Latest results from the procedure calling test, Ackermann's function

B A WICHMANN
National Physical Laboratory, Teddington, Middlesex
Division of Information Technology and Computing

March 1982

Abstract

Ackermann's function has been used to measure the procedure calling overhead in languages which support recursion. Two papers have been written on this which are reproduced[1] in this report. Results from further measurements are included in this report together with comments on the data obtained and codings of the test in Ada and Basic.

1 INTRODUCTION

In spite of the two publications on the use of Ackermann's Function [1, 2] as a measure of the procedure-calling efficiency of programming languages, there is still some interest in the topic. It is an easy test to perform and the large number of results obtained means that an implementation can be compared with many other systems. The purpose of this report is to provide a listing of all the results obtained to date and to show their relationship. Few modern languages do not provide recursion and hence the test is appropriate for measuring the overheads of procedure calls in most cases.

Ackermann's function is a small recursive function listed on page 2 of [1] in Algol 60. Although of no particular interest in itself, the function does perform other operations common to much systems programming (testing for zero, incrementing and decrementing integers). The function has two parameters M and N, the test being for (3, N) with N in the range 1 to 6.

Like all tests, the interpretation of the results is not without difficulty. One problem is that of optimization, at least as far as hand-coding is concerned. John Reiser has noted that the calls of Ackermann consist of one external call (in the test program) and numerous internal calls. The two can be distinguished by the stack level. The innermost call in the last leg of the algorithm must be with N>0 and hence the first test of the algorithm can be avoided. Dave Messham of ICL went further in analysing the case when N=1 to use essentially the result that Ackermann (1,M) = M+2. These versions are noted in the listings but should be discounted for comparative purposes.

In previous publications, the speed of the computation has been reported. This is no longer added to the data because of measurement problems and also because the data can be rapidly overtaken by hardware improvements. Such data for a 360/370

[1] Not in this electronic version.

implementation is much less useful than the instructions executed and the code space in bytes.

The instructions executed can be found by an examination of the machine-code produced. On some machines very complex instructions are used for procedure linkage. No account is taken of this; it would be nice to analyse the number of store accesses but this is too difficult and not meaningful with cache memory. For the implementation with a high overhead, the time taken has been used to estimate the number of machine instructions executed. These estimates are shown in brackets in the tables of results.

The size of the code for Ackermann is of considerable interest. It shows a major influence from the machine architecture. The code size is measured in 8-bit bytes (or number of bits divided by 8 if appropriate) and includes any constants needed. The size is just that for the function itself and excludes any subroutines that may be called for performing procedure linkage. The fact that such subroutines are called is indicated by a '+' after the size column in the tables. Such subroutines clearly contribute to the number of instructions executed.

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. Of course, some of the best figures are produced by software-designed machines such as the P-code of UCSD Pascal. One could reasonably argue that this is not "fair" since such machines would not be much good at executing, say, FORTRAN.

Bishop and Barron discuss procedure calling in a structured architecture in [6]. The measurements presented here do not support their contention that the B6700 is better than the ICL 2900. The results for the implementation language S3 on the 2900 are superior in both space and instructions executed to the B6700. The reason for this is that Ackermann does not need a display, which as Bishop and Barron point out is poorly supported on the 2900. As often happens with such comparisons, it is not clear which approach is more effective overall.

2 Coding in various languages

In most languages, the original Algol 60 coding can be followed without change. Indeed, no change should be made as sometimes better results are possible. For instance, inverting the order of the parameters may produce better code in the last leg of the algorithm (given the obvious stack implementation). In two cases, Basic and Ada, a coding is given below. The Basic coding is not obvious, and Ada (perhaps unfairly) uses more features of the language. A similar coding to that of Basic, for FORTRAN is given in [5].

2.1 Ada version

In Ada, several actions can be taken. Firstly, the timing statements can be included as part of the code although the interpretation of the figures may be upset by multiprogramming. Secondly, the parameters can be constrained to be positive (or zero) which then tests the ability of the compiler to remove unnecessary constraint checking. Lastly, the exception Storage_Error can be used to trap stack overflow so as not to go too far in the evaluation. The resulting coding is:

```
with Text_IO, Calendar; use Text_IO, Calendar;
   procedure Time_Ackermann is
```

```
   package Out_Times is new Fixed_IO(Duration);
   package Out_Int is new Integer_IO(Integer);
   use Out_Times, Out_Int;

   Before, After: Time;
   I, J, K, K1, Calls: Integer;

   subtype Positive is Integer range 0..Integer’Last;

   function Ackermann( M, N: Positive) return Positive is
   begin
      if M = 0 then
         return (N + 1);
      elsif N = 0 then
         return Ackermann( M - 1, 1);
      else
         return Ackermann( M - 1, Ackermann(M, N - 1) );
      end if;
   end Ackermann;

begin
K := 16; K1 := 1; I := 1;
while K1 < Integer’Last/512 loop
   Before := Clock();
   J := Ackermann( 3, I);
   After := Clock();
   if J /= K - 3 then
      Put( "Wrong value");
   end if;
   Calls := (512*K1 - 15*K + 9*I + 37)/3;
   Put( "Number of Calls " );
   Put( Calls );
   Put( " Time per call " );
   Put( (After - Before) / Calls );
   New_line;
   I := I + 1;
   K1 := 4 * K1;
   K := 2 * K;
end loop;
exception
   when Storage_Error =>
      New_line;
      Put( "Stack space exceeded for Ackermann( 3, " );
      Put( I );
      Put( ")" );
      New_line;
end Time_Ackermann;
```

Note that this version is for Revised Ada (July 1980) and will require a small change for the ANSI standard.

2.2 Basic version

The following coding was run on a ZX81 and took about 12 minutes to calculate Ackermann(3,3). On the TRS-80, it took about 2 hours for Ackermann(3,3). The only coding known to be slower was for ML/1 on a PDP11. The coding executes 23.5 statements per call.

```
10 DIM S(510) sufficient for 3,3
20 LET P=1
30 PRINT "ACKERMANN(";
40 INPUT S
50 PRINT S; ",";
60 GOSUB 400
70 INPUT S
80 PRINT S; ")";
90 GOSUB 400
100 GOSUB 150
110 PRINT "="; S
l20 STOP
150 GOSUB 500 Ackermann
160 LET N=S
170 GOSUB 500
180 LET M=S
190 IF M <> 0 THEN GOTO 220
200 LET S=N+1
210 RETURN
220 IF N < > O THEN GOTO 290
230 LET S=M-1
240 GOSUB 400
250 LET S=1
260 GOSUB 400
270 GOSUB 150
280 RETURN
290 LET S=M-1
300 GOSUB 400
310 LET S=M
320 GOSUB 400
330 hET S=M-1
340 GOSUB 400
350 GOSUB 150 -
360 GOSUB 400
370 GOSUB 150
380 RETURN
400 LET S(P)=S Push
410 LET P=P+1
420 RETURN
500 LET P=P-1 Pop
510 LET S=S(P)
520 RETURN
```

Because of the interpretive nature of Basic, the number of machine instructions is not easily determined. Hence the time per call and the space are measured. The space figure is for lines 150-520 above.

2.2.1 Basic results

Language	Compiler	Instr./call	Size (bytes)	Source
ZX81		0.301 (slow mode)	421	B A Wichmann
TRS-80		3.0	?	R F Maddock

3 Tables of Results

Each result is listed in the following tables under five columns. The first gives the language, the second the compiler, the third the instructions executed, the fourth the code size and lastly the person supplying the information. Readers are warned that it has not always been possible to check the data provided.

Each result is listed three times (in general). Firstly under the machine architecture, secondly under the language used, and lastly in order of the number of machine instructions executed. Within the first two headings, the results are listed in terms of increasing number of instructions executed.

N. B. + after *Size(bytes)* denotes use of out-of-line code. *Instr./call* in brackets is an estimate based upon time taken.

3.1 List of results, by machine architecture

3.1.1 IBM 360/370

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	BAL-360	3.994	64	J F Reiser
Assembler	BAL-360	6	106	D B Wortman
CDL2	Berlin,opt+mods	13.5	?	C Oeters
RTL/2	ICI-360	14.5	102	J Barnes
IMP	Edinburgh,360	18	122	P D Stephens
BCPL	Cambridge	19	116+	M Richards
CDL2	Berlin,opt	19	?	C Oeters
MARY	360	20	114	Sven Tapelin
ALGOL 60	Edinburgh,360	21	128	P D Stephens
CDL2	Berlin	22.5	?	C Oeters
LIS	Siemens-360	26.5	192	J Teller
PASCAL	Siemens	32	224+	M Sommer
PASCAL	Manitoba	42.5	?	W B FouIkes
PL/I	OPT v1.2.2	(61)	?	M Healey
ALGOL W	Stanford Mk2	(74)	?	Sundblad
ALGOL 60	Delft,360	(142)	?	B Jones
PL/I	F v5.4	(212)	?	M Healey
SIMULA	NCC v5.01	(230)	146	R Babcicky
ALGOL 60	IBM-F	(820)	?	Sundblad

3.1.2 ICL 1900

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	PLAN	7.5	57	W L Findlay
PASCAL	Belfast Mk2	27	111+	J Welsh
ALGOL 68-R	Malvern(no heap)	28	153	P Wetherall
PASCAL	Belfast Mk1	32.5	129+	W L Findlay
ALGOL 60	Manchester,l900	33.5	?	J S Rohl
ALGOL 68-R	Malvern(heap)	34	162+	P Wetherall
ALGOL 60	ICL XALV	(120)	?	M McKeag

3.1.3 DEC PDP11

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	PAL-11	3.496	30	J F Reiser
Imp77	Edinburgh(PDP11)	7	28	P S Robertson
Assembler	PAL-11	7.5	32	W A Wulf
Bliss	CMU-11, opt	7.5	32	W A Wulf
Bliss	CMU-11	10	64	W A Wulf
PALGOL	NPL	13	86	M J Parsons
MODULA	Univ York	13.5	74	J Holden
BCPL	RMCS v7	17.5	76	Robert Firth
CORAL 66	RMCS v7	17.5	76	Robert Firth
ALGOL 60	RMCS v7	18.5	80	Robert Firth
BCPL	Cambridge-11	20.5	104	M Richards
PASCAL	Unix-11,Amsterdam	22	126	A Tannenbaum
C-opt	Unix V7, 11/45	25	64+	W Findlay
C	UNIX	26	62+	P Rlint
Sue-11	Toronto	26.5	176	J J Horning
C	Unix V7, 11/45	27.5	82+	W Findlay
RTL/2	ICI-11	30.5	70+	J Barnes

3.1.4 DEC-10

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	PAL-DEC10	3.010	54	J P Reiser
Assembler	PAL-DEC10	5	85	J Palme
Bliss	CMU-DEC10	15	103+	W A Wulf
Imp77	Edinburgh(DEC10)	16	130	I A Young
PASCAL	Hamburg-DEC10	20	166	D Burnett-Hall
C	MIT	23.5	157	A Synder
ALGOL 68	DEC-10-C	27	140	I C Wand
Ada	Intermetrics,v1	44	936	Ben Brosgol
SIMULA	DEC-Stockholm	(158)	?	J Palme

3.1.5 Burroughs - stack m/cs

Language	Compiler	Instr./call	Size (bytes)	Source
ALGOL 60	XALGOL-6700	16	57	G Goos
ALGOL 60	XALGOL-5500	19.5	57	R Backhouse
PASCAL	Tasmania	24.5	73	A H J Sale

3.1.6 ICL 2900 Series

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	2900-ICL	0.2	52	A Montgomery
S3	2900-ICL	11	52	A Montgomery
PASCAL	2900-ICL	17.5	88	B A Wichmann
IMP	2900-ERCC	19	86	P D Stephens
ALGOL 60	2900-ICL	19.5	84	A Montgomery
ALGOL 60	2900-ERCC	21	96	P D Stephens
ALGOL 68	2900-SWURCC	25	94	A Montgomery
SCL	2900-ICL interp	(3409)	?	A Montgomery
SCL	2900-ICL semi-comp	(22 200)	?	A Montgomery

3.1.7 CDC 6000

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	Compass	6.490	67.5	J P Reiser
Assembler	Compass, fast	9	97.5	A Lunde
Assembler	Compass-opt	9.5	60	D Grune
Assembler	Compass,conservative	9.5	112.5	A Lunde
Assembler	Compass	15.5	83	W M Waite
Bliss	Oslo	17	127.5	A Lunde
PASCAL	Zurich, March 76	36.5	?	U Amman
PASCAL	Zurich 3.4	38.5	232	N Wirth
ALEPH	Amsterdam 17.1	41.5	292	D Grune
Mini-ALGOL 68	Amsterdam	51	292	L Ammeraal
ALGOL 68	CDC v1.0.8	(60)	?	H Boom
SIMULA	CDC	(800)	?	R Conradi
JCL	Kronos	(140 000)	?	A W Colijn

3.1.8 Univac 1100

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	1108	(9)	?	R Conradi
ALGOL 60	Univac	(175)	?	R Conradi
SIMULA	Univac	(120)	?	R Conradi

3.1.9 CTL Modular 1

Language	Compiler	Instr./call	Size (bytes)	Source
CORAL 66	CTL	15.5	66	V Hathway
BCPL	CTL	25	66+	V Hathway

3.1.10 Norwegian SM4

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	SM4	11	?	R Conradi
MARY	Trondheim	30.5	?	R Conradi

3.1.11 KDF9

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	KDF9	14	?	B Wichmann
PALGOL	KDF9 - NPL	24.5	74	D Schofield
ALGOL 60	Kidsgrove	68.5	?	B Wichmann
ALGOL 60	Whetstone,interp.	(1550)	?	B Wichmann
BABEL	KDF9	(310)	?	B Wichmann

3.1.12 VAX

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	VAX MAC	7	32	Robert Firth
Ada	York,library unit	8	52	C Forsyth
C-opt	Unix 32V, 11/780	9	56	W Findlay
C	Unix 32V, 11/780	10	80	W Findlay
Ada	York,nested unit	11	64	C Forsyth
BCPL	RMCS VAX generator	11.5	65	Robert Firth
CORAL 66	RMCS VAX generator	11.5	65	Robert Firth
Imp77	Edinburgh(VAX)	12	71	G Toal
ALGOL 60	RMCS VAX generator	12.5	69	Robert Firth
PASCAL	DEC VAX V1.2	25	187	Robert Firth

3.1.13 TI 9900 (microp)

Language	Compiler	Instr./call	Size (bytes)	Source
Imp77	Edinburgh(TI990)	17.5	110	G Toal
Eh	Waterloo	32	92	M Gentleman

3.1.14 Hungarian R10

Language	Compiler	Instr./call	Size (bytes)	Source
CDL	Budapest	(60)	?	Z Mocsi

3.1.15 GEC 4080 (mini)

Language	Compiler	Instr./call	Size (bytes)	Source
BCPL	Warwick, GEC4000	19.5	58+	J M Collins
PASCAL	GEC 4080	(175)	220+	B A Wichmann

3.1.16 IRIS 80

Language	Compiler	Instr./call	Size (bytes)	Source
LIS	CII	24	192	J D Ichbiah

3.1.17 CDC 3000

Language	Compiler	Instr./call	Size (bytes)	Source
SIMULA	3300 - NRDE	369	324	E Heistad

3.1.18 E1 - X8

Language	Compiler	Instr./call	Size (bytes)	Source
ALGOL 60	Karlsruhe	(4400)	?	J Winkler

3.1.19 MI1 - microprocessor

Language	Compiler	Instr./call	Size (bytes)	Source
PALGOL	MI1-NPL	12	36	R E Milne

3.1.20 P1000 - Philips m/c no longer marketed

Language	Compiler	Instr./call	Size (bytes)	Source
ALGOL 60	Eindhoven	98.5	120	Kruseman Aretz

3.1.21 MODCOMP IV

Language	Compiler	Instr./call	Size (bytes)	Source
CORAL 66	Leasco-MODCOMP	21.5	96	Hetherington

3.1.22 NORD-10

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	NORD-10	7	50	T Noodt
PASCAL	Oslo	27	82+	Terje Noodt
PASCAL	NORD-10 (P)	102.5	136+	T Noodt

3.1.23 PE 3200 (formerly Interdata 8/32)

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	CAL	9.5	50	Robert Firth
Imp77	Edinburgh(PE3220)	11.5	66	G Toal
CORAL 66	RMCS, PE 3200	16.5	90	Robert Firth
BCPL	RMCS, PE 3200	17.5	96	Robert Firth
ALGOL 60	RMCS, PE 3200	17.5	94	Robert Firth

3.1.24 Computer Automation NM4

Language	Compiler	Instr./call	Size (bytes)	Source
BCPL	HPAC generator,NM4	15.5	70	Robert Firth
CORAL	66 HPAC generator,NM4	15.5	68	Robert Firth
ALGOL 60	HPAC generator,NM4	16.5	70	Robert Firth

3.1.25 Ferranti Argus 700

Language	Compiler	Instr./call	Size (bytes)	Source
Imp77	Edinburgh(F700)	13	50	F King

3.1.26 Motorola 6809

Language	Compiler	Instr./call	Size (bytes)	Source
Imp-	Edinburgh(6809)	16.5	61	G Toal

3.2 Listing of results by language types

3.2.1 ALGOL 60

Language	Compiler	Instr./call	Size (bytes)	Source
Imp77	Edinburgh(PDP11)	7	28	P S Robertson
Imp77	Edinburgh(PE3220)	11.5	66	G Toal
PALGOL	MI1-NPL	12	36	R E Milne
ALGOL 60	RMCS VAX generator	12.5	69	Robert Firth
PALGOL	NPL	13	86	M J Parsons
Imp77	Edinburgh(F700)	13	50	F King
ALGOL 60	XALGOL-6700	16	57	G Goos
Imp77	Edinburgh(DEC10)	16	130	I A Young
ALGOL 60	HPAC generator,NM4	16.5	70	Robert Firth
Imp-	Edinburgh(6809)	16.5	61	G Toal
ALGOL 60	RMCS, PE 3200	17.5	94	Robert Firth
Imp77	Edinburgh(TI990)	17.5	110	G Toal
IMP	Edinburgh,360	18	122	P D Stephens
ALGOL 60	RMCS v7	18.5	80	Robert Firth
IMP	2900-ERCC	19	86	P D Stephens
ALGOL 60	XALGOL-5500	19.5	57	R Backhouse
ALGOL 60	2900-ICL	19.5	84	A Montgomery
ALGOL 60	2900-ERCC	21	96	P D Stephens
ALGOL 60	Edinburgh,360	21	128	P D Stephens
PALGOL	KDF9 - NPL	24.5	74	D Schofield
ALGOL 60	Manchester,l900	33.5	?	J S Rohl
ALGOL 60	Kidsgrove	68.5	?	B Wichmann
ALGOL 60	Eindhoven	98.5	120	Kruseman Aretz
ALGOL 60	ICL XALV	(120)	?	M McKeag
ALGOL 60	Delft,360	(142)	?	B Jones
ALGOL 60	Univac	(175)	?	R Conradi
BABEL	KDF9	(310)	?	B Wichmann
ALGOL 60	IBM-F	(820)	?	Sundblad
ALGOL 60	Whetstone,interp.	(1550)	?	B Wichmann
ALGOL 60	Karlsruhe	(4400)	?	J Winkler
Imp77	Edinburgh(VAX)	12	71	G Toal

3.2.2 ALGOL 68

Language	Compiler	Instr./call	Size (bytes)	Source
S3	2900-ICL	11	52	A Montgomery
ALGOL 68	2900-SWURCC	25	94	A Montgomery
ALGOL 68	DEC-10-C	27	140	I C Wand
ALGOL 68-R	Malvern(no heap)	28	153	P Wetherall
ALGOL 68-R	Malvern(heap)	34	162+	P Wetherall
Mini-ALGOL 68	Amsterdam	51	292	L Ammeraal
ALGOL 68	CDC v1.0.8	(60)	?	H Boom

3.2.3 ALGOL W

Language	Compiler	Instr./call	Size (bytes)	Source
ALGOL W	Stanford Mk2	(74)	?	Sundblad

3.2.4 SIMULA

Language	Compiler	Instr./call	Size (bytes)	Source
SIMULA	Univac	(120)	?	R Conradi
SIMULA	DEC-Stockholm	(158)	?	J Palme
SIMULA	NCC v5.01	(230)	146	R Babcicky
SIMULA	3300 - NRDE	369	324	E Heistad
SIMULA	CDC	(800)	?	R Conradi

3.2.5 PL/I

Language	Compiler	Instr./call	Size (bytes)	Source
PL/I	OPT v1.2.2	(61)	?	M Healey
PL/I	F v5.4	(212)	?	M Healey

3.2.6 BCPL

Language	Compiler	Instr./call	Size (bytes)	Source
C-opt	Unix 32V, 11/780	9	56	W Findlay
C	Unix 32V, 11/780	10	80	W Findlay
BCPL	RMCS VAX generator	11.5	65	Robert Firth
BCPL	HPAC generator,NM4	15.5	70	Robert Firth
BCPL	RMCS v7	17.5	76	Robert Firth
BCPL	RMCS, PE 3200	17.5	96	Robert Firth
BCPL	Cambridge	19	116+	M Richards
BCPL	Warwick, GEC4000	19.5	58+	J M Collins
BCPL	Cambridge-11	20.5	104	M Richards
C	MIT	23.5	157	A Synder
BCPL	CTL	25	66+	V Hathway
C-opt	Unix V7, 11/45	25	64+	W Findlay
C	UNIX	26	62+	P Rlint
C	Unix V7, 11/45	27.5	82+	W Findlay
Eh	Waterloo	32	92	M Gentleman

3.2.7 Bliss

Language	Compiler	Instr./call	Size (bytes)	Source
Bliss	CMU-11, opt	7.5	32	W A Wulf
Bliss	CMU-11	10	64	W A Wulf
Bliss	CMU-DEC10	15	103+	W A Wulf
Bliss	Oslo	17	127.5	A Lunde

3.2.8 PASCAL

Language	Compiler	Instr./call	Size (bytes)	Source
PASCAL	ICL-Perq	13	41	B A Wichmann
PASCAL	UCSD-Apple II	14.5	40	B A Wichmann
PASCAL	2900-ICL	17.5	88	B A Wichmann
PASCAL	Hamburg-DEC10	20	166	D Burnett-Hall
PASCAL	Unix-11,Amsterdam	22	126	A Tannenbaum
LIS	CII	24	192	J D Ichbiah
PASCAL	Tasmania	24.5	73	A H J Sale
PASCAL	DEC VAX V1.2	25	187	Robert Firth
LIS	Siemens-360	26.5	192	J Teller
PASCAL	Belfast Mk2	27	111+	J Welsh
PASCAL	Oslo	27	82+	Terje Noodt
PASCAL	Siemens	32	224+	M Sommer
PASCAL	Belfast Mk1	32.5	129+	W L Findlay
PASCAL	Zurich, March 76	36.5	?	U Amman
PASCAL	Zurich 3.4	38.5	232	N Wirth
PASCAL	NORD-10 (P)	102.5	136+	T Noodt
PASCAL	Manitoba	42.5	?	W B FouIkes
PASCAL	GEC 4080	(175)	220+	B A Wichmann

3.2.9 Ada

Language	Compiler	Instr./call	Size (bytes)	Source
Ada	York,library unit	8	52	C Forsyth
Ada	York,nested unit	11	64	C Forsyth
Ada	Intermetrics,v1	44	936	Ben Brosgol

3.2.10 Assembler

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	2900-ICL	0.2	52	A Montgomery
Assembler	PAL-DEC10	3.010	54	J P Reiser
Assembler	PAL-11	3.496	30	J F Reiser
Assembler	BAL-360	3.994	64	J F Reiser
Assembler	PAL-DEC10	5	85	J Palme
Assembler	BAL-360	6	106	D B Wortman
Assembler	Compass	6.490	67.5	J P Reiser
Assembler	NORD-10	7	50	T Noodt
Assembler	VAX MAC	7	32	Robert Firth
Assembler	PLAN	7.5	57	W L Findlay
Assembler	PAL-11	7.5	32	W A Wulf
Assembler	Compass, fast	9	97.5	A Lunde
Assembler	1108	(9)	?	R Conradi
Assembler	Compass-opt	9.5	60	D Grune
Assembler	Compass,conservative	9.5	112.5	A Lunde
Assembler	CAL	9.5	50	Robert Firth
Assembler	SM4	11	?	R Conradi
Assembler	KDF9	14	?	B Wichmann
Assembler	Compass	15.5	83	W M Waite

3.2.11 CORAL 66

Language	Compiler	Instr./call	Size (bytes)	Source
CORAL 66	RMCS VAX generator	11.5	65	Robert Firth
CORAL 66	CTL	15.5	66	V Hathway
CORAL 66	HPAC generator,NM4	15.5	68	Robert Firth
CORAL 66	RMCS, PE 3200	16.5	90	Robert Firth
CORAL 66	RMCS v7	17.5	76	Robert Firth
CORAL 66	Leasco-MODCOMP	21.5	96	Hetherington

3.3 Listing of results on order of instructions per call

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	2900-ICL	0.2	52	A Montgomery
Assembler	PAL-DEC10	3.010	54	J P Reiser
Assembler	PAL-11	3.496	30	J F Reiser
Assembler	BAL-360	3.994	64	J F Reiser
Assembler	PAL-DEC10	5	85	J Palme
Assembler	BAL-360	6	106	D B Wortman
Assembler	Compass	6.490	67.5	J P Reiser
Assembler	NORD-10	7	50	T Noodt
Assembler	VAX MAC	7	32	Robert Firth
Imp77	Edinburgh(PDP11)	7	28	P S Robertson
Assembler	PAL-11	7.5	32	W A Wulf
Bliss	CMU-11, opt	7.5	32	W A Wulf
Assembler	PLAN	7.5	57	W L Findlay
Ada	York,library unit	8	52	C Forsyth
Assembler	1108	(9)	?	R Conradi
Assembler	Compass, fast	9	97.5	A Lunde
C-opt	Unix 32V, 11/780	9	56	W Findlay
Assembler	Compass-opt	9.5	60	D Grune
Assembler	Compass,conservative	9.5	112.5	A Lunde
Assembler	CAL	9.5	50	Robert Firth
Bliss	CMU-11	10	64	W A Wulf
C	Unix 32V, 11/780	10	80	W Findlay
Assembler	SM4	11	?	R Conradi
Ada	York,nested unit	11	64	C Forsyth
S3	2900-ICL	11	52	A Montgomery
BCPL	RMCS VAX generator	11.5	65	Robert Firth
Imp77	Edinburgh(PE3220)	11.5	66	G Toal
CORAL 66	RMCS VAX generator	11.5	65	Robert Firth
Imp77	Edinburgh(VAX)	12	71	G Toal
PALGOL	MI1-NPL	12	36	R E Milne
ALGOL 60	RMCS VAX generator	12.5	69	Robert Firth
Imp77	Edinburgh(F700)	13	50	F King
PALGOL	NPL	13	86	M J Parsons
CDL2	Berlin,opt+mods	13.5	?	C Oeters
MODULA	Univ York	13.5	74	J Holden
Assembler	KDF9	14	?	B Wichmann
RTL/2	ICI-360	14.5	102	J Barnes
Bliss	CMU-DEC10	15	103+	W A Wulf
Assembler	Compass	15.5	83	W M Waite
BCPL	HPAC generator,NM4	15.5	70	Robert Firth
CORAL	66 HPAC generator,NM4	15.5	68	Robert Firth
CORAL 66	CTL	15.5	66	V Hathway
ALGOL 60	XALGOL-6700	16	57	G Goos
Imp77	Edinburgh(DEC10)	16	130	I A Young
ALGOL 60	HPAC generator,NM4	16.5	70	Robert Firth
CORAL 66	RMCS, PE 3200	16.5	90	Robert Firth
Imp-	Edinburgh(6809)	16.5	61	G Toal

Language	Compiler	Instr./call	Size (bytes)	Source
Bliss	Oslo	17	127.5	A Lunde
ALGOL 60	RMCS, PE 3200	17.5	94	Robert Firth
Imp77	Edinburgh(TI990)	17.5	110	G Toal
BCPL	RMCS v7	17.5	76	Robert Firth
BCPL	RMCS, PE 3200	17.5	96	Robert Firth
PASCAL	2900-ICL	17.5	88	B A Wichmann
CORAL 66	RMCS v7	17.5	76	Robert Firth
IMP	Edinburgh,360	18	122	P D Stephens
ALGOL 60	RMCS v7	18.5	80	Robert Firth
IMP	2900-ERCC	19	86	P D Stephens
CDL2	Berlin,opt	19	?	C Oeters
BCPL	Cambridge	19	116+	M Richards
ALGOL 60	2900-ICL	19.5	84	A Montgomery
BCPL	Warwick, GEC4000	19.5	58+	J M Collins
ALGOL 60	XALGOL-5500	19.5	57	R Backhouse
MARY	360	20	114	Sven Tapelin
PASCAL	Hamburg-DEC10	20	166	D Burnett-Hall
BCPL	Cambridge-11	20.5	104	M Richards
ALGOL 60	2900-ERCC	21	96	P D Stephens
ALGOL 60	Edinburgh,360	21	128	P D Stephens
CORAL 66	Leasco-MODCOMP	21.5	96	Hetherington
PASCAL	Unix-11,Amsterdam	22	126	A Tannenbaum
CDL2	Berlin	22.5	?	C Oeters
C	MIT	23.5	157	A Synder
LIS	CII	24	192	J D Ichbiah
PALGOL	KDF9 - NPL	24.5	74	D Schofield
PASCAL	Tasmania	24.5	73	A H J Sale
ALGOL 68	2900-SWURCC	25	94	A Montgomery
BCPL	CTL	25	66+	V Hathway
C-opt	Unix V7, 11/45	25	64+	W Findlay
PASCAL	DEC VAX V1.2	25	187	Robert Firth
C	UNIX	26	62+	P Rlint
LIS	Siemens-360	26.5	192	J Teller
Sue-11	Toronto	26.5	176	J J Horning
ALGOL 68	DEC-10-C	27	140	I C Wand
PASCAL	Belfast Mk2	27	111+	J Welsh
PASCAL	Oslo	27	82+	Terje Noodt
C	Unix V7, 11/45	27.5	82+	W Findlay
ALGOL 68-R	Malvern(no heap)	28	153	P Wetherall
MARY	Trondheim	30.5	?	R Conradi
RTL/2	ICI-11	30.5	70+	J Barnes
Eh	Waterloo	32	92	M Gentleman
PASCAL	Siemens	32	224+	M Sommer
PASCAL	Belfast Mk1	32.5	129+	W L Findlay
ALGOL 60	Manchester,l900	33.5	?	J S Rohl
ALGOL 68-R	Malvern(heap)	34	162+	P Wetherall
PASCAL	Zurich, March 76	36.5	?	U Amman
PASCAL	Zurich 3.4	38.5	232	N Wirth

Language	Compiler	Instr./call	Size (bytes)	Source
ALEPH	Amsterdam 17.1	41.5	292	D Grune
PASCAL	Manitoba	42.5	?	W B FouIkes
Ada	Intermetrics,v1	44	936	Ben Brosgol
Mini-ALGOL 68	Amsterdam	51	292	L Ammeraal
CDL	Budapest	(60)	?	Z Mocsi
ALGOL 68	CDC v1.0.8	(60)	?	H Boom
PL/I	OPT v1.2.2	(61)	?	M Healey
ALGOL 60	Kidsgrove	68.5	?	B Wichmann
ALGOL W	Stanford Mk2	(74)	?	Sundblad
ALGOL 60	Eindhoven	98.5	120	Kruseman Aretz
PASCAL	NORD-10 (P)	102.5	136+	T Noodt
SIMULA	Univac	(120)	?	R Conradi
ALGOL 60	ICL XALV	(120)	?	M McKeag
ALGOL 60	Delft,360	(142)	?	B Jones
SIMULA	DEC-Stockholm	(158)	?	J Palme
ALGOL 60	Univac	(175)	?	R Conradi
PASCAL	GEC 4080	(175)	220+	B A Wichmann
PL/I	F v5.4	(212)	?	M Healey
SIMULA	NCC v5.01	(230)	146	R Babcicky
BABEL	KDF9	(310)	?	B Wichmann
SIMULA	3300 - NRDE	369	324	E Heistad
SIMULA	CDC	(800)	?	R Conradi
ALGOL 60	IBM-F	(820)	?	Sundblad
ALGOL 60	Whetstone,interp.	(1550)	?	B Wichmann
SCL	2900-ICL interp	(3409)	?	A Montgomery
ALGOL 60	Karlsruhe	(4400)	?	J Winkler
SCL	2900-ICL semi-comp	(22 200)	?	A Montgomery
JCL	Kronos	(140 000)	?	A W Colijn

4 Notes

ALGOL 60 on P1000 uses thunks and a compatibility check to maintain security with formal calls. The full thunk mechanism is avoided with simple variables and constants. This implementation method is comparable with ALGOL W.

The two assembler versions for the DEC10 use instructions which do two actions at once: add one to a register & jump, subtract one from a register & jump, and also uses stacking instructions.

The codings from John Reiser do two pieces of optimisation: distinguishing between external and internal calls (the stack depth can be used to distinguish the cases), and avoiding the test for n=0 when n>0 on internal calls. The count of the instructions per call is based upon figures for Ackermann(3,6). Similar codings have been produced by Dave Messham of ICL which go down to .02 instructions per call by essentially assuming the value of Ack(1,n) = n+2. All these versions should probably be rejected since they avoid the procedure call which the test is designed to measure.

The Leasco-MODCOMP version uses a conditional expression and hence the compiled code is somewhat more compact than the ‘ordinary’ coding.

The Berlin CDL2 results come in three flavours: without optimization, with machine-independent optimization and with machine-dependent optimization. The last case is designed but not implemented (figures from hand-coding).

The Pascal implementation for the NORD-10 is based upon the Zurich P compiler with the simplest modification to generate code instead of interpreting. The code generator is being enhanced.

The MODULA code for the PDP11 uses SP addressing for local addressing similar to the PALGOL compiler.

The PASCAL compiler for the NORD-10 has an additional option to test for stack overflow which adds three instructions on execution and 8 bytes to the size.

The Pascal compiler on the 2900 does not avoid the multiple assignment to the function result and also produces a prelude at the end of the code with a jump to the start of the procedure. One the 2972, it takes 9.54 microseconds or .557 microseconds per instruction.

The high figure for Pascal on the GEC4080 is caused by the software overhead to allow the stack to extend beyond the directly addressable limit of 16K bytes (for a single segment).

The RMCS generator for BCPL goes via the usual OCODE route. However, generators for both Coral 66 and ALGOL 60 are available via the same route giving essentially the same figures, as shown under Robert Firth. He notes that BCPL loses some efficiency because procedures are called via procedure variables. His Algol 60 generator loses one instruction by assignment of the function result.

The UCSD P-code implementation of Pascal has not been added to the list under machine architectures because of its (usually) interpretive nature. The Apple II implementation takes about 600 microseconds per P-code instruction. Stack overflow on (3,4) crashes the system.

The ICL Perq runs a very similar instruction set to P-code but this is microcoded. Ackermann(3,7) crashed the system, although (3,6) ran too fast to time accurately. Fifty calls of (3,6) took 355 seconds, or 41.2 microseconds per call. This is about 200 times faster than UCSD Pascal on the Apple. The 13 instructions for a call were estimated by hand compiling the function into Q-code. The call instruction is relatively long so that the average time for an instruction is 3.17 microseconds, quite a bit less than the maximum of one microsecond.

The two codings in the command language for the ICL 2900 series(SCL) are with an interpreter and with partial compilation. Note that partial compilation gives a better figure than one true compiler. The instructions executed are derived from an instruction counter on the machine and hence should be reliable.

The code generated by the Imp 77 compiler for the PDP11 is quite remarkable. It is better than what was thought to be the optimal hand coding. Peter Robertson, the designer of the compiler, explains this by noting that the compiler is in three passes. Allocating the two parameters to registers means that no stack frame is required. Classical optimization is then effective, including code motion. The improvement over Bliss is achieved by making the entry address the third instruction — the two previous incrementing and decrementing each parameter. This means that instructions are saved due to the commonality of the expressions in the calls. This trick with the entry address does not appear to be used with the other Imp code generators although good results are obtained in most cases. The 6809 generator is for a subset of Imp.

The two results from the York Ada compiler are for a library unit and for a function nested within the main program (needing a display). The coding is the obvious one without the subtype Positive and the exception Storage_Error. The machine code for the library unit is essentially the same as that produced by C. The Unix C optimiser is used by the York compiler to reduce the code size.

References

[1] B. A. Wichmann, 'Ackermann's function: a study in the efficiency of calling procedures', BIT, 16, 103-110 (1976).

[2] B. A. Wichmann, 'How to call procedures, or second thoughts on Ackermann's Function'. Software — Practice and Experience. Vol 7 pp317-329 (1977).

[3] M. Broy, Program Development: The Ackermann Function as an example. Report TUM-INFO-7716. Technical University of Muenchen. (1977 .

[4] Boris Allan, The limits of my world. Practical Computing. Sept 1981.

[5] A. C. Day, Fortran Techniques, with special reference to non-numerical applications, CUP (1972).

[6] J. M. Bishop and D. W. Barron. Procedure calling and structured architecture. Computer Journal, Vol23, No2 ppll5-123. (1980).

For further references, the above papers should be consulted. In particular, the paper of Broy gives the reference to W. Ackermann and an iterative algorithm for the function.

A Heading material

ISSN 0143-7348

National Physical Laboratory Teddington, Middlesex TWll OLW, UK

Acknowledgement is given to John Wiley & Sons Ltd for their agreement to reproduce in this report an article which first appeared in "Software, Practice and Experience".

Acknowledgement is given to the Editor of BIT for agreement to the reproduction of an article which appeared in that journal,

Approved on behalf of Director, NPL, by Mr E Albasiny, Superintendent, Division of Information Technology and Computing.

NPL Report DITC 3/82

B Document details

Scanned and converted to LaTeX, February 2002. The material was reorganised slightly to reflect the LaTeX article style— heading material placed in an appendix, the two articles not reproduced since they are available separately, and the layout of the tables changed for additional clarity.