International Journal of Computer and Information Sciences, Vol. 14, No. 1, 1985

Architectural Support for Variable Addressing in Ada[1]— A Design Approach

Prasenjit Biswas[2] and Subrata Dasgupta[3]

Received July 1984; revised April 1985

In designing a language-directed machine architecture, the choice of the technique used in interpreting machine instructions has considerable influence on machine performance. Yet, there does not appear to exist any well established design method for choosing an interpretive mechanism; or for determining the hardware/firmware support for an efficient implementation of such a mechanism. The purpose of this paper is to propose such a design method, based on the use of an architecture description language. The specific architectural focus of the paper is the variable-addressing mechanism in Ada and the implications that such mechanisms have on the implementation of procedure CALL/RETURN and block ENTRY/EXIT functions. The analysis presented in this paper clearly establishes that either Dijkstra's mechanism or "local display method" is not suitable for adoption in designing architectural support for variable-addressing in Ada.

KEY WORDS: language-directed architecture; virtual and real transfer complexity; variable-addressing mechanisms proposed by Rohl, Tanenbaum, Dijkstra and the "local-display method".

1. INTRODUCTION

In designing an instruction set for a language-directed machine, usage statistics on various high-level language (HLL) constructs play an

---

[1] Ada is a registered trademark of the United States Government Ada joint program office.

[2] Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas 75275.

[3] Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, Louisiana 70504.

0091-7036/85/0200-0051$04.50/0 © 1985 Plenum Publishing Corporation

obviously important role.[1–3] Furthermore, the choice of the technique used in interpreting these instructions will have considerable influence on machine performance. Yet, there does not appear to exist any well established design method for choosing an interpretive mechanism or for determining the appropriate hardware/firmware support for an efficient implementation of such a mechenism. The purpose of this paper is to propose such a design method. The methodology is used in designing the architectural support for variable-addressing in Ada. In a more general sense it is also intended as a contribution to architecture design methodology.[4]

The architectural focus of this paper is the variable-addressing mechanism in programs written in block-structured language Ada and the implications that such mechanisms have on implementation of procedure CALL/RETURN and block ENTRY/EXIT functions.[4] It is well known that these are some of the most important and frequently used operations in block-structured HLL environment; hence they are highly appropriate candidates for architectural support through semantically "close" instructions in a language directed machine. A one-to-one correspondence between such HLL functions and machine level instructions can lead to effective exploitation of the available processor/memory bandwidth.[2] This of course, implies that a single instruction like CALL (for example) would include in its semantics and its implementation a sequence of operations over and above those necessary for actual transfer of control.

The execution performance of the CALL/RETURN and ENTRY/EXIT instructions will clearly depend on the technique used to implement the variable-addressing mechanism. So the architect of a language-directed architecture is faced with the problem of choosing the most efficient implementation technique.

The results presented in this paper may be compared to De Prycker's work.[5] De Prycker attempted to evaluate two different variable-addressing methods implemented in existing architectures. We, on the other hand, concentrate primarily on the development of a method for designing the hardware/firmware support for the HLL procedure CALL/RETURN and block ENTRY/EXIT functions for a language-directed architecture.

Various techniques have been proposed by system designers for implementing the variable-addressing mechanism in block-structured languages. We will consider the following four techniques:

(a) The "classical" display mechanism as suggested by Dijkstra.[6]

(b) A modified display mechanism proposed by Rohl.[7]

[4] Block structuring in Ada is similar to that of Algol with the important exception that Ada does not allow formal procedure parameters.

(c) A local display mechanism as implemented in the ICL 2900 Pascal compiler.[8] This was also implemented as part of a virtual architecture for Ada.[9]

(d) Tanenbaum's proposal.[3]

We shall use the term architectural *component* to denote any part of an architecture that can be studied, analyzed and designed in reasonable isolation. For example, the architectural support provided for the HLL functions of procedure CALL and RETURN can be viewed as an architectural component. An examination of the execution[5] of such components within the framework of von Neumann style architectures reveals that the transfer operation is by far the predominating operation. Transfers can occur between:

(i) two memory locations;

(ii) a processor register and a memory location;

(iii) two processor registers.

Thus, it is quite reasonable to compare different implementation techniques in terms of the number of transfer operations to implement an architectural component in a given host processor.[6] If we assign appropriate weights for the different kinds of transfers and if the necessary statistics on program behavior are known, the average cost of execution of a particular implementation of an architectural component on a given host processor can be determined. This is, therefore, essentially a technique for analyzing the suitability of host processors for implementing given HLL functions. A similar study for two of the previously mentioned techniques for implementing variable-addressing mechanisms [(a) and (d)] was reported by De Prycker.[5] However, as we shall show, the method we propose here should be regarded essentially as a design method rather than a technique for analysis.

To begin with, our only assumption concerning the underlying host processor organization is that it conforms to the von Neumann style. Thus, in the absence of any *particular* host structure it is necessary that we use an abstract way of describing architectural components and their implementations. The architecture description language (ADL) S^*A is used for this

[5] For convenience we use the phrase "execution of a component" to denote the collective sequence of events that would take place if a realization of the component were to be activated.

[6] To avoid any terminological confusion we shall use the term "host" to denote any processor on which an architectural component supporting HLL functions are being implemented. We have borrowed this term, for obvious analogical reasons, from the terminology of microprogramming and emulation.

purpose.[4,10,11] From an S^*A description of an architectural component we derive two measures termed the Virtual Transfer Complexity (VTC) and the Average Virtual Transfer Complexity (AVTC). As explained in Section 3, the VTC measure is a design aid in that it indicates the minimum transfer complexity that could be achieved using a particular implementation technique. Note that the VTC and AVTC measures as determined from an S^*A description are independent of the characteristics of any particular host structure. The AVTC measure, however, is dependent on the usage statistics obtained from the programming language environment under consideration.

The rest of the paper is organized as follows. In Section 2, we develop the notion of virtual transfer complexity. We also include some example VTCs computed for a few typical S^*A statements. In Section 3, we review briefly, the relevant notions of maintenance and variable access in the run time representation of a typical Algol-like block-structured environment. Complete S^*A descriptions of the procedure CALL/RETURN (CR) and block ENTRY/EXIT (BE) components for all four implementation techniques have been developed and are described in Ref. 12. For the sake of brevity, however, we include in this paper only two of these descriptions, viz., those of the techniques suggested by Rohl and Tanenbaum. In Section 4, we present the AVTs for the CR and BE components corresponding to each implementation technique.

In the remaining sections we derive a host processor organization from the S^*A descriptions and evaluate the Real Transfer Complexity (RTC) of the components for this organization. A comparison of the AVTC and ARTC values indicate that the choice of implementation techniques could be restricted to those suggested by Rohl and Tanenbaum. It is, moreover, concluded that the choice between these two schemes depends critically on the relative frequency of procedures being declared at an intermediate (i.e., neither local nor global) lexical level with respect to the calling level.

2. VIRTUAL TRANSFER COMPLEXITY

The ADL S^*A has previously been described in several papers (see Refs. 10, 11, and 13). A complete definition of the language is available and has appeared in a book.[4] The use of S^*A in describing architectures facilitates the design process in a number of ways:

(a) It makes it possible to describe the functioning and behavior of an architectural component at a level that is essentially independent of any particular hardware/firmware implementation of the component.

(b) The description helps identification of the specific statistics of the HLL environment upon which the performance of the architectural component depends. This, in turn, facilitates the collection of the appropriate usage statistics.

(c) The description clearly indicates the possible trade-offs between M and R measures[14] necessary for performance improvement. Furthermore, if all the architectural components of a language-directed architecture are considered together, the S^*A descriptions of the components collectively provide a fairly complete picture of the hardware/firmware requirements of the underlying processor.

(d) Because S^*A has been formally defined, it becomes possible to formally verify the correctness of an architectural design at a very early stage in the design process.[13]

Given an S^*A description of an architectural component, its virtual transfer complexity (VTC), when using a particular technique, is a measure of the number of transfers necessary for the "execution" of the component. In computing the VTC, all the variables are assumed to be available in the registers of a virtual processor. The array types are considered as register banks with the usual access mechanism, and a transfer through an ALU is viewed as a register—register transfer between the input and output buffers of an ALU.

In the next section we will present the VTC and AVTC of the CALL/RETURN and ENTRY/EXIT components for all four of the variable-addressing mechanisms mentioned in Section 1. Before that, however, we illustrate the method of computing the VTC by taking some typical S^*A statements.

Let us assume the following data type and data object declarations:

type M = seq [..] bit;
glovar A, B : array [..] of M;
glovar X, Y, P, Q : M

Here, we have defined a sequence data type named M; (global) variables X, Y, P, and Q are declared as instances of this type, while A and B are declared as arrays of elements of type M. The square brackets [..] in the first two declarations simply represent the (unspecified) sizes of the sequence and array respectively.

Example 1.

$$A[X] := B[P + A[Q]]$$

The transfers are: from Q to the selection mechanism of A; from the selected element of A to ALU_1; from P to ALU_2; transfer through the ALU; from ALU_0 to the selection mechanism of B; transfer from X to the selection mechanism of A; and, finally, transfer from the selected element in B to the selected element in A. The VTC of this statement is 7.

Example 2.

$$\text{while } P = Q \text{ do } S \text{ od}$$

Here S denotes the body of the *while* statement. If the VTC of S is w and assuming that S is iterated m times, the VTC of the *while* statement is $mw + 3(m+1)$; the predicate is evaluated $m+1$ times and in each evaluation three transfers are involved—from P to the comparator; from Q to the comparator; and a transfer through the comparator. The transfer involved in the branching operation is ignored—that is, we ignore the information transfer resulting from a transfer of control.

3. VARIABLE ADDRESSING TECHNIQUES

A program written in a block-structured language may be depicted as a tree which would highlight the nested structure of program blocks.[15,16] Each block would be associated with a level in the tree called its *static lexical level.*

Execution of a program may be viewed as a dynamic changing of lexical levels, hence, the lexical level of a block during program execution is termed its dynamic lexical level.

From a "static" point of view (i.e., the program as viewed at compile time) blocks and procedures are treated identically in that the body of a procedure is viewed as a block and is assigned a lexical level one higher than the level of declaration of the procedure identifier. Within a block, a variable is uniquely identified by the ordered pair

$$\langle \text{lexical level, sequence number} \rangle$$

where "lexical level" denotes the static lexical level of the block in which the variable is declared and "sequence number" indicates the relative position of the declaration in the block.

A dynamic change in lexical levels occurs during a block invocation or a procedure call. However, the nature of the change in the two cases are quite different. The new (dynamic) level due to a block invocation is identical to the static level of the invoked block whereas, calling a procedure

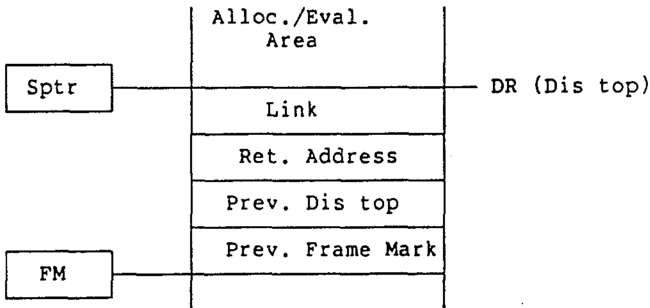
changes the dynamic level to correspond to the static level of the procedure body—that is, to a level one higher than the static level of the procedure declaration. An exit of a block or a return from a procedure requires the dynamic level to be reset to the level at which the block was invoked or the procedure called.

According to the scope rule for Ada a variable may be accessed if it is declared in the same block or in a statically surrounding block—i.e., if the variable is declared in one of the levels corresponding to the nodes in the path from the root of the tree to the node corresponding to the level at which the variable is referenced. Thus, the set of nodes on the path from the root to the active node characterizes the legitimate lexical levels in the variable accessing environment. In the case of procedure calls, the procedure identifier is treated as a variable, identified by means of the usual ⟨lexical level, sequence number⟩ pair, and the rule for procedure accessibility is the same as that for a variable.

In a typical Ada environment blocks may be treated as degenerate procedures called from the level at which they are defined. The foregoing discussion indicates that we need two linked list structures to represent the static and dynamic aspects of the environment. Usually, the expression evaluation/storage allocation stack is combined with these two linked list structures to form an *activation stack*. The stack frames corresponding to the static program structure are connected through a static link, while dynamic program activity is recorded and maintained through a dynamic link in each stack frame. Note that the dynamic and static links in a frame allocated as a result of a block invocation contain the same values, that both point to the previous frame; whereas in a frame allocated to the body of a procedure, the dynamic link points to the previous frame, while the static link points to the frame in which the procedure identifier was allocated. The dynamic link information is used in reverting back to the dynamic procedure call/block invocation level when the procedure/block execution has been completed. The static link is utilized in accessing non-local variables.

With this general description in mind, accessing a variable in the "parent" static environment requires following the static link chain to the appropriate level of declaration. Thus, depending on the difference in the lexical levels of access and declaration, a variable access could involve several levels of indirection. To reduce this overhead, Dijkstra suggested the use of an extra set of *display locations*, in the form of a stack.[6] According to this scheme any accessible lexical level can be directly reached through the display location associated with that level. The top of the display stack points to the current active frame, while the remaining display contains copies of the static links of the accessing environment.

A major problem with this now classical technique, is in rebuilding the display stack on return from a procedure; the number of locations to be reset is directly proportional to the lexical level difference between the calling level and that of the procedure declaration. A modification to this technique was suggested by Rohl.[7] He observed that in Dijkstra's scheme, there is a redundancy of information within the static links, the dynamic links, and the display. Rohl's modified version is described by the S^*A mechanism M_2 (Fig. 2) while Fig. 1a depicts the base of the corresponding stack frame. The static and dynamic links are replaced here by a single link that connects only the frames associated with the same lexical level. Thus, whenever we overwrite the contents of a display location (during a CALL or an ENTRY) the corresponding information is stored in the stack frame associated with that dynamic lexical level. So we can see that the display rebuilding overhead after returning from a procedure is independent of the lexical level difference between the calling level and the level of the



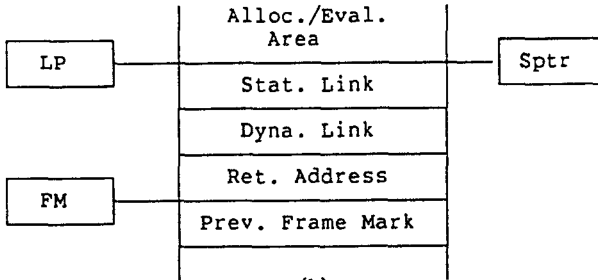


Fig. 1. A view of the current stack frame just after execution of the procedures. (a) $M_2 \cdot$ Call; (b) $M_4 \cdot$ Call

```
Proc RETURN ;
    Sptr = Frame-mark ; (1) Frame-mark := Stack [sptr] ; (2)
    /* Deallocation of frame */
    Local := Display [Dis-top] ; (2)
    Display [Dis-top] := Stack [Local-1] (5)
    /* reinstated the content that was destroyed due
         to call */
    Dis-top := Stack [Local - 3] ; (5)
    Pctr := Stack [Local - 2] ; (5)
EndProc ;

Proc ENTRY ;

    Stack [Sptr] := Frame-mark ; (2)
    Frame-mark := Sptr; (1) Sptr := Sptr + 1 ; (3)
    Dis-top := Dis-top + 1 ; (3)
    Stack [Sptr] := Display [Dis-Top] ; Sptr := Sptr + 1 ;
    /* As in call */

    Display [Dis-top] := Sptr ; (3)
    /* Display [Dis-top] points to present environment */
endProc ;

Proc EXIT ;
    Sptr := Frame-mark ; (1)
            Frame-mark := stack [sptr] ; (2)
    Display [Dis-top] :=
         Stack [Display [Dis-top] - 1] ; (6)
    /* restored the content overwritten due to
       Block entry */
    Dis-top := Dis-top - 1 ; (3)
EndProc ;

End Mech M2 ;
```

Fig. 2. $S*A$ descriptions of CR and BE components of mechanism M_2 (the numbers in parentheses indicate VTCs of the statements).

```
Mech M4 ; /* TANENBAUM's METHOD */
       type memword = seq [..] bit;
       glovar mainmem: array [..] of memword;
       syn stack = mainmem;
       glovar sptr, Frame-mark, Pctr : memword;
       glovar LP, GP : memword;
       glovar Inst-reg : tuple
                              adr : tuple
                                    Dlex: seq[..] bit;
                                    Offset : seq [..] bit;
                                    endtup
                              endtup;
                              : memword;
       privar chain, Ptr : memword ;

Proc CALL;
       Stack [Sptr] := Frame-mark ; (2)
       Frame-mark := Sptr ; (1)  Sptr:=Sptr + 1 ; (3)
       Stack [Sptr] := Pctr ; (2) Sptr := Sptr  + 1 ; (3)
       Stack [Sptr] := LP ; (2) sptr := Sptr + 1 ; (3)
       /* Dynamic link stored */
       If Inst-reg. Adr. Dlex = 'Global' (3)
             Stack [Sptr] := GP ; (2)  Ptr := GP; (1)
       || Inst-reg. Adr. Dlex = 0 (3) Stack [sptr] := LP ; (2)
                                                Ptr := LP; (1)
       || DO chain := Inst-reg.Adr. Dlex ; (1)

         WHILE Chain ¬= 0 (3 [m + 1])
         DO    Ptr := Stack [Ptr - 1] ; (4)
                     Chain := Chain - 1 ; (3)
         OD

           Stack [Sptr] := ptr ; (2) Sptr := Sptr + 1 ; (3)
           /* static link set */
         OD  /* m = lexical level difference between
           the level of call and the level of declaration
           of the procedure identifier , when the procedure
           identifier is an intermediate variable . */
       fi ;
         LP := Sptr ; (1) /* New LP set */
         Pctr := Stack [ptr + Inst.reg. adr. offset] ; (5)
EndProc ;
```

```
Proc RETURN ;
      Sptr := Frame-mark ; (1)
              Frame-mark := stack [sptr] ; (2)
      Pctr := Stack [LP - 3] ; (5) /* ret. address  */
      LP : Stack [LP - 2] ; (5) /* LP points to
            Calling environment */
endProc;
Proc ENTRY ;
      Stack [sptr] := Frame-mark ; (2)
      Frame-mark := Sptr; (1)  Sptr := Sptr + 1 ; (3)
      Stack [sptr] := LP ; (2) sptr := sptr + 1 ; (3)

      Stack [sptr] := LP ; (2) sptr := sptr + 1 ; (3)
      /* both links are stored, for uniformity in var
         access mem */
      LP := Sptr ; (1)
endProc ;


Proc  EXIT ;
      Sptr := Frame-mark ; (1)
              Frame-mark := Stack [Sptr] ; (2)
      LP := Stack [LP - 1] ; (4)


endProc


end Mech M4 ;
```

Fig. 3. S^*A descriptions of the CR and BE components of mechanism M_4 (numbers in parentheses indicate VTCs of the statements).

procedure declaration—it involves only the resetting of a single display location associated with the static lexical level of the procedure body.

Another technique for implementing variable-addressing through display locations is the so-called "local display" method.[8,9] In this method, the display locations necessary to represent the current variable accessing environment are stored on the current frame of the activation stack. The current frame is deallocated on return from a procedure or on exit from a block. Thus, there is practically no overhead for rebuilding the display on return from a procedure.

A significant variation on the basic theme was suggested by Tanenbaum[3] whose basic premise (which may not altogether be true for Algol-like languages) was that most variable accesses are local or global in nature—thus dedicated display locations for nonlocal references were unnecessary. He proposed, instead, the use of two dedicated display locations or registers, one for pointing to the base of the current frame, the other to the global frame of the activation stack (Fig. 1b). The S^*A mechanism M_4 (Fig. 3) describes the operations.

4. EVALUATION OF VIRTUAL TRANSFER COMPLEXITIES

Based on the foregoing S^*A descriptions we can now evaluate the virtual transfer complexities of the archtectural components (in the following, the legend M_i. CR and M_i. BE denote respectively, the CALL/RETURN and ENTRY/EXIT components in mechanism M_i).

1A: $\mathrm{VTC}(M_1 \cdot \mathrm{CR}) = 64 + 12\delta_p$
(where δ_p represents the difference in lexical levels between the point of calling and the point of declaration of the procedure identifier)
$\mathrm{AVTC}(M_1 \cdot \mathrm{CR}) = 64 + 12\hat{\delta}_p$
(where $\hat{\delta}_p$ is the average δ_p computed over a large set of sample programs characterizing the programming language environment under consideration)

1B: $\mathrm{VTC}(M_1 \cdot \mathrm{BE}) = 23 + 6 = 29$
$\mathrm{AVTC}(M_1 \cdot \mathrm{BE}) = 29$

2A: $\mathrm{VTC}(M_2 \cdot \mathrm{CR}) = 37 + 20 = 57$
$\mathrm{AVTC}(M_2 \cdot \mathrm{CR}) = 57$

2B: $\mathrm{VTC}(M_2 \cdot \mathrm{BE}) = 12 + 12 = 24$
$\mathrm{AVTC}(M_2 \cdot \mathrm{BE}) = 24$

3A: $\mathrm{VTC}(M_3 \cdot \mathrm{CR}) = 35 + 12\beta + 3(\beta + 1) + 12 = 50 + 15\beta$
(where β is the lexical level of the procedure identifier)
$\mathrm{AVTC}(M_3 \cdot \mathrm{CR}) = 50 + 15\hat{\beta}$
(where $\hat{\beta}$ is the average β computed over a large number of sample programs)

3B: $\mathrm{VTC}(M_3 \cdot \mathrm{BE}) = 20 + 9v + 6(v + 1) + 7 = 33 + 15v$
(where v is the number of displays to be transferred from the previous frame)
$\mathrm{AVTC}(M_3 \cdot \mathrm{BE}) = 33 + 15\hat{v}$
(where $\hat{v}$ is the average v computed over a large number of sample programs)

4A: $\mathrm{VTC}(M_4 \cdot \mathrm{CR}) =$
if procedure identifier is a global variable
then $(22+6)+13=41$
if procedure identifier is a local variable at calling level
then $(22+3+6)+13=44$
otherwise $(22+3+3+(10\delta'_p+13))+13=54+10\delta'_p$
$\mathrm{AVTC}(M_4 \cdot \mathrm{CR}) = 35+\eta_{pg}(6)+\eta_{pl}(9)+\eta_{pi}(29+10\hat{\delta}'_p)$

where

η_{pg} = relative frequency of a procedure identifier being a global variable

η_{pl} = relative frequency of a procedure identifier being a local variable

η_{pi} = relative frequency of a procedure identifier being an intermediate variable

We assume that the relative frequencies are obtained by considering all the (static) procedure calls in a large number of sample programs characterizing the programming language environment.

4B: $\mathrm{VTC}(M_4 \cdot \mathrm{BE}) = 24$
$\mathrm{AVTC}(M_4 \cdot \mathrm{BE}) = 24$

4.1. Comparison of the AVTCs

For purposes of comparison we use the following preliminary statistics on Algol 60 obtained by De Prycker[1]:

$$\hat{\delta}_p \simeq 2; \qquad \hat{\beta} \simeq 1; \qquad \hat{v} \simeq 2$$

Using these values we observe that:

$$\mathrm{AVTC}(M_1 \cdot \mathrm{CR}) \simeq 88; \qquad \mathrm{AVTC}(M_2 \cdot \mathrm{CR}) = 57; \qquad \mathrm{AVTC}(M_3 \cdot \mathrm{CR}) \simeq 65$$

Thus, as far as the CR component is concerned, $M_2 \cdot \mathrm{CR}$ appears to be the most superior. Similarly, in the case of the BE component we obtain:

$$\mathrm{AVTC}(M_1 \cdot \mathrm{BE}) = 29; \qquad \mathrm{AVTC}(M_2 \cdot \mathrm{BE}) = 24; \qquad \mathrm{AVTC}(M_3 \cdot \mathrm{BE}) \simeq 63$$

Again, $M_2 \cdot \mathrm{BE}$ appears to be the most efficient.

It is difficult to make an absolute comparison between $M_4 \cdot \mathrm{CR}$ and the other CR components as the statistics η_{pg}, η_{pl}, and η_{pi} are unavailable. Nonetheless, it is possible to make an interesting observation when we compare $\mathrm{AVTC}(M_2 \cdot \mathrm{CR})$—the best of the first three CR components—and $\mathrm{AVTC}(M_4 \cdot \mathrm{CR})$.

Let $\eta_{pl}+\eta_{pg}$ be denoted by η_{lg}. Then

$$\text{AVTC}(M_4 \cdot \text{CR}) \simeq 35+\eta_{lg}(7.5)+\eta_{pi}(29+20)=35+7.5\eta_{lg}+49\eta_{pi}$$

We know that $\text{AVTC}(M_2 \cdot \text{CR})=57$, and $\eta_{lg}+\eta_{pi}=1$. Thus, we see that if the relative frequency of declaring a procedure at an intermediate level (η_{pi}) is greater than 0.35, then

$$\text{AVTC}(M_4 \cdot \text{CR}+M_4 \cdot \text{BE}) > \text{AVTC}(M_2 \cdot \text{CR}+M_2 \cdot \text{BE})$$

(Note: $\text{AVTC}(M_2 \cdot \text{BE})=\text{AVTC}(M_4 \cdot \text{BE})$)

If we want to compare these implementation techniques for the purpose of selecting a variable accessing mechanism for a block-structured environment, we need to take into account the VTCs of variable accesses. For the description of the four mechanisms it is obvious that the VTC of accessing a variable is the same for M_1 and M_2; this is less than the VTC of a variable access in M_3. The discussion on the average cost (per average program) of using any of these techniques is deferred to a later section (see Section 6) where we include the cost of variable access in the evaluation.

5. THE REAL TRANSFER COMPLEXITY

The AVTCs indicate the best that the designer can extract from the implementation techniques. He is now faced with the problem of choosing one of the techniques for implementation on a real host architecture. Note that a choice based on AVTC computations might not be the best for a 'real' hardware/firmware realization unless we have an "ideal" range for mapping the domain of $S*A$ constructs onto "real" hardware/firmware. (The following discussion is based on the use of a microprogram controlled processor; however, the conclusions reached would remain valid for an equivalent hardwired control scheme.)

The *real transfer complexity* (RTC) of an $S*A$ statement is expressed as a matrix of three components, viz., the number of register-register transfers $R(S_1)$, the number of transfers through the ALU $A(S_1)$, and the number of transfers across the processor-memory interface $\text{MP}(S_1)$, required for the execution of a statement S_1, appearing in an $S*A$ description. Thus:

$$\text{RTC}(S_1)=[R(S_1)\ A(S_1)\ MP(S_1)]$$

We denote the RTC of a component X in mechanism M_t as $\text{RTC}(M_t \cdot X)$ where

$$\text{RTC}(M_t \cdot X)=\sum_i \text{RTC}(S_i)$$

the summation taken over all the statements in the execution of the component $M_t \cdot X$. The Average RTC(ARTC) is calculated as before.

In the S^*A description of the architectural components we have intentionally avoided any possible parallelisms among the various statements. Similarly, we assume that the microprogram control of the real host processor is of a purely vertical nature. We make these simplifying assumptions so that the fundamental issues in our design method are clearly visible. In passing, we note that the language S^*A has adequate constructs to express parallelism.[4,10] The mechanisms could easily be modified to include parallel operations, in which case the underlying firmware control (to be designed) would be changed to a horizontal scheme. The evaluation of the corresponding VTCs and RTCs would require very simple modifications.

Before we can compute the RTCs we need to propose an organization for the hardware/firmware complex that will support these components. The requirements that the host must meet to achieve a RTC measure reasonably close to the VTC measure can be directly extracted from the S^*A descriptions.

Rather than describe four different processor organizations for the four mechanisms, we adopt a basic von Neumann style structure similar to Fuller's canonical processor[14] and indicate any special requirements needed for each mechanism. This organization is shown in Fig. 4. Though not clearly indicated in the diagram, it is assumed that any two of the registers can participate in a register-register transfer through a common bus.

The stack is assumed to be in main memory. The *glovar* Display is mapped onto a bank of registers, while the *privar* Dis_top is mapped onto a register that is used as a selector index for the register bank, Display. The *glovar*'s Sptr, Pctr, and Frame_mark are mapped onto the processor registers SP, PC, and FM; Inst_reg is mapped onto IP (instruction register). The *privar's* local (in M_1 and M_2) as well as the *glovar* LP (in M_4) are mapped onto the processor register L. The *privar* Ptr is mapped onto the register PT. The *privar's* Stat (in M_1), Base (in M_3), and the *gloval* GP is mapped onto the register T1. At this point we do not specify which of the registers are user addressable and which ones are only used by the microprograms. Such a specification is quite straightforward if we restrict ourselves to a specific mechanism. In any case, the microcode can address any one of the registers. The register T2 is to be only used by the microcode. The register ACC serves as one of the inputs to the ALU. ALU_2 and ALU_0 are, respectively the second input and the output registers of the ALU. For any monadic ALU operation, the operand is expected to be in ACC.

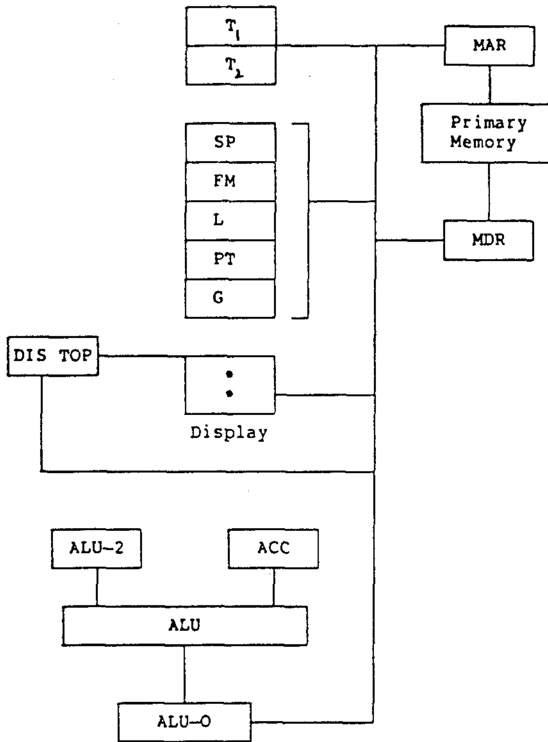


Fig. 4. The processor organization (control lines are not shown).

The choice of this particular processor structure represents a direct implementation of the S^*A description on a typical von Neumann style microarchitecture with a "pure" vertical microprogram control in which obvious design constraints lead us to implement the *glovar* stack in main memory with a constant access base (st_base) known to the firmware. It should be noted that the processor is not biased towards any particular mechanism and, thus, the comparison of RTCs based on this processor structure can be considered to be fair.

We now present examples of RTC evaluations for some typical S^*A statements involved in describing the mechanisms.

1. Stack [Sptr] := Display [Dis_top];
   As mentioned earlier Display is a bank of registers is in the processor indexed by the register Dis_top. The transfers for this statement are:

Display [Dis_top] → MDR; 2(R)
Stack-base → ALU_2; (R) SP → ACC; (R)
ADD; (A) ALU_0 → MAR; (R) WRITE; (M)
The RTC of the statement is: $[5_R 1_A 1_M]$

2. Pctr := Stack [Display [Dis_top + Inst_reg. adr. offset]]
The Inst_reg.adr. offset field of Inst_reg is known to the firmware and appropriate mask in known to the firmware for extracting the field.
The transfers are:
IR → ACC; (R) Mask (offset) → ALU_2; (R)
AND; (A) ALU_0 → ALU_2; (R) Dis_top → ACC; (R)
ADD; (A) Display [ALU_0] → ALU_2; 2 (R)
Stack-base ACC; (R) ADD; (A) ALU_0 → MAR; (R)
READ; (M) MDR → PC; (R)

(**Note:** The offset field was assumed to be right aligned; in cases of other fields e.g. Inst_reg. adr. Lex, the SHIFT operations required is counted as an average of one ALU transfer.)
The RTC of the statement is: $[9_R 3_A 1_M]$
The RTCs for the other statements can be evaluated in a similar fashion.

The real transfer complexity (RTC) and the Average RTC(ARTC) of components discussed earlier are as follows:

1a. $$\text{RTC of } (M_1 \cdot \text{CR}) = [87_R 27_A 11_M] + (n+1)[2_R 1_A 0_M] + n[10_R 3_A 1_M]$$
$$= [(79 + 12\delta_p)_R (24 + 4\delta_p)(10 + \delta_p)_M]$$
$$= [1\ \delta_p] \cdot \begin{bmatrix} 79_R & 24_A & 10_M \\ 12_R & 4_A & 1_M \end{bmatrix}$$

The ARTC ($\text{M}_1 \cdot \text{CR}$) is obtained by replacing δ_p by $\hat{\delta}_p$ in the previous expression.

1b. $\text{RTC}(M_1 \cdot \text{BE}) = [32_R 9_A 4_M] = \text{ARTC}(M_1 \cdot \text{BE})$

2a. $\text{RTC}(M_2 \cdot \text{CR}) = [70_R 22_A 9_M] = \text{ARTC}(M_2 \cdot \text{CR})$

2b. $\text{RTC}(M_2 \cdot \text{BE}) = [32_R 9_A 4_M] = \text{ARTC}(M_2 \cdot \text{BE})$

3a. $$\text{RTC}(M_3 \cdot \text{CR}) = [1\ \beta] \cdot \begin{bmatrix} 63_R & 22_A & 8_M \\ 13_R & 6_A & 2_M \end{bmatrix}$$

The ARTC($M_3 \cdot \text{CR}$) is obtained by replacing β by $\hat{\beta}$ in the a previous expression.

3b. $\mathrm{RTC}(M_3 \cdot \mathrm{BE}) = [1\ v] \cdot \begin{bmatrix} 38_R & 13_A & 6_M \\ 15_R & 7_A & 3_M \end{bmatrix}$

The expression for $\mathrm{ARTC}(M_3 \cdot \mathrm{BE})$ is obtained by replacing v by $\hat{v}$ in the expression for $\mathrm{RTC}(M_3 \cdot \mathrm{BE})$.

4a. $\mathrm{RTC}(M_4 \cdot \mathrm{CR}) = [57_R 17_A 7_M]$ (if procedure identifier is a global variable)

or

$= [60_R 17_A 7_M]$ (if procedure identifier is local to the calling block)

or

$= [1\ \delta'_p] \begin{bmatrix} 70_R & 20_A & 8_M \\ 10_R & 3_A & 1_M \end{bmatrix}$

(if procedure identifier is an intermediate variable)

$$\mathrm{ARTC}(M_4 \cdot \mathrm{CR}) = [1 \eta_{pg} \eta_{pl} \eta_{pi}] \begin{bmatrix} 49_R & 16_A & 6_M \\ 8_R & 1_A & 1_M \\ 11_R & 1_A & 1_M \\ (21 + 10\hat{\delta}_p)_R & (4 + 3\hat{\delta}_p)_A & (2 + \hat{\delta}_p)_M \end{bmatrix}$$

where $\eta_{pg}, \eta_{pl}, \eta_{pi}$, and $\hat{\delta}_p$ have their usual meanings as defined earlier.

4b. $\mathrm{RTC}(M_4 \cdot \mathrm{BE}) = [30_R 9_A 5_M] = \mathrm{ARTC}(M_4 \cdot \mathrm{BE})$

On comparing the ARTs of the CR and BE components of the three mechanisms M_1, M_2, and M_3, we come to the same conclusion, as in the previous section that is, in evaluating the average transfer cost of a program we need only consider the mechanisms M_2 and M_4.

5.1. Overall Performance

As discussed earlier, in order to estimate the overall performance of an implementation technique for an "average" program it is essential to estimate the AVTC and ARTC of variable access. Furthermore, we need only compute these measures for the mechanisms M_2 and M_4.

The variable access (VARAC) component of a mechanism should be composed of two separate procedures for Read and Write access. As the ARTC/AVTC of both subcomponents are identical, we will only describe the procedure for Read access (RVAR) for the two mechanisms. The declarative part of M_2 and M_4 will not be repeated. We assume one more (system) *glovar* declaration for the variable Read_Value. In the description of the real processor architecture, Read_Value maps onto MDR.

The procedure RVAR for $M_2 \cdot$ VARAC is as follows:

Proc RVAR; /* for mech M_2 */
Read_Value := Stack [Display [Inst_reg. Adr. Lex] + Inst_reg. Adr. Offset] (6)
endproc

Thus:
AVTC($M_2 \cdot$ VARAC) = 6;
ARTC($M_2 \cdot$ VARAC) = $[11_R 5_A 1_M]$
The Procedure RVAR for $M_4 \cdot$ VARAC is as follows:
Proc RVAR /* for mech M_4 */
If Inst_reg. adr. Dlex = 'Global' (3) ⇒ Read_value := Stack [GP + offset]; (5)
|| Inst_reg. adr. Dlex = 0 (3) ⇒
|| Read_value := Stack [LP + offset] (5)
|| *DO* Chain := Inst_reg. adr. Dlex; (1)
Ptr := Stack [LP − 1]; (4)
WHILE Chain ≠ 0 $(3[p+1])$
DO Ptr := Stack [Ptr − 1]; (4)
Chain := Chain − 1; (3)
OD /* p = lex. level difference between the level of declaration and access of the intermediate var. */ *fi*;
endproc

The AVTC($M_4 \cdot$ VARAC) $= 5 + \eta_g \cdot 8 + \eta_l \cdot 11 + \eta_i[14 + 10\hat{\delta}_v]$, where η_g, η_l, and η_i are relative frequencies of global, local, and intermediate variable access respectively. $\hat{\delta}_v$ = The mean lex.lev. difference between the level of declaration and level of access of an intermediate variable computed over a large number of sample programs.

$$\text{ARTC}(M_4 \cdot \text{VARAC}) = [1 \eta_g \eta_l \eta_i] \begin{bmatrix} 4_R & 2_A & 0_M \\ 10_R & 3_A & 1_M \\ 13_R & 3_A & 1_M \\ (22 + 10\hat{\delta}_v)_R & (5 + 3\hat{\delta}_v)_A & (2 + \hat{\delta}_v) \end{bmatrix}$$

For comparing the two mechanisms, as far as variable access is concerned, we use the preliminary statistics in Ref. 5. The statistics are:

$$\eta_g \simeq 0.38, \qquad \eta_l \simeq 0.28, \qquad \eta_i \simeq 0.34, \qquad \hat{\delta}_v \simeq 1.0$$

On substituting, ARTC($M_4 \cdot$ VARAC) = $[22.3_R\ 6.5_A\ 1.7_M]$

AVTC($M_4 \cdot$ VARAC) = 13.77

We use De Prycker's "program activity characterization" model[1,5] to evaluate the average cost of using an implementation technique.

The average VTC of a block structured program using the technique proposed by Rohl is represented as:

$$\mathrm{AVTC}(P(M_2)) = n_b \cdot 24 + n_p \cdot 57 + n_v \cdot 6$$

where:

n_b = mean number of block ENTRY/EXIT's in a program

n_p = mean number of procedure CALL/RETURN's

n_v = mean number of variable accesses.

Similarly, for Tanenbaum's technique we obtain

$$\mathrm{AVTC}(P(M_2)) = n_b \cdot 24 + n_p(35 + 7.5 + 41.5\eta_{pi}) + 13.77n_v$$

Thus,

$$\mathrm{AVTC}(P(M_2)) < \mathrm{AVTC}(P(M_4))$$

provided that

$$24n_b + 57n_p + 6n_v < 24n_b + (42.5 + 41.5\eta_{pi})\, n_p + 13.77n_v$$

or, considering n_v/n_p to be a small positive quantity

$$\eta_{pi} \geqslant 0.35$$

(**Note:** The preliminary statistics obtained in Ref. 1 from nine numerical programs for digital filtering and speech recognition is used for these evaluations.)

So we see that our earlier observation regarding the effectiveness of using either of these implementation techniques remains unaltered. It is, however, extremely difficult to predict the effectiveness of either of the techniques in terms of ARTC unless some reliable statistics are available regarding n_p, n_v, n_b, and η_{pi}. We can only note that

$$\begin{aligned}\mathrm{ARTC}(P(M_4)) &- \mathrm{ARTC}(P(M_2)) \\ &= n_p[(31.5\eta_{pi} - 11.5)_R(9\eta_{pi} - 5)_A(5\eta_{pi} - 2)_M] \\ &\quad + n_b[-2_R 0_A 1_M] + n_v[10.2_R - 0.8_A 0.7_M]\end{aligned}$$

6. CONCLUSIONS

In this paper we have suggested a method for selecting the hardware/firmware support for the addressing mechanism in block-structured HLL environments. This method can obviously be applied to other architectural components.

The original problem that prompted this particular investigation was the choice of one of these four addressing techniques as part of the design of a machine directed towards the programming language Ada.[17] In the absence of usage statistics for an Ada environment, available data on Algol 60 was used. We were surprised to discover the effectiveness of the relatively less known technique suggested by Rohl. It might be noted that Rohl's technique is suitable as long as formal procedure parameters are not taken into consideration. It can be considered almost as good as Tanenbaum's technique for the Ada environment as Ada does not allow formal procedure parameters. In the absence of any reliable statistics for η_{pi} we could not come to a definite conclusion regarding the choice between Tanenbaum's and Rohl's techniques. But considering some of the available statistics, e.g., $\hat{\beta} = 1$, $\hat{\delta}_p = 2$, we may reasonably assume that the probability that $\eta_{pi} \geqslant 0.5$ is quite high. In such a case, Rohl's technique appears to be superior. The analysis clearly demonstrates that either Dijkstra's mechanism or the "local display" method is not suitable for adoption in designing the architectural support for variable-addressing of an architecture directed towards Ada.

REFERENCES

1. M. DePrycker, On the Development of a Measurement System for High Level Language Program Statistics, *IEEE Trans. Comput.*, **C-31**(9) 883–891, (September 1982).
2. G. J. Myers, *Advances in Computer Architecture*, Wiley-Interscience, John Wiley & Sons, New York, (1982).
3. A. S. Tanenbaum, Implications of Structured Programming for Machine Architecture, *Comm. ACM*, **21**(3) 237–245, (March 1978).
4. S. Dasgupta, *The design and description of computer architectures*, Willey-Interscience, John Wiley and Sons, New York (1984).
5. M. DePrycker, A Performance Analysis of the Implementation of Addressing Methods in Block Structured Languages, *IEEE Trans. Comput.*, **C-31**(2) 155–163, (February 1982).
6. E. W. Dijkstra, Recursive programming, *Numer. Math.*, **2**, (1960).
7. J. S. Rohl, *An Introduction to Compiler Writing*, MacDonald/Elsevier, London, (1975).
8. M. J. Rees, *et al.* Pascal on an Advanced Architecture, *Pascal–The Language and Its Implementations*, D. W. Barron (ed), John Wiley & Sons, New York, (1982).
9. O. Dommergaard, The Design of a Virtual Machine for Ada, *Towards a Formal Description of Ada*, D. Bjorner and O. N. Oest (eds), Springer-Verlag, Lecture Notes in Comp. Science, 98, Heidelberg, (1980).

10. S. Dasgupta, Computer design and description languages, *Advances in Computers* Vol. 21, M. C. Yovits (ed), Academic Press, New York, (1982).
11. S. Dasgupta and M. Olafsson, Towards a Family of Languages for the Design and Implementation of Machine Architectures, *Proc. 9th Annual Symp. on Comp. Architecture*, IEEE Comp. Soc. Press, New York, (1982).
12. P. Biswas and S. Dasgupta, Architectural support for variable addressing in block-structured languages, Tech Report 84-CSE-3, Southern Methodist University, (April 1984).
13. S. Dasgupta, On the Verification of Computer Architectures Using an Architecture Description Language, *Proc. 10th Annual Symp. on Computer Architecture*, IEEE Com. Soc. Press, Nex York, (1983).
14. S. H. Fuller and W. E. Burr, Measurement and Evaluation of Alternative Computer Architectures, *IEEE Computer*, (October 1977).
15. R. W. Doran, *Computer Architecture: A Structured Approach*, Academic Press, New York, (1979).
16. E. Horowitz, *Fundamentals of Programming Languages*, Computer Science Press, Rockville, Maryland, (1983).
17. P. Biswas, A capability architecture for Ada, *IEEE Comp. Soc. Conf. on Ada Appl. and Env.*, St. Paul, Minnesota, (October 1984).

Printed in Belgium