SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 12, 367–373 (1982)

Exploiting Machine-Specific Pointer Operations in Abstract Machines*

CHRISTOPHER W. FRASER AND DAVID R. HANSON

Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

Increasingly powerful machine instructions complicate abstract machine design for portability. Abstract machine instructions must be "larger" than the target machine instructions that they are to exploit, but they must not grow so large as to complicate the realization of the abstract machine on real machines. This paper presents related techniques for low-level yet machine-independent access to typical stack and string-processing instructions.

KEY WORDS abstract machines instruction sets portability stacks strings

INTRODUCTION

Abstract machine modelling[1] is widely used in the implementation of portable software, especially programming languages.[2,3] An "ideal" machine is tailored to a class of computers, programming languages, or applications. The software is written in the assembly language for the ideal machine or in high-level languages whose compilers generate code for the ideal machine. Once the ideal machine is realized on a real machine, typically through macro expansion or direct translation, all of the associated software is available on that real machine. Numerous successful language implementations use abstract machine modelling. Early examples include SNOBOL4[4] and BCPL;[5] more recent examples are MACRO SPITBOL,[6] Pascal,[7,8] Concurrent Pascal,[9] and Edison.[10]

Unfortunately, it is hard to design abstract machines that match a broad range of targets, and it is easy to let familiarity with a real machine overly influence the design of the abstract machine, defeating much of its purpose. One of the major difficulties is the level of the abstract machine. Making it too low-level may complicate the use of target machine instructions that are at a level higher than the abstract machine instructions. Making it too high-level complicates the realization of the abstract machine on a target machine. Moveover, the growing complexity of recent computers complicates choosing the appropriate level.

Although computer instruction sets vary widely, many computers, including most microprocessors, have stack and pointer instructions, which, although common in concept, vary in implementation. For example, many machines offer "auto-increment" instructions, which dereference a pointer register and increment it in the same instruction. Most auto-increment instructions adjust the pointer *after* the address calculation (e.g. PDP-11, Motorola 68000, TI 9900), but some adjust the pointer *before* (e.g. DEC-10). Likewise, many machines offer instructions that implement the familiar stack operations. On some (e.g. DEC-10) the stack grows *up*; on others (e.g. PDP-11) it grows *down*.

*This work was supported by the National Science Foundation under Grant MCS-7802545.

0038–0644/82/040367–07$01.00
© 1982 by John Wiley & Sons, Ltd.

Received 10 August 1981
Revised 25 November 1981

These variations complicate abstract machine design because even if the multiple operations involved are provided, choosing a direction of stack growth or point of pointer adjustment precludes the use of such features on computers with the opposite orientation. For example, MINIMAL, the MACRO SPITBOL abstract machine, assumes the stack grows down. This suits the PDP-11, for instance, but not the DEC-10. As such, the implementation of MACRO SPITBOL on the DEC-10 does not use the hardware stack instructions.

The remainder of this paper describes the design of abstract machine instructions so that these features can be used without complicating the realization process.

POINTER INSTRUCTIONS AND SEQUENCE TRAVERSAL

It is difficult to use auto-increment instructions in a machine-independent fashion because pre- and post-incrementing require slightly different algorithms. Consider, for example, the programming language C.[11] If p is a pointer variable, the expression ++p increments p and then yields its value, whereas p++ yields its value and then increments it. The unary operator "*" dereferences a pointer, and the assignment operator "=" yields the value assigned, so the null-bodied loop

```
while (*d++ = *s++)
        ;
```

copies the string referenced by s onto the string referenced by d and stops when it copies a zero. If d and s occupy registers, this yields a tight two-instruction loop on the post-incrementing PDP-11 and a suboptimal five-instruction loop on the pre-incrementing DEC-10. The similar loop

```
while (*++d = *++s)
        ;
```

yields the optimal three-instruction DEC-10 code but results in a poorer four-instruction PDP-11 sequence. It also requires a slightly different initialization sequence: s and d must point just *before* the first character, not *at* it. This change is hard to make automatically, so most programs suffer on one machine or the other.

Sequence pointers, a new abstract machine datatype, solve this problem. Sequence pointers are not so "close" to the hardware as to encode knowledge of whether pre- or post-incrementing is desired, but they are not so "distant" from the hardware as to be hard to implement on real machines. There are four operators that manipulate sequence pointers. The abstract machine instruction

```
pinit   p,s
```

creates a sequence pointer to s and deposits it in p. On post-incrementing machines, p references the first element of s. On pre-incrementing machines, it references the first element *before* s. The instruction

```
popen  p
```

"starts" incrementing p. That is, it increments p on pre-incrementing machines, but it does nothing on post-incrementing machines. Likewise,

```
pclose  p
```

"completes" an incrementing operation started by a popen. It increments p on post-

incrementing machines. but it does nothing on pre-incrementing machines. The "*" addressing operator dereferences a sequence pointer. For example.

```
load    r1,*p
```

loads abstract machine register r1 with the character referenced by p. The following program. which copies data from s to d until it copies a zero. illustrates the use of these features.

```
        pinit   r1,s
        pinit   r2,d
loop:   popen   r1
        load    r0,*r1
        pclose  r1
        popen   r2
        store   r0,*r2
        pclose  r2
        jumpeq  r0,0,loop
```

The popen and pclose instructions permit the increment to be placed in the best location for the target machine, and even a simple peephole optimizer can combine the increment with the indirect reference. An operator to abbreviate the open-dereference-close idiom above would reduce the volume of code and the load on the peephole optimizer. It would not. however, completely replace the operators above because it is sometimes useful to dereference an opened pointer in several places before closing it. and to open a pointer in one routine and close it in another.

High-level languages with built-in string or vector operations like SNOBOL4 already have features "large" enough to exploit sequence pointers. In contrast, systems programmming languages need minor extensions to offer source language operations equivalent to pinit, popen, and pclose. These extensions may be implemented by the abstract machine or by source language macros[11,12] if machine-dependent macro definitions are tolerable. For at least some applications, these minor extensions may be justified by their 30-50% speed-up of common string and vector processing operations.

STACK INSTRUCTIONS AND PROCEDURE INVOCATION

It is difficult to use a target machine's stack in a machine-independent fashion because calling sequences and access to parameters and local variables depend on the direction of stack growth. It is important to use the hardware stack because efficient calling sequences and efficient access to variables are of utmost importance in high-level language implementations.[13]

In typical implementations of recursive procedures, a "frame pointer" points at an activation record, or frame, for the currently executing procedure. Parameters and locals are stored in this activation record and are accessed via based addressing off the frame pointer. The procedure invocation and entry sequences allocate and initialize the activation record, adjusting the frame pointer accordingly. The exit sequence deallocates the record, restoring the frame pointer to its previous value in the process.†

Assuming that the stack grows up, that sp is the stack pointer, and that fp is the frame pointer, typical machine-dependent sequences on the DEC-10 are as follows.

†This approach does not handle nested procedures as in Pascal, which can be handled by dedicating some abstract machine registers for use as a display (c.f. Reference 14).

calling sequence:

```
        push    sp,arg1     ; push argument 1
        push    sp,arg2     ; push argument 2
        ...
        push    sp,argn     ; push argument n
        pushj   sp,name     ; call procedure
        decr    sp,n        ; remove arguments
```

entry sequence:

```
name:   push    sp,fp       ; save the frame pointer
        move    fp,sp       ; establish new frame pointer
        incr    sp,m        ; allocate space for m locals
```

exit sequence:

```
        move    sp,fp       ; deallocate locals
        pop     sp,fp       ; restore the frame pointer
        popj    sp,         ; return to caller
```

The layout of the activation record during execution of a procedure is shown in Figure 1. Parameters are accessed via negative offsets from fp, and locals by positive offsets from fp.† A similar layout can be arranged for a stack that grows down.

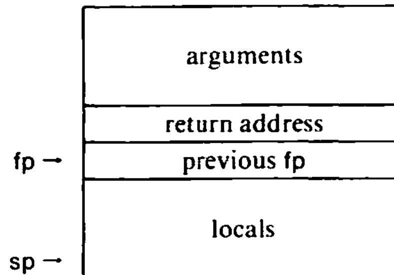

Figure 1. Activation Record Layout.

While it is possible to conceal the direction of stack growth within abstract machine instructions for procedure entry and exit, it is more difficult to do so for procedure invocation and variable accessing. The direction of stack growth dictates both the order in which arguments are pushed and the offset values used in references to parameters and locals.

The first problem can be resolved by concealing argument transmission within the call abstract machine instruction much as entry and exit sequences are concealed within procedure definition instructions. The abstract machine call instruction is

call *name*, arg_1, arg_2, ..., arg_n

where the arguments may be arbitrary operands. This form has the additional advantage of permitting the order of evaluation of the arguments to be defined by the language instead of its implementation.

The second problem—offsets in operands—cannot be solved in a similar manner. Some arrangement is needed that permits parameters and locals to be referenced in a way that is

†This includes locals declared explicitly in the source program as well as any 'implicit' locals needed to save registers or to buffer intermediate computations.

independent of stack growth direction. This is typically accomplished through extra operations for accessing parameters and locals.[7] It is, however, simpler to use *two* logical frame pointers, one for accessing parameters (ap), and one for accessing locals (lp). As described below, only *one* pointer is required in the realization of the abstract machine, but it is useful to describe the technique using two pointers.

Since ap and lp are set during procedure entry and exit, this technique amounts to shifting the dependence on the direction of stack growth from the operands to the entry and exit instructions. Assuming that parameters and locals are referenced with positive offsets from ap and lp, respectively, Figure 2 illustrates the frame layout for both directions of stack growth.

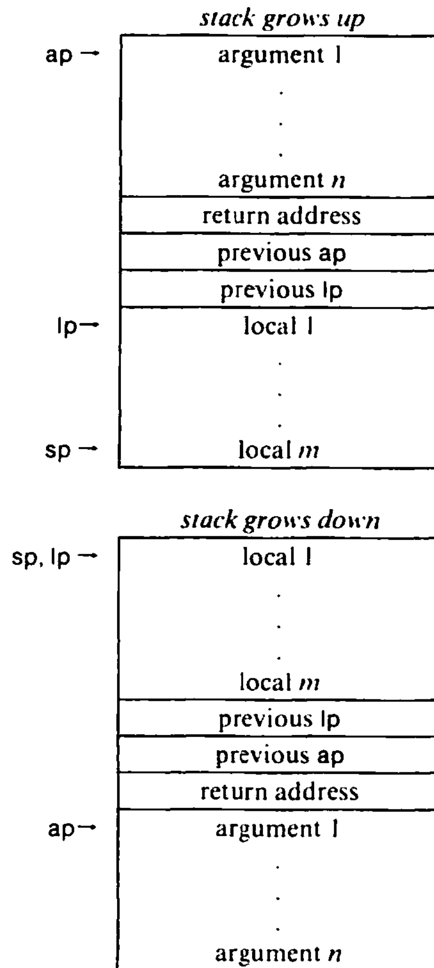

Figure 2. Activation Record Layout for both Stack Growth Directions.

Note that the arguments to a procedure must be pushed in the correct order by the call instruction. Calling, entry, and exit sequences corresponding to Figure 2 for a procedure with n arguments and m locals for the DEC-10 are as follows. The second column gives code for a hypothetical DEC-10 in which the stack grows down.

	stack grows up		*stack grows down*	
calling sequence:				
	push	sp,arg_1	push	sp,arg_n
	...		...	
	push	sp,arg_n	push	sp,arg_1
	pushj	sp,name	pushj	sp,name
	decr	sp,n	incr	sp,n
entry sequence:				
	push	sp,ap	push	sp,ap
	push	sp,lp	push	sp,lp
	movei	ap,-n-2(sp)	movei	ap,3(sp)
	incr	sp,m	decr	sp,m
	movei	lp,-m+1(sp)	movei	lp,0(sp)
exit sequence:				
	movei	sp,-1(lp)	movei	sp,m(lp)
	pop	sp,lp	pop	sp,lp
	pop	sp,ap	pop	sp,ap
	popj	sp,	popj	sp,

It is possible to dispense with either ap or lp because the difference between them depends only on n, m, and l, the space occupied by linkage information, all of which are constants. For example, for the layouts shown in Figure 2, if the stack grows up, lp is ap+n+l; if it grows down, it is ap-m-l. Thus, the translator can implement one of ap or lp and convert references to the other into offsets in terms of the chosen pointer. For instance, if lp is chosen on a machine whose stack grows down, references of the form x(ap) are converted to x+m+l(lp) during translation. In this way, parameters and locals may be accessed using simple indexing off of a single register, achieving in a machine-independent fashion the simplicity and efficiency of typical machine-dependent subroutine mechanisms.

DISCUSSION

The techniques described in the two sections above are related more closely than it may appear. Both techniques exploit increasingly common auto-increment and auto-decrement addressing. Accordingly, both techniques are increasingly likely to prove useful on the same class of target machines.

Many recent architectures imitate the PDP-11's post-increment and pre-decrement. These operations are presented as general addressing operations, but they are perhaps most used when looping over strings and vectors, and when manipulating stacks. They suit programs that increment pointers after dereferencing them, and they suit stacks that grow down, because, with this direction, simple indirection through the stack pointer accesses the item atop the stack. However, when the hardware instead offers a *pre*-increment and a *post*-decrement, pointer incrementation time and stack growth direction must be switched simultaneously. The techniques of the previous sections handle this reversal for two of the most common uses of auto-incrementing and auto-decrementing.

REFERENCES

1. M. C. Newey, P. C. Poole and W. M. Waite, 'Abstract machine modelling to produce portable software—a review and evaluation', *Software—Practice and Experience*, 2, 107-136 (1972).
2. S. S. Coleman, P. C. Poole and W. M. Waite, 'The mobile programming system: Janus', *Software—Practice and Experience*, 4, 5-23 (1974).
3. P. Kornerup, B. B. Kristensen and O. L. Madsen, 'Interpretation and code generation based on intermediate languages', *Software—Practice and Experience*, 10, 635-658 (1980).
4. R. E. Griswold, *The Macro Implementation of SNOBOL4: A Case Study in Machine-Independent Software Development*, W. H. Freeman, San Francisco, 1972.
5. M. Richards, 'The portability of the BCPL compiler', *Software—Practice and Experience*, 1, 135-146 (1971).
6. R. B. K. Dewar and A. P. McCann, 'MACRO SPITBOL—a SNOBOL4 compiler', *Software—Practice and Experience*, 7, 95-113 (1977).
7. B. K. Haddon and W. M. Waite, 'Experience with the universal intermediate language Janus', *Software—Practice and Experience*, 8, 601-616 (1978).
8. K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli and C. H. Jacobi, 'Pascal-P implementation notes', in *Pascal—The Language and its Implementation*, D. W. Barron (ed.), Wiley-Interscience, Chichester, UK, 1981, 125-170.
9. P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.
10. P. Brinch Hansen, 'Edison—a multiprocessor language', *Software—Practice and Experience*, 11, 325-361, (1981).
11. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1978.
12. W. A. Wulf, D. B. Russell and A. N. Habermann, 'BLISS: a language for systems programming', *Communications of the ACM*, 14, 780-790 (1971).
13. N. Wirth, 'The design of a PASCAL compiler', *Software—Practice and Experience*, 1, 309-333 (1971).
14. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.