THE KDF9 AND BENCHMARKING

by **Bill Findlay**

1:BACKGROUND

This paper is one of a series on aspects of the English Electric (EE) KDF9 computer. For the nearest we have to a KDF9 reference manual, see [ICL69]. An overview of the KDF9, with a basic description of both the hardware and the software, can be found in [FindlayEE]. I present a fuller account of the KDF9 hardware in a second companion paper [FindlayHW], and of its software—both operating systems and compilers—in a third [FindlaySW]. The purpose of the present work is rather different. Here I focus on the rôle KDF9 played in the development of benchmarking, the attempt to measure and compare the performance of hardware and software platforms objectively, repeatably and accurately (which is to say, *scientifically*).

EE undertook performance measurements on their prototype KDF9, in order to provide support for a tender [Fell63]. Some small programs, implementing algorithms of particular interest to the prospective customer, were timed using the hardware clock [FindlayHW, §4.1], as delivered by Director's's 'OUT 3' system call (Director is the operating system). In a few cases this was supplemented by direct electronic observation using an oscilloscope. A 'theoretical' time for each example was calculated (not easy, see [FindlayHW, §9]) for comparison with observation.

The results indicated that the KDF9 ran about 5%-15% slower than the theoretical time. This was attributed to extra time taken: in aligning and standardizing floating point, in shifting fixed point values, in overhead from instructions needed to set up each test correctly, and in extra instruction fetches due to examples not starting in first syllable of a word. Where an oscilloscope result was available, its μs-accurate time tended to lie between the theoretical and measured values, suggesting that Director was slightly pessimistic. The **ee9** KDF9 emulator [FindlayUG] reports times that are a little slower again. This is largely due to ignoring the internal concurrency of the KDF9 CPU, a shortcoming that is particularly significant when there are many relatively slow FPU operations, as in the Fell tests.

Early attempts at *machine independent* benchmarking were based on 'mixes' of instructions thought to be typical of particular workloads [Longbottom]. The Gibson mix was derived from observations of scientific codes. It consists of a set of weights to be applied to the execution times of various instruction types, the result being an estimate of the mean instruction time. The Gibson mix rate for KDF9 is 170KIPS (0.170MIPS). The Whetstone benchmark, from historical figures, runs at 178KIPS. This indicates that, despite the simplistic basis of the technique, KDF9's actual performance is reasonably well represented by its mix rating.

It is interesting to compare KDF9 with other computers of its era (1962-64). In the following table, using data from [Longbottom], the second column gives the main store cycle time, the third column is the Gibson mix rating, and the fourth is a 'figure of merit' for the architecture, defined as the product of the store cycle time and the Gibson rating. Consider the Burroughs 5000 and the KDF9. They have the same main store cycle time (6μs) and word length (48 bits), but KDF9 is nearly three times faster.

To put it slightly differently, KDF9 needs about one store cycle time on average for each mix 'instruction' executed, while the B5000 needs nearly three. I conjecture that this is due to the B5000 keeping its expression evaluation stack and its return address stack in main store, whereas the KDF9 has special stacks for these purposes, the 'nesting stores', implemented with 1.5μs cores and in a way that overlaps much of their access times [FindlayHW, §4.2].

KDF9	**6μs**	**170 KIPS**	**1020**	**0.98**	**80 KWIPS**	**2.125**
ATLAS 1	2μs	350 KIPS	700	1.43	170 KWIPS	2.05
Burroughs B5500	**4μs**	**144 KIPS**	**576**	**1.72**	**64 KWIPS**	**2.25**
CDC 3600	1.5μs	337 KIPS	506	1.96		
Burroughs B5000	**6μs**	**60 KIPS**	**360**	**2.78**		
System	**Store cycle time**	**Gibson Mix rate**	**Figure of merit**	**Store cycles per G.M. order**	**Whetstone (FORTRAN)**	**G.M. KIPS per KWIPS**

One of the ways in which KDF9 ALGOL 60 left an important legacy was in the advent of application based performance measurement. This paper describes two seminal projects: the benchmark based on the Whetstone ALGOL system, and the Ackermann function probe of the efficiency of procedure calling.

We have FORTRAN Whetstone results for only KDF9 and the B5500 (the Atlas figure is for ALGOL 60 [C75]), but they do serve to confirm the plausibility of the remarks above. KDF9 EGDON FORTRAN achieves 80K Whetstone Instructions/second (KWIPS) and B5500 FORTRAN gets 64KWIPS, roughly in proportion to their Gibson mix rates, with the KDF9 doing slightly better, *pro rata*, at running the 'high-level' language!

But that does not tell the whole story. When it comes to an ALGOL version of the Whetstone Benchmark, the B5500 is 50% faster than the KDF9; largely, I conjecture, thanks to descriptor mechanisms that efficiently implement ALGOL constructs such as dynamic array access, 'call by name', and procedure calls. These are architectural features that would make the B5500 relatively slow in FORTRAN, which gets no benefit from them. FORTRAN was designed to work well with simple—IBM 704-style—indexed addressing, which KDF9 does superbly.

The paper *Basic statement times for ALGOL 60* [Wichmann73b] gives the following. With KDF9 EGDON ALGOL, `e1[1,1,1]:=1` takes 247μs, and with Kidsgrove ALGOL, `p3(x,y,z)` takes 136μs. B5500 ALGOL takes 66.6μs and 53μs respectively. That the KDF9 nevertheless outclasses the B5500 on the full FORTRAN benchmark, which is perhaps more representative of workloads outside the ALGOL niche, is testament to the general excellence of its architecture.

2: THE WHETSTONE BENCHMARK

ALGOL 60 [Naur60], the first 'structured' language, and the common ancestor of many of the most popular languages of the present day, had relatively early implementations on the KDF9 [FindlaySW, §2.3]. In fact, KDF9 has two ALGOL compilers that run under the Time Sharing Director: the Kidsgrove ('KAlgol') and Whetstone ('WAlgol') systems, which are named after the EE installations where they were written. The EGDON system also eventually gained a third ALGOL 60 compiler. The Whetstone and Kidsgrove compilers accept a common variant of the ALGOL 60 standard, defined to remove ambiguities and other defects of the language described in the *Revised Report*.

WAlgol [RR64] consists of a compiler (the 'Translator') that generates code for an ALGOL-oriented Whetstone virtual machine instruction (WI) set, and a WI interpreter (the 'Controller'). It is focussed on fast compilation and on error checking execution. The KAlgol compiler [HH63], by contrast, generates native KDF9 code focussed on fast execution and has global optimization features. The complementary pair anticipated IBM's PL/I 'checkout' and optimizing compilers by nearly a decade.

A benefit of using an interpreter is that it is relatively easy to augment it with code that produces useful data about the programs it runs. This can be used for debugging, for source code improvement, and for insight into program behaviour. The **ee9** KDF9 emulator offers such facilities for native KDF9 code, including execution traces, histograms of opcode usage, and execution frequency profiles. It is not hard to instrument the Whetstone Controller to gather similar statistical information about the run time characteristics of ALGOL programs.

Brian Wichmann carried out systematic studies of this kind. He began in 1967 by measuring the times taken by a selection of typical ALGOL source statements, and was able to compare the KDF9 results (see Appendix 1) with measurements made on a small number of other architectures [Wichmann71]. The publication of these figures encouraged others to do likewise, and eventually a wide range of systems had been investigated.

It is clear from these statement times that ALGOL performance tended to be at least as much a matter of compiler competence as of computer architecture. The times for particular statements, using different compilers on the same computer model, could vary widely—often by a factor of 2 and in one case by a factor of 30. This is at least partly because computer manufacturers (especially American manufacturers) did not see ALGOL as much of a selling point, and devoted their resources to competitive implementations of FORTRAN, a more widely available and more performance oriented language. This cast doubt on the suitability of ALGOL for *hardware* speed measurement.

These figures would have allowed for the creation of an 'ALGOL mix'—analogous to the Gibson mix—had their 'weights'—their relative execution frequencies—been known. The second part of the work was therefore to determine these weights. This was done in a comprehensive survey of the static and dynamic characteristics of about 1000 Whetstone ALGOL programs that were submitted for running as part of the everyday workloads at the National Physical Laboratory and at Oxford University [Wichmann73b]. The result was a record of about 155 million WI executions. Because of the close semantic relationship between ALGOL source code and the Whetstone virtual machine instruction set, it was then possible to work back to weightings for the individual statements.

These studies used the Atlas as a point of reference, but it was realized that using the speed of the obsolescent Atlas as a standard was not a satisfactory basis for future work. Few programmers had the extensive personal experience of Atlas that would afford an intuitive grasp of its speed, fewer still would do so in future, and no one did so outside the UK. A less parochial, machine independent measure was wanted. The WI database made that possible.

The UK government's Central Computer Agency, a body concerned with the purchase of computers by the state, had a vital interest in obtaining value for money, and therefore in validating the performance claims of manufacturers. Harold Curnow, working at the CCA, collaborated with Wichmann in constructing an ALGOL program that reproduces the distribution of WI executions in the database [CW76, Longbottom]. Its authors referred to it simply as 'a synthetic benchmark', but the world at large took it up as 'the Whetstone Benchmark'. A typical run approximates the execution of 1 million WI. Speeds were initially measured in kiloWIs per Second (KWIPS), and later in MWIPS. Modern computers achieve gigaWIPS, but KAlgol got only 62KWIPS, and WAlgol itself crawled along at a mere 2.4KWIPS.

Unofficial FORTRAN translations of the benchmark were soon doing the rounds, but a problem—not so apparent in the ALGOL version—became evident. Some FORTRAN compilers were capable of invalidating the whole basis of the test by performing optimizations that reduced the effective number of WI executed. A revised official version, in FORTRAN, and designed to defeat over-enthusiastic optimization, was therefore produced. In this form it provided a simple and easily deployed tool for estimating computer performance in scientific applications. Its uptake was rapid and it served for many years to provide a degree of objectivity in performance measurement, its benchmark results being quoted in advertisements that touted the superior speed of new computers.

An enhancement was later produced by Roy Longbottom that additionally gave ratings in Millions of Instructions Per Second (MIPS) and Millions of Floating-point Operations Per Second (MFLOPS). MIPS were scaled to a speed relative to the DEC VAX 11/780, which was deemed (somewhat generously) to be capable of 1MIPS. This too became a widely used metric, known as a 'VAX Unit of Power' or 'VUP'— the VAX range, unlike Atlas, being ubiquitous.

At the National Physical Laboratory a large collection of performance indicators for many then-current systems was collated by Wichmann and Nott. Unfortunately it was deemed to be commercially sensitive by the Director of NPL and was suppressed, all printed copies being trashed. Brian Wichmann undertook no further work on benchmarking. Presciently, he preserved the data and it has recently been made available online [NW77]. Note that the measurements listed there are stated as hardware instructions per millisecond, not in terms of WIPS.

The success of the Whetstone Benchmark was ultimately its downfall. As the sophistication of compilers increased it became easy for them to recognise specific statements of the program and put extra effort into compiling them well. It is rumoured that one system, instead of compiling individual statements, substituted a complete, hand-crafted machine code program designed to show that computer in the best possible light!

These and other problems led ultimately to the Whetstone Benchmark being superseded. The System Performance Evaluation Cooperative, latterly the Standard Performance Evaluation Corporation (SPEC), developed a series of benchmarks [`www.spec.org`] which are based on non-trivial real-world applications and are not limited to a narrow view of 'scientific' computing. Even the SPEC benchmarks are not immune to 'benchmarketing' shenanigans, and so SPEC identifies a set of good practices that must be followed when applying them.

3: THE ACKERMANN FUNCTION BENCHMARK

The measurements that prompted the creation of the Whetstone Benchmark pointed up the (in)efficiency of procedure calls as being both highly variable and highly characteristic of the systems investigated. It therefore seemed worthwhile to Wichmann to look into this matter more closely.

A recursive procedure makes it easy to invoke a large number of more or less similar calls—for ease of timing—and stresses the allocation of stack space—a particular weakness in many implementations.

Others had been thinking along similar lines. A function that meets these requirements is a common variant of Ackermann's function [Sundblad71]:

```
A(m, n) :: if m = 0 then n+1 else if n = 0 then A(m-1, 1) else A(m-1, A(m, n-1))
```

Evaluated as seen, when m = 3, this invokes $(128\times4^n - 120\times2^n + 9\times n + 37) / 3$ function calls, taking the maximum depth of the stack to $2^{n+3} - 4$ activation records and yielding the result $2^{n+3} - 3$. This is is the most time- and space-consuming case that it is feasible to compute: A(3, 4) is 125 but A(4, 3) is $2^p - 3$, with $p = 2^{65536}$.

In ALGOL 60 the function looks like this:

```
integer procedure Ackermann(m, n);
   value   m, n;
   integer m, n;
if m = 0 then
   Ackermann := n + 1
else if n = 0 then
   Ackermann := Ackermann(m - 1, 1)
else
   Ackermann := Ackermann(m - 1, Ackermann(m, n — 1));
```

Removing "**value** m, n;" would have a catastrophic effect. With that declaration the formal parameters m and n are, in effect, local variables initialized to the actual parameter values given in the call. Without it, i.e. specifying m and n as 'name' parameters (closures), they are re-evaluated on every access, causing a hyper-exponential increase in the already enormous number of calls executed.

Note the opportunities for tail recursion elimination in the n = 0 case, and in the outer call of the general case. We can take advantage of it thus:

```
integer procedure Ackermann(m, n);
   value   m, n;
   integer m, n;
HEAD:
   if m = 0 then
      Ackermann := n + 1
   else if n = 0 then
      begin
      n := 1;
      m := m - 1;
      goto HEAD
      end
   else
      begin
       n := Ackermann(m, n - 1);
      m := m - 1;
      goto HEAD
      end;
```

Inspired by Sundblad's paper, Wichmann obtained and analysed measurements of about 25 architecture/language combinations, with a particular view to 'System Implementation Languages'—subjects of great interest before C achieved its later dominance. Looking at absolute speed, number of instructions executed, and per-invocation stack space consumed, he concluded [Wichmann76]:

The results show a very wide variation in performance even for languages containing no inherent complications. Additional instructions required in ALGOL 68, PL/I and PASCAL to check for stack overflow are quite insignificant compared to the hundreds of extra instructions executed by the inefficient implementations of ALGOL 60. There is no doubt that 'System Implementation Languages' give very much better results on this test without reducing the facilities to the programmer. [...]

Does Ackermann's function represent a good test for a system implementation language? Unfortunately no statistical information is available to the author on the use of procedures in operating systems and compilers etc. Hence it is not known if, for instance, two parameters is typical. [...] *The computational part of testing for the equality with zero, jumping and adding or subtracting one seems very typical of non-numeric work.*

The response to this paper was an enthusiastic inflow of new measurements [Wichmann77] that more than doubled the sample size, allowing pertinent comparisons to be made between the same language on different architectures, and between different languages on the same architecture. Abandoning CPU time and stack space as useful criteria, Wichmann instead focussed on the number of instructions executed per call, and the function's code size, in bytes. In regard to the XALGOL compiler on the Burroughs B6700 he noted: *No improvement to the code generated seems possible*. Other systems did not fare so well.

Wichmann returned to the subject for the last time [Wichmann82], focussing on architectural features intended to facilitate procedure calling. He remarked:

In general terms, it appears that both languages and machines are getting better at subroutine linkage. The VAX is better than the DEC-10 and the ICL 2900 better than the ICL 1900.

It would be good to think that at least some of this improvement was due to the light he shone on the topic himself. Here are some figures from comparatively early language systems (estimated numbers indicated by an asterisk):

IBM System/360 Language & Compiler	**orders/call**
PL/I OPT v1.2.2	61*
ALGOL W Stanford Mk2	74*
ALGOL 60 Delft	142*
PL/I F v5.4	212*
SIMULA NCC v5.01	230*
ALGOL 60 IBM-F	820*

By 1982 Edinburgh University's ALGOL 60 compiler for the 360 architecture achieved 21 instructions per call.

[Wichmann82] considered, for the first time, an Ada version of the program as well as one in BASIC—from the sublime to the ridiculous! Ada compared well with C (and still does):

DEC VAX Language & Compiler	**orders/call**	**bytes**
York Ada global function	8	52
C-opt	9	56
C	10	80
York Ada nested function	11	64

Considering just ALGOL 60 shows the enormous discrepancies between some manufacturers' compilers and those developed independently:

ALGOL 60, by architecture	**orders/call**	**bytes**
ALGOL 60 RMCS, VAX	12.5	69
ALGOL 60 XALGOL, B6700	16.0	57
ALGOL 60 RMCS, PE 3200	17.5	94
ALGOL 60 XALGOL-5500	19.5	57
ALGOL 60 ICL,2900	19.5	84
ALGOL 60 Edinburgh, 360	21.0	128
ALGOL 60 Manchester, 1900	33.5	N/A
ALGOL 60 ICL XALV, 1900	120 *	N/A
ALGOL 60 IBM-Delft, 360	142 *	N/A
ALGOL 60 IBM-F, 360	820 *	N/A

What of KDF9, our point of departure? With the recent resurrection of the Kidsgrove ALGOL compiler, it is possible to make some interesting comparisons. Running under **ee9**, we get the following measurements for `A(3,7)`, including now the results from `paskal` [FindlayPC], the new Pascal cross-compiler for the KDF9:

KDF9 Language	**orders/call**	***μ*s/call**
ALGOL 60, Whetstone, fully recursive	1981	10900
Whetstone, tail recursions eliminated	1562	8746
ALGOL 60, Kidsgrove, fully recursive	83 (75)	474 (435)
Kidsgrove, tail recursions eliminated	54 (43)	302 (254)
Pascal, `paskal` fully recursive	29	189
`paskal`, tail recursions eliminated	20	130

The Kidsgrove figures in parentheses were obtained by applying peephole optimizations such as those described in [HW71], which are similar to those performed for Pascal. It is extraordinary that the interpretive Whetstone system takes only a small multiple (2.4) of the number of instructions used by object code of the somewhat later, native code, IBM F-level compiler.

Modern computers are much too fast for a single run of `A(3, 7)` to yield useful information. The following table shows results from `A(3, 10)`, run 100 times in succession to get an accurate measurement, on a 2017 Apple iMac. I measured versions in Ada 2012 and C, respectively the best and the worst of the present-day descendants of ALGOL. Note that these times are given in nanoseconds, not microseconds.

Present-day Language on Intel x86_64	**ns/call**
Ada 2012	0.835
C	0.912
Usercode, real **ee9** time	85

Ada is *still* 10% faster than C, and **ee9**—which is written in Ada 2012—runs at least some KDF9 code over 800 times faster than the KDF9 hardware did!

The Ada and C times are for object programs produced by the 2017 releases of the GNU compilers at optimization level -O2. With -O3 optimization they run, respectively, in 0.418ns and 0.585ns per call—Ada now being 40% faster! These speedier runs are achieved by loop-unrolling the tail recursions to a quite extraordinary degree, so that the object code is almost unrecognizable. While this is a legitimate tactic for a modern compiler aiming at the best possible performance, and having gigabytes to play with, it takes such results out of direct comparability with historical figures.

The Usercode routine reported on above is my own go at an efficient implementation, using every trick in the book:

```
P1; (To compute A(C7, C8), with m in C7, n in C8 and result returned in C8);
99;
   J1C7NZ;           (to 1 if m not equal to 0);
      I8; =+C8;      (n := n + 1);
   EXIT 1;           (return);
*1;
   J2C8NZ;           (to 2 if n not equal to 0);
      I8; =C8;       (n := 1);
      DC7;           (m := m - 1);
   J99;              (tail recursion for A(C7 => m-1, C8 => 1));
*2;
      LINK; =M0M2;   (push return address);
      C7; =M0M2QN;   (push m);
      DC8;           (n := n - 1);
   JSP1;             (full recursion for A(C7 => m, C8 => n-1));
      M1M2; =C7;     (m := top of stack);
      DC7;           (m := m - 1);
      M-I2;          (pop stack);
      M0M2; =LINK;   (return address := top of stack);
   J99;              (tail recursion for A(C7 => m-1, C8 => A(m, n-1)) );
```

It uses 9.5 orders and 70 KDF9 μs per call. A different routine, now lost, was used by Wichmann in the cited papers.

4: KIDSGROVE, WHETSTONE AND PASCAL STATEMENT TIMES

Historically, timing on the KDF9 was done using the OUT 3 system call, which depended on:

(a) the resolution of the hardware real-time clock, i.e. 32μs

(b) the accuracy with which Director accounted for its own time consumed in servicing the OUT interrupt

(c) time 'stolen' from the CPU by concurrent DMA transfers.

Variation due to (c) is unknowable, as it would depend on which peripherals were active during the run. A paper tape reader consumes 1000 core cycles per second, a line printer perhaps 2000 cycles/s, and a magnetic tape at most 5000 cycles/s. On a very busy machine with two TRs, an LP, and 4 MTs working flat out, I/O would consume 15% of the available store cycles. As KDF9 executed about 1 instruction per store cycle on average, this could slow the CPU by up to 15%, a significant effect on attempts at precision timing. I do not know whether the tests were historically run on an otherwise empty machine, or if they could have been subjected to timing variations of this order. I assume here that they were run alone, which conforms with Brian Wichmann's recollection [*personal communication*].

Considerable effort was put into (b) by the authors of Director, so I suspect that jitter in accessing the clock is the main issue. Therefore we cannot expect a hardware clock reading to be accurate to better than 32μs. This means that, unless there is a consistent pattern, a difference of order 64μs between two historic timings of a statement is probably not significant. **ee9** times are accumulated in steps of 1μs, and are completely reproducible, although they may not be historically faithful. **ee9** does not attempt to do cycle-accurate microcode emulation, but it does try to record a correct per-instruction time, based on EE documentation. It falls short of a complete simulation of the hardware by ignoring internal CPU concurrency, which can make the emulated KDF9 run slightly slower (in emulated time!) than the original hardware. This effect is seldom more than 2% or so.

When timing machine code it is necessary to take into account instruction fetches. KDF9's Main Control brings a word of 6 order syllables from core store to an empty Instruction Word Buffer (IWB) when all the instructions within it have been executed (this simplifies the matter considerably, see [FindlayHW, §7]). In **ee9** this time is approximated as either 7μs or 8μs per *pair* of fetched order words, as **ee9** loads both IWBs on a successful jump and when they both empty. This is not authentic, but it is reasonable. Fetching a word took at least 6μs—and could take as long as 12μs if the core store was busy when the fetch was initiated—but that time was at least partly overlapped with the execution of orders in the alternate IWB. The best argument for this fudge factor in **ee9** is that the Whetstone Benchmark records the same virtual CPU time, to the second, as it did in real CPU time on the hardware. A further complication is that the number of instruction fetches needed to execute a sequence of orders is dependent on the starting position of its first instruction. If that does not begin in syllable 0, one more than the optimum number of instruction word fetches may be necessary. Bearing these points in mind, the following times for native object code were measured using twelve consecutive repetitions of each statement, so that instruction word fetches were distributed evenly among them, and so that variation due to the starting syllable position was scaled down by a factor of 12.

These complications do not much apply to Whetstone ALGOL timings, in which such sources of variability are insignificant by comparison with the time required to interpret WVMIs.

OVERVIEW

These abbreviations are useful: HI: KDF9 Hardware Instructions; WI: Whetstone (virtual machine) Instructions.

The time that passes between calls of OUT 3, with an intervening null ALGOL statement, was obtained and subtracted from all other measured time differences to obtain the time consumed by the statements under examination. **ee9** provides hardware instruction counts similarly—a facility not available on the real KDF9.

Using the Whetstone ALGOL system:

The timing overhead is	3.791ms		
A null-body **for** loop takes, per iteration	3.685ms	651HI	5.66μs/HI
A null-body **for** loop of 0 iterations takes	5.229ms	946HI	5.53μs/HI
A null-body **while** loop takes, per iteration	3.992ms	709HI	5.63μs/HI
A null-body **while** loop of 0 iterations takes	5.872ms	1066HI	5.51μs/HI

The Whetstone Benchmark executes 74670550HI, averaging 5.6μs/HI and 75HI/WI, achieving just 2.4KWIPS.

Using the Kidsgrove ALGOL system, which generates native machine code:

The timing overhead is	38(35)μs	5(5)HI	7.60(7.00)μs/HI
A null-body **for** loop takes, per iteration	114(107)μs	18(18)HI	6.33(5.94)μs/HI
A null-body **for** loop of 0 iterations takes	104(86)μs	18(17)HI	5.78(5.06)μs/HI
A null-body **while** loop takes, per iteration	71(51)μs	13(9)HI	5.46(5.67)μs/HI
A null-body **while** loop of 0 iterations takes	73(63)μs	15(12)HI	4.87(5.25)μs/HI

The Kidsgrove figures in parentheses are obtained by applying, with an additional pass, peephole optimizations such as those identified in the paper *Improving the Usercode Generated by the Kidsgrove ALGOL Compiler* [HW71]. These optimizations were used for the Whetstone Benchmark test, but not for the tests enumerated in the following.

With a Kidsgrove compilation, the Whetstone Benchmark executes 3452492HI, averaging 6.05μs/HI and 3.45HI/WI according to **ee9**, and achieving 20.88μs/WI for 47.9KWIPS—somewhat slower than is given in [Longbottom], which is flagged as being for a (globally) optimized compilation. This may be because the published figure is for a run without overflow checking and tracing, which is enabled by default in the resurrected compiler.

Using the new `paskal` Pascal cross-compiler, which generates Usercode, KDF9 achieves these results:

The timing overhead is	34μs	4HI	8.5μs/HI
A null-body **for** loop takes, per iteration	44μs	8HI	5.5μs/HI
A null-body **for** loop of 0 iterations takes	12μs	1HI	12μs/HI
A null-body **while** loop takes, per iteration	47μs	9HI	5.3μs/HI
A null-body **while** loop of 0 iterations takes	24μs	3HI	8μs/HI

It is clear that having a tractable language to compile, rather than ALGOL 60, makes a big difference.

The paper *Basic statement times for ALGOL 60* [Wichmann73b] (BSTA) gives results that are generally somewhat faster than those in the earlier work *Timing of ALGOL* [Wichmann67] (TOA), and are generally closer to the times measured with **ee9**. In particular, BSTA gives the Whetstone null **while** loop time as 3.93ms—within 2% of the **ee9** figure—and reports the Kidsgrove **for** loop iteration time as 129μs.

It is plausible that these later measurements, reported in BSTA, were made with a version of the Controller that had been somewhat improved over that in TOA, shortening the WI interpretation cycle by about 15HI. There are also a few notable differences between **ee9**'s times and those in BSTA. In these cases I suspect that there is some significant, but unknown, difference in the versions of the Controller being measured.

TOA gives the null **for** loop iteration time of Whetstone as 10.4ms. This is a big anomaly, and difficult to explain. It is out of line with other individual statement times, which match within a few percent. I suspect a typo, especially as the next line of the document gives exactly the same figure for the total loop time with 1 iteration. I cannot see either entry as being correct. Because of these problems with TOA, I have decided not to include its results in this analysis.

MEASUREMENT AND ANALYSIS

The number of WI per ALGOL statement is taken from [Wichmann70a]. The measured cases are displayed in this format:

<statement>

x {**ee9** ms} *y* {BSTA ms} *n* WI, *m* HI: (*x*/*n*) *μ*s/WI, *m*/*n* HI/WI; *Kμ*/*KI*; *Pμ*/*PI;*

where *Kμ* is **ee9'**s logged Kidsgrove ALGOL time in *μ*s, including instruction fetch overhead, *KI* is the number of machine code instructions executed by the Kidsgrove object code, *Pμ* is **ee9'**s KDF9 Pascal (`paskal`) time in *μ*s, and *PI* is the number of machine code instructions executed by the Pascal object code.

`x := 1.0`
1.30 0.99 3WI, 228HI: 433*μ*s/WI, 76HI/WI; 16*μ*s/2HI; 16*μ*s/2HI

`x := 1`
1.34 1.03 3WI, 228HI: 446*μ*s/WI, 76HI/WI; 48*μ*s/6HI; 16*μ*s/2HI

`x := y`
1.38 1.08 3WI, 242HI: 460*μ*s/WI, 81HI/WI; 16*μ*s/2HI; 16*μ*s/2HI

`x := y + z`
2.02 1.75 4WI, 360HI: 505*μ*s/WI, 90HI/WI; 33*μ*s/4HI; 32*μ*s/4HI

`x := y × z`
2.03 1.76 4WI, 361HI: 508*μ*s/WI, 90HI/WI; 40*μ*s/4HI; 40*μ*s/4HI

`x := y / z`
2.02 1.71 4WI, 352HI: 505*μ*s/WI, 88HI/WI; 61*μ*s/4HI; 61*μ*s/4HI

`k := 1`
1.31 1.00 3WI, 244HI: 437*μ*s/WI, 81HI/WI; 14*μ*s/2HI; 16*μ*s/4HI

`k := 1.0`
1.40 1.09 3WI, 244HI: 467*μ*s/WI, 81HI/WI; 83*μ*s/16HI; 16*μ*s/4HI

`k := l + m`
2.07 1.80 4WI, 369HI: 518*μ*s/WI, 92HI/WI; 26*μ*s/4HI; 26*μ*s/4HI

`k := l × m`
2.10 1.81 4WI, 371HI: 525*μ*s/WI, 93HI/WI; 42*μ*s/5HI; 42*μ*s/5HI

`k := l ÷ m`
2.06 1.78 4WI, 365HI: 515*μ*s/WI, 91HI/WI; 111*μ*s/15HI; 64*μ*s/5HI

`k := l`
1.39 1.10 3WI, 244HI: 463*μ*s/WI, 81HI/WI; 16*μ*s/2HI; 16*μ*s/2HI

`x := l`
1.42 1.12 3WI, 244HI: 473*μ*s/WI, 81HI/WI; 50*μ*s/6HI; 29*μ*s/4HI

`l := y`
1.48 1.19 3WI, 244HI: 493*μ*s/WI, 81HI/WI; 83*μ*s/16HI; 85*μ*s/14HI

From this we see that `paskal` converts constant operands between integer and real at compile time, unlike Kidsgrove. `paskal` also generates inline code for converting integer expressions to real, whereas Kidsgrove calls a subroutine.

The better performance of Pascal for integer division rests on its subrange types. In the Pascal version of the program the variables are declared to be non-negative, allowing the compiler to use a simple hardware division order. That order yields the floor of a negative quotient, but both languages demand truncation toward zero, so Kidsgrove invokes a subroutine to correct the hardware result for ALGOL, and `paskal` does likewise for potentially negative integers.

`x := y ↑ 2`
2.16 1.89 4WI, 379HI: 540*μ*s/WI, 95HI/WI; 206*μ*s/37HI; 34*μ*s/4HI

`x := y ↑ 3`
2.18 1.92 (BSTA actually has 11920!) 4WI, 384HI: 545*μ*s/WI, 96HI/WI; 233*μ*s/43HI; 61*μ*s/6HI

`x := y ↑ z`
2.45 2.55 4WI, 454HI: 613*μ*s/WI, 114HI/WI; 720*μ*s/108HI; 720*μ*s/108HI

Kidsgrove uses different subroutines for reals raised to integer powers and reals raised to real powers.

`paskal` generates inline code for manifest squares, cubes, and fourth powers. It also uses a specific subroutine for integer powers of integers. That is not possible in ALGOL 60, because of an implication in the language definition that the type of a power depends on the value of the exponent. In practice, implementers have chosen to deliver a real.

`e1[1] := 1`
2.07 1.78 5WI, 356HI: 414μs/WI, 71HI/WI; 57μs/10HI; 16μs/4HI

`e2[1,1] := 1`
2.42 2.16 6WI, 415HI: 403μs/WI, 69HI/WI; 172μs/26HI; 16μs/4HI

`e3[1,1,1] := 1`
2.80 2.51 7WI, 474HI: 400μs/WI, 68HI/WI; 331μs/48HI; 16μs/4HI

The Kidsgrove object code for `e2[1,1]` and for `e3[1,1,1]` invokes a subroutine to calculate the address of the array element, but the address calculation for `e1[1]` is done in-line. There is a specific subroutine for 2-dimensional arrays and a generic subroutine for arrays of greater dimensionality. `paskal` calculates the address of an element with constant subscripts at compile time, so these array elements have the same performance as simple variables.

`begin real` `a` **`end`**
1.91 1.67 4WI, 357HI: 478μs/WI, 89HI/WI; 24μs/4HI

`begin array` `a[1:1]` **`end`**
3.24 3.13 7WI, 599HI: 463μs/WI, 86HI/WI; 635μs/103HI

`begin array` `a[1:1, 1:1]` **`end`**
3.90 3.76 9WI, 708HI: 433μs/WI, 79HI/WI; 782μs/126HI

`begin array` `a[1:1, 1:1, 1:1]` **`end`**
4.55 4.36 11WI, 811HI: 414μs/WI, 74HI/WI; 933μs/149HI

These blocks have no analogue in Pascal.

`begin goto` `abcd; abcd:` **`end`**
1.27 0.94 3WI, 221HI: 420μs/WI, 77HI/WI; 22μs/3HI; 15μs/1HI

`begin switch` `ss := pq;` **`goto`** `ss[1]; pq:` **`end`**
5.71 5.61 16WI, 1039HI: 373μs/WI, 65HI/WI; 312μs/49HI

The Whetstone code for these **`goto`** statements includes an execution-trace WI, so the jump is effected by just 2WI. The Kidsgrove code for **`goto`** `abcd` includes 2 execution-trace HIs, so the jump is effected by a single HI.

`p0`
2.27 2.01 5WI, 417HI: 450μs/WI, 83HI/WI; 51μs/10HI; 20μs/2HI

`p1(x)`
3.49 3.55 8WI, 641HI: 434μs/WI, 80HI/WI; 67μs/12HI; 73μs/11HI

`p2(x,y)`
4.44 4.57 10WI, 812HI: 442μs/WI, 81HI/WI; 85μs/14HI; 97μs/14HI

`p3(x,y,z)`
5.39 5.72 12WI, 984HI: 447μs/WI, 82HI/WI; 101μs/16HI; 113μs/16HI

The Whetstone code for these calls of null-bodied procedures includes several execution-trace WIs: specifically 3WI per call, plus 1WI per parameter. For example, `p0` is actually invoked by just 2WI, and there is a 3WI tracing overhead. This overhead was optional, but is included in the historical figures and in the present **ee9** measurements.

It is of some interest to dig a little deeper into the differences between ALGOL and Pascal. The first point is that Kidsgrove is a multi-pass, globally optimising compiler, whereas `paskal` is a single-pass compiler that generates code 'on the fly'. So Kidsgrove generates the code for the bodies of these procedures after having made a deep analysis of their characteristics, but `paskal` has irrevocably converted them to Usercode before its own analysis is complete.

This means that `paskal` cannot specialise a procedure body as Kidsgrove does, but it can significantly specialise the entry/exit code—which is generated at the end of the body—by omitting components that support operations not present in the body. Specifically, `paskal` determines whether a procedure: is a leaf (i.e. makes no calls itself), is a function, accesses non-local variables other than globals, has parameters, or has local variables. `p0`, `p1`, `p2` and `p3` are all non-function leaf procedures, do not access non-local variables, and do not declare local variables. Moreover `p0` does not have parameters. This lets `paskal` reduce its entry/exit protocol to the absolute minimum: a call instruction and an exit instruction. For the other procedures, `paskal` must assume that their bodies require a stack-frame pointer and so the entry/exit protocols it creates must deal with that. Kidsgrove knows that no frame pointer is required.

Making the procedure bodies a little more realistic changes the balance. For example, this:

```
integer global;
…
procedure q1(n); value n; integer n; begin integer j; j := n; global := j end;
```

takes 408μs/70HI with Kidsgrove, while the Pascal equivalent takes only 109μs/15HI.

The ALGOL object code for `q1` is much more involved than that of Pascal, partly because it uses utility subroutines to update a 'display' of pointers to the most recently active stack frames of the statically enclosing procedures. `paskal` maintains a single 'static link' to the immediately enclosing routine. If a procedure—as here—includes no accesses to non-locals, it avoids even that (globals are addressed directly).

```
x := sin(y)
```
5.18 4.72 8WI, 804HI: 645μs/WI; 1195μs/ 209HI

```
x := cos(y)
```
5.20 4.61 8WI, 804HI: 547μs/WI; 1206μs/ 212HI

```
x := abs(y)
```
4.11 4.11 8WI, 753HI: 511μs/WI; 104μs/15HI

```
x := exp(y)
```
4.40 4.70 8WI, 753HI: 550μs/WI; 442μs/71HI

```
x := ln(y)
```
4.42 4.72 8WI, 794HI: 550μs/WI; 426μs/57HI

```
x := sqrt(y)
```
4.32 4.44 8WI, 782HI: 538μs/WI; 331μs/46HI

```
x := arctan(y)
```
6.05 5.13 8WI, 987HI: 753μs/WI; 1974μs/238HI

```
x := sign(y)
```
4.17 4.13 8WI, 764HI: 507μs/WI; 146μs/20HI

```
x := entier(y)
```
4.24 4.28 8WI, 777HI: 528μs/WI; 225μs/33HI

The ALGOL standard functions in the Kidsgrove system present something of a puzzle. `sin`, `cos` and `arctan`, as listed in BSTA, were much faster than is found with **ee9**, whereas the BSTA times for some other functions were much slower. It is difficult to see how `sin` could have been as fast as was reported, given that its instruction count (which is exact) implies an average execution time—according to BSTA—of only 3μs per order; or how a simple routine such as `sign` could have been as slow as BSTA claims, implying an average execution time of almost 10μs per order.

The `arctan` function is unusual, clocking 6.7μs/HI with Whetstone. It makes heavy use of relatively slow KDF9 orders such as floating point arithmetic and internal subroutine calls. The other analytical functions do most of their work with fixed point fractions, converting the result to floating point only at the end of the calculation. This effect is seen even more clearly in the Kidsgrove `arctan` time, which—being free of interpretive overhead—implies 8.3μs/HI.

The `paskal` runtime system uses a copy of the Kidsgrove standard functions, so its performance in these tests is essentially the same.

References and Further Reading

Many of the following documents can be viewed or downloaded. Online sources that I am aware of at time of writing are indicated thus:

§ See: `sw.ccs.bcs.org/CCs/KDF9/Wichmann`

* See: `www.findlayw.plus.com/KDF9`

[CCA71] *A comparison of computer speeds using mixes of instructions*
Central Computer Agency, Technical Support Unit; Note 3806; 1971.

[C75] *The Design of Synthetic Programs-II*
H. J. Curnow; in *Benchmarking: computer evaluation and measurement* ; Hemisphere Pub. Corp.; 1975.

[*CW76] *A Synthetic Benchmark*
H. J. Curnow and B. A. Wichmann; *Computer Journal*, Vol. 19 No. 1; 1976.

[*EELM68a] *Egdon System Reference Manual*
Publication 103 550566; English Electric–Leo–Marconi Computers Limited; 1968.

[*EELM69] *KDF9 ALGOL Programming*
Publication 1002 mm (R) 1000565, English Electric–Leo–Marconi Computers Limited; 1969.

[§Fell63] *Programmes to Investigate Timings on KDF9*
Miss P. Fell; English Electric – Leo Computers Report No. K/AA u 29; 10 June 1963

[*FindlayEE] *The English Electric KDF9;* W. Findlay; 2021.

[*FindlaySW] *The Software of the KDF9;* W. Findlay; 2021.

[*FindlayHW] *The Hardware of the KDF9;* W. Findlay; 2021.

[*FindlayKB] *The KDF9: a Bibliography;* W. Findlay; 2020.

[*FindlayUG] *Users' Guide for **ee9**: An English Electric KDF9 Emulator;* W. Findlay; 2022.

[*FindlayPC] *PASKAL: a Pascal cross-compiler for the KDF9;* W. Findlay; 2022.

[*HW71] *Improving the Usercode Generated by the Kidsgrove ALGOL Compiler*
R. Healey and B. A. Wichmann; National Physical Laboratory CCU TM3; 1971.

[*ICL69] *KDF9 Programming Manual*
Publication 1003 mm, 2nd Edition, International Computers Limited; October 1969.

[Longbottom]
`www.webarchive.org.uk/wayback/archive/20130303230651/www.roylongbottom.org.uk/cpumix.htm`
`www.webarchive.org.uk/wayback/archive/20130303230703/www.roylongbottom.org.uk/whetstone.htm`
Roy Longbottom.

[Naur60] *Report on the Algorithmic Language ALGOL 60*
ed. P. Naur; 1960. A copy can be seen in [EELM68a], Chapter 7, 'EGDON ALGOL'.

[*NW77] *A Guide to the Processing Speeds of Computers*
C.W. Nott and B.A. Wichmann; 1977.

[*RR64] *ALGOL 60 Implementation*
B. Randell and L.J. Russell; Academic Press; 1964.

[*Sundblad71] *The Ackermann function. A theoretical, computational and formula manipulative study*
Y. Sundblad; *BIT Numerical Mathematics*; Vol. 11 No. 1; 1971.

[*Wichmann67] *Timing of ALGOL*
B. A. Wichmann; National Physical Laboratory; March 1967.

[*Wichmann70a] *Some statistics from ALGOL programs*
B. A. Wichmann; National Physical Laboratory CCU Report No.11; 1970.

[*Wichmann70b] *Estimating the execution speed of an ALGOL program*
B. A. Wichmann; National Physical Laboratory CCU TM1; 1970.

[*Wichmann71] *The Performance of Some ALGOL Systems*
B.A. Wichmann; IFIP 71, North Holland Publishing Company; 1972.

[§Wichmann72] *Five ALGOL compilers*
B. A. Wichmann; *Computer Journal*, Vol.15 No.1; 1972.

[§Wichmann73a] *ALGOL 60 Compilation and Assessment*
B. A. Wichmann; Academic Press; 1973.

[Wichmann73b] *Basic statement times for ALGOL 60*
B.A. Wichmann; National Physical Laboratory, Teddington, Middlesex; 1973.

[*Wichmann76] *Ackermann's function: a study in the efficiency of calling procedures*
B. A. Wichmann; *BIT*, Vol.16, pp. 103-110; 1976.

[*Wichmann77] *How to call procedures, or second thoughts on Ackermann's Function*
B. A. Wichmann; *Software: Practice and Experience*, Vol.7, pp. 317-329; 1977.

[*Wichmann82] *Latest results from the procedure calling test, Ackermann's function*
B. A. Wichmann; NPL Report DITC 3/82; 1982.