A synthetic benchmark

H J Curnow and B A Wichmann
Central Computer Agency, Riverwalk House, London SW1P 4RT
National Physical Laboratory, Teddington, Middlesex TW11 OLW

Computer Journal, Vol 19, No 1, pp43-49. 1976

Abstract

A simple method of measuring performance is by means of a benchmark program. Unless such a program is carefully constructed it is unlikely to be typical of the many thousands of programs run at an installation. An example benchmark for measuring the processor power of scientific computers is presented: this is compared with other methods of assessing computer power. (Received December 1974)

An important characteristic of computers used for scientific work is the speed of the central processor unit. A simple technique for comparing this speed for a variety of machines is to time some clearly defined task on each one. Unfortunately the ratio of speeds obtained varies enormously with the nature of the task being performed. If the task is defined informally in words, large variations can be caused by small differences in the tasks actually performed on the machines. These variations can be largely overcome by using a high level language to specify the task. An additional advantage of this method is that the efficiency of the compiler and the differences in machine architecture are automatically taken into account. In any case, most scientific programming is performed in high level languages, so these measurements will be a better guide to the machine's capabilities than measurements based on use of low level languages.

An example of the use of machine-independent languages to measure processing speed appears in Wichmann [7] which gives the times taken in microseconds to execute 42 basic statements in ALGOL 60 on some 50 machines. The times were measured by placing each statement in a loop executed sufficiently often to give a reasonable interval to measure. The time for the statement is found by taking from this interval the same measurement with a dummy statement and dividing by the number of repetitions. The two thousand or so time measurements provide a lot of information about the various implementations of ALGOL 60 but do not directly give a performance measure. With basic statement times for only two machines, the average of the 42 ratios between the times provides a simple comparative measure. This technique can be generalised by assuming that the times T_{ij} for a statement i ($i = 1$ to n) on machine j ($j = 1$ to m) satisfies

$$T_{ij} \approx S_i \times M_j$$

where S_i is a time depending only upon the statement and M_j depends only upon the machine (arbitrarily take one machine as unity). A least-squares fitting process gives the S_i and M_j from the times (see [8], page 70, for details). The M_j provide a comparative measure without having to assign weights to the individual statements.

Figure 1: Some interpretive instruction counts

Whetstone instruction	*Algol context*	*Dynamic frequency per thousand*	*Static frequency per thousand*
Take Integer Result	Access to a simple integer variable	129	73.5
Take Real Address	Access to real array element or storing simple real variable	68	48.4
$\times$	Integer or Real multiplication	36.2	18.9
Procedure entry		20.1	18.7
GoTo Accumulator	**goto**	2	7.4
Call Block	Entry to a block other than procedure body or for statement	0.2	1.3
IMPlies	Boolean operator	0	0

1 Statistics of language usage

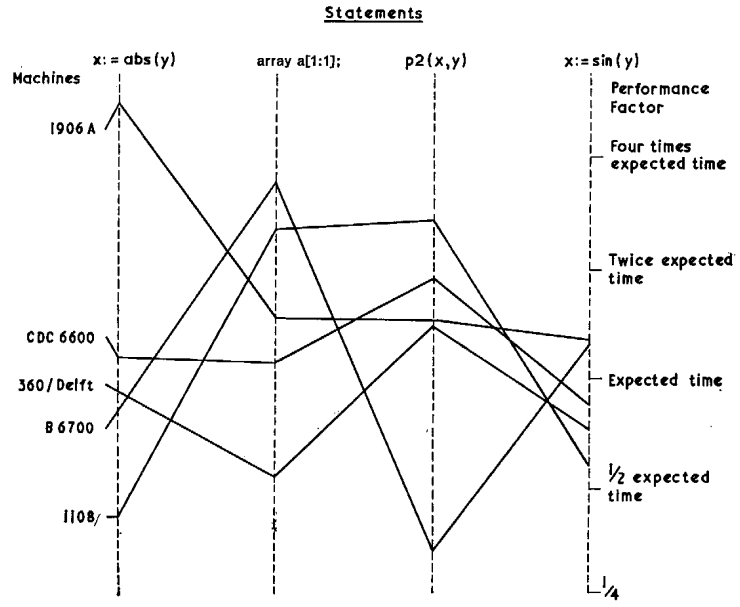
The comparative measure from the basic statements given above is not a true performance indicator, because the measure does not take into account that some features of ALGOL 60 are much more frequently used than others. Moreover, many of the basic statements are handled with enormous variation in efficiency. For instance, executing a block containing only a simple variable declaration produces no instructions with the 1900 XALT compiler but involves two supervisor calls with the 360 ALGOL 60 F compiler. This variation is illustrated in Fig. 2, which shows the times divided by the basic machine speed (the M_j) for a few of the basic statements.

Weights for the basic statements must be based upon statistics of language usage. Such information has been collected at NPL and Oxford University from 949 programs (Wichmann [6]). In the Whetstone system (Randell and Russell, [3]) the ALGOL program is first translated into an intermediate code which is then executed by an interpreter program. By modification of the translator and interpreter programs both static and dynamic frequency counts of some 124 instructions were obtained. Part of this information is produced in Fig. 1. Weights for the basic statements have been calculated from these statistics to give a performance measure similar to the Gibson mix.

The ALGOL statement mix has a number of disadvantages. Firstly, it is not always easy to obtain in one program sufficiently accurate processor times for the 42 measurements. Secondly, many of the basic statements were too simple to reflect the general language features they were supposed to represent. The most obvious case of this was that constant subscripts were used in the array accessing statements. The program containing the statements was too simple to be typical of scientific programs in general and in consequence an optimising compiler could perform better than on average programs. Fortunately, moving code out of loops and similar extensive optimisation is rarely performed by ALGOL 60 compilers and hence the technique was viable when confined to that language.

A single program more typical of ALGOL usage was therefore required to measure processor speed. Such programs are called synthetic benchmarks.

Figure 2: Variation in performance of ALGOL basic statements



2 The design of the benchmark

The design of the program was a compromise between requirements for simplicity and complexity. The program had to be simple enough to be easily transferred from one machine to another and translated into different languages. It also had to be possible to produce the Whetstone intermediate code, calculate the frequencies of the different instructions and match them to the available statistics. One simple program was constructed using the basic statements described above, but when this was, translated into FORTRAN and put through optimising compilers, large parts of the program were not executed because they were not logically necessary. Less dramatically but also of significance such a compiler could make a simple program run at an untypically high speed by using the fast registers as temporary data stores. Thus it became apparent that the program should be complex enough to seem typical when presented to an intelligent compiler. It was impossible to meet this requirement completely because the necessary information about program structure was not available, and the architecture of computers is so diverse. The best course to follow appeared to be to ensure that the program could not be logically optimised and to hope that by writing code which looked natural the use of fast registers and other hardware features would be approximately correct. The program would then provide a useful measure of performance with well written source code. Language peculiarities were to be avoided so that the program would be useful for comparing languages having a similar subset.

Thus one is led to having a number of modules of different types using different language features, each executed many times by inclusion within a **for** loop. Each module should represent a genuine calculation, preferably producing different results

on each execution but capable of indefinite repetition without leading to numeric overflow or other mishap. The structure of the modules and the number of repetitions of each should be chosen to match the frequencies of the Whetstone instructions which would be generated by the benchmark program to the instruction frequencies obtained from the 949 programs.

The choice of the method of fitting the benchmark program to the analysis is an important matter. As many instructions as possible should be fitted by the choice of a comparatively small number of module repetition counts. One method would be to minimise the root mean square difference between the instruction frequencies in the benchmark and in the analysis. A refinement would be to weight the instructions according to their frequency or to their execution times, thus giving greater weight to fitting the more important instructions. The modules themselves would be redesigned until the fit obtained by the minimisation process was a good one. The methods actually used will be described later.

3 Construction of the program

3.1 The modules

The computations to be carried out should be genuine calculations producing different results on each execution, and capable of indefinite repetition. For the elementary mathematical operations a set of transformation statements was investigated. The statements in ALGOL were:

$$x1 := (x1 + x2 + x3 - x4) \times 0.5;$$

$$x2 := (x1 + x2 - x3 + x4) \times 0.5;$$

$$x3 := (x1 - x2 + x3 + x4) \times 0.5;$$

$$x4 := (-x1 + x2 + x3 + x4) \times 0.5;$$

This transformation provides a convergent sequence of x_i values which theoretically tend to the solution

$$x1 = x2 = x3 = x4 = 1.0$$

but the convergence for approximate computation and for other values of the x's are of interest. It was found that the transformation is convergent for all values of the x's, but to allow for machine accuracy limits and to provide some changing computations the factor 0.5 in the statements was replaced by 0.499975. An arbitrary choice of initial values such as $x1 = 1.0$, $x2 = x3 = x4 = -1.0$ gives a plausible sequence of calculations. This scheme of computation was used as the basis for three modules (Nos. 1. 2 and 3) in which the variables were respectively simple identifiers, array elements and elements of an array parameter of the procedure *pa* (see program text in the Appendix). A programmed loop was included in *pa* to control the proportion of parameter references to procedure calls. Other almost stationary transformations were used in Modules Nos. 7 and 11. The first used trigonometric functions as follows:

$$x := t \times arctan(2.0 \times sin(x) \times cos(x)/(cos(x \times y) + cos(x - y) - 1.0));$$

$$y := t \times arctan(2.0 \times sin(y) \times cos(y)/(cos(x + y) + cos(x - y) - 1.0));$$

With a value of $t = 0.499975$ and starting with $x = y = 0.5$ these almost transform x and y into themselves, and give a slowly varying calculation. This module was designed to give the correct proportion of calls to *sin* or *cos* and *arctan* and to avoid carrying forward common sub-expressions. The second, module No. 11, exercised the other standard functions in a single transformation:

$$x := sqrt(exp(ln(x)/t1));$$

With $t = 0.50025$ and an initial value of $x = 0.75$ a stable series of values of x is obtained with repeated execution of this statement. Note that in these modules multiplication or division was chosen so that overall there would be a good fit to the statistics.

Conditional jumps were exercised in Module No. 4 by a set of conditional statements. Starting from an initial value of $j = 1$, repeated execution of this group of statements alternates j between 0 and 1, and each condition is alternately true and false.

Integer arithmetic and array addressing were used in Module No. 6 in a simple calculation. With initial values of $j = 1$, $k = 2$, $l = 3$ these values remain unchanged with repeated executions, but this is unlikely to be detected by the compiler. Procedure calls and parameter addressing were exercised in Module No. 8 by a procedure (*p3*) with three parameters. The global variable t has the same value as in other modules. Values of the actual parameters corresponding to x and y are unchanged.

Array references appeared in Module No. 9 which was made the body of a parameterless procedure ($p0$) to increase the number of procedure calls. Global variables are set up as j = 1, k = 2, l = 3, el[l] = 1.0, e1[2] = 2.0, e1[3] = 3.0, and these values are permuted each time the module is executed.

Finally, in Module No. 10, simple integer arithmetic was used. With initial values of j = 2, k = 3, this interchanges values with each execution.

4 Fitting

Each of the modules was translated into Whetstone instructions, including the coding of the containing **for** loop, using a vocabulary of some 40 different instructions. To simplify the task of fitting their frequencies to the analysis, and also to take account of some of the other instructions not directly represented in the benchmark, some combinations were made to reduce the number of fitted values to 25. These accounted for over 95% of the instructions executed in the analysed programs. The problem was thus to choose the execution frequencies of the ten modules so that the 25 instruction frequencies matched. The modules had been designed with this fit in mind, but there would not be an exact solution. The first approach was to obtain a least-squares fit using a standard method. This solution suffered from two disadvantages. Firstly, it gave negative frequencies for some of the modules, which would have been difficult to implement, and secondly it gave equal importance to fitting each of the instructions. The first problem was overcome by using a direct search method with the range of the parameters restricted to be positive, taking the previously calculated result as a starting point. The second was overcome by allotting to each instruction a total time, which was the product of the instruction frequency and an instruction time. The instruction times were derived from the basic statement times, which in turn were derived from measurements on a large number of machines [6]. Using these times, a weighted root

mean square deviation of the benchmark from the analysis was defined, and this was minimised by the direct search method. The result was that two of the modules, numbers 1 and 10, were eliminated by the restriction to positive frequencies. The remaining eight modules gave a weighted root mean square deviation over the 25 instructions of 15% which was considered satisfactory. The total nominal time of 5.93 seconds compared with a target of 6.08, and the total instruction count of 963 thousand compared with the target of one million.

Having decided upon the modules and their relative frequencies the whole program could be assembled. The modules are contained within a framework which controls the number of executions of each and provides for the output of results. This output is only required to ensure that the calculations are logically necessary; it is not intended to represent the output from a typical program. The execution frequency of each module is proportional to the input value of i and the scaling factors are such that a value of i = 10 gives a total weight to the modules corresponding to one million Whetstone instructions. Although Modules Nos. 1 and 10 have zero frequencies they have been retained in the program because study of the object code they produce might be of interest. For accurate measurements the effect of the framework, and if necessary of compilation and loading, may be eliminated by performing several runs with different values of i. From these results the speed of the machine may be expressed in Whetstone instructions per second.

5 Results from several machines

Using the benchmark described above, a survey was made of the execution speeds which might be expected from one machine using different languages, compilers and precisions. The object was partly to obtain information about the languages and their implementations and partly to study the usefulness of the synthetic benchmark technique for such a study.

The first machine studied was an IBM 360/65. The operating system provided information about CPU time used (in micro-hours), peripheral channel program execution and core occupied. Object code listings were obtained from all the compilers, except the ALGOL one. Brief descriptions of the compilers follow:

ALGOL F:	The standard IBM ALGOL compiler. Known to produce inefficient code.
FORTRAN G:	The standard IBM FORTRAN medium sized compiler.
FORTRAN H:	The superior IBM FORTRAN compiler. Produces better code than G at the expense of compilation time and does logical optimisation.
PL/I F:	The standard IBM PL/I compiler.
PL/I OPT:	The new IBM PL/l optimising compiler.

Controlling parameters were chosen to give the maximum execution speed of the compiled program, and in the case of ALGOL a compiler option also controlled the precision of the representation of real quantities. The ALGOL program was used as described above with insignificant changes to the input and output procedures to suit the IBM implementation. For FORTRAN an equivalent program was written to perform the same calculations using similar language features. These do not give exactly

Figure 3: **360/65 execution speeds** (speeds in thousands of Whetstone instructions per second)

Representation		*Algol*	*FORTRAN*		*PL/I*	
real	*integer*	F	G	H	F	OPT
6HEX	31BIN	72	430	409 521 [1]	372	443
14HEX	31BIN	65 [2]	321	421	302	335
6HEX	6DEC	-	-	-	163	262
6HEX	15BIN	-	370	-	-	-

the same facilities as in ALGOL but the differences are probably not significant in most applications. The types of the variables were defined either implicitly by their identifiers according to the FORTRAN convention, or by suitable declarations; the names of the standard functions were also modified as necessary. For PL/I the program was derived from the ALGOL version and was very similar to it.

Various internal representations of real and integer quantities were used. Real quantities were either short or long precision floating point (6 hexadecimal digits, 6HEX or 14 hexadecimal digits, 14HEX). Integer quantities were either whole-word or half-word binary (31 bits, 31BIN or 15 bits, 15BIN) or nine digit packed decimal (9DEC). The execution speeds of the programs produced by the various compilers with different combinations of these representations are given in Fig. 3; the speeds quoted are in thousands of Whetstone instructions per second. The program had been designed to minimise the improvement which an optimising compiler could make by re-arrangement of the logic, and examination of the object code showed that this had been achieved. The two figures shown for FORTRAN H with standard precision refer to no and full optimisation. The better performances of H compared with G and of PL/I OPT compared with PL/I F were the result of the use of fewer machine instructions and, to a lesser extent, of faster ones; H was particularly good at holding values temporarily in registers. Without the object code listing the reasons for the slow performance of ALGOL could not be determined, but it is known that there is an array check on each array access which was not made in FORTRAN or PL/I as run. Also the procedure call and parameter checking and passing mechanisms were more complex.

The use of long precision (14HEX) instead of short (6HEX) caused a 20-25%, loss of speed in both FORTRAN and PL/I. This could have been caused by extra machine instruction times for double-length working and by extra computations needed for precision in standard functions. In ALGOL the proportional reduction in speed was less, but the increase in execution time was about double that with the other languages. Using decimal (9DEC) instead of binary (31BIN) representation for integers in PL/I had a severe effect, particularly with the F compiler. This might be important since decimal representations could inadvertently result from such simple declarations as FIXED, which would give 5DEC or REAL(9,2). In comparing the languages it must be remembered that although the programs were made as nearly equivalent as possible, faster execution was obtained at the expense of programming language power, run-time error checks and compilation time. The ALGOL compiler took only one tenth of the CPU time taken by the PL/I Optimizer, but produced a program that took six times as long to execute. The program is not, however, intended to provide a good measure of compilation speed.

The other machine studied was an ICL 1904A. Programs were run from a MOP

Figure 4: **ICL 1904A execution speeds** (speeds in thousands of Whetstone instructions per second)

	Algol	*FORTRAN*			
	XALT	*XFAT*		*XFEW*	
Trace level	*single*	*single*	*double*	*single*	*double*
0	125	159	20.8	192	21
1	58	91	19.2	100	19
2	52	17	9.4	59	17.5
3	55	-	-	-	-

terminal under George 3 and the mill-time in seconds and the core occupied were obtained from the system messages. Measurements of simple programs on the same machine had shown agreement with the quoted instruction times, so the conversion factor to true seconds was probably correct. The compilers were all standard ICL products for ALGOL and FORTRAN:

ALGOL XALT 32K	disc compiler
FORTRAN XFAT 32K	disc compiler good diagnostics
XFEW 48K	disc compiler optimised code

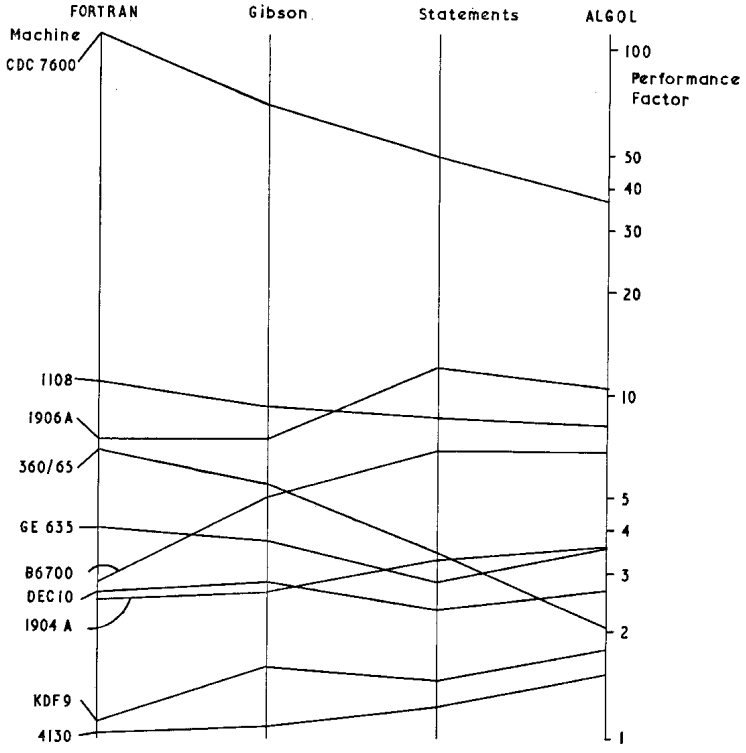
For ALGOL the program was used with only minor changes. Real quantities were represented in the machine to standard single precision, 37 binary digits floating point. For FORTRAN two versions were used which gave single and double (74 binary digit) precisions. The execution speeds of the programs are given in Fig. 4. In comparing the ALGOL and FORTRAN speeds it is necessary to consider the different facilities provided at run for error checking and for tracing program flow. These are controlled by TRACE statements which direct the compilers and come in different combinations in the two languages. At the lowest level, TRACE 0, both languages are blind to all but the grossest errors; also at the TRACE 2 level the facilities are roughly comparable. Table 3 shows the speeds of execution of programs compiled with different compilers on the same machine, at different TRACE levels. The main degradation in ALGOL comes with TRACE 1 which adds frequent overflow and array subscript checks; the error traceback with TRACE 2 only adds a small overhead to the already complex ALGOL procedure call mechanism. In FORTRAN both steps involve heavy penalties; TRACE 1 gives a subroutine error traceback while TRACE 2 gives overflow checks and a detailed statement traceback. This last seems a large price to pay if all one requires are the overflow and array subscript checks which come in the same package deal; using explicit program checks would be preferable.

5.1 A comparison of four performance measures

The four performance measures considered are:

1. The Gibson mix. This is a weighted average of instruction times, reflecting usage expected of scientific work. The values were calculated by the Technical Services Division of the CCA and include an additional allowance for the address structure and the number of registers.

Figure 5: Four measures compared



2. The ALGOL statement mix.

3. The ALGOL synthetic benchmark.

4. The FORTRAN synthetic benchmark.

All these measures are true performance indicators in the sense that they take into account the relative usage of the various instruction types in a machine. These are in low-level terms in the Gibson mix whereas the other three are expressed in high-level language form.

The four measures can be compared directly in diagrammatic form as in Fig. 5. The scales are logarithmic to include computers of widely differing powers. In any case, hardware and software improvements tend to yield a percentage rather than an absolute gain. Since the units of each measurement are more or less arbitrary, the scales have been adjusted vertically to correspond as closely as possible. (In fact, the average gradient of the lines joining each measure is zero.)

One must add a note of caution about the actual values plotted. With four measures and twelve computers, 48 measurements are needed. However, to display as many machines as possible, when three measures were available the fourth has been estimated (in seven cases). Also, the compiling options used on different machines on the same range or with the two ALGOL programs have not always been the same. These differences have arisen because of the varying choices made at different computer installations.

Visual inspection of the diagram reveals that measurements on members of a range of computers (using the same software) shows a similar pattern. The two ALGOL measures are clearly more correlated than the others. The ALGOL and FORTRAN benchmark results differ enormously in some cases by nearly a factor of three.

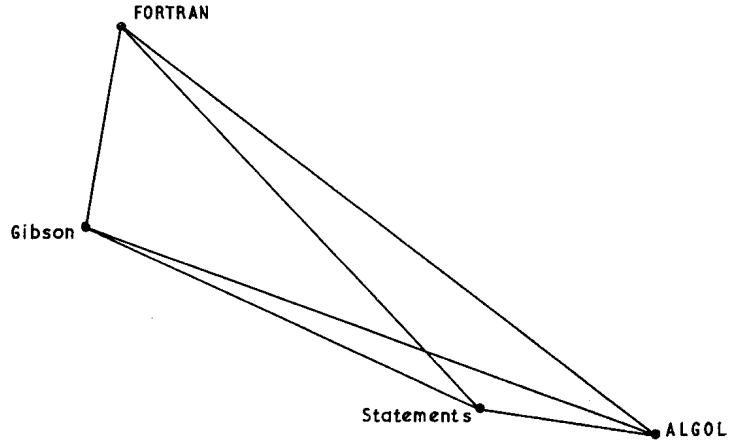
The correspondence between the measures can be quantified by calculating the average gradient of the lines joining two measures (ignoring the sign of the gradient). This average slope corresponds to the mean difference (a ratio) between the two scales. The six ratios are:

ALGOL —	Statements	ratio	1.21
Gibson —	FORTRAN		1.24
Gibson —	Statements		1.57
FORTRAN —	Statements		1.69
Gibson —	ALGOL		1.85
FORTRAN —	ALGOL		2.00

Hence two computers with one twice as powerful as the other on the FORTRAN scale, could be the same on the ALGOL scale. For an example of this, see the KDF9 and the 360/65 values. In contrast, much smaller differences are to be expected between the ALGOL and Statement scales.

The six ratios given above can be used to plot the distances between four points representing the scales. With a very small amount of distortion the logarithm of these ratios can be plotted as distance between points on a plane giving Fig. 6. This figure again illustrates the high correlation between the ALGOL and Statement scales. The close relationship between FORTRAN and Gibson is particularly remarkable in view of the diverse routes by which they were derived. Note that there is a roughly linear relationship — FORTRAN, Gibson. Statements, ALGOL. This can be seen from the previous graph where a large number of the lines maintain a roughly constant gradient.

Figure 6: The relationship between the four measures



6 Limitations and applications

The benchmark program described here has been presented as a model of the large number of programs originally analysed. The intention is that by running it upon a new type of machine one may learn something of the performance the machine would have if it ran the original programs. It is not possible to validate the benchmark by direct comparison with the original programs so one must estimate how faithful a model it is likely to be. The performance of any machine depends upon the characteristics of the stream of machine code instructions executed and the data values, so the benchmark program must generate code which is typical from a number of aspects. This benchmark is probably typical in the mixture of machine code functions it exercises and in their sequencing. For example, on the 1904A, examination of the object code showed that the proportion of branch instructions was 19%, which is a reasonable figure in the light of experience on Atlas [4].

The number of executions of each loop is high and it would have been better to arrange the repeated execution of the whole program by obtaining a measurable load, rather than increasing the repetition counts of the modules individually. The values of the operands of arithmetic imistructions take varying values as the program rums, but no attempt was made to make these representative, the main object being to avoid repetition of a small number of values which would certainly have been abnormal. The utilisation of such system features as the standard function library should be typical but, of course, calls to the input/output system and to supervisory software are not represented. The most important way in which this program is not typical is in the size of the area of store that it addresses, which is very small, both as regards data and code. This is not significant on unsophisticated machines but becomes so when machines with multi-level stores of various types are considered. It may well be true that on the 360/65 the use made by the FORTRAN H compiler of the general purpose registers was reasonably typical, although it did manage to perform the whole of module 4 in registers. On a machine with one or more of the various types of buffered, slaved or virtual stores all one can say is that the program is unusually small and simple in its patterns of access. A modified version has been produced which accesses a larger area of store, but this must clearly be an arbitrary exercise. When more is known about

the characteristics of programs running on these multi-level store machines it may be possible to produce a typical program for particular types of machine. It will clearly be impossible to produce one valid for any conceivable machine.

Despite these limitations the program described should be of some value, particularly in relation to smaller machines. It provides a measure of computer speed of a similar level of usefulness as the Gibson mix, etc. but taking account of the actual compilers available on the machine. It is particularly suitable on machines with unusual architecture where the Gibson mix is difficult to interpret (e.g. KDF9. Burroughs). By comparison with other measures it provides some inidication of the software inefficiency incurred by using the various high-level languages. The different compilers and options can be evaluated on one machine. Another possible application for this program would be as the processor load component in a more comprehensive total load benchmark applied to a complete system. Although this benchmark program is of limited use, since it represents only small scientific programs, the general principles used in its construction could be applied more widely. The chief obstacle to progress here is, as in other areas of computer performance measurement, the lack of information about the characteristics of real programs.

Appendix: The Benchmark

```
begin
    real x1, x2, x3, x4, x, y, z, t, t1, t2;
    array e1[1:4];
    integer i, j, k, l, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11;
    procedure pa(e);
        array e;
        begin
        integer j;
        j := 0;
    lab:
        e[1] := (e[1] + e[2] + e[3] - e[4]) ×t;
        e[2] := (e[1] + e[2] - e[3] + e[4]) ×t;
        e[3] := (e[1] - e[2] + e[3] + e[4]) ×t;
        e[4] := (-e[1] + e[2] + e[3] + e[4]) / t2;
        j := j + 1;
        if j < 6 then
            goto lab;
        end procedure pa;
    procedure p0;
        begin
        e1[j] := e1[k];
        e1[k] := e1[l];
        e1[l] := e1[j];
        end procedure p0;
    procedure p3(x, y, z);
        value x, y;
        real x, y, z;
        begin
        x := t×(x + y);
```

```
        y := t×(x + y);
        z := (x + y)/t2;
        end procedure p3;
procedure pout(n, j, k, x1, x2, x3, x4);
        value n, j, k, x1, x2, x3, x4;
        integer n, j, k;
        real x1, x2, x3, x4;
        begin
        outreal(1, n);
        outreal(1, j);
        outreal(1, k);
        outreal(1, x1);
        outreal(1, x2);
        outreal(1, x3);
        outreal(1, x4);
        end procedure pout;
comment initialise constants;
t := 0.499975;
t1 := 0.50025;
t2 := 2.0;
comment read value of i, controlling total weight: if i=10 the
        total weight is one million Whetstone instructions;
inreal(2, i);
n1 := 0;
n2 := 12 ×i;
n3 := 14 ×i;
n4 := 345 ×i;
n5 := 0;
n6 := 210 ×i;
n7 := 32 ×i;
n8 := 899 ×i;
n9 := 616 ×i;
n10 := 0;
n11 := 93 ×i;
comment module 1: simple identifiers;
x1 := 1.0;
x2 := x3 := x4 := -1.0;
for i := 1 step 1 until n1 do
        begin
        x1 := (x1 + x2 + x3 - x4) ×t;
        x2 := (x1 + x2 - x3 + x4) ×t;
        x3 := (x1 - x2 + x3 + x4) ×t;
        x4 := (-x1 + x2 + x3 + x4) ×t;
        end module 1;
pout(n1, n1, n1, x1, x2, x3, x4);
comment module 2: array elements;
e[1] := 1.0;
e[2] := e[3] :=e[4] :=1.0;
for i := 1 step 1 until n2 do
        begin
```

```
        e1[1] := (e1[1] + e1[2] + e1[3] - e1[4]) ×t;
        e1[2] := (e1[1] + e1[2] - e1[3] + e1[4]) ×t;
        e1[3] := (e1[1] - e1[2] + e1[3] + e1[4]) ×t;
        e1[4] := (-e1[1] + e1[2] + e1[3] + e1[4]) ×t;
        end module 2;
pout(n2, n3, n2, e1[1], e1[2], e1[3], e1[4]);
comment module 3: array as parameter;
for i := 1 step 1 until n3 do
        pa(e1);
pout(n3, n2, n2, e1[1], e1[2], e1[3], e1[4]);
comment module 4: conditional jumps;
j := 1;
for i := 1 step 1 until n4 do
        begin
        if j = 1 then
                j := 2
        else
                j := 3;
        if j > 2 then
                j := 0
        else
                j := 1;
        if j < 1 then
                j := 1
        else
                j := 0;
        end module 4;
pout(n4, j, j, x1, x2, x3, x4);
comment module 5: omitted;
comment module 6: integer arithmetic;
j := 1;
k := 2;
l := 3;
for i := 1 step 1 until n6 do
        begin
        j := j × (k − j) × (l − k);
        k := l × k − (l − j) × k;
        l := (l − k) × (k + j);
        e1[l − 1] := j + k + l;
        e1[k − 1] := j × k × l;
        end module 6;
pout(n6, j, k, e1[1], e1[2], e1[3], e1[4]);
comment module 7: trig. functions;
x := y := 0.5;
for i := 1 step 1 until n7 do
        begin
        x := t × arctan(t2 × sin(x) × cos(x)/
                (cos(x + y) + cos(x − y) − 1.0));
        y := t × arctan(t2 × sin(y) × cos(y)/
                (cos(x + y) + cos(x − y) − 1.0));
```

```
        end module 7;
    pout(n7, j, k, x, x, y, y);
    comment module 8: procedure calls;
    x := y := z := 1.0;
    for i := 1 step 1 until n8 do
        p3(x, y, z);
    pout(n8, j, k, x, y, z, z);
    comment module 9: array references;
    j := 1;
    k := 2;
    l := 3;
    e[1] := 1.0;
    e[2] := 2.0;
    e[3] := 3.0;
    for i := 1 step 1 until n9 do
        p0;
    pout(n9, j, k, e1[1], e1[2], e1[3], e1[4]);
    comment module 10: integer arithmetic;
    j := 2;
    k := 3;
    for i := 1 step 1 until n10 do
        begin
        j := j + k;
        k := j + k;
        j := k - j;
        k := k - j - j;
        end module 10;
    pout(n10, j, k, x1, x2, x3, x4);
    comment module 11: standard functions;
    x := 0.75;
    for i := 1 step 1 until n11 do
        x := sqrt(exp(ln(x)/t1));
    pout(n11, j, k, x, x, x, x);
    end
```

FORTRAN and PL/I versions of this program are available on request.

References

[1] Heinhold, J (1962) and Bauer, F.L. (Editors), *Fachbegriffe der Programmierungstechnik.* Ansgearbeitet vom Fachausschatz Programmieren der Gesellshaft für Angewandte Mathematik und Mechanik (GAMM) Oldenbourg, Munchen.

[2] KNUTH, D. E. (1971). An emipirical study of FORTRAN programs, *Software Practice and Experience*, Vol. 1, No. 2, pp. 105-133.

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

[4] SUMNER, F. H. (1974). Measurement techniques in computer hardware design, *State of the Art Report No. 18*, pp. 367-390, Infotech, Maidenhead.

[5] WICHMANN, B. A. (1969). *A comparison of ALGOL 60 execution speeds*, National Physical Laboratory Report CCU3.

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

[7] WICHMANN, B. A. (1973a). *Basic statement times for ALGOL 60*, National Physical Laboratory Report NAC42.

[8] WICHMANN, B. A. (1973b). *ALGOL 60 Compilation and Assessment*, Academic Press, London.

A Document details

Partly converted just before I retired, but finished in October 2001. The original was set in two columns. Other small changes have been made to reflect the default LaTeX style. Typo corrected, July 2010.