Five ALGOL compilers

B. A. Wichmann

Division of Numerical Analysis and Computing, National Physical Laboratory, Teddington, Middlesex

A detailed comparison of the times taken to perform elementary statements in ALGOL 60 has revealed wide differences in performance. An examination of the machine code produced by five compilers (Atlas, KDF9 (Kidsgrove), 1900 (XALT), B5500 and 1108 (Trondheim compiler)) has been undertaken to find the reasons for the disparities. The large range of machine architecture means that very different techniques have been used for code generation. This enables one to give guide lines for a suitable architecture for good ALGOL 60 code generation to be possible.

(Received September 1971)

Very little is known about the relative merits of computer software—except by personal experience. ALGOL 60 is an excellent vehicle for a comparative study because of its machine independence and the fact that compilers are available for a wide range of machines. This paper considers only the characteristics of the machine code generated by the compiler. Many other factors can be just as important—such as compile-time, run-time and failure-time diagnostics, compiling times, library facilities, etc., but they are not considered here.

In some previous work (Wichmann, 1969, 1970, 1971), the author found that the time taken for various simple ALGOL 60 statements was often five times faster or slower than would be expected. The expected statement times were found from the model

(time for statement S on machine A) = (statement factor for S) × (machine factor for A)

The statement factors and machine factors were calculated by a least squares fit. Hence this took the overall speed of the machine into account—as judged from all the statement times. Such a large variation in the ratio of the expected to observed statement times was, therefore, not to be expected. The statements used are listed in **Table 1.**

The reason for this discrepancy is that totally different techniques have been used by compiler writers in generating machine code. In an extreme case, one system invokes a supervisor function for **begin real** *a*; **end** whereas another system produces no machine-code at all! On the other hand, relatively small variations were found in the time taken to evaluate the standard functions *sin*, *cos*, *exp*, *ln*, *sqrt* and *arctan*.

It has also been possible to assign weights to the ALGOL statements. This gives an ALGOL mix similar to the Gibson mix (which uses basic instruction times). The mix figure of merit is not very closely paralleled by the machine factor found from the previous method. The reason for this is that some compiler writers have been noticeably more successful than others in choosing which features to implement efficiently.

It was clear from this work that the basic choices open to the compiler writer would not be apparent without a detailed study of the machine and the compiled code. This would obviously not be possible for all of the 25 systems for which statement times are available. Hence, four ALGOL compilers were selected for further study. These were the Atlas ALGOL compiler, the Kidsgrove compiler for KDF9, the 1900 compiler XALT (Mark 1A) and the Extended ALGOL compiler for the B5500. These were chosen to include a wide range of computer architectures apart from being conveniently available to the author. In addition Dr. P. T. Cameron of Univac has kindly run all the tests on the 'NU ALGOL' compiler for the 1108. This compiler is a recent replacement of the manufacturer's original compiler, and has been written in Trondheim, Norway.

The tests

The method adopted was to take four sample programs, compile them with each compiler, and make a detailed examination of the machine-code produced. To avoid any possibility of mispunching, the programs were converted automatically from the KDF9 source text. Four main characteristics are measured for the programs: the number of instructions compiled, the number of instructions executed, the size in bits of the compiled program, and the execution time.

Test 1:
The program consisted of CACM Algorithm 271 Quickersort (Scowen, 1965) used for sorting an array containing 10,000 random numbers. The four characteristics are measured for each of the three main procedures of the program, that is, *quickersort*, *inarray* (which fills the array with the random numbers) and *checksort* (which checks that the array is sorted).

Test 2:
This really consists of 13 separate programs. They are ALGOL compiler tests obtained by the author from M. Woodger. Many of them were written by Randell and Russell to test the Whetstone ALGOL compiler (Randell and Russell, 1964). The major test is to see that they compile and produce the correct answers. Execution time is not measured in this case, except for the last test which is Knuth's 'man or boy' (Knuth, 1964). This test uses call-by-name and recursive procedure calls very extensively.

Test 3:
This is a coding of GAMM loops sometimes used as a measure of computer performance (Heinhold, 1962). The five loops used consist of (a) adding two vectors of 30 elements, (b) multiplying two similar vectors, (c) calculating a polynomial of 10 terms, (d) finding the maximum of a vector of 10 elements, and (e) calculating a square root by Newton's method for five iterations. This test can, of course, be coded in any computer language, but in ALGOL 60 it is a rather severe test of one dimensional arrays and simple **for** loops. The program repeats the test twice, once with formal arrays to a procedure, and once with the actual arrays. The main characteristics are measured for the five loops, both in and out of the procedure.

Test 4:
This consists of a program to time the 41 statements used in previous tests (Wichmann, 1969, 1970, 1971). By examining the machine-code produced from these statements, it is possible to see if any 'special coding' has been used in these rather simple statements. For each of these statements, the four characteristics are measured as before. To these statements are

Table 1

The declaration of identifiers used in the timing of simple statements was as follows:

```
integer k, l, m;
real x, y, z;
integer array e1[1:1], e2[1:1, 1:1], e3[1:1, 1:1, 1:1];
procedure p0;;
procedure p1(x); value x; real x;;
procedure p2(x, y); value x, y; real x, y;;
procedure p3(x, y, z); value x, y, z; real x, y, z;;
```

The variables were initialised as follows:

```
x := y := z := 1·0; l := k := m := e1[1] := 1;
```

The statements were:

$x := 1{\cdot}0$	$x := y/z$	$k := l \div m$
$x := 1$	$k := 1$	$k := l$
$x := y$	$k := 1{\cdot}0$	$x := l$
$x := y + z$	$k := l + m$	$l := y$
$x := y \times z$	$k := l \times m$	$x := y \uparrow 2$
$x := y \uparrow 3$	$l := e1[1]$	
$x := y \uparrow z$	**begin real** a; **end**	
$e1[1] := 1$	**begin array** $a[1:1]$; **end**	
$e2[1, 1] := 1$	**begin array** $a[1:500]$; **end**	
$e3[1, 1, 1] := 1$	**begin array** $a[1:1, 1:1]$; **end**	
begin array $a[1:1, 1:1, 1:1]$; **end**		$x := abs(y)$
begin goto *abcd*; *abcd*: **end**		$x := exp(y)$
begin switch $s := p$; **goto** $s[1]$; p: **end**		$x := ln(y)$
$x := sin(y)$		$x := sqrt(y)$
$x := cos(y)$		$x := arctan(y)$
$x := sign(y)$	$p3(x, y, z)$	
$x := entier(y)$		
$p0$		
$p1(x)$		
$p2(x, y)$		

In addition the loop time in **for** $i := 1$ **step** 1 **until** n **do** is also taken to make 42 'statements'.

added the characteristics of the loop code involved in **for** $i := 1$ **step** 1 **until** n **do.** These statements ensure that the principal ALGOL statements have been investigated.

The results

Each characteristic was analysed in the same way as the statement times, that is, it is assumed that

(Number of instructions generated for S on machine A) $\simeq$ (Factor for S) (Factor for machine A).

The factors are again calculated by a least squares fit. If a particular machine always generates more instructions, then the corresponding factor for that machine will have a large value. The factors for the statements, procedures and programs are of no particular consequence, but allow estimates to be made when any data is missing.

The factors for the four systems are only ratios, so Atlas has been taken as unity. The factors for the four characteristics are:

	Atlas	1108	KDF9	1907 (1·1 μs)	B5500
Instructions compiled	1·0	0·41	0·78	0·65	0·63
Compiled code size in bits	1·0	0·31	0·35	0·32	0·16
Execution time	1·0	0·28	3·5	1·2	1·8
Instructions executed	1·0	0·83	1·5	0·94	0·96

One can see from this that as far as the number of instructions generated is concerned, Atlas generates significantly more and the 1108 significantly less than the other three systems. With the code size in bits, Atlas is very much worse and the B5500 very much better than the others. The speed comparison in the third row is in line with the previous work (Wichmann, 1969). The number of instructions executed to do a fixed task in ALGOL varies remarkably little in view of the wide range in architecture, except that KDF9 comes out significantly worse.

The exceptional values

A complete list of those cases where the actual value was less than half or more than twice the expected value are listed in **Table 2.** Some statements give a wide range for this ratio indicating that completely different methods of implementation have been used. For instance, with the statement $x := abs(y)$ two methods are used. The B5500 and 1108 produce open code thus giving very fast times. On the other hand, Atlas and KDF9 use the standard procedure calling mechanism giving relatively longer times.

With entering and exiting from a dummy block (**begin real** a; **end**), the code generated depends upon the method used to assign storage. If storage is assigned at procedure level as on KDF9 and B5500, then there is virtually no code generated. On the other hand Atlas and the 1108 assign storage at block level so that several pointers must be updated. In fact Atlas generates further code for diagnostic purposes. Array declarations also vary substantially. Atlas performs very well by using a large amount of open code. On the other hand the B5500 takes up to 12 times longer than might be expected as storage allocation is a supervisor function. The 1108 is the only compiler to take longer to declare an array of 500 elements than an array of one element which is due to zeroising its contents.

Type conversion is another area where compilers differ substantially. There is also a big difference in the speed of the relevant hardware for conversion from fixed to floating point form (and the converse). With the statement $x := 1$, Atlas and the B5500 store the 1 as an integer but can do the conversion rapidly. On the 1108 and 1900 the integer 1 is converted to floating point form at compile time. KDF9 is the worst, using quite a slow subroutine for the conversion. The conversion in $k := 1{\cdot}0$ is dealt with in a similar manner by the compilers although it is much less worthwhile due to its rarity.

The exponential operator is handled very differently by the compilers. $\uparrow 2$ and $\uparrow 3$ on the B5500 are done by repeated multiplication using only the stack, whereas the other systems use short subroutines except for Atlas which uses open code. The coding of $x := y \uparrow z$ is done incorrectly on the KDF9 and the 1108. In both cases z is checked to see if it is integral, in which

Table 2

Anomalous values for measured characteristics

All the anomalous values are listed, together with the reason if this is known. The ratio is actual value/expected value of the instruction time, instructions executed, size in instructions or size in bits. A small value for the ratio implies good implementation of this feature in relation to the compiler's overall performance.

Execution times

RATIO	ALGOL SOURCE TEXT	REASON
ATLAS		
0·43	'man or boy'	Display is held in registers
0·49	$x := l$	Little difference between reals and integers
0·21 to 0·36	array declarations	Open code used
4·7	**goto** *abcd*	Very general subroutine used

RATIO	ALGOL SOURCE TEXT	REASON
3·5	$x := abs(y)$	General procedure mechanism used
2·6	$x := sign(y)$	General procedure mechanism used
KDF9		
0·37	'man or boy'	Call-by-name handled quite well
4·0	$x := 1$	Type conversion done by subroutine
2·6	$k := 1{\cdot}0$	Type conversion done by subroutine
2·9	$x := l$	Type conversion done by subroutine
2·3	$e3[1, 1, 1] := 1$	Slow subroutine used, suitable for any dimension
0·3	**begin real** a; **end**	Storage is assigned at procedure level
0·47	**array** $a[1:500]$	Bad time for this on 1108, makes KDF9 good in comparison
2·3	$x := abs(y)$	General procedure mechanism used
0·4 to 0·42	$p0$, $p1$, $p2$ and $p3$	These are 'simple' procedures, which are optimised by using what is ordinarily the stack pointer as the environment pointer within the procedure
B5500		
0·74	$x := y \uparrow z$	Not known in detail, but all standard functions are slow
0·39	**begin real** a; **end**	Storage is assigned at procedure level
3·5 to 12	array declarations	Supervisor call, must set up descriptors and allocate storage
0·33	$x := abs(y)$	Open code
2·5	$x := ln(y)$	Not known
0·32	$x := entier(y)$	Open code
0·3 to 0·32	$p0$, $p1$, $p2$ and $p3$	Stack mechanism and special instructions
1900		
5·9	'man or boy'	Environment control is poor due to too few index registers
0·45	$k := 1{\cdot}0$	Type conversion done at compile time
0·39 to 0·46	**array** $a[1:500]$ **array** $a[1:1, 1:1]$ **array** $a[1:1, 1:1, 1:1]$	Only just outside range due to poor 1108 and B5500 times
0·3	$x := ln(y)$	Special coding for $y = 1{\cdot}0$! (This was the only value tested)
1108		
0·35	$k := 1{\cdot}0$	Type conversion at compile time
0·26	$x := y \uparrow z$	Error in $\uparrow$, coded as $x := y \uparrow 1$
5·2	**begin real** a; **end**	Storage at block level, assigns zero to a
7·2	**array** $a[1:500]$	Assigns zero to elements of the array
0·28	**goto** *abcd*	Single instruction
2·9	**switch**	Not known

RATIO	ALGOL SOURCE TEXT	REASON
0·23	$x := abs(y)$	Only two instructions; load magnitude y, store x
0·44	$x := exp(y)$	Standard procedure mechanism not used
5·5 to 7·4	$p0$, $p1$, $p2$ and $p3$	Slow set-up, fast per parameter

Instruction executed

Since this is very similar to the execution times, only those exceptional cases of listing which do not appear above, are listed.

RATIO	TEXT	REASON
ATLAS		
2·2	$k := l \div m$	Requires to test the sign of the numbers—done as open code
KDF9		
3·0	$x := ln(y)$	Large number of register (stack) manipulation instructions used
B5500		
0·41	$l := y$	Type conversion done by hardware
0·42	$x := y \uparrow 2$ $x := y \uparrow 3$	Done by repeated multiplication in the stack
1900		
0·44	**switch**	Simple switches are optimised
2·0	$p3(x, y, z)$	

Code size in instructions

Very much less variation occurs. All the anomalous ones are listed.

RATIO	TEXT	REASON
ATLAS		
0·43	$x := l$	Type conversion very straightforward
2·2	**array** $a[1:1, 1:1, 1:1]$	Open code used
B5500		
2·1	**goto** *abcd*	Dummy instructions used so that label starts at word boundary
1900		
0·32	$p0$	Special coding used when no parameters
1108		
2·5	$p0$	Produces code in the same way as if parameters were used. Sets aside word containing number of parameters.

Code size in bits

The variations in this characteristic are similar to those of the code size in instructions. The only additional anomalous one was:

RATIO	TEXT	REASON
ATLAS		
2·0	**begin real** a; **end**	Block level storage, also sets up information for diagnostics.

case repeated multiplication is used. This has since been corrected by the author for the KDF9—which of course means that $x := y \uparrow z$ with $z = 1{\cdot}0$ is now much slower (since it involves *ln* and *exp*).

gotos in ALGOL are, in general, very complex. Atlas does no elementary optimisation with them but uses a subroutine. KDF9 and the 1108 generate a single instruction for a **goto** within a block, but KDF9 generates extra code for passing a label, which is used for diagnostic purposes. Switches also present problems because the element of a switch list can be a complex designational expression rather than a label. The B5500 and 1900 compilers optimise simple switches.

A measure of the difficulty of handling call-by-name parameters with the associated environment control is given by the time to execute Knuth's 'man or boy'. Atlas performs very well because all the environment information (the display) is kept in registers. On the other hand, with only three index registers on the 1900, the time taken for Knuth is nearly six times the expected value.

The times taken to execute the simple procedure calls varies substantially. Not surprisingly the B5500 comes out best, because of the stack mechanism and special instructions for procedure calls. KDF9 does well because the dummy procedures are classified as simple. In this case, the stack pointer is used as the environment pointer within the procedure. This substantially simplifies the calling mechanism. The 1108 does very badly due to a long set-up process, although the length of time to deal with a parameter is quite short.

The anomalous values for execution time and the number of instructions executed are, of course, very similar. In the same way the anomalous values for the code size in instructions and the code size in bits are very similar. However, there is not the large variation in the size of object code compared with the execution times. This means that the overall ratios used to predict the number of instructions generated and the compiled code size in bits is fairly accurate.

The individual compilers

Atlas:

The large number of registers ($\simeq$90), and the rich instruction set is a great advantage. The compiler does very little optimisation and yet produces very tolerable machine-code. The current environment is stored entirely in registers, so that every variable is accessed by a single instruction which is a small offset from one of the address registers. The disadvantage of Atlas is that a large proportion of the 24 bits of the address field in every instruction is zero. Added to this, the compiler produces open code for virtually everything except **gotos.** Hence the size of the compiled code is large in terms of the number of instructions and enormous in terms of bits. This is only tolerable due to the paging mechanism. Because of the lack of optimisation and use of open code, the machine code is extremely easy to understand, so the author would recommend study of this for anybody who wished to know how ALGOL 60 is compiled.

KDF9:

The Kidsgrove compiler for KDF9 (Hawkins and Huxtable, 1963) was produced with the aim of doing very extensive array subscript optimisation. Unfortunately this sometimes produces incorrect code, so the option to do the optimisation is rarely used. Without this optimisation, the array accessing code is very poor. Some improvements have been made by Oxford University and the author to overcome this defect, but they are not considered here. The KDF9 itself presents substantial problems to the compiler writer. The stack mechanism does not allow for automatic overflow, so the compiler must empty the stack on procedure and function calls. Environment control and array accessing are not very convenient on KDF9 because the address registers must be loaded and unloaded via the stack. Although KDF9 does execute more instructions than many machines quite a large proportion of the instructions only involve the stack and so are relatively fast.

B5500:

The stack mechanism for this machine is designed explicitly for ALGOL. Unfortunately only two registers are available for the top of the stack and hence the stack instructions are not necessarily very fast. The special instructions for procedure call, array access, name call, etc. ensure that no more instructions are executed on this machine than with conventional ones even though it is a zero address computer. The real advantage of the B5500 is that it has very compact instructions (12 bits) and a short address length (10 bits). This allows B5500 ALGOL programs to be half the size (in bits) of their conventional counterparts. The space required for data, is, of course, unaffected. 'Man or boy' does not work on the B5500 due to a restriction on the stack size.

1900:

This computer series has a conventional one-address architecture. From the point of view of ALGOL it has the disadvantage of only three index registers. This makes environment control and call-by-name very difficult, and is reflected in the time taken to execute 'man or boy'. The compiler assigns store sometimes at procedure level and sometimes at block level. The author has been told that the current versions of this compiler now assign all simple variables at procedure level. The compiler does a fairly extensive amount of simple optimisation, so that there are very few short sequencies of code that could be radically improved.

The address of the front of the stack is not kept in a register, which means that procedure entry and exit is not very fast. In order to simplify the parameter handling problem, a simple 'thunk' mechanism is used for all parameters (Ingerman, 1961).

1108:

The 1108 is a multi-register one-address machine. It has a large instruction set which is moderately well utilised by the compiler. Storage allocation is at block level which incurs a significant penalty on block entry, and probably on procedure call. Unfortunately the compiler specification states that all variables are assigned a value of zero on declaration. Apart from the substantial overhead on array declaration, this means that block entry must always generate some code. The intelligent use of the different registers on the machine is the main reason why the compiled code is notably more compact than with the other conventional machines.

The compiler does some surprising optimisation including the evaluation of constant subexpressions. In one case this evaluation was incorrect. The main weakness of the compiler is the long time for a procedure call.

Conclusions

The main advantage of a non-conventional architecture for the compilation of ALGOL 60 appears to be the production of extremely compact object code. This is achieved with the B5500 by a very short address length within an instruction. Because of the dynamic storage allocation of ALGOL 60, access to simple variables is always by a small offset from an environmental pointer. Hence an address length within an instruction of only 9 bits is adequate. Anything in excess of 9 bits is likely to be wasted. On the other hand, several index registers or their equivalent are necessary for environment control and array accessing. Such registers must be capable of being updated rapidly for procedure entry and exit, and for access to name parameters.

Access to array elements is usually via an array word which

can be addressed in the same way as a simple variable. A short address length may preclude some array access optimisation, for instance if '*a*' is a global array of fixed size *a*[200] could be accessed by a single instruction provided the address field was large enough. In fact the B5500 does not allow array accessing optimisation because the storage protection system depends upon access via the array word (descriptor). The optimisation produced by the ALCOR compilers (Grau, 1967), could be done on a machine with a short address length, but not the B5500.

Array bound checking is an area where special hardware can be used to great advantage. Unfortunately the hardware on the B5500 does not deal with the general value of the lower bound, so that explicit code must be generated by the compiler to subtract the value of this lower bound if it is non-zero. Options to do bound checking on other machines tend to be very expensive in processor time. The 1108, although having no built-in hardware for array accessing, has a convenient instruction for bound checking. With this instruction, a single test can be made to see if the operand lies within the range defined by two registers.

Apart from the production of compact code from ALGOL 60, it is clear that in many scientific fields non-conventional machines can have other substantial advantages. Array bound checking has already been mentioned, but other examples lie outside the scope of this paper, for instance distinction between data and program and the ability to share the available core store between processes. The majority of these advantages are in the field of operating system design, and so are not considered here. Such advantages are likely to have a substantial effect upon the performance of the compiling system itself, and the easy way in which such systems can be developed.

References

Burroughs B5500 Extended ALGOL Language Manual, 1966.

GRAU, A. A., HILL, U., and LANGMAACK, H. (1967). *Translation of ALGOL* 60, Berlin; Springer.

HAWKINS, E. N., and HUXTABLE, D. H. R. (1963). A multipass translation scheme for ALGOL 60, *Annual review in Automatic Programming*, Oxford; Pergamon Press.

HEINHOLD, J., and BAUER, F. L. (Editor). (1962). Fachbegriffe der Programmierungstechnik, Ansgearbeitet vom Fachausschutz Programmieren der Gesellschaft für Angewandte Mathematik und Mechanik (GAMM), 2 Anfl. Munchen, Oldenbourg-Verlag.

INGERMANN, P. Z. (1961). Thunks—A way of compiling procedure statements with some comments on procedure declarations. *CACM*, Vol. 4, No. 1, pp. 55-58.

KNUTH, D. E. (1964). Man or boy? Algol Bulletin No. 17, page 7, Mathematische Centrum, Amsterdam.

RANDELL, B., and RUSSELL, L. J. (1964). *ALGOL* 60 *Implementation*. APIC Studies in Data Processing No. 5, London; Academic Press.

SCOWEN, R. S. (1965). Quickersort, Algorithm 271. *CACM*, Vol. 8, No. 11, page 669.

WICHMANN, B. A. (1969). A comparison of ALGOL 60 execution speeds. National Physical Laboratory, CCU 3.

WICHMANN, B. A. (1970). Some statistics from ALGOL programs. National Physical Laboratory, CCU 11.

WICHMANN, B. A. (1971). The performance of some ALGOL systems. To appear in the proceedings of the IFIP congress 1971.

Correspondence

To the Editor
The Computer Journal

Sir,

Suggested Extension to FORTRAN IV

When endeavouring to translate an ALGOL program to FORTRAN recently, I came across a statement of the type:

for $I = 1$ **step** 1 **until** 10, 15, 20 **step** 10 **until** 100, $I + 100$
while $(B \wedge (A \leqslant C)$ **do**

A statement of this type obviously cannot be translated into FORTRAN without a great deal of complication.

On the other hand, it would seem a logical extension to FORTRAN to allow DO loops of an alternative type, of the general form as follows (or a similar form):

DO *n* I = /*n*1, *n*2/*n*3/*n*4, *n*5, *n*6/*I* + (integer variable or expression), (Boolean variable or expression)/

With this, the ALGOL expression above would become, in Extended FORTRAN:

DO *n* I = /1, 10/15/20, 100, 10/I + 100, (B. OR. (A.LE.C))/

The usual form of the DO loop would, of course, be retained. Parsing of the above would be distinguished firstly by the slash following the equals. The items between the slashes would constitute a 'DO' list element, similar to the for list element in ALGOL.

A further improvement might be to allow the use of negative and/or real stepping values in the suggested form, which would increase the power of FORTRAN considerably.

Yours faithfully,

A. J. FINN

'Aeschi'
Salthaugh Road
Keyingham
Hull HU12 9RT
11 October 1971