How to Call Procedures, or Second Thoughts on Ackermann's Function

B. A. WICHMANN
National Physical Laboratory, Teddington,
Middlesex TW11 OLW, England *

June 1977

Abstract

Various problems have been encountered in using Ackermann's function to measure the efficiency of procedure calIs in System Implementation Languages. Although measurements have been made of some 60 systems, the ones presented here are included only when comparisons are possible. For conventional machine design, it is possible to draw some conclusions on desirable instruction set features. A surprising result from the analysis is that implementation quality is far more important than the overheads implied by language design.

KEY WORDS Procedure call, Recursion, Efficiency, System Implementation Languages.

1 INTRODUCTION

In the design of System Implementation Languages, efficiency is a prime consideration because otherwise programmers will be forced to use conventional assembly languages. It is well known that procedure calling in conventional high-level languages is relatively expensive, and hence this aspect of System Implementation Languages (SILs) is worth special attention. Measurements using Ackermann's function have been made with this in mind[1]. In this paper comparisons are made between the implementations on the same machine architecture by examination of the machine code produced. Several machine architectures are also compared for their suitability as target machines for recursive languages. The architectures chosen for this study are the IBM 360/370, ICL 1900, DEC PDP11, DEC10, Burroughs stack machines and the CDC Cyber Series. Although other results have been obtained, further analysis is not, in general, worthwhile due to the lack of comparative data on the same architecture.

2 COLLECTING THE DATA

The short nature of the test combined with its 'unusual' characteristics has encouraged language implementors to try it out on their favourite SIL. Currently, data is available on about 60 systems—a large sample which in itself makes a more detailed comparison

*Received July 1976, Revised 21 October 1976

worthwhile. Larger tests which have been constructed recently have been much less successful in collecting data.

Problems have arisen with the information received at the National Physical Laboratory (NPL). Sometimes this has excluded a listing of the machine-code produced. Even when a code listing is available, it is sometimes in a form which makes analysis awkward (octal dump or without source names). When one is not familiar with the machine, following low-level listings is quite difficult. Sometimes the listings of control routines are not available, but fortunately systems which require such subroutines are of less interest. In contrast, counting the instructions executed is straightforward.

In a few cases, it was not clear if the test was actually run or if the timing was deduced from an instruction timing manual.

3 SPECIFICATION

The ALGOL 60 specification proved to be accurate and easy to convert to any of the SILs considered. In a language with conditional expressions the code can be followed exactly, although it can be made more compact by writing

$$Ackermann(m-1, \text{if } n = 0 \text{ then } 1 \text{ else } Ackermann(m, n-1))$$

for the second and third legs. Nobody actually made this change. A more serious program change, since it could have significant effects on the efficiency, is swapping the two parameters. This change allows one to avoid stacking a parameter before making the inner call in the third leg of the algorithm. One coding was rejected for this reason.

Languages such as PASCAL without conditional expressions lose on code size but not on instructions executed. Languages with a dynamic RETURN gain a small amount on both size and instructions executed. The impact of these coding changes is slight compared with the procedure calling mechanism itself.

The specification of the machine code examples is quite difficult since a line has to be drawn as to what optimization is legitimate. The following changes are allowed: the two parameters are interchanged to avoid some stacking, the parameters are placed in registers, the two outer calls are made by jumping to the beginning of the procedure, check on stack overflow is not required. The reason for permitting the 'interchanging' of the parameters is that the 'order' is not evident from the machine-code. With all these changes, one can go further and note that there is now only one (recursive) call of the function and hence the stack depth can be used to determine whether a recursive exit or jump out of the function is needed at the end. This change is not regarded as legitimate since there is essentially no subroutine mechanism left at all (the return mechanism becomes an unconditional jump). This last form of optimization was produced by Professor Wortman on the 360 but a less ambitious coding of his appears in the analysis below.

The original ALGOL 60 procedure and the machine-language equivalent coded in ALGOL 60 are as follows:

```
integer procedure Ackermann(m, n);
    value m, n; integer m, n;

    Ackermann := if m = 0 then
            n + 1
    else if n = 0 then
```

```
            Ackermann(m-1, 1)
        else
            Ackermann(m-1, Ackermann(m, n-1));

integer procedure Ackermann(m, n);
        value m, n; integer m, n;
        comment machine-language equivalent;
        begin
        integer procedure innerack;
        start: if m = 0 then
            innerack := n + 1
        else if n = 0 then
            begin
            n := 1; m := m-1;
            goto start
            end
        else
            begin
            comment stack m (and return address);
            stack[stp] := m; stp := stp+ 1;
            n := n-1;
            n := innerack;
            stp := stp - 1; m := stack[stp] - 1;
            goto start
            end;

integer array stack[1: 1000];
integer stp;
stp := 1;
comment s, p, m, n, and result of innerack kept in registers;
Ackermann := innerack
end;
```

4 HOW GOOD IS THE TEST?

There are four serious objections to the test. The first is that recursion itself is not a common requirement and hence measurement of the procedure calling overhead should not use recursion. However, in almost all cases, the SIL's have a procedure calling mechanism which supports recursion. With the Modular 1, CORAL 66 is more efficient with recursion than without, although this is certainly exceptional. The author has been told that even the PL/I optimizer does not take advantage of non-recursion. Recursion is often the simplest way to achieve re-entrant code.

The second objection is that it cannot measure the overhead in languages which support more than one calling convention. A good example of the two-call strategy is LIS[2] which has ACTION's (simple and cheap) and PROCEDURE's (as usual). The recursion forces the use of the more general calling method which one would hope could be avoided with most calls in practical cases. Other tests are being designed for simple cases, including ones where open code could be generated.

The third objection to the test is that some very unusual optimization is possible. One compiler (Bliss, opt) removes the recursion on the two outer calls by performing a jump to the beginning of the procedure. The gain made was about 20 per cent and hence does not seriously undermine the value of the test. Optimization is made easier because all the working store of the procedure can be kept in registers on almost all machines. It is clear that results from any one small procedure cannot test all the aspects of procedure call when optimization is taken into account.

A fourth objection is that the test uses only value parameters and not variable (or name/address) parameters. However, statistics given on p. 90 of Reference 4 give the percentage of value parameters as 88 and hence the objection is not serious. Since the second leg of the algorithm is rarely executed, the complex third leg is very important. The nested function call in this leg is certainly not typical and could cause a compiler to generate less efficient code.

Other minor objections are that the test is too small and that excessive stack space is required to give a reasonable time interval for measurement. The 4K addressing limit problem on the 360 means that a clever compiler can run this test more efficiently. An example of this is the IMP compiler, but the improvement is only 15 instructions per call. The excessive stack space can be overcome by repeating a call for a small case in a loop and calculating the time per call slightly differently (from the coding given in Reference 1).

5 FURTHER ANALYSIS

There are several problems with making a more detailed analysis of the code produced. The number of instructions executed per call is clearly dependent upon the architecture— some machines definitely have more powerful instructions than others. Even within one machine range (say 360), the number of instructions executed is not necessarily a good comparison. One compiler may generate more register-only instructions giving quicker code. This difference is partly reflected in the size figures. There is some uncertainty about the size since the pooling of constants makes the allowance for them hard to determine. With this test, the code can contain an untypically high proportion of jumps. Such a code can upset the instruction look-ahead and pipelining on the faster computers.

5.1 THE 360 FAMILY

The architecture of the 360 Series is well known. The machine has 16 general-purpose registers in which integer arithmetic is performed. All addressing is offset from the sum of two registers, and may be further modified by a 12-bit number in long instructions. Instructions are 2, 4 or 6 bytes long and addressing is always at byte level. Hence to access the Ith element of an integer array, one must multiply I by four. A subset of integer instructions is available for 16-bit operands.

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	BAL	6	106	D. B. Wortman
RTL/2	ICI	14.5	102	J. Barnes
IMP	Edinburgh	19	140	P. D. Stephens
BCPL	Cambridge	19	116+	M. Richards
ALGOL 60	Edinburgh	21	128	P. D. Stephens

+ denotes the use of subroutines.

5.1.1 Assembler

The 360 machine code example is tightly coded by having an 'inner' subroutine which merely calls itself and hence can be assumed to have various registers set. The registers include the constants 1 and 4 so that almost all the instructions in the inner loop are short (2 bytes) and fast. The two outer calls merely jump to the first instruction of the inner subroutine without stacking a return address. In the inner call, the return address (24 bits) and the value of m (8 bits) is stacked in one word. This is only legitimate if it can be shown that m is limited to 8 bits (as could be stated in a range declaration in PASCAL).

The size is mainly in the initialization code including constant values for the registers on entry to the main (outer) subroutine.

5.1.2 RTL/2[3]

RTL/2 performs relatively well by a carefully designed subroutine linkage. The code analyzed here makes no check on the availability of stack space although an option for this is implemented. The last parameter is left in a register and the result of the function is also left in a register. The compiler keeps m and n in registers when possible so that compact code is produced. The store multiple and load multiple instructions are used to preserve the old environment in the procedure entry and exit code respectively. Since RTL/2 requires no display (merely locals and globals) nor permits second order working store, stack maintenance is very straightforward. In fact, two registers are used, one for locals and the other for the top of the stack to push/pop parameters.

Apart from the 102 bytes for instructions, about 20 bytes of constants are placed in the code, apparently to assist in diagnostics.

5.1.3 IMP

The IMP language is essentially a superset of ALGOL 60 with features to make it suitable as a SIL[5]. The language requires a complete display which is held in the general-purpose registers. This strategy imposes restrictions on the static block structure of programs as well as inhibiting the use of the registers in expressions within deeply nested programs.

The IMP language allows one to specify a routine as **short** in which case the 4K byte addressing problem is avoided. The program analyzed here is without **short** and is 16 bytes longer.

The code uses the store multiple instruction just before the call to dump both the parameters and the display in the stack. A load multiple instruction is used after the return to recover the old environment. This technique produces a longer calling sequence than RTL/2 but executes much the same number of instructions. Since IMP uses a full display, the store/load multiple instructions dump/restore more registers and hence takes longer than RTL/2. Although both compilers do register optimization, it appears that RTL/2 is marginally more successful in terms of code space by loading the parameters before they are strictly required.

5.1.4 BCPL

The BCPL compiler produces code that is broadly similar to that of RTL/2. Both languages do not need a full display and use store multiple at the beginning of the routine to dump the parameters and the environment. One difference is that BCPL uses

a small run-time subroutine for procedure exit (instead of load multiple to restore the old environment). The quality of the code is not quite as good as the other systems so far—for instance, the result of the function is stored in core three times once for each leg of Ackermann.

5.1.5 ALGOL 60

This compiler was produced by modifying the Edinburgh IMP system and hence the code is very similar. The only changes are that minor register optimizations are not performed. The register optimization, although it increased the speed of execution, actually produced bigger code. Hence the ALGOL 60 code is marginally more compact.

5.2 THE ICL 1900 FAMILY

The ICL 1900 Series has a relatively simple instruction set. Eight integer registers are provided but only 3 are available for address modification. The word length is 24 bits: instructions and integers occupy one word, floating point numbers two words. Within an instruction, 12 bits are available for addressing either the first 4K words of the program or 4K words offset from an index register.

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	PLAN	7.5	57	W. Findlay
ALGOL 68-R	Malvern (no heap)	28	153	P. Wetherall
PASCAL	Belfast	32.5	129+	W. Findlay
ALGOL 68-R	Malvern (heap)	34	162+	P. Wetherall

5.2.1 Assembler

The 1900 machine code implementation is a straightforward coding of the algorithm. As before, the outer calls are replaced by jumps and only the inner call stacks a link and one parameter in two words. Only one register needs to be initialized and hence a significant space gain is made over the 360 coding.

5.2.2 ALGOL 68-R

The two ALGOL 68-R versions are identical in the code for Ackermann itself. The difference lies in the fact that with the complete program as written, the loader can determine that no heap is required. Under these circumstances, six instructions are omitted from the procedure entry code which are otherwise necessary to check that stack space is still available. Without the heap, no explicit software check is made on the stack height since the hardware protection mechanism on the machine can fault the program.

The ALGOL 68-R system uses a complete display with storage allocated at block level. The display cannot, of course, be held in registers and hence it is placed in the innermost block. Copying this display on procedure entry accounts for most of the overhead on procedure call. The size of the display could be reduced by allocating storage at procedure level, as has been done by a second ALGOL 68 compiler produced at Malvern (but not released).

5.2.3 PASCAL

The 1900 PASCAL system[6] uses a different method of handling the display than ALGOL 68-R. Although a complete display is necessary, it will be smaller because storage can only be allocated at procedure level. This restriction has a disadvantage to users that declarations cannot be made so 'local' as with a language with blocks. The method of handling the display is by back-chaining from the current level. To set up the chain on procedure entry, the easiest method is to pick up the pointer before the call. The number of instructions that this will require depends upon the difference in static level between the call and the procedure. In the case of Ackermann, the pointer is not necessary at all since the procedure is at global level—however, the 1900 implementation does not spot this.

This PASCAL system does not use a loader and in consequence a jump table is used to link to the two subroutines for entry and exit. The use of subroutines produces more compact code than ALGOL 68-R. In spite of the check on the stack height, PASCAL ought to give slightly faster code than ALGOL 68-R but the extra jump instructions stop this. A similar quality code to ALGOL 68-R would give about six instructions fewer per call.

5.3 THE PDP11 FAMILY

The PDP11 Series has a relatively large instruction set for a minicomputer with a 16-bit word length. Like the 360, byte addressing is used, necessitating multiplication by two in many high-level language constructs. Although all instructions are 2 bytes, a literal can (and must) be placed after the instruction unless all the operands are within the six general-purpose registers. A wide selection of addressing modes are available, including auto-increment and auto-decrement. It is difficult to produce compact code on the PDPll since ordinary store access involves 4 bytes—2 for the instruction and two for the literal (address). It is very easy to use any register as a stack address register—in fact register 6 is used in this way for interrupt handling.

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	PAL	7.5	32	W. A. Wulf
Bliss	CMU, opt	7.5	32	W. A. Wulf
Bliss	CMU	10	64	W. A. Wulf
PALGOL	NPL	13	86	M. J. Parsons
BCPL	Cambridge	20.5	104	M. Richards
C	UNIX	26	62+	P. Klint
Sue-ll	Toronto	26.5	176	J. J. Horning
RTL/2	ICI	30.5	70+	J. Barnes

5.3.1 Assembler

The machine coding for the PDP11 follows the same strategy as for the other eases. The ability to increment and decrement registers in one 16-bit instruction gives a compact program of 32 bytes. As before, n is kept in a register while m is stacked with the link on the inner call. The stacking of the link is part of the subroutine call instruction.

5.3.2 Bliss[7]

The optimized Bliss/11 (i.e. Bliss for the PDP11) coding uses a facility within the language to declare a parameter linkage convention. This allows the parameters to be

passed in registers and makes further optimization by the compiler that much easier. Only the declaration of the linkage method is machine-dependent. Other 'tricks' are used to give better code as can be seen from the listing:

```
linkage B01 = Bliss(register = 0, register = 1);

macro Inc(Var) = (Var:= .Var+l),
          Dec(Var) = (Var: = .Var-1);

routine B01 Ack(N, M) =
      begin
      if .M = 0 then return Inc(N)
      Ack(if Dec(N)≥0 then Ack(.N, .M) else 1,
            Dec(M))
      end;
```

Although the program is a reasonably clear expression of the algorithm it is not clear that a programmer who did not know the machine and some of the properties of the compiler would express Ackermann's function in this way.

The second version for Bliss/11 is a straightforward encoding of the ALGOL 60 version. The main penalty is not the speed (instructions per call) but the doubling in the code size. This is caused by addressing core rather than using registers. The code uses a single stack register for both stacking parameters and addressing variables. This is straightforward on the PDP11 provided the languages do not have second order working store.

5.3.3 PALGOL

The NPL PALGOL system is a dialect of ALGOL 60 with the restriction of no second order working store and access only to locals and globals. In this way, only one stack register is required. The compiler is a simple one and performs no significant optimization. In particular, the increment and decrement instructions are not used (can they be used ?— they do not set the carry bit). All the PDP11 compilers have to manipulate the stack because the 'obvious' calling sequence places the parameters beneath the link which is the wrong way round for the procedure exit code.

5.3.4 BCPL

This compiler produces open code for entry and exit but does not make very good use of the available registers. Several unnecessary MOV instructions are generated which has a significant effect on the code size. The code uses two registers for the stack, one for locals and the other for pushing/popping parameters although only one is strictly necessary. Unlike PALGOL, the increment and decrement instructions are used. Comparatively small modifications (by hand) to the code reduce the instructions executed to 13.5 per call.

5.3.5 C

This language is a dialect of BCPL but with additional 'type' information added to permit the declaration of structures with different field lengths[8]. The C code generator is evidently designed to produce compact rather than fast code. Subroutines are used for

entry and exit. Two registers are used for the stack as with the Cambridge system. A post-processor attempts some further optimization which was successful in this case. Code for the last two legs were combined giving the code that would be generated from:

$$Ackermann(m-1,\ \text{if } n = 0 \text{ then } 1 \text{ else } Ackermann(m, n-1))$$

5.3.6 Sue-11

This language is based upon PASCAL and requires a complete display[9]. This is handled by back-chaining using open code throughout. The registers are not well used, so the code is very voluminous. Clearly, generality rather than space or time has been the main aims of the system. With only eight registers, one cannot hold the display within them, so back-chaining or maintaining a core copy would seem the best method. However, the code produced here shows the penalty that can be involved if no optimization is performed.

5.3.7 RTL/2

The code generator for the PDP11 is rather less successful than that for the 360. The system aims at producing compact code and hence uses subroutines for entry and exit. However, these subroutines are entered via a TRAP instruction rather than the ordinary subroutine call. Although this is marginally more compact, the speed overhead in decoding a TRAP instruction is significant, this being the main cause of its relative slowness. This is the only PDP11 system to insert a check on the size of the stack, incurring an additional overhead. In fact the system records the amount of stack space used, involving 6 instructions altogether.

5.4 THE DEC10 FAMILY

The DEC PDP10 Series is a 36-bit architecture with a very extensive instruction set. Push and pop instructions are available to manipulate stacks provided the address is within one of the 16 registers. Hardware is also available to address fields within a word. An analysis has been produced showing that it is difficult to exploit such a large instruction set and so many registers[10].

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	PAL	5	85	J. Palme
Bliss	CMU	15	103+	W. A. Wulf

5.4.1 Assembler

This coding is roughly equivalent to the other machine code examples. The outer calls are replaced by jumps, leaving the one inner recursive call. The fact that only five instructions are required per call is a consequence of the rich instruction set—for instance, one can decrement a register and jump unconditionally in one instruction. The stacking orders allow one to handle the recursive call in three instructions.

5.4.2 Bliss[11]

This coding is a straightforward conversion of the ALGOL 60. Two subroutines are used for procedure entry and exit for setting and restoring environments. The compiler

does not do the optimization of the Bliss/11 compiler although the reloading of register values is avoided. The compiler does not seem to be able to exploit the rich instruction set in the way that is possible with hand coding. Two registers are used for addressing, one for locals and the other for pushing/popping parameters.

5.5 THE BURROUGHS FAMILY

The two Burroughs machines—the B5500 and B6700—are not compatible but share a common design philosophy. Both are naturally programmed in ALGOL 60 and, indeed, the coding given by the compiler is the only conceivable one. The B5500 has fixed length instructions of 12 bits (4 per word) whereas the B6700 has variable length instructions. Both instruction sets are stack organized, but the B5500 only permits access to locals and globals while the B6700 has a complete display in hardware. It is difficult to make any comparison between these machines and conventional hardware because the basic approach is so different. The advantages in terms of code size and instructions executed are apparent.

Language	Compiler	Instr./call	Size (bytes)	Source
ALGOL 60	XALGOL	19.5	57	R. Backhouse
ALGOL 60	XALGOL	16	57	G. Goos

5.5.1 B5500

The fact that both the Burroughs machines give the same size of code is just luck since the binary machine code is very different. Stack addresses are usually 10 bits and are loaded into the stack in one instruction. Similarly, a value or descriptor may also be loaded. The other possibility for the top 2 bits of an instruction gives a 10-bit addressless operation code. This gives compact code except for common operations such as subtracting one or testing for equality with zero which takes two instructions. The only improvement that could be made to the code by hand coding is removing three instructions at the end of the procedure which stores the result of the function in a local variable and then reloads it into the stack.

5.5.2 B6700

With this machine, variable length instructions are used, the addressless ones being 8 bits. Special instructions are available for loading 0 and 1 making the code more compact. This gain is offset by additional space needed in the longer instructions to address a complete display (5 bits). The display is updated by the procedure calling instructions without any explicit code. No improvement to the code generated seems possible. A small piece of optimization is performed by removing the storing and reloading of the result of the function.

5.6 THE CDC 6000/CYBER SERIES

The CDC Series sacrifices simplicity in design for the need to produce very high-speed processors. Within a 60-bit word, instructions are 15 or 30 bits long but cannot overlap word boundaries. Hence it is sometimes necessary to pad out with dummy instructions— amounting to 20 per cent of the code in some cases. Jumps can only be to the beginning of word boundaries. Pre-normalization rather than post-normalization

makes the generation of accurate code awkward as does the absence of integer multiply on the early members of the range. Eight general-purpose registers and eight index registers are provided, but the registers are divided into 'fetch' and 'store' type. Data can only be fetched into one type and stored from the other type. This means that more instructions are required for store access and a higher premium is paid for keeping calculations within the registers. In general, instructions involving different registers will be executed in parallel on the faster machines— a challenge for a good code generator.

Language	Compiler	Instr./call	Size (bytes)	Source
Assembler	Compass	15.5	83	W. M. Waite
Assembler, Opt	Compass	9.5	60	D. Grune
PASCAL	3.4 Zurich	38.5	232	N. Wirth
ALEPH	Amsterdam 17.1	41.5	292	D. Grune
Mini-ALGOL 68	Amsterdam	51	292	L. Ammeraal

5.6.1 Assembler

This coding has been produced to follow as closely as possible the other machine code examples. The extra number of instructions are mainly due to the lack of power of individual instructions on the 6000 Series. Several register to register instructions are required which would be unnecessary on more conventional architectures. Dummy instructions are not included in the execution count but contribute to the space. There is a smaller percentage of dummy instructions with hand coding because more information is kept in registers (permitting 15-bit instructions to be used with no alignment problems).

5.6.2 Assembler, Opt

This version was produced by very careful use of the registers but following the same logic as above. Dummy instructions were avoided as were several register to register instructions. This example illustrates the difference that can arise with machine-level coding due to the amount of effort spent. Dr. Grune made the following observation: 'This is probably not the limit. By abandoning all sense of decent programming I could possibly combine M and the return address into a floating point number, and then by judicious use of the floating point divide instruction, I could . . . etc.'

5.6.3 PASCAL

This compiler produces good code for a difficult machine. The language requires a complete display which is back-chained, but unlike 1900 PASCAL, the chain is avoided with global procedures (like Ackermann). Parameters are passed through registers (in this case). Since the subroutine instruction places the link within the code area, the call is handled by an ordinary jump preceded by setting the link in a register. The absence of conditional expressions loses 6 per cent on space and a further 11 per cent is lost through dummy instructions for padding.

A check on the stack is necessary, but this is performed twice, once at the beginning of the procedure and then on the inner recursive call because the stack must be incremented. This is a very cautious approach—the usual method adopted is to make the check less often but ensure 50 or so words are left spare. As the addresses for the check are in registers the penalty is quite small[1].

[1] I have been told that the March 1976 release of the compiler no longer generates the second stack test

5.6.4 ALEPH[12]

This language is designed for compiler writing rather than being general purpose. Although the language does not require a full display and has no second order working store, flexible global arrays are available. Such arrays give some flexibility in storage allocation without adversely affecting the procedure calling mechanism. Parameters are passed through registers, and so the comparatively unoptimized code is quite efficient. The code includes a stack overflow check.

5.6.5 Mini-ALGOL 68[13]

The language is a true subset of ALGOL 68 including as much as possible without a heap. The compiler produces relatively straightforward code which on this machine is rather long. A display is maintained in store which is updated by copying on procedure entry. A check is made for stack overflow on procedure entry as well. Tests for equality uses the general (safe) method of performing subtraction both ways round and oring the two results. Testing for equality with zero can, of course, be performed directly as with the PASCAL code. A slight gain is made over PASCAL by use of conditional expressions.

6 GENERAL COMMENTS

It is difficult to draw many conclusions from this study for a number of reasons. Firstly, the quality of code produced varies considerably, even though only the better systems have been analyzed. Secondly, it is not easy to tell why the code produced takes the form that it does. Unused generality in this test may result in redundant instructions whose purpose is obscure. Lastly, comparison between machine ranges is clearly dangerous unless one has very detailed knowledge of each range, which the author does not have.

One reason for the user to write procedures is to reduce the volume of machine code produced by the compiler. This reduction will depend upon the organizational overhead in the procedure call, procedure entry and procedure exit. Making the space occupied by the procedure entry and exit code small is not easy. The problem is that the open code probably amounts to 4 or 5 instructions, but control routines would need extra parameters so that the space reduction in their use is hardly worthwhile. The compiler writer would like to augment the machine instruction repertoire by two instructions for this purpose. The TRAP instruction on the PDP11 and the undefined operations on the DEC10 would apparently provide this facility but in both cases the decoding overhead is too large for it to be cost-effective.

The usefulness (to the compiler writer) of the subroutine call instruction varies considerably from machine to machine. The CDC 6000 Series and many minicomputers have a call instruction which plants the return address in the location before the first instruction of the subroutine. This is very awkward since it inhibits recursion (and re-entrant code). The majority of machines seem to be like the IBM 360 and leave the return address in a register. This provides the compiler writer with a useful call instruction and the flexibility he needs for handling the return address. The PDP11 and DEC10 are some of the few machines to place the return address in the main memory

saving two instructions per call. This illustrates the problem of keeping performance information about software up to date.

in a fashion which is acceptable to almost any high-level language. The only problem with the PDP11 stacking mechanism is that when parameters and local variables are also stacked, the return sequence is awkward.

Apart from the return address, the main function of the procedure entry and exit code is to establish and then restore the appropriate 'environment'. This environment ensures addressibility to the parameters, local variables and non-locals. On the Burroughs machines, the environment is established as part of the procedure call instruction, but in other machines explicit loading and storing of pointers is necessary. For this purpose, the load/store multiple instructions as on the IBM 360 are particularly convenient. If the environment is kept mainly as a display in core, then a move or block copy instruction is more useful. However, it sometimes happens that such instructions are ineffective because of the registers that must be set up first.

The best results obtained to date in high-level languages are as follows:

Instructions executed

- Languages with access to locals and globals only and no variable array space:

  10 instructions per call Bliss/11

- Full display but no variable array space:

  30.5 instructions per call MARY(SM4)

- Additional storage management apart from a stack:

  34 instructions per call ALGOL 68-R

Space The most compact results have been for B5500 and B6700 ALGOL with a total of 57 bytes of code.

7 ACKNOWLEDGEMENTS

This comparison would not have been possible without the work of all the contributors in sending the author listings and timings. The timings and machine code versions are particularly difficult to produce. The author would also like to thank the members of IFIP Working Group 2.4 (Machine Oriented Higher Level Languages) for both encouragement and criticism of earlier drafts of this paper. The highly perceptive comments of the referee has also permitted several improvements to be made.

References

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

[2] J. D. Ichbiah, J. P. Rissen, J. C. Heliard and P. Cousot, 'The system implementation language LIS', *Technical Report 4549 E1/EN CII*, Louveciennes, 1976.

[3] Imperial Chemical Industries, *RTL/2 Language Specification*, 1974.

[4] B. A. Wichmann, *ALGOL 60 Compilation and Assessment,* Academic Press, London, 1973.

[5] P. D. Stephens, 'The IMP language and compiler', *Computer J.* 17, 216-223 (1974).

[6] J. Welsh and C. Quinn, 'A PASCAL compiler for ICL 1900 series computers', *Software—Practice and Experience*, 2, 73-77 (1972).

[7] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.

[8] D. M. Ritchie, *C Reference Manual*, Bell Telephone Laboratories, Murray llill (undated).

[9] B. L. Clark and F. J. B. Ham, 'The Project SUE systems language reference manual', *Technical Report CSRG 42*, University of Toronto, 1974.

[10] A. Lunde, *Evaluation of instruction set processor architecture by program tracing*, Ph.D. thesis, Carnegie-Mellon University, 1974.

[11] W. A. Wulf, D. B. Russell and A. N. Habermann, 'Bliss: a language for systems programming', *Comm. ACM*, 14, 780-790 (1971).

[12] R. Bosch, D. Grune and L. Meertens, 'ALEPH: a language encouraging program hierarchy', *Proc. Int. Computing Symposium Davos*, North-Holland Publishing Co., Amsterdam, pp. 73-79, 1974.

[13] L. Ammeraal, 'An implementation of an ALGOL 68 sublanguage', *Proc. Int. Computing Symposium*, 1975 North Holland Publishing Co., Amsterdam, pp. 49-53, 1975.

A Document details

Scanned and converted to LaTeX, February 2002.