ACKERMANN'S FUNCTION: A STUDY IN THE EFFICIENCY OF CALLING PROCEDURES

B. A. WICHMANN
National Physical Laboratory, Teddington,
Middlesex, TW11 0LW, UK

March 5, 1999

Abstract

A six line recursive procedure is used to assess the efficiency of the procedure calling mechanism in ALGOL-like languages. The results from some 40 systems varying from ALGOL 68 and PL/I to System Implementation Languages for minicomputers are presented and compared. A hundred to one variation in performance occurs with this test, the major reasons for which are given.

Received April 17, 1975. Revised January 22, 1976.

1 Introduction

There are several areas where traditional high-level languages are notably less efficient than hand-coding. Two of these areas are accessing array elements and calling procedures. Although in the former case much research has been done to improve the efficiency (resulting in the ALCOR and FORTRAN H2 compilers), little work has been published on calling procedures. In many scientific computations procedures are not called very frequently but this is not true in non-numeric work. Since modularisation by means of procedures is one of the most powerful programming tools in structuring complex problems, an inefficient calling mechanism can be very detrimental. In recent years there has been an increased use of high-level languages as assembler replacements — the so-called 'System Implementation Languages'. Such languages must achieve a high degree of object code efficiency if their use is to become more widespread. The purpose of this paper is to compare the procedure calling mechanism of traditional high-level languages with both system implementation languages and machine-code.

2 Ackermann's function

In view of tile paper by Sundblad [1], a reasonable test case for study seemed to be the evaluation of Ackermann's function. This function has the advantage of being very short and not requiring large integer values for typical cases, yet being complex enough to be a significant test. The figures given in Sundblad's paper also provided a basis for comparison before many values had been obtained.

Following Sundblad, we calculate Ackermann (3,n) for increasing n, in accordance with the complete program listing below. Three figures are determined from the test — the average execution time per call, the number of instructions executed per call and the amount of stack space required for each call. Although the average

time per call can be determined from running tile program, the other two figures cannot he found so easily and indeed, in many cases the figures are not available.

3 Program listing

```
begin
    integer i, j, k, k1;
    real t1, t2;
    integer procedure Ackermann(m, n); value m, n; integer m, n;
        Ackermann := if m = 0 then n + l
            else if n = 0 then Ackermann(m - 1, 1)
            else Ackermann(m - 1, Ackermann(m, n - 1));

    k:= 16; k1 := 1;
    for i := 1 step 1 until 6 do
        begin
        t1 := cputime;
        j : = Ackermann(3, i);
        t2 := cputime;
        if j ≠ k − 3 then
            outtext(1, 'WRONG VALUE');
        outreal(1, t2 − t1);
        comment Now output time per call;
        outreal(1, 3 × (t2 − t1)/(512 × k1 − 15 × k + 9 × i + 37)));
        k1 := 4 × k1; k := 2 × k
    end
end
```

4 Properties of the algorithm

To determine the execution characteristics of the test one needs the following properties of the algorithm (which can be easily proved by induction). The number of executions of each leg of the algorithm in calculating Ackermann (3, n) is:

first leg:	$(64 \times 4 \uparrow n-$	$72 \times 2 \uparrow n + 6 \times n + 26)/3$
second leg:	(	$24 \times 2 \uparrow n - 3 \times n - 12)/3$
third leg:	$(64 \times 4 \uparrow n-$	$72 \times 2 \uparrow n + 6 \times n + 23)/3$
total		
no. of calls =	$(128 \times 4 \uparrow n-$	$120 \times 2 \uparrow n + 9 \times n + 37)/3$

Hence for large n, the average number of instructions per call is half the number of instructions in the first and third legs. Both legs contain the procedure entry and exit overheads and the third leg includes two unsuccessful tests for equality with zero. So the number of instructions executed per call can be found by examining the machine code produced by the compiler. Unfortunately this is not always easy and sometimes almost impossible. To provide an approximate comparative value, an estimate has been made from the time using a Gibson mix value [2] to extrapolate from a similar machine. In a few cases, the instructions executed arc known but the time has not been measured. In these cases, an estimate for the time is given based upon published instruction times.

The amount of stack space required by the program is determined by the maximum depth of procedure calls. This is given by Sundblad as

$Ackermann(3, n) - 1 = 2 \uparrow (n + 3) - 4$

and occurs when the third leg is being evaluated at all but the last of those levels. Hence the amount of stack space required doubles when n is increased by one. For a more detailed discussion on the storage requirements and the definition of the 'maximum depth of recursion' see [5]. To measure this space, Sundblad observed the maximum value of 11 for which *Ackermann* (3, n) could be calculated using 26K words of store (which he called the 'capability'). Although this measurement is easy to take, it only provides a rough estimate of the space needed per call. Examination of the machine code of the third leg of the algorithm gives the precise value. In fact, the number of words per call is the difference in the address of $4m$ in the inner call to the address of the parameter in the routine. It is not easy to calculate this value from the code, and so a post-mortem print may be the easiest way to find the value. Rather than give the capability, the number of words per call is listed with the results. If only the capability is known, bounds can be given for the number of words per call by assuming that between 3K and 10K words are needed out of the 26K store for the program and run-time system.

5 Notes on the results

Estimates of missing values are included in the table in brackets and have been calculated in the manner described above. The program listing in all cases has followed the coding given very closely. The only exceptions are 1) the machine code examples, 2) the PASCAL and SUE systems which have no conditional expressions, and 3) PL516 which follows the hardware of the machine in not supporting recursion (stacking is performed by subroutines).

6 Results

Language/ Computer	Time per call (microseconds)	Instructions per call	Words per call	Characteristic (see below)
ALGOL 60				
B6700	41.2	16	13	ND2VO
B5500 Mk XV.1.01	135	19.5	7	NL2VO
EMAS 4/75	46.7	21	18	ND2VO
1906A Manchester	29.2	33.5	30	ND2VR
KDF9 Mk2	532	68.5	10	CD2VR
1906S XALV	70.9	(120)	13	CD2TR
370/165 Delft	43.8	(142)	?	CD2TR
NU 1108	175	(175)	9	CD2TR
ALGOL W				
360/67 Mk2	121	(74)	(16-45)	HD2TR
IMP				
ICL 4/75	46	17.5	18	ND2VO
SIMULA				
1108	120	(120)	9	HD2TR
DEC10(KI10)	317	(158)	?	HD2TR
CYSEN 74	380	(800)	(15)	HD2TR
ALGOL 68				
1907F (no heap)	134	28	15	NO2VO
1907F (heap)	167	34	15	HD2VR
COC 6400(subset)	45.3	51	?	HD2VO
Cyber 73 vl.0.8	67.8	(60)	7	HD2VR
Bliss 10				
DEC10(KA10)	53.15	15	5	NLWVR
PL/I				
360/65 OPT v1.2.2	101	(61)	68	HD2AO
360/65 F v5.4	351	(212)	(70)	HD2AR
PASCAL				
1906S	19.1	32.5	11	HDWVR
COC 6400	34	38.5	6	HDWVO
370/158	39	42.5	30	HDWVE
RTL/2				
4/70	46	14.5	14	NLWVO
PDP11/20	(107)	30.5	?	CLWVH
PALGOL				
PDP11/20	(46)	13	3	NLWVO
Bliss/11				
POP11/20 OPT	31	8	2	NLWVO
MARY				
SM4	105	30.5	9	COWVR
CORAL 66				
MOD 1	(21)	15.5	3	NLWVO
PL516				
DDP516	84.5	37	2	CLWVH
C (UNIX)				
POP 11/45	50.4	26	?	NLWVR
BCPL				
370/165	5.9	19	9	NLWVR
POP 11/45	48	20.5	7	NLWVO
MOO 1	(35)	25	7	NLWVR
Machine code				
Most machines	?	5-14	1-2	NLWVO

7 Factors influencing the execution speed

Factors influencing the call of a recursive procedure vary from inherent problems with the architecture of the machine to purely software problems on the design of the procedure calling mechanism. Against each of the results above is a sequence of letters and digits which describes the principle characteristics governing the execution performance.

Recursion. On most modern machines with a few base registers and indexing facilities, the basic overhead of recursion is very small indeed. A few minicomputers do not have adequate base registers or addressing facilities to support recursion. The Honeywell DDP5l6 is of this type, hence with the high-level assembler PL516 stacking must be performed explicitly. On some machines, although the addressing logic is available an additional time penalty occurs by addressing via pointers. On the Modular 1 implementation of CORAL 66, recursion is an option. In fact the code produced with recursion is more efficient than that without recursion. The reason for this is that the short address length of the Modular 1 means that base registers must be loaded to access the variables of another procedure. This is a larger overhead than that incurred using a single stack register which only needs to he incremented and decremented by a small amount. Without recursion, the number of instructions is increased to 19.5 (30% more).

Storage allocation. In order to execute this program, a stack storage scheme is necessary. It is sometimes possible on conventional hardware to use an area in the store as stack without performing an explicit software check for stack overflow. One can then rely upon a hardware address violation which should permit the output of a suitable diagnostic. Such systems are marked by an `N`, whereas `C` denotes a software stack overflow check. Languages such as ALGOL 68, PL/I and SIMULA require more than a simple stack and hence must perform an overflow check in a manner which allows a recovery to perform some garbage collection. Systems like this arc marked with an `H`. PASCAL permits additional storage apart from the stack but without garbage collection. Although no garbage collection is involved, PASCAL is marked with an `H`. Only SIMULA requires an actual garbage collection during the execution of this test. The method used to administer the stack is partly a matter of language design and partly a matter of implementation. For instance, although ALGOL 68 requires a heap, the ALGOL 68-R implementation will run a stack only if the program does not require the heap. The difference, shown above, is that an additional subroutine is called on procedure entry to check that adequate stack space is available.

Complete display. Block structured languages in general require a base pointer for every block whose variables can be accessed at the relevant part of the program. This facility is marked with a `D`. Languages such as BCPL and Burroughs B5500 ALGOL restrict access to local and global identifiers only, permitting a significant economy in pointers. These languages are marked with an `L`. The actual method of implementing the display can vary independently of the language. For instance, ALGOL W and IMP have every base pointer in a register, the PASCAL implementations above backchain the display (from the local procedure) and ALGOL 68-R keeps a copy in core of the full display in the local block. In almost all cases, there will be small variations in the calling overhead depending upon the relevant positions (in the display) of the call and the procedure.

Dynamic stack storage. In ALGOL 60, the sizes of arrays are not, in general, known until program execution. This facility, implemented with second order working store, requires a small overhead on the administration of stack storage even when no arrays are being used, as in this test. Languages requiring such storage are marked with a 2 whereas systems allowing only storage whose size can be determined at compile time are marked with a `W`. PALGOL and RTL/2 are `LW` languages and hence require only one stack pointer (and one pointer for static storage depending upon the addressing structure).

Parameter handling. The most expensive mechanism is marked with a `T` which denotes a 'thunk' that is required to implement the ALGOL 60 call by name [4]. A thunk can be avoided with this test by performing all parameter checking at compile time and using only the value mechanism (`V`). The KDF9, B5500 and Atlas ALGOL compilers all use the value mechanism. The use of the thunk mechanism by the other ALGOL 60 compilers is caused by the problem of handling the call of formal procedures whose parameters cannot be specified [3]. With value parameters, the ICL 1900 ALGOL compiler uses a simplified thunk mechanism but the Manchester ALGOL compiler uses the value mechanism. The PL/I systems use call by address which is marked with an `A`. With recursion, the addresses of parameters are dynamic and in consequence this method is less efficient than call by value.

Open code or subroutines. Systems which handle this test entirely by open code are marked with an `O`, whereas the use of subroutines is marked with an `R`. The PL/I optimising compiler generates a subroutine call, but it is not invoked unless insufficient stack space is given (which did not happen in this case).

8 Conclusion.

Ackermann's function provides a simple method of comparing the efficiency of the procedure calling mechanism of a language permitting recursion. 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. Machine independence seems to be realised in this case without any measurable cost as BCPL shows.

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 large amount of stack space used is certainly not typical and can result in pathological situations. For instance, stack space is claimed in 64 word units under George 3 on a 1906A, but is not released. Hence during the first part of the algorithm when the stack is being increased a large operating system overhead occurs. During the second part when the stack is rarely increased beyond its previous maximum, there is no significant overhead. The computational part of testing for the equality with zero, jumping and adding or subtracting one seems very typical of non-numeric work. On the 360 computer, the fact that the the algorithm is very short (<4K bytes) results in a small saving, but on the IMP compiler which can take advantage of this the speed was only the increased by 3%.

From the better figures produced by system implementation languages, the code for Ackermann is roughly divided as follows:

	instructions
subroutine entry and exit	2
stacking return address	1
setting up environment	3
checking for stack overflow	2 (if check made)
restoring old environment	3
(on procedure exit)	
setting parameters	$2 \times 1 = 2$
body of Ackermann	8
Total	21

9 Acknowledgements

This work has been inspired by the International Federation for Information Processing Working Group 2.4. The desire of the group to obtain information on the performance of system implementation languages has led to tests such as this. It would not have been possible to write this paper without the active help of the following persons in running the test: — Mr. L. Ammeraal (Mini-ALGOL 68), Dr R. Backhouse (B5500), Dr J. G. P. Barnes (RTL/2), Dr D. A. Bell (PL516), Dr H. Boom (Cyber 73 ALGOL 68), Mr P. Klint (C-UNIX), Mr R. Conradi (MARY, CYBER 74, 1108, CDC3300), Mr W. Findlay (PASCAL 1906A & machine code), Mr W. B. Foulkes (PASCAL 370/158). Professor G. Goos (B6700), Mr. V. Hathway (BCPL MOD 1), Mr M. Healey (PL/I OPT), Professor J. J. Horning (Sue 11), Mr B. Jones (ALGOL 60, Dclft 370/165), Dr M. MeKeag (PASCAL & ALGOL 60, 1906S), Mr Z. Mocsi (R10), Dr J. Palme (DEC10 SIMULA & machine code), Mr M.J. Parsons (PALGOL), Dr M. Richards (BCPL), Professor J. S. Rohl (Manchester 1900 ALGOL), Mr P. D. Stephens (IMP), Dr P. Wetherall (ALGOL 68-R), Professor N. Wirth (PASCAL), Professor D. B. Wortman (370/165 machine code) and Professor W. A. Wulf (Bliss 10, Bliss 11 & machine code).

References

[1] Y. Sundblad, The Ackermann function. A theoretical, computational and formula manipulative study. BIT 11 (1971), 107119.

[2] Central Computer Agency, A comparison of computer speeds using mixes of instructions. Technical Support Unit, Note 3806, (1971).

[3] B. A. Wichmann, ALGOL 60 Compilation and Assessment, Academic Press, London, (1973).

[4] P. Z. Ingerman, Thunks — A way of compiling procedure statement with some comments on procedure declarations, Comm ACM. 4, (1961), 5558.

[5] B. J. Cornelius and G.H. Kirby, Depth of recursion and the Ackermann function, BIT 15 (1975), 144150.