(13)

Productivity Engineering in the UNIX† Environment

Improved RISC Support for the Smalltalk-80 Language

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

of each look up is saved by replacing the look up code with a direct call on the method found. Each method, in turn, has prologue code that checks that the first argument is of the appropriate class, and invokes a full look up otherwise.

Method caching is an execution-time strategy. Other investigators have considered compile-time strategies. These involve inferring the classes of operands so that the compiler can predict the methods that would be selected at execution time. The resulting sends can then be treated as ordinary non-polymorphic calls would be in a straightforward implementation of a standard procedural language. The compiler can also go further, and expand these calls in line, producing still more opportunities for simplification. Such in-line expansion can pay off handsomely in Smalltalk-80, simply because methods tend to be small. In general, researchers that have taken compile-time approaches have required class declarations in the programs they process in order to get achieve the best results. This is true, for example, in the Classy system at Berkeley [4], Atkinson's Hurricane compiler [1], and the Kiku system of Suzuki and Terada [6].

In this paper, I consider ways in which the performance of SOAR's method caching might be improved. These involve both possible changes to the SOAR architecture that are in keeping with the RISC philosophy behind its design, and software strategies that resemble in-line expansion, but are invoked dynamically.

The treatment assumes that the reader is somewhat with Smalltalk-80 and somewhat familiar with the SOAR architecture (as described by Ungar [7] and in detail by Samples, Klein, and Foley [5]). The next section describes SOAR's current caching scheme. Section 3 discusses possible changes to the SOAR architecture and how they could be used to improve the performance of sends. Section 4 demonstrates that some of the advantages of compile-time open coding can be achieved at execution time. Finally, section 5 discusses the implications of these changes.

2 Fast Calls In SOAR

For a few operators, the SOAR Smalltalk compiler will translate a send by open-coding it as one of the SOAR machine instructions. It uses SOAR machine instructions for such operations as the binary '+' operator, which

Before first call:	*Before later calls:*
call lookup	call *address of method*
address of selector	*class of receiver*

Figure 1: In-line caching in SOAR: sender.

```
loadc    (r14)classOffset,r6   ; Fetch class of receiver into r6
%load    (r15)0,r5             ; Load class of last receiver
%trap3   ne, r5,r6             ; Trap if classes not equal
```

Figure 2: In-line caching in SOAR: called method's prologue.

is compiled into an integer addition. The tags on SOAR values distinguish integers from pointers, and the hardware checks that the operands of an addition are, in fact, integers, causing a trap if they are not.

For a send involving an ordinary operator (or *selector*, to use the Smalltalk term), the compiler produces the following two-word sequence shown on the left of Figure 2. When this is executed, the routine *lookup* determines the address of the code for the appropriate method, which depends on the selector and the class of the first operand (the *receiver*). It gets the selector from the return address of the call, and the receiver from its standard location in register 6. As a side-effect, it changes the initial sequence to the following. Subsequent executions of this code, therefore, go directly to the method found on the previous encounter.

Of course, should the call turn out to be used polymorphically, the above scheme alone will cause erroneous results on subsequent executions of the call. Therefore, the sendee verifies each cached send with the prologue shown in Figure 2. If the trap is taken, the trap handler can interpolate a call to *lookup*, fetching the necessary selector from a known location relative to the trap instruction (just before the 'loadc' instruction[2]).

SOAR handles sends to the superclass differently. However, since these are quite rare, they can be ignored for the purposes of this paper.

The code sequences above yield an overhead of 8 machine cycles for a send and matching return (1 for the call itself, 2 for the return, 2 for each

[2]'Loadc' is a special "load class" instruction whose purpose is to catch the case where the receiver is an integer, and therefore does not have its class within it. I will ignore this issue for the purposes of this paper.

load, and 1 for the trap), assuming that the cache hits. Ungar's figures [7] indicate that roughly 8% of execution time is spent in the prologue alone in the larger standard Smalltalk-80 benchmarks. The figure is 22% if we restrict measurement to user code—code generated by the compiler, as opposed to runtime support routines. His figures imply that if the costs of the call and return are included, the total minimum call overhead on cached methods amounts to 13% of execution time, or 35%, considering just user code.

3 Effects of Possible Changes to the SOAR Architecture

One could imagine trying to improve SOAR's call overhead using various schemes for bundling the entire call sequence and method prologue into a single complex call instruction. However, this approach would drastically change the flavor of the architecture. In this section, I'll consider changes that maintain the "RISCness" of SOAR. SOAR follows the RISC philosophy in that it has a simple, microcode-like instruction set and its support of high-level languages consists only of providing a variety of simple tag checks that can be performed in parallel with other operations.

First, consider the possibility of changing the representation of the inline cache to put checking at the call site. To do so, we first change the representation of an object's class in its header to be an integer index into a class table. The sole purpose of this change is to make it feasible to represent a class in the immediate field of a RISC instruction. Figure 3 shows possible new before-and-after call sequences. Here, the previous class for each call is stored in the instruction that tests that the class has not changed. This saves two cycles per dynamic call, at the cost of increasing the size of each call by 50%. However, Ungar's data indicates that the resulting change in image size would be small.

When the cache misses and the trap is taken, the trap handler can find the selector to use for the look up operation by following the call to its destination. As before, the selector can be stored in the word preceding the call target.

There is a slight awkwardness imposed by this scheme. SOAR imme-

Before first call:	*Before later calls:*	
call lookup	loadc	(r6)classOffset,rt
address of selector	%trap3	ne,r6,*previous class*
nop	call	*address of method*

Figure 3: An alternative in-line caching technique.

Before first call:	*Before later calls:*	
call lookup	tagtrap	ne,r6,*previous class*
address of selector	call	*address of method*

Figure 4: In-line caching with bigger tags.

diates are 12 bits long, which might seem to accomodate a moderately generous 4096 classes. However, the upper 4 bits of the immediate go to specifying the tag. If one is willing to tolerate an odd encoding for the class index (in which the upper 4 bits and lower 8 bits are significant) this need not matter. The major impact occurs when one must find the actual class object. An alternative is to change the SOAR architecture so to eliminate tagged immediates. Ungar suggests that this would slow SOAR down by 10%, but it is not clear that he considered all possible work-arounds, such as keeping the common immediates like **nil** in global registers.

It is more interesting to consider a more radical change. In the SPUR system [3], the 8-bit tags are not part of the 32 bit data portion of a word, so that an entire word is 40 bits long. Tags serve essentially the same purpose as in SOAR—that of allowing easy checks of data types. However, they can carry 6 bits of type information. Suppose that we extend the type information to 14 bits[3] and carry a type index with every object. Now a call may be implemented as shown in Figure 3. Here, the checking overhead for a cached call has been reduced to a single cycle. The new instruction 'tagtrap' compares an immediate field against the tag part of a register.

[3]In fact, the current implementation of SPUR actually uses an entire 64-bit word to carry 40 bits. This, however, was purely a matter of expedience in the development of a first implementation of the architecture.

4 Dynamic Open Coding

At this point, we have eliminated all but 1 cycle of the checking overhead for cached calls. This still leaves the overhead of the bare call-return sequence itself. This overhead would be inconsequential if it were not for the tendency for Smalltalk code to contain many sends.

A traditional way of dealing with the overhead of calls to small procedures is to open-code these procedures, placing a copy of the body of the procedure, suitably modified, at the call site. With suitable care, the same strategy may be adopted in SOAR, but with the expansion taking place at execution time.

Some obvious candidates for this strategy are the "pseudo-primitive" methods. In the standard Smalltalk-80 virtual machine, these are methods whose bodies are encoded in their headers, and which do nothing but return the receiver or return an instance variable. In each of these cases, the body of the method, exclusive of call and return, consists of a single SOAR instruction. Consider the right-hand code sequence in Figure 3. Instead of a call, the routine *lookup* can insert the actual single instruction to be executed (suitably modified to account for being used in the sender's context.) The compiler would recognize cases where this is possible and indicate to *lookup* that the substitution is desired by a suitable flag in the method. The resulting call overhead would then be a single cycle.

This approach complicates the handling of cache misses, since the address of the method is no longer immediately available at the call site. One solution is to provide a separate data structure in which to store a mapping of call sites to method selectors. The effect of this on the time required to handle cache misses is uncertain, since it depends on how often such pseudo-primitive sends are polymorphic.

Pseudo-primitive methods are not the only ones that might benefit from open codings. Any method whose body, exclusive of calls and returns, consists of a single instruction might be handled by the same technique, assuming the compiler determines this to be safe. Certain chains of calls, in which the body of the called method consists of a send to the receiver of the same arguments in the same order, may be eliminated by this technique.

5 Discussion

The purpose of the above exercise was to explore some architectural alternatives to compile-time type inference and user type declarations. By reducing the overhead of a non-polymorphic call from 8 cycles to 4 cycles, we reduce the total time spent in the check-call-return sequence from 13% of total execution time to 7.5%, and from 35% of execution time for user code to 17.5%. About 33% of all sends in SOAR (D'Ambrosio) are pseudo-primitive, and for these, at least, we can reduce the overhead to a single cycle.

This is not to say that compile-time type analysis along the lines of the Classy system [4] is unnecessary. Open coding a call realizes only part of the possible optimization of that call. Constant folding, register allocation, and other classical techniques can in some cases squeeze a great deal more speed out of a program. Rather, I have tried to show that there are substantial gains to be made by relatively straightforward modifications to the SOAR architecture.

6 Acknowledgement

This research was sponsored by the Defense Advanced Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

References

[1] Robert G. Atkinson. Hurricane: an optimizing compier for smalltalk. In *OOPSLA '86 Conference Proceedings*, pages 151–158, Association for Computing Machinery, September 1986. Also published as SIGPLAN Notices, volume 21, number 11 (Nov., 1986).

[2] Bruce D'Ambrosio. Smalltalk-80 language measurements: dynamic use of compiled methods. In David A. Patterson, editor, *Smalltalk on a RISC: Architectural Investigations*, pages 110–125, University of California, Berkeley, 1983. Berkeley technical report.

[3] Mark D. Hill, Susan J. Eggers, James R. Larus, and George S. Taylor, et al. SPUR: a VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–24, November 1986.

[4] James Larus and William Bush. Classy: a method for efficiently compiling smalltalk. In David A. Patterson, editor, *Smalltalk on a RISC: Architectural Investigations*, pages 186–202, University of California, Berkeley, 1983. Berkeley technical report.

[5] A. Dain Samples, Michael Klein, and Peter Foley. *SOAR Architecture.* Technical Report UCB/CS/85/226, University of California, Berkeley, Computer Science Division, March 1985.

[6] Norihisa Suzuki and Minoru Terada. Creating efficient systems for object-oriented languages. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 290–296, Association for Computing Machinery, January 1984.

[7] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* PhD thesis, University of California, Berkeley, 1986. Issued as technical report UCB/CSD 86/287.

SECURITY CLASSIFICATION OF THIS PAGE — AD-A179 325

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION The Regents of the University of California 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION SPAWAR
6c. ADDRESS (City, State, and ZIP Code) Berkeley, California 94720	7b. ADDRESS (City, State, and ZIP Code) Space and Naval Warfare Systems Command Washington, DC 20363-5100
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
* Improved RISC Support for the Smalltalk-80 Language

12. PERSONAL AUTHOR(S) Paul N. Hilfinger
*

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
technical	FROM _____ TO _____	* 3/16/87	* 8

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Enclosed in paper.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE