ‘y’, PR ‘z’, CF ‘p3’, REJECT.
    (TRACE ‘p3’, PE, CSR, CSR, CSR, RETURN)

It has been thought worthwhile to add one additional calculation to the list of Algol statements. This is the loop code involved in **for** i := 1 **step** 1 **until** n **do**. The reason for this is that this code can be given a significant weight and the length of time is calculated as a by-product of timing the other statements. It is also shown that some compilers optimise step 1 to reduce this time whereas others do not. The additional code, notational for case 42: gives

42. loop time FORS2, TIC1, LINK, TIA, LINK, TIR, LINK, FR.

5.3 Calculation of the weighting factors

Denote the weights by an **array** wt[1:42]. Many of the weights can be determined from single operations.

- wt[28] = 94 (105) from Take Switch Address.
- wt[27] = 2010 (51) from the frequency of GoTo Accumulator[1].

The ↑ operator appears in the three statements 15, 16 and 17. So the frequency for ↑ must be split between these three as the weights. The last statement $x := y \uparrow z$ is very different from the other two since it must be implemented (in general) by calling the ln and exp routines. Also, there is no statement involving (integer)↑(integer) which is rather different since even the type of the answer is now known until execution, since it depends on the sign of the exponent. The figures given in 4.1.3 can be used to estimate the split between the statements. Say that half of the general cases are ↑(integer) and is to be assigned to the first two statements. Also group ↑ 3, 4, 5 under ↑ 3 and then add the half from the general case in proportion to the previous weights. This gives a split of 315:35:50, from which one has

- wt[15] = 2780 (59).
- wt[16] = 309 (59).
- wt[17] = 442 (59).

In section 4.1.2 the observed proportion of one, two and three dimensional array declarations was 161:54:1. Splitting the one dimensional declaration in half, the frequency of MSF can be used to weight statement 23, 24, 25 and 26. This gives,

- wt[23] = 59 (49).
- wt[24] = 59 (49).
- wt[25] = 39 (49).
- wt[26] = .73 (49).

Unfortunately this means that Call BLock is oversubscribed since the ratio Make Storage Function : Call BLock is 1.25:1 which cannot be preserved by the statements. For this reason it would seem best to put the weight for statement 22 to zero.

- wt[22] = 0 (0).

The weight for statement 38 is determined by the CFZ count in this context, as given in section 2.3.5.1, so

- wt[38] = 788 (104).

The weights for statements 39, 40 and 41 can be obtained by matching the PE and check operations. Since only Check and Store Real appears in the statements, it seems reasonable to match the total count of all the check operations, that is, the average number of parameters. Of course, PE and CSR have already appeared in the standard function calls and so only the remaining weight need be assigned. This gives two conditions on the three weights so take arbitrarily the weight for 39 and 40 to be the same. So we have

- wt[39] = 2316 (87).
- wt[40] = 2316 (87).
- wt[41] = 6053 (28).

[1] Other operations are wt[6] (/), wt[11] (DIV), wt[29] (sin), wt[30] (cos), wt[31] (abs), wt[32] (exp), wt[33] (ln), wt[34] (sqrt), wt[35] (arctan), wt[36] (sign) and wt[37] (entier).

Statement 42 can be weighted directly from the FORS2 frequency so

- wt[42] = 17 800 (21).

The statements involving INDex Address and INDex Result (18 to 21) cannot be dealt with very well, since it is impossible to keep both the INDA/INDR ratios and the dimension ratio correct. So the total of INDA and INDR is split amongst the four statements to keep the dimension ratios correct and taking the same weight for 18 and 21. This gives

- wt[18] = 23 796 (7).
- wt[19] = 15 962 (7).
- wt[20] = 296 (7).
- wt[21] = 23 796 (7).

To assign weights to the statements 4, 5, 9 and 10 one requires an estimate of the proportion of real and integer use of the operators + and $\times$. The estimate used is based upon the analysis given in section 2.3.9. The categories 3), 4) and 8) are taken as an estimate of the use of integers, and 5), 6), 9) and 10) as that for reals. This gives

- wt[4] = 26 694 (16).
- wt[5] = 31 212 (12).
- wt[9] = 4 300 (23).
- wt[10] = 4 978 (17).

The eight remaining statements are not characterized by an operations since they only involve store and fetch operation. In fact, the store and fetch instructions are already overweighted, which is basically due to the fact that the 42 statements are somewhat shorter than is typical of most programs. On the other hand about ten per cent of the elementary operations remain unaccounted for. The total discrepancy of about 40 000 is therefore divided amongst the remaining statements.

An examination of a number of programs by hand has shown that integers occur not infrequently in real expressions (as in statements 2 and 13) but that the converse is much less common. So arbitrarily take the following weights

- wt[1] = 10 000.
- wt[2] = 7 000.
- wt[3] = 10 000.
- wt[7] = 3 000.
- wt[8] = 500.
- wt[12] = 5 000.
- wt[13] = 4 000.
- wt[14] = 500.

The complete weight matrix is reproduced in Appendix C.

5.4 Comments on the Algol mix

Times for the simple statements are available for 18 systems which are tabulated in Appendix D. This is double the number that was listed in [3]. Also, some of the times have been revised because of alterations to the system or improved accuracy of the measurement.

Two methods are available to assess the performance of the system. The technique used in [3], which does not give weights to the statements, and nix figure based upon the weight matrix given in Appendix C.

The table with the unweighted measure, and the seven mix figures together with the average of the seven appears in Appendix E.

5.4.1 Notes on the machine times

Atlas This compiler produces open code for almost everything except **goto**'s. The subroutine used for **goto**'s is a general one catering for formal labels and jumps out of blocks and switches. Consequently this time is very long. In contrast, most of the other times are very good. The large number of registers on Atlas (about 90) means that a display can be maintained in these. This makes procedure entry very straightforward in all cases. The general procedure entry mechanism is used for the standard functions. This has a noticeable effect on abs, sign and entier which could be done by a few instructions compiled in line. One difficulty with not treating the standard functions as ordinary procedures is that it may not then be possible for the user to redefine them, at least by independent compilation (if this exists).

The compiler does virtually no optimisation, but produces very acceptable results without. It is one of the few compilers not to treat **step** 1 as a special case so the **for** loop control code is very slow. One difficulty is that the machine-code produced is very long, although this is partly due to the long address length of the machine (24 bits) which is not used very fully. The size of the machine-code is rather less critical on a paged machine. The timing was done by F.R.A. Hopgood at the Atlas Computer Laboratory.

KDF9 - Kidsgrove Algol compiler This compiler is described in more detail in [2]. It was produced rather early in the development of Algol compilers and contains optional facilities for optimisation. Unfortunately these facilities were rather over-ambitious and in consequence sometimes fail to translate programs or even translate them incorrectly. This has meant that optimisation has not been used very much. There is no interrupt condition on KDF9 for overflow so this system checks overflow on each assignment. This makes the short statements significantly larger than those of Egdon Algol which make no such check.

Procedures are classified to see if open code can be produced for procedure entry and exit. This is successful for the standard functions and the dummy procedures p0, p1, p2 and p3. However, other procedures except input-output ones are unlikely to be so simple. The time taken for complex procedures is slightly longer than Egdon Algol. This means that the weighted figures may be rather optimistic. Without optimiser, no special coding is used for **step** 1 and so the loop time is long. The time for $x := y \uparrow z$ is less than that of (ln + exp). This is because the run-time system spots that z is integral (actually 1.0) and performs the calculation by repeated multiplication.

Egdon Algol compiler on KDF9 This compiler was produced very much later than the last one. Although it was produced very quickly it relies on some of the experience gained with the Kidsgrove compiler. No optimisation in a global sense has been done although subscript handling is better and **step** 1 is done by special coding. The machine-code produced in substantially more compact than that of the last compiler.

The standard functions are treated specially, abs, sign and entier are done by open code and the rest use the "external" Fortran calling convention. This is quicker than the ordinary procedure calling mechanism which is used for p0, p1, p2 and p3. Unfortunately about half the weight assigned to p0 etc is for input-output procedures which are called by this external mechanism. Consequently the mix figure may be rather pessimistic.

The timing was done by Dr. M.D. Poole at Culham Laboratory, UKAEA, but the shorter statements were timed by execution of a number of copies of the machine code sequences as described in [3].

ICL 4130 (2μs and 6μs) and ICL 503 The compilers for the first two machines are identical, and the compiler for the 503 was coded from the same overall logic as the 4130 one. Hence the times for the statements show a rather similar pattern. Sub script checking is done in all three cases, but the 4130 has special hardware to enable this to be done rapidly. This is not true of the 503 as can be seen from the residual matrix. The coding of the procedure ln in all three cases is about twice as long as might be expected. The 4130 compiler evidently optimises $\uparrow 2$ and $\uparrow 3$ but the 503 one does not. Simple procedures are dealt with very quickly on the 503 which is thought to be due to language restrictions allowing fixed locations to be used for parameters and links.

RREAC This machine (and compiler) was produced at the Royal Radar Establishment, Malvern. It was never commercially produced, so the figures are not of wide interest. The compiler assigns storage at procedure level (technique is described in [2]). This means that **begin real** a, **end** produces no code. The time taken for this has been set at one microsecond otherwise it would introduce a singularity into the unweighted analysis. The Kidsgrove Algol compiler also assigns storage at procedure level but does produce code for block entry. This code is unnecessary if the block contains no array declarations but is nevertheless generated.

The timing was done by S.N. Higgins of the Royal Radar Establishment.

1907 (2μs) These timing figures were produced some time ago. The compiler (XALE) has now largely been superseded by (XALT) which is the next one to be considered. However, these times are the same for the simple statements, and the 2μs 1907 is more widely available than the machine upon which the XALT timings were done. The residuals vary very little, so the weighting is not very critical as can be seen from the various mix figures.

The fast time for $k := 1.0$ is apparently due to the fact that the type conversion is done at compile time. This is not very important, but the converse conversion of integers appearing in real expressions is much more common. Doing this conversion would usually give identical times for $x := 1.0$ and $x := 1$, which can be seen to be not always the case.

There is apparently some special coding in the ln function which gives a fast time for ln(1.0) which is rather unfortunate.

ALGOL 60 at Royal Radar Establishment These times are included for comparison with the next ones which are for an Algol 68 compiler on the same computer. As already stated this compiler is an enhancement of the one used for the 1907 (it is XALT Mark 1D). Storage is assigned at procedure level, so the zero time for begin real a; end has been increased slightly to avoid the singularity. The machine is a dual-processor 1907. This is slower (using only one processor, of course) than an ordinary 1907 due to the logic time required to overcome problems of clashes of access to the core-store.

The reason for some differences from the 1907 times is not known. For instance, the long time for $k := l \times m$ and declaring an array with 500 elements. The array declaration time may be due to an initialisation to check for non-assignment as a post mortem facility.

The times were measured by Dr. C.T. Sennett at the Royal Radar Establishment and are not thought to be very accurate (10% error).

Algol 68 at the Royal Radar Establishment The next two sets of times are not for a subset of Algol 60, but for Algol-like languages in advance of Algol 60. Hence the comparison is not necessarily a fair one. Since Algol 68 contains many more features, it would seem unlikely (at first sight) that these statements could be compiled any better. Storage cannot apparently be assigned at procedure level so the statement **begin real** a; **end** does produce some executable code. The procedure mechanism is simpler since it is never necessary to generate a "thunk" for a parameter (see [8] for an explanation of this). The times given are the ones which apply when "garbage collection" is not necessary — a reasonable restriction since this is not necessary in Algol 60 (without dynamic own arrays which most systems omit).

The switch has been replaced by a case statement, otherwise the coding of the statements is straightforward. The procedures abs, sign and entier can be replaced by operators — which allows the compiler writer to use open code without violating any redefinition facility.

The times were produced by Dr. C.T. Sennett at the Royal Radar Establishment and are not thought to be very accurate.

Although the Algol 60 times come out faster without taking the weights into account, the Algol 68 figures are significantly faster by about 20% when the weighting is done. This is mainly due to the procedure times.

Algol W on an IBM 360/67 at Stanford University As with the last times, these figures are not for Algol 60 but for a language which is a direct development of Algol 60 (see [2] and [5]). Similar remarks apply to the coding of the statements: the switch was done by a case statement and abs, sign and entier are coded by the use of operators.

As with the other compilers, array accessing time is in the fastest mode, which in this case is without subscript checking. The time for the case statement included a bound check on the case number. The shorter instruction sequences were taken from the sum of the average instruction times as given by the manual. The longer ones were timed using a program by E. Satterthwaite at Stanford University.

At least two major enhancements have been made to the compiler since the timing test was first done. So, to a certain extent, the compiler has been timed to do well with this test. The improvement made has been about 50%, mainly with procedure entry.

These last two sets of times, shows that radical additions can be made to an Algol-like language without having a detrimental effect upon the execution of the Algol-60 features. Indeed, due no doubt to the competence of the compiler-writing teams, these new languages do distinctly better than their Algol 60 counterparts.

Algol 60 in an IBM 360/65 at University College, London From the point of view of instruction execution speeds this machine is virtually the same as the 360/67 at Stanford. The contrast in execution speed of the statements is rather surprising.

This compiler, and the next three to be considered, apparently do checking of formal-actual correspondence of parameters at run-time. This makes procedure entry very slow. In addition, on entry to a block the compiler requests (in general) core-store from the supervisor. Although the operating system uses fixed partitions, this supervisor call obviously increases the time substantially. It can clearly be seen that the standard functions do not involve this mechanism.

The compiler optimises constant subscripts, as does the ALCOR compiler. This is rather unfortunate since it really invalidates the weighting given to the array access statements (giving the mix too optimistic a value). However, the poor procedure/block entry performance degrades the system alarmingly. It must be remembered that about half the weight assigned to procedures was for input/output. If this is done by a faster mechanism, then the mix figure will be pessimistic. me mix figures vary very widely because of the uneven performance of the statements.

The times were obtained by Miss J. Garrett for Dr. P.A. Samet of University College, London, in January 1969.

The Norwegian compiler for the CDC 3600 Timing figures for this system were sent to the author by Per M. Kjeldas at the Kjeller Computer Installation, Norway.

From the residual matrix, one can see that this compiler gives a poor performance with procedure entry, but unlike the 360/65 gives good results for block entry. One would imagine that a small amount of work on the compiler to improve $x := y \uparrow 2, x := y \uparrow 3$ and **goto** would be worthwhile. The fast block entry times, which has a noticeable effect on the unweighted analysis has little effect with the mix since the weight to these statements are very low. Hence this does little to stop the degradation in performance due to the procedure times. I have been told that the procedure entry times are long due to optimisation of call by name when the actual parameter is a simple variable. This seems unfortunate, since value parameters are so much more common (see 2.3.5).

CDC 6600 Of the machines considered in this report, this is usually acknowledged to be the most powerful. The performance of the Algol system is very disappointing.

Most of the simple assignment statements are very fast, although array accessing is not quite so good. The statements $x := y \uparrow 2, x := y \uparrow 3$ are particularly noteworthy. Block entry is rather poor, but this is not too significant for the mix calculation. The time for **goto** is not good. This is thought to be due to a division of the program into blocks which means that jumps, even implicit ones, are done by a subroutine which has the relevant storage information.

The standard functions are not particularly fast, although the general procedure entry mechanism is obviously not used. This is presumably because of the extra calculation involved to get the 60 bit precision of the 6600.

Improvements have been made to the compiler to make procedure entry faster (by about 50%). However this consists of removing some parameter checking. This seems of doubtful benefit since it is very difficult to test all possible calls of procedures in order to be confident that the checking can be removed. Clearly it is very much better if the parameter checking can be done at compile time.

The times were produced by A.G. Bell when he was at C.D.C. France and are dated May 1969.

Univac 1108 (Standard compiler) The times for these statements show a similar pattern to the 360/65 with poor procedure and block entry times. In fact, subscript checking was done, so the array accessing statements are about 50% longer than they would be otherwise. This clearly degrades the mix figures, since subscript checking has not, in general, been done with the other systems.

In Univac Algol, variables including array variables, are initialised to zero on declaration. This clearly puts an additional overhead on the block entry statements — particularly the declaration of an array of 500 elements. This has rather an unfortunate effect on the analysis. It makes the statement factor for declaring the 500 word array about 25% more than that for an array of one element. Since this and the Algol 60 compiler at RRE are the only compilers to give different times for these statements, it upsets rather a lot of the other figures. A more elaborate mathematical technique could possibly alter this, but would hardly be worthwhile.

The procedure entry times are poor due to parameter checking at run-time. No type conversion is allowed for parameters, for instance, a real value parameter may not have as the actual parameter an integer. The input/output facilities are not done by procedures but by Fortran-type statements. Hence the weight associated with input/output given to the procedure calls would give a pessimistic value for the mix figure.

In contrast, the statements involving, and the standard functions are very fast.

The times were produced by Dr. M.D. Poole when he was at the National Engineering Laboratory with the compiler dated April 1968.

The B5500 Algol Compiler As is well known, the computer has special hardware to facilitate the execution of Algol 60. In fact, there is no assembler (in the ordinary sense) available for the 5500 and so Algol is used instead. This means that the design aim of B5500 Algol has been to provide effective access to the machines facilities rather than provide a strict Algol 60 system.

Only two registers are available for the top of the stack. This means, from the point of view of core-accessing, the 5500 does little better than a one-address machine with evaluation of expressions. The instruction length on the 5500 is only twelve bits which means that code is extremely compact, typically one third the size of most other systems.

One consequence of the 5500 architecture is that array declaration involves a supervisor call. This accounts for the very long time for these statements. The statements $x := y \uparrow 2$ and $x := y \uparrow 3$ are optimised by using the "bit pattern" technique for exponentiation with any integer less than 1024 (see [7] page 399).

The stack mechanism has a substantial effect on the procedure entry mechanism which is relatively the fastest where no special coding has been done (as on 2 and 6). A restriction on B5500 Algol makes the procedure entry mechanism somewhat simpler — namely variables declared in one procedure cannot be accessed in a procedure nested within this.

The times were produced by Mr. J. Thomas of the Post Office Telecommunications Headquarters.

ALCOR - Illinois compiler on the 7094/1 at NEUCC Copenhagen This compiler is described in [6]. It is one of the few Algol compilers to do extensive for loop optimisation. Times were taken with this

in action since this is necessary to suspend subscript checking. As with the 360/65, this really invalidates the weight assigned to the array accessing statements, giving an optimistic value to the mix. Clearly, to assess the performance of the array optimisation one would require some detailed statistics in the overall structure of Algol programs. Unfortunately this report does not give such information.

Apart from procedure entry, the times are fairly consistent. abs, sign and entier appear to be done by open code and presumably ln(1.0) has been coded exceptionally. The general procedure entry mechanism has obviously not been used for the standard functions. Since input/output is Fortran- like rather than by procedure calls, the mix figure may be somewhat pessimistic. Parameter checking is apparently done at compile time, so the reason for the slow procedure entry times is not clear.

The times were produced by Dr. E. Hansen at NEUCC in October 1969. An additional test showed that programs with heavy array accessing can run three times quicker with the optimisation rather than doing the subscript check.

ICL 4/70 This computer is a member of the system 4 range of ICL and is compatible with the 360 series. It is the most powerful of the generally available machines in this range.

The times show a completely different pattern to any of the other machines. If $k := 1, k := l + m, k := l \div m, k := l$ is taken as a measure of the speed of doing integer working, then the computer is quite fast. However the standard functions sin, cos, exp, ln, sqrt and arctan are on average twice as long as one would expect or five times slower than the average performance of the integer statements. Hence the time taken for any benchmark is likely to depend critically on the proportion of integer to real working. Surprisingly the basic floating point assignment statements appear to execute at about the expected speed. The slowness of the standard functions could be due to extreme care over numerical accuracy or lack of hardware to deal with guard bits required to maintain accuracy in the calculation.

The times were produced by the Post Office Communications Headquarters taking substantial care to get figures consistent to five per cent and agreeing with the manufacturers specification for the instruction times.

5.4.2 The relationship of the Algol mix to the Gibson Mix

A measure sometimes used for computer processor performance is the Gibson mix. This is a measure of the rate of execution of instructions where some allowance is made for the addressing structure of the machine. Gibson mix figures are often quoted by manufacturers in the technical specification of the machine. Weights are assigned to about a dozen machine instructions in calculating the mix in a very similar way to the Algol mix.

If the Gibson Mix is regarded as a measure of processor “hardware” performance, and the Algol mix as a measure of this plus the compiler, then the ratio should give some indication of the efficiency of the compiler.

The ratio for machines for which the Gibson Mix figure was available is in the following order.

1. ALGOL W 360/67
2. B5500
3. ICL 4130 (6μs)
4. ALCOR 7094/1
5. ICL 4130 (2μs)
6. ICL 503
7. ICL 190 7 (2μs)
8. KDF9 - Egdon
9. CDC 3600

10. Atlas
11. ICL 4/70
12. KDF9
13. Univac 1108
14. IBM 360/65
15. CDC 6600

A major contributory factor to the substantial differences (a factor of 8 difference between the top and the bottom) is the ease with which a machine architecture allows one to compile efficient code. Producing a compiler for the B5500 (at least with the restrictions imposed by the machine) is substantially easier than for most other computers. In contrast, it is very difficult to make good use of the autonomous accumulators on the CDC 6600 without increasing compilation times inordinately. CDC have clearly decided to put the major part of their compiler-writing expertise into solving this problem for the compilation of FORTRAN.

It is possible to calculate a machine instruction mix from the Algol mix. To do this, one takes a hypothetical one-address (say) computer and compiles (by hand) the 42 statements. This should be done on the basis of a simple compiler without extensive optimisation — otherwise **for** i := 1 **step** 1 **until** n **do** $x := y$; could be optimised to $x := y$, and constant subscripts could be calculated at compile-time invalidating the mix assumptions. From the Algol weights for the statements, weights can be deduced for the relevant machine instructions. The general availability of the Gibson Mix figures makes this preferable to inventing another machine instruction mix.

The important advantage of the Algol mix is that it is measured through the software and so gives a figure of performance in terms of the high level language program. It necessarily takes full account of the architecture of the machine, since this will have had its effect on the compiled code of the user program.

References

[1] RANDELL, B and RIJSSELL, L.J. ALGOL 60 Implementation. APIC Studies in Data Processing No.5. 1964, London Academic Press.

[2] HUXTABLE, D.H.R. and HAWKINS, E.N. A multipass translation scheme for ALGOL 60. Annual review in Automatic Programming 1963, 3, Oxford, Pergamon Press.

[3] WICHMANN, B.A. A comparison of A1GOL 60 execution speeds. National Physical Laboratory. CCU Report No.3, January 1969.

[4] Algol W, Language Description. Computer Science Department, Stanford University. Report CS110. September 1968.

[5] WIRTH, Niklaus and HOARE, C.A.R. A contribution to the development of Algol. Comm ACM. 9 Jun 1966, pages 413-431.

[6] GRIES, D, PAUL, M., and WIEHLE, H.R. Some techniques used in the ALCOR-ILLINOIS 7090. Comm A.C.M. 8 Aug 1965, pages 496-500.

[7] KNUTH, D.E. Semi-numerical Algorithms. Vol. 2. The Art of Computer Programming. Addison-Wesley 1969.

[8] INGERMAN, P.Z. Thunks — A way of compiling procedure statements with some comments on procedure declarations. Comm. A.C.M. 4, 1 (1961) pages 55-58.

A Statistics of each Whetstone Interpretive Instruction

Eaeh interpretive instruction is listed in the order of decreasing use.

Column 1 gives the frequency per million executed interpretive instructions. This is the average of the seven frequencies obtained from 155 million operations.

Column 2 gives the variance of the first column as calculated from the seven samples, expressed as a percentage. So a variance of ten per cent or less implies that the first decimal digit is significant, whereas a variance of 100 per cent suggests that little reliance can be placed on the frequency.

Column 3 gives the static frequency per thousand of eaeh instruction. Hence this measures the frequency of occurrenee of the instruction in the compiled eode.

frequency per million	*variance*	*static frequency per thousand*	*operation*
129 000	10	73.5	Take Integer Result
84 200	20	44.2	LINK
68 000	17	29.3	Take Real Result
55 100	18	48.4	Take Real Address
52 200	12	31.0	Take Integer Address
49 000	6	47.5	STore
43 200	4	39.5	Take Integer Constant 1
39 700	10	26.4	INDex Result
36 200	9	18.9	$\times$
31 000	14	15.7	+
26 300	11	13.4	$-$
24 600	20	64.8	Unconditional Jump
24 400	39	10.2	Take Formal Address
24 200	12	21.8	INDex Address
23 500	22	24.8	TRACE
22 500	33	37.6	Take Integer Constant
20 100	26	18.7	Procedure Entry
19 400	29	46.1	Call Function
19 300	18	10.2	For Return
18 100	26	12.9	Check and Store Real
17 800	21	9.7	FOR S2
15 400	16	12.8	If False Jump
13 400	26	13.6	Block Entry
13 200	27	12.3	End Implicit Subroutine
13 100	26	15.9	Check and Store Integer
11 000	27	6.7	/
7 450	13	33.1	REJECT
6 980	26	5.9	=
6 630	25	10.0	Take Integer Constant 0
6 280	41	14.0	Real DOWN
5 450	44	9.8	Take Real Constant
5 230	21	3.2	NEGate
3 600	24	2.7	$>$
3 530	59	2.1	$\uparrow$
3 380	28	12.5	Call Function Zero
3 290	38	2.1	$<$
2 730	44	3.6	RETURN
2 700	24	10.2	For Block Entry
2 690	40	1.3	Take Formal Real
2 510	27	10.2	For Statement End

Table 12: Whetstone Statistics, Part 1

frequency per million	*variance*	*static frequency per thousand*	*operation*
2 480	79	2.60	Check Arithmetic
2 330	29	9.70	FORS1
2 230	56	1.80	$\neq$
2 180	32	1.20	$\vee$
2 010	51	8.90	Take Label
2 010	51	7.40	GoTo Accumulator
1 890	42	1.80	Take Formal Address Integer
1 820	109	3.50	Take Formal Integer
1 790	76	2.00	Check STring
1 750	49	0.22	SQRT
1 700	93	1.20	Take Formal Address Real
1 490	101	0.06	COS
1 390	40	0.19	ABS
1 370	207	1.60	Decrement Switch Index
1 330	125	4.60	FOR Arithmetic
1 030	70	1.20	$\leq$
1 020	108	0.06	SIN
987	154	0.81	Integer DOWN
977	48	2.30	STore Also
909	122	0.17	ENTIER
914	86	0.59	$\geq$
890	120	1.60	UP1
884	74	1.30	$\wedge$
831	92	0.09	EXP
817	85	0.85	Take Boolean Address
753	61	0.93	Take Boolean Result
664	112	1.60	UP2
656	34	4.60	Check Array Real
644	83	0.13	LN
591	149	0.03	ARCTAN
481	70	1.00	DIV
471	180	0.63	Check Array Integer
377	91	0.10	FOR While
197	51	1.30	Call Block
157	49	3.40	Make Storage Function
163	194	0.36	Call Formal Function
124	68	0.31	Take Boolean Constant False
115	107	0.24	$\neg$
113	93	0.49	Check and Store Boolean
103	92	0.22	DUMMY

Table 13: Whetstone Statistics, Part 2

frequency per million	*variance*	*static frequency per thousand*	*operation*
94	105	0.08	Take Switch Address
85	143	0.03	Take Formal Boolean
82	82	0.05	SIGN
78	126	0.27	Take Boolean Constant True
59	91	0.06	Check Label
43	85	0.08	Copy Real Formal Array
24	170	0.05	Check Boolean
9	153	0.04	Check PRocedure
4	56	0.52	FINISH
3	169	0.65	Check Function Real
2	91	0.59	Boolean DOWN
2	208	0.03	Check and Store Label
1	241	—	EQUIVALENT
0	0	0.08	End Switch List
0	223	—	Copy Integer Formal Array
0	223	—	TEST
0	151	0.09	Take Formal Label
0	0	—	Check Function Boolean
0	0	—	Check Function Integer
0	0	—	Avoid Own Array
0	0	—	Call Formal Function Zero
0	0	—	IMPlies
0	0	—	Make Own Storage Function
0	0	—	Call Segment
0	0	—	Check Switch
0	0	—	Check Array Boolean
0	0	—	Copy Boolean Formal Array
—	—	—	Parameter Switch
—	—	—	Parameter Boolean Array
—	—	—	Parameter Boolean Constant
—	—	6.50	Parameter Real Array
—	—	0.93	Parameter Real Constant
—	—	0.80	Parameter Integer Array
—	—	33.50	Parameter Integer Constant
—	—	0.12	Parameter Procedure
—	—	1.90	Parameter Formal
—	—	0.16	Parameter Label
—	—	14.10	Parameter String
—	—	0.12	Parameter Boolean
—	—	8.60	Parameter Real
—	—	19.40	Parameter Integer
—	—	12.20	Parameter SubRoutine
—	—	—	Parameter Function Boolean
—	—	0.70	Parameter Function Real
—	—	0.04	Parameter Function Integer

Table 14: Whetstone Statistics, Part 3

Symbol	*Frequency/million*
a	21 101
b	7 707
c	11 877
d	15 597
e	2 537
f	9 064
g	3 115
h	4 621
i	24 809
j	7 882
k	5 144
1	8 786
m	12 121
n	18 652
o	1 172
P	9 776
q	2 685
r	18 341
s	14 540
t	24 664
u	5 106
v	3 518
w	6 446
x	12 136
y	6 225
z	1 917

Table 15: Lower case letters

B Basic symbol frequencies

The frequency of Algol basic symbols as produced on KDF9 (see section 4). Programs can use both lower and upper case alphabetic characters but because of the one case on the line printer listing, the use of upper case characters is discouraged.

The statistics are listed as follows

1. Lower case letters, total frequency 286 539 / million.
2. Upper case letters, total frequency 50 270 / million.
3. Digits, total frequency of 71 105 / million.
4. Other basic symbols in order of decreasing use

Symbol	*Frequency/million*
A	3 938
B	1 706
C	2 208
D	2 002
E	3 721
F	2 008
G	951
H	1 123
I	3 831
J	1 093
K	661
L	2 853
M	2 210
N	2 912
O	2 330
P	2 318
Q	404
R	3 344
S	2 282
T	2 703
U	1 270
V	1 393
W	489
X	1 166
Y	1 085
Z	269

Table 16: Upper case letters

Symbol	*Frequency/million*
0	16 402
1	23 002
2	10 518
3	9 233
4	2 789
5	3 241
6	2 079
7	1 417
8	1 298
9	1 126

Table 17: Digits

Symbol	*Frequency/million*
(space)	143 506
(tab)	125 162
(new line)	54 026
;	42 232
,	40 770
:=	20 618
(	17 307
)	17 306
⌈	15 377
⌉	15 377
-	7 377
×	7 041
+	7 023
(open string quote)	6 405
(close string quote)	6 404
(String space)	4 775
end	4 474
begin	4 473
if	4 137
then	4 137
for	3 534
do	3 534
step	3 411
until	3 411
=	2 807
:	2 795
.	2 721
/	2 541
procedure	2 169
real	2 117
integer	2 062
else	1 872
value	1 502
goto	1 465
comment	1 438
KDF9	1 277
ALGOL	1 277
array	1 211

Table 18: Remaining symbols, Part 1

Symbol	*Frequency/million*
$<$	775
$>$	737
$\neq$	722
10	712
$\uparrow$	478
$\wedge$	365
$\vee$	257
$\leq$	233
$\geq$	222
$\rightarrow$	209
$\div$	206
library	200
string	106
true	102
false	100
Boolean	95
$\neg$	85
label	51
while	31
switch	19
segment	8
own	3
$\equiv$	0
$\supset$	0

Table 19: Remaining symbols, Part 2

C Matrix of statement weights

Table 20 gives the matrix of statement weights from seven samples as calculated in section 5.3.

10 000.00	10 000.00	10 000.00	10 000.00	10 000.00	10 000.00	10 000.00	$x := 1.0$
7 000.00	7 000.00	7 000.00	7 000.00	7 000.00	7 000.00	7 000.00	$x := 1$
10 000.00	10 000.00	10 000.00	10 000.00	10 000.00	10 000.00	10 000.00	$x := y$
22 000.00	25 400.00	26 900.00	29 900.00	34 600.00	26 400.00	21 700.00	$x := y + z$
29 100.00	36 200.00	25 000.00	29 100.00	35 400.00	31 500.00	32 200.00	$x := y \times z$
11 800.00	6 470.00	13 400.00	11 800.00	15 400.00	10 800.00	7 130.00	$x := y/z$
3 000.00	3 000.00	3 000.00	3 000.00	3 000.00	3 000.00	3 000.00	$k := 1$
500.00	500.00	500.00	500.00	500.00	500.00	500.00	$k := 1.0$
4 370.00	2 360.00	5 470.00	5 570.00	3 970.00	4 410.00	3 950.00	$k := l + m$
5 800.00	3 360.00	5 080.00	5 420.00	4 060.00	5 260.00	5 860.00	$k := l \times m$
725.00	910.00	224.00	947.00	193.00	287.00	83.00	$k := l \div m$
5 000.00	5 000.00	5 000.00	5 000.00	5 000.00	5 000.00	5 000.00	$k := l$
4 000.00	4 000.00	4 000.00	4 000.00	4 000.00	4 000.00	4 000.00	$x := l$
500.00	500.00	500.00	500.00	500.00	500.00	500.00	$l := y$
5 970.00	4 480.00	1 550.00	1 920.00	2 710.00	1 160.00	1 680.00	$x := y \uparrow 2$
664.00	497.00	172.00	214.00	302.00	129.00	187.00	$x := y \uparrow 3$
948.00	710.00	246.00	305.00	431.00	184.00	267.00	$x := y \uparrow z$
23 300.00	23 900.00	25 700.00	24 500.00	20 500.00	23 000.00	25 600.00	$e1[1] := 1$
15 600.00	16 100.00	17 300.00	16 400.00	13 700.00	15 400.00	17 200.00	$e2[1,1] := 1$
289.00	298.00	320.00	304.00	255.00	285.00	319.00	$e3[1,1,1] := 1$
23 300.00	23 900.00	25 700.00	24 500.00	20 500.00	23 000.00	25 600.00	$l := e1[1]$
0.00	0.00	0.00	0.00	0.00	0.00	0.00	**real** a;
32.00	119.00	73.00	66.00	32.00	46.00	40.00	**array** a[1:1];
32.00	119.00	73.00	66.00	32.00	46.00	40.00	**array** a[1:500];
22.00	80.00	49.00	44.00	22.00	31.00	27.00	**array** a[1:1,1:1];
0.40	1.50	0.91	0.82	0.40	0.57	0.50	**array** a[1:1,1:1,1:1];
1 450.00	1 710.00	2 410.00	3 110.00	1 260.00	3 640.00	508.00	**goto**
0.00	131.00	50.00	306.00	13.00	131.00	29.00	**switch**
2 600.00	104.00	539.00	646.00	2 860.00	264.00	109.00	$x := sin(y)$
4 430.00	883.00	555.00	683.00	3 200.00	482.00	232.00	$x := cos(y)$
747.00	741.00	1 090.00	1 710.00	1 750.00	2 420.00	1 310.00	$x := abs(y)$
1 700.00	317.00	449.00	493.00	377.00	2 310.00	167.00	$x := exp(y)$
172.00	888.00	31.00	130.00	1 530.00	592.00	1 160.00	$x := ln(y)$
719.00	1 120.00	2 200.00	1 420.00	3 600.00	1 630.00	1 580.00	$x := sqrt(y)$
257.00	40.00	1.00	370.00	2 710.00	385.00	372.00	$x := arctan(y)$
103.00	47.00	2.00	65.00	119.00	22.00	216.00	$x := sign(y)$
132.00	20.00	827.00	1 180.00	3 430.00	767.00	9.00	$x := entier(y)$
308.00	1 020.00	798.00	2 630.00	91.00	573.00	89.00	$p0$
5 430.00	6 780.00	2 150.00	1 770.00	340.00	192.00	2 070.00	$p1(x)$
5 430.00	6 780.00	2 150.00	1 770.00	340.00	192.00	2 070.00	$p2(x, y)$
7 380.00	2 520.00	6 010.00	7 440.00	6 540.00	7 750.00	4 960.00	$p3(x, y, z)$
13 400.00	21 000.00	17 400.00	15 200.00	14 100.00	18 700.00	24 800.00	loop time

Table 20: Matrix of statement weights

D Observed and estimated statement times

Observed and estimated times for 18 machines — estimated times are shown in italics.

ATLAS	KDF9	KDF9-EGDON	ICL 4130 2μs	ICL 4130 6μs	*Statement*
6.0	27.3	22.0	*18.7*	36.0	$x := 1.0$
6.0	118.0	26.5	24.0	44.0	$x := 1$
6.0	27.3	22.0	17.0	37.0	$x := y$
9.0	45.0	32.7	28.0	56.0	$x := y + z$
12.0	53.0	32.7	47.0	74.0	$x := y \times z$
18.0	73.0	45.2	80.0	109.0	$x := y/z$
9.0	24.3	18.3	11.0	24.0	$k := 1$
18.0	93.0	54.6	38.0	66.0	$k := 1.0$
12.0	40.0	32.7	16.0	35.0	$k := l + m$
15.0	53.0	34.0	36.0	70.0	$k := l \times m$
48.0	121.0	1340.	40.0	64.0	$k := l \div m$
9.0	28.0	22.0	11.0	23.0	$k := l$
6.0	124.0	28.7	21.0	41.0	$x := l$
18.0	101.0	54.6	38.0	66.0	$l := y$
39.0	209.0	402.0	55.0	92.0	$x := y \uparrow 2$
48.0	231.0	435.0	170.0	339.0	$x := y \uparrow 3$
120.0	288.0	360.0	1700.0	2920.0	$x := y \uparrow z$
21.0	78.0	38.7	46.0	99.0	$e1[1] := 1$
27.0	181.0	67.8	60.0	136.0	$e2[1,1] := 1$
33.0	367.0	247.0	120.0	174.0	$e3[1,1,1] := 1$
15.0	88.0	42.7	36.0	74.0	$l := e1[1]$
45.0	33.0	16.3	350.0	820.0	**real** a;
96.0	771	226.0	1000.0	2340.0	**array** a[1:1];
96.0	764	226.0	1000.0	2330.0	**array** a[1:500];
156.0	995.0	509.0	1600.0	3790.0	**array** a[1:1,1:1];
216.0	1050.0	600.0	2300.0	5200.0	**array** a[1:1,1:1,1:1];
42.0	31.0	10.0	3.0	5.0	**goto**
129.0	591.0	465.0	900.0	1990.0	**switch**
210.0	654.0	547.0	1100.0	1560.0	$x := sin(y)$
222.0	696.0	595.0	1000.0	1650.0	$x := cos(y)$
84.0	193.0	22.7	60.0	138.0	$x := abs(y)$
270.0	676.0	434.0	700.0	1430.0	$x := exp(y)$
261.0	783.0	301.0	1100.0	2020.0	$x := ln(y)$
246.0	422.0	269.0	600.0	1140.0	$x := sqrt(y)$
272.0	1120.0	1040.0	1100.0	1710.0	$x := arctan(y)$
99.0	191.0	32.0	60.0	152.0	$x := sign(y)$
99.0	326.0	63.1	80.0	164.0	$x := entier(y)$
54.0	80.0	276.0	390.0	900.0	$p0$
69.0	110.0	295.0	410.0	930.0	$p1(x)$
75.0	121.0	325.0	430.0	980.0	$p2(x, y)$
93.0	137.0	356.0	400.0	1010.0	$p3(x, y, z)$
57.0	141.0	*62.6*	*77.3*	*152.0*	loop time

Table 21: Statement times, Part 1

ICL 503	RREAC	ICL 1907	ALGOL60/RRE	RRE ALGOL 68	*Statement*
20.0	47.0	15.0	15.0	14.0	$x := 1.0$
53.0	98.0	14.0	15.0	54.0	$x := 1$
20.0	56.0	14.0	*11.3*	*11.8*	$x := y$
42.0	94.0	23.0	23.0	23.0	$x := y + z$
63.0	129.0	31.0	39.0	39.0	$x := y \times z$
87.0	185.0	47.0	72.0	71.0	$x := y/z$
20.0	45.0	8.0	8.0	8.0	$k := 1$
186.0	84.0	8.0	8.0	121.0	$k := 1.0$
36.0	80.0	15.0	12.0	13.0	$k := l + m$
85.0	147.0	37.0	78.0	75.0	$k := l \times m$
281.0	410.0	53.0	56.0	45.0	$k := l + m$
18.0	53.0	11.0	8.0	8.0	$k := l$
46.0	107.0	36.0	22.0	44.0	$x := l$
182.0	92.0	37.0	113.0	122.0	$l := y$
401.0	676.0	97.0	104.0	180.0	$x := y \uparrow 2$
457.0	750.0	117.0	120.0	213.0	$x := y \uparrow 3$
2970.0	3170.0	232.0	286.0	978.0	$x := y \uparrow z$
216.0	109.0	17.0	29.0	22.0	$e1[1] := 1$
347.0	208.0	37.0	71.0	54.0	$e2[1,1] := 1$
480.0	272.0	57.0	120.0	106.0	$e3[1,1,1] := 1$
236.0	118.0	20.0	29.0	22.0	$l := e1[1]$
210.0	1.0	64.0	0.1	52.0	**real** a;
910.0	1060.0	357.0	664.0	242.0	**array** a[1:1];
910.0	1060.0	324.0	4200.0	232.0	**array** a[1:500];
1220.0	1590.0	407.0	854.0	352.0	**array** a[1:1,1:1];
1530.0	1960.0	507.0	1010.0	452.0	**array** a[1:1,1:1,1:1];
28.0	58.0	14.0	16.0	16.0	**goto**
288.0	208.0	109.0	54.0	62.0	**switch**
1270.0	1120.0	328.0	394.0	692.0	$x := sin(y)$
1180.0	1300.0	357.0	404.0	462.0	$x := cos(y)$
36.0	97.0	70.0	84.0	22.0	$x := abs(y)$
1340.0	1110.0	386.0	494.0	562.0	$x := exp(y)$
1520.0	1710.0	97.0	116.0	462.0	$x := ln(y)$
518.0	842.0	357.0	464.0	432.0	$x := sqrt(y)$
1730.0	1750.0	490.0	564.0	622.0	$x := arctan(y)$
122.0	195.0	64.0	114.0	72.0	$x := sign(y)$
211.0	151.0	96.0	164.0	152.0	$x := entier(y)$
27.0	255.0	104.0	284.0	72.0	$p0$
42.0	332.0	187.0	414.0	92.0	$p1(x)$
59.0	381.0	232.0	464.0	132.0	$p2(x, y)$
79.0	430.0	282.0	504.0	162.0	$p3(x, y, z)$
110.0	*146.0*	*42.7*	*50.7*	*53.0*	loop time

Table 22: Statement times, Part 2

ALGOL W 360/67	IBM 360/65	CDC 3600	CDC 6600	UNIVAC 1108	*Statement*
2.1	5.6	4.0	3.0	1.9	$x := 1.0$
9.0	21.7	4.0	2.0	1.9	$x := 1$
2.1	4.2	4.0	2.0	1.8	$x := y$
4.6	4.4	9.0	2.6	3.7	$x := y + z$
6.5	6.3	11.0	3.0	4.8	$x := y \times z$
9.4	7.3	18.0	5.2	11.0	$x := y/z$
2.1	6.0	4.0	2.4	2.0	$k := 1$
38.5	32.4	4.0	2.0	17.9	$k := 1.0$
3.5	15.7	9.0	2.8	2.9	$k := l + m$
11.0	22.6	11.0	4.2	4.4	$k := l \times m$
14.9	26.5	24.0	5.6	13.8	$k := l + m$
2.1	12.4	4.0	2.2	1.6	$k := l$
9.0	31.6	4.0	2.0	3.6	$x := l$
38.5	31.5	9.0	4.6	17.4	$l := y$
6.2	49.6	124.0	4.2	11.0	$x := y \uparrow 2$
93.0	54.2	130.0	4.8	10.7	$x := y \uparrow 3$
209.0	131.0	99.0	22.6	12.4	$x := y \uparrow z$
5.6	6.7	11.0	13.6	12.6	$e1[1] := 1$
12.5	6.9	29.0	17.6	20.4	$e2[1,1] := 1$
19.5	6.8	40.0	19.8	44.4	$e3[1,1,1] := 1$
5.6	5.1	9.0	13.0	13.0	$l := e1[1]$
21.3	327.0	5.0	1ll.0	52.3	**real** a;
61.5	649.0	65.0	162.0	114.0	**array** a[1:1];
61.5	708.0	65.0	162.0	1020.0	**array** a[1:500];
87.9	685.0	89.0	163.0	126.0	**array** a[1:1,1:1];
114.0	661.0	113.0	166.0	141.0	**array** a[1:1,1:1,1:1];
2.2	*7.2*	12.0	29.0	1.l	**goto**
11.7	344.0	98.0	153.0	157.0	**switch**
96.9	83.7	131.0	112.0	42.9	$x := sin(y)$
91.3	72.0	137.0	109.0	44.2	$x := cos(y)$
3.0	2.8	23.0	4.2	1.9	$x := abs(y)$
104.0	78.0	141.0	113.0	40.8	$x := exp(y)$
78.1	71.5	130.0	112.0	38.3	$x := ln(y)$
74.1	68.5	113.0	103.0	36.8	$x := sqrt(y)$
84.7	54.0	159.0	113.0	74.9	$x := arctan(y)$
10.7	25.6	25.0	6.4	5.4	$x := sign(y)$
46.2	43.3	30.0	90.0	16.4	$x := entier(y)$
21.3	373.0	114.0	235.0	73.9	$p0$
41.9	500.0	246.0	278.0	171.0	$p1(x)$
61.3	603.0	362.0	304.0	268.0	$p2(x,y)$
80.8	726.0	504.0	328.0	362.0	$p3(x,y,z)$
4.9	*25.9*	*18.9*	*12.6*	7.5	loop time

Table 23: Statement times, Part 3

B 5500	ALCOR 7094/1	ICL 4/70	*Statement*
12.1	6.5	6.9	$x := 1.0$
8.1	10.0	19.8	$x := 1$
11.6	7.5	4.6	$x := y$
18.8	9.5	9.2	$x := y + z$
50.0	*15.3*	17.5	$x := y \times z$
32.5	16.5	24.7	$x := y/z$
8.1	6.5	2.5	$k := 1$
25.0	30.0	48.5	$k := 1.0$
18.8	11.0	5.1	$k := l + m$
35.0	15.0	9.7	$k := l \times m$
34.6	45.0	17.2	$k := l + m$
11.6	8.5	3.1	$k := l$
11.8	6.5	22.2	$x := l$
26.1	29.5	49.5	$l := y$
46.6	15.5	102.0	$x := y \uparrow 2$
85.0	33.5	110.0	$x := y \uparrow 3$
1760.0	381.0	582.0	$x := y \uparrow z$
24.0	25.0	14.2	$e1[1] := 1$
42.8	21.0	25.2	$e2[1, 1] := 1$
66.6	22.0	120.0	$e3[1, 1, 1] := 1$
23.5	9.0	21.5	$l := e1[1]$
22.3	7.0	3.2	**real** a;
2870.0	308.0	155.0	**array** a[1:1];
2870.0	308.0	155.0	**array** a[1:500];
8430.0	418.0	188.0	**array** a[1:1,1:1];
13000.0	526.0	214.0	**array** a[1:1,1:1,1:1];
31.5	*7.8*	7.5	**goto**
98.3	205.0	175.0	**switch**
598.0	265.0	564.0	$x := sin(y)$
758.0	311.0	571.0	$x := cos(y)$
14.0	11.5	158.0	$x := abs(y)$
740.0	206.0	585.0	$x := exp(y)$
808.0	80.0	587.0	$x := ln(y)$
605.0	168.0	470.0	$x := sqrt(y)$
841.0	336.0	755.0	$x := arctan(y)$
37.5	16.0	174.0	$x := sign(y)$
41.1	28.0	233.0	$x := entier(y)$
31.0	281.0	135.0	$p0$
39.0	407.0	143.0	$p1(x)$
45.0	503.0	150.0	$p2(x, y)$
53.0	598.0	358.0	$p3(x, y, z)$
38.5	*28.1*	49.5	loop time

Table 24: Statement times, Part 4

E Analysis of the times

In order to make this report self-contained, a brief explanation is given here of the analysis which is given in full in [3].

The time for any statement on any machine could be expected to be the product of a factor depending on the machine (i) times a factor depending on the statement (j) i.e.

$$t_{ij} \approx m_i \times s_j$$

or $t_{ij} = r_{ij} \times m_i \times s_j$ where the r_{ij} are on "average" equal to 1.0. The method of calculating m_i and s_j[2] ensures that the product of the elements in each row or column of r_{ij} is equal to 1.0. Missing values for the times are eatimated by putting the corresponding value of r_{ij} equal to 1.

One abnormal value can upaet the machine or statement factors (usually both), but with the 18 machines and 42 statements in this report the effect is insignificant. For instance, recent substantial changes to the times for many ALGOL W statements only altered the machine factors for other computers by at most one per cent.

Hence a number greater than one in the r_{ij} matrix given in Appendix F means that the corresponding statement on that machine took longer than expected.

[2] The s_j are printed in Appendix E.

Time in μs	*Statement*
6.5	$x := 1.0$
10.4	$x := 1$
6.0	$x := y$
10.1	$x := y + z$
14.6	$x := y \times z$
21.2	$x := y/z$
5.1	$k := 1$
18.6	$k := 1.0$
8.8	$k := l + m$
17.2	$k := l \times m$
29.1	$k := l \div m$
5.7	$k := l$
11.7	$x := l$
25.9	$l := y$
44.1	$x := y \uparrow 2$
68.7	$x := y \uparrow 3$
234.0	$x := y \uparrow z$
17.2	$e1[1] := 1$
29.0	$e2[1, 1] := 1$
48.0	$e3[1, 1, 1] := 1$
15.9	$l := e1[1]$
17.3	**real** a;
237.0	**array** a[1:1];
295.0	**array** a[1:500];
337.0	**array** a[1:1,1:1];
414.0	**array** a[1:1,1:1,1:1];
7.4	**goto**
117.0	**switch**
228.0	$x := sin(y)$
231.0	$x := cos(y)$
17.4	$x := abs(y)$
226.0	$x := exp(y)$
188.0	$x := ln(y)$
175.0	$x := sqrt(y)$
285.0	$x := arctan(y)$
30.9	$x := sign(y)$
54.6	$x := entier(y)$
84 0	$p0$
117.0	$p1(x)$
142.0	$p2(x, y)$
166.0	$p3(x, y, z)$
26.8	loop time

Table 25: Statement times for an Atlas-powered machine

Unweighted measure, $= 1/m_i$	Seven mix figures							*Average mix*	*Machine*
1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	ATLAS
0.276	0.293	0.284	0.281	0.282	0.286	0.290	0.277	0.285	KDF9
0.428	0.344	0.364	0.411	0.389	0.410	0.446	0.429	0.399	KDF9-EGDON
0.347	0.278	0.304	0.346	0.330	0.311	0.367	0.357	0.328	ICL 4130 2μs
0.176	0.144	0.150	0.171	0.163	0.170	0.182	0.178	0.165	ICL 4130 6μs
0.244	0.179	0.185	0.186	0.189	0.188	0.194	0.183	0.186	ICL 503
0.183	0.152	0.154	0.167	0.16s	0.165	0.175	0.168	0.164	RREAC
0.629	0.546	0.581	0.595	0.583	0.616	0.627	0.645	0.599	ICL 1907
0.529	0.363	0.363	0.388	0.380	0.450	0.431	0.432	0.401	ALGOL60/RRE
0.506	0.454	0.513	0.521	0.517	0.469	0.531	0.543	0.507	RRE ALGOL 68
2.380	2.090	2.310	2.370	2.310	2.420	2.490	2.580	2.370	ALGOL W 360/67
1.030	0.518	0.499	0.608	0.553	0.967	0.727	0.694	0.652	IBM 360/65
1.420	0.653	0.691	0.765	0.721	0.981	0.845	0.845	0.786	CDC 3600
2.130	0.912	0.884	1.050	0.966	1.430	1.230	1.200	1.100	CDC 6600
2.440	1.050	1.060	1.140	1.080	1.680	1.300	1.280	1.230	UNIVAC 1108
0.539	0.484	0.526	0.581	0.582	0.461	0.563	0.589	0.541	B 5500
0.955	0.488	0.511	0.590	0.549	0.739	0.674	0.661	0.602	ALCOR 7094/1
0.750	0.568	0.669	0.716	0.686	0.554	0.691	0.729	0.659	ICL 4/70

Table 26: Performance data

F Residual matrix

A value greater than one indlcates that the statement took longer than expected (as judged from the other statement times). Values in italics are estimated and so are automaticaly 1.0. The product of every row and column is 1.0 (by construction).

ATLAS	KDF9	KDF9-EGDON	ICL 4130 2μs	ICL 4130 6μs	*Statement*
0.927	1.170	1.460	*1.000*	0.978	$x := 1.0$
0.575	3.130	1.090	0.798	0.743	$x := 1$
1.000	1.260	1.580	0.987	1.090	$x := y$
0.895	1.240	1.390	0.965	0.980	$x := y + z$
0.821	1.000	0.959	1.120	0.891	$x := y \times z$
0.851	0.954	0.916	1.310	0.907	$x := y/z$
1.760	1.310	1.530	0.745	0.825	$k := 1$
0.969	1.380	1.260	0.710	0.626	$k := 1.0$
1.370	1.260	1.600	0.632	0.702	$k := l + m$
0.870	0.850	0.845	0.724	0.715	$k := l \times m$
1.650	1.150	1.970	0.476	0.387	$k := l \div m$
1.580	1.360	1.650	0.669	0.710	$k := l$
0.513	2.930	1.050	0.622	0.617	$x := l$
0.696	1.080	0.904	0.509	0.449	$l := y$
0.884	1.310	3.90	0.432	0.367	$x := y \uparrow 2$
0.699	0.930	2.720	0.858	0.869	$x := y \uparrow 3$
0.514	0.341	0.661	2.520	2.200	$x := y \uparrow z$
1.220	1.260	0.965	0.928	1.010	$e1[1] := 1$
0.930	1.720	1.000	0.716	0.824	$e2[1,1] := 1$
0.687	2.110	2.210	0.866	0.637	$e3[1,1,1] := 1$
0.946	1.530	1.150	0.787	0.821	$l := e1[1]$
2.610	0.528	0.405	7.030	8.360	**real** a;
0.406	0.900	0.409	1.460	1.740	**array** a[1:1];
0.325	0.716	0.328	1.170	1.390	**array** a[1:500];
0.463	0.816	0.647	1.650	1.980	**array** a[1:1,1:1];
0.522	0.703	0.621	1.930	2.210	**array** a[1:1,1:1,1:1];
5.650	1.150	0.576	0.140	0.118	**goto**
1.100	1.390	1.700	2.660	2.990	**switch**
0.919	0.792	1.030	1.670	1.200	$x := sin(y)$
0.962	0.834	1.100	1.500	1.260	$x := cos(y)$
4.840	3.070	0.560	1.200	1.400	$x := abs(y)$
1.200	0.829	0.824	1.080	1.120	$x := exp(y)$
1.390	1.150	0.687	2.030	1.890	$x := ln(y)$
1.410	0.667	0.659	1.190	1.150	$x := sqrt(y)$
0.954	1.080	1.560	1.340	1.060	$x := arctan(y)$
3.200	1.7l0	0.444	0.674	0.866	$x := sign(y)$
1.870	1.650	0.495	0.508	0.529	$x := entier(y)$
0.643	0.263	1.410	1.610	1.890	$p0$
0.588	0.259	1.080	1.210	1.390	$p1(x)$
0.528	0.235	0.979	1.050	l.210	$p2(x, y)$
0.559	0.228	0.916	0.833	1.070	$p3(x, y, z)$
2.130	1.450	*1.000*	*1.000*	*1.000*	loop time

Table 27: Residual matrix, Part 1

ICL 503	RREAC	ICL 1907	ALGOL60/RRE	RRE ALGOL 68	*Statement*
0.752	1.330	1.460	1.220	1.090	$x := 1.0$
7.240	1.720	0.844	0.767	2.620	$x := 1$
0.876	1.720	1.470	*1.000*	*1.000*	$x := y$
1.020	1.710	1.440	1.270	1.160	$x := y + z$
1.050	1.620	1.330	1.470	1.350	$x := y \times z$
1.000	1.600	1.400	1.800	1.700	$x := y/z$
0.952	1.670	0.982	0.826	0.790	$k := 1$
2.440	0.829	0.271	0.228	3.290	$k := 1.0$
0.999	1.670	1.070	0.723	0.749	$k := l + m$
1.200	1.560	1.350	2.390	2.200	$k := l \times m$
2.350	2.580	1.740	1.020	0.787	$k := l \div m$
0.769	1.700	1.210	0.742	0.709	$k := l$
0.958	1.680	1.930	0.994	1.900	$x := l$
1.770	0.652	0.899	2.370	2.380	$l := y$
2.270	2.870	1.380	1.250	2.060	$x := y \uparrow 2$
1.620	2.000	1.070	0.924	1.570	$x := y \uparrow 3$
3.700	2.490	0.625	0.648	2.720	$x := y \uparrow z$
3.060	1.760	0.622	0.893	0.647	$e1[1] := 1$
2.970	1.310	0.801	1.290	0.940	$e2[1,1] := 1$
2.430	1.040	0.746	1.320	1.720	$e3[1,1,1] := 1$
3.630	1.360	0.793	0.967	0.701	$l := e1[1]$
2.960	0.071	2.330	0.003	1.520	**real** a;
0.936	0.822	0.948	1.480	0.517	**array** a[1:1];
0.751	0.660	0.690	7.530	0.398	**array** a[1:500];
0.881	0.866	0.759	1.340	0.528	**array** a[1:1,1:1];
0.899	0.870	0.770	1.300	0.552	**array** a[1:1,1:1,1:1];
0.977	1.430	1.780	1.740	1.090	**goto**
0.598	0.325	0.584	0.244	0.267	**switch**
1.360	0.900	0.903	0.972	1.530	$x := sin(y)$
1.240	1.030	0.972	0.926	1.070	$x := cos(y)$
0.505	1.020	2.530	2.560	0.641	$x := abs(y)$
1.450	0.903	1.080	1.160	1.260	$x := exp(y)$
1.970	1.670	0.325	0.327	1.240	$x := ln(y)$
0.727	0.882	1.280	1.400	1.250	$x := sqrt(y)$
1.470	1.720	1.080	1.050	1.100	$x := arctan(y)$
0.962	1.760	1.300	1.950	1.180	$x := sign(y)$
0.942	0.507	1.110	1.590	1.410	$x := entier(y)$
0.078	0.556	0.778	1.790	0.433	$p0$
0.087	0.518	1.000	1.860	0.396	$p1(x)$
0.107	0.491	1.030	1.730	0.470	$p2(x, y)$
0.176	0.474	1.070	1.600	0.492	$p3(x, y, z)$
1.000	*1.000*	*1.000*	*1.000*	*1.000*	loop time

Table 28: Residual matrix, Part 2

ALGOL W 360/67	IBM 360/65	CDC 3600	CDC 6600	UNIVAC 1108	*Statement*
0.773	0.894	0.878	0.989	0.715	$x := 1.0$
2.060	2.150	0.545	0.409	0.444	$x := 1$
0.838	0.727	0.952	0.715	0.735	$x := y$
1.090	0.452	1.270	0.552	0.897	$x := y + z$
1.060	0.446	1.070	0.438	0.801	$x := y \times z$
1.060	0.357	1.210	0.525	1.270	$x := y/z$
0.977	1.210	1.110	1.000	0.952	$k := 1$
4.940	1.800	0.306	0.230	2.350	$k := 1.0$
0.951	1.850	1.460	0.681	0.806	$k := l + m$
1.520	1.360	0.907	0.520	0.622	$k := l \times m$
1.220	0.940	1.170	0.410	1.150	$k := l \div m$
0.877	2.250	0.997	0.823	0.684	$k := l$
1.830	2.790	0.486	0.365	0.750	$x := l$
3.550	1.260	0.494	0.379	1.640	$l := y$
0.335	1.160	3.990	0.203	0.608	$x := y \uparrow 2$
3.230	0.816	2.690	0.149	0.380	$x := y \uparrow 3$
2.130	0.578	0.603	0.207	0.129	$x := y \uparrow z$
0.777	0.403	0.910	1.690	1.790	$e1[1] := 1$
1.030	0.246	1.420	1.290	1.710	$e2[1,1] := 1$
0.967	0.146	1.180	0.880	2.250	$e3[1,1,1] := 1$
0.842	0.332	0.807	1.750	2.000	$l := e1[1]$
2.940	19.600	0.412	13.700	1.390	**real** a;
0.619	2.830	0.390	1.460	1.170	**array** a[1:1];
0.497	2.480	0.313	1.170	8.400	**array** a[1:500];
0.621	2.100	0.375	1.030	0.909	**array** a[1:1,1:1];
0.658	1.650	0.388	0.856	0.831	**array** a[1:1,1:1,1:1];
0.705	*1.000*	2.290	8.320	0.361	**goto**
0.238	3.030	1.190	2.780	3.260	**switch**
1.010	0.379	0.815	1.040	0.458	$x := sin(y)$
0.943	0.322	0.844	1.010	0.467	$x := cos(y)$
0.412	0.167	1.83	0.516	0.267	$x := abs(y)$
1.100	0.358	0.889	1.070	0.441	$x := exp(y)$
0.992	0.394	0.985	1.280	0.498	$x := ln(y)$
1.010	0.405	0.918	1.260	0.513	$x := sqrt(y)$
0.708	0.196	0.793	0.849	0.641	$x := arctan(y)$
0.826	0.857	1.150	0.442	0.426	$x := sign(y)$
2.020	0.820	0.781	3.520	0.733	$x := entier(y)$
0.604	4.600	1.030	5.960	2.140	$p0$
0.850	4.400	2.980	5.060	3.540	$p1(x)$
1.030	4.390	3.620	4.560	4.590	$p2(x, y)$
1.160	4.510	4.300	4.210	5.300	$p3(x, y, z)$
0.438	*1.000*	*1.000*	*1.000*	0.682	loop time

Table 29: Residual matrix, Part 3

B 5500	ALCOR 7094/1	ICL 4/70	*Statement*
1.010	0.959	0.803	$x := 1.0$
0.419	0.916	1.430	$x := 1$
1.050	1.200	0.584	$x := y$
1.000	0.902	0.690	$x := y + z$
1.850	*1.000*	0.899	$x := y \times z$
0.829	0.745	0.878	$x := y/z$
0.853	1.210	0.362	$k := 1$
0.726	1.540	1.960	$k := 1.0$
1.160	1.200	0.433	$k := l + m$
1.100	0.831	0.422	$k := l \times m$
0.641	1.470	0.442	$k := l \div m$
1.100	1.420	0.473	$k := l$
0.544	0.531	1.430	$x := l$
0.544	1.090	1.440	$l := y$
0.570	0.335	1.730	$x := y \uparrow 2$
0.668	0.466	1.200	$x := y \uparrow 3$
4.070	1.560	1.870	$x := y \uparrow z$
0.754	1.390	0.622	$e1[1] := 1$
0.795	0.691	0.653	$e2[1, 1] := 1$
0.748	0.437	1.870	$e3[1, 1, 1] := 1$
0.799	0.542	1.020	$l := e1[1]$
0.697	0.387	0.138	**real** a;
6.550	1.240	0.493	**array** a[1:1];
5.250	0.997	0.395	**array** a[1:500];
13.500	1.180	0.419	**array** a[1:1,1:1];
16.900	1.210	0.388	**array** a[1:1,1:1,1:1];
2.280	*1.000*	0.757	**goto**
0.452	1.670	1.120	**switch**
1.410	1.110	1.850	$x := sin(y)$
1.770	1.290	1.860	$x := cos(y)$
0.435	0.632	6.830	$x := abs(y)$
1.770	0.872	1.950	$x := exp(y)$
2.320	0.407	2.350	$x := ln(y)$
1.860	0.917	2.020	$x := sqrt(y)$
1.590	1.130	1.990	$x := arctan(y)$
0.655	0.495	4.240	$x := sign(y)$
0.406	0.490	3.20	$x := entier(y)$
0.199	3.190	1.201	$p0$
0.179	3.310	0.914	$p1(x)$
0.171	3.380	0.794	$p2(x, y)$
0.172	3.430	0.712	$p3(x, y, z)$
0.775	*1.000*	1.390	loop time

Table 30: Residual matrix, Part 4

G Document details

1. Converted in October 2001. The text is much superior to the original. This report seems to be the last that was produced without a word-processor. Additional comments on the text have been added as footnotes. The estimated times were marked with a star, rather than italics.