

Compiling Smalltalk-80 to a RISC

William R. Bush, A. Dain Samples, David Ungar, Paul N. Hilfinger*

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley CA 94720

*Computer Systems Laboratory
Electrical Engineering Department
Stanford University, Stanford CA 94305

*Proceedings of the Second International Conference on
Architectural Support for Programming Languages and Operating Systems
(ASPLOS II)
October 1987
pages 112-116*

The Smalltalk On A RISC project at U.C. Berkeley proves that a high-level object-oriented language can attain high performance on a modified reduced instruction set architecture. The single most important optimization is the removal of a layer of interpretation, compiling the bytecoded virtual machine instructions into low-level, register-based, hardware instructions. This paper describes the compiler and how it was affected by SOAR architectural features. The compiler generates code of reasonable density and speed. Because of Smalltalk-80's semantics, relatively few optimizations are possible, but hardware and software mechanisms at runtime offset these limitations. Register allocation for an architecture with register windows comprises the major task of the compiler. Performance analysis suggests that SOAR is not simple enough; several hardware features could be efficiently replaced by instruction sequences constructed by the compiler.

1. Introduction

The goal of the Smalltalk On A RISC (SOAR) project at U.C. Berkeley was to produce a high-performance execution engine for the Smalltalk-80¹ language. The heart of the effort is a Berkeley RISC processor extended to support Smalltalk. The processor was designed in conjunction with the runtime system, which together have yielded substantial performance improvements over conventional Smalltalk-80 implementations. An extensive performance evaluation can be found in Ungar's dissertation [1]; results with an instruction-level simulator using large benchmarks indicate that the SOAR system could run Smalltalk roughly 2.5 times as fast as a Motorola

¹Smalltalk-80 is a trademark of Xerox Corporation.

68010 with a similar cycle time, or about the same speed as the Xerox Dorado high-performance workstation.

One reason for the SOAR system's speed is its compilation of Smalltalk. It has been estimated that compiling Smalltalk to the SOAR instruction set produces a factor of two speedup over conventional interpreted systems, making compilation the single most important Smalltalk speedup technique of the SOAR system. This paper describes the SOAR compiler. We have detailed the processor [2] and runtime system [3] elsewhere, and will assume some familiarity with these, as well as with the Smalltalk-80 language. An overview of the SOAR project, emphasizing architectural results, is available [4].

2. The Nature of the Compiler

The Smalltalk-80 language is defined operationally in terms of a virtual machine that executes stack-based instructions called bytecodes. The Smalltalk-80 programming environment is a binary image that runs on the virtual machine. This virtual machine can be inefficient if naively implemented. It is commonly realized as a bytecode interpreter, which requires special hardware (such as that possessed by the Dorado) to avoid interposing a layer of overhead between the virtual machine and the native machine.

An alternative is to compile bytecodes to native machine instructions, an approach successfully taken by Deutsch and Schiffman [5]. The Deutsch and Schiffman system dynamically translates procedures (called *methods* in Smalltalk) as needed, keeping a cache of native-code methods and flushing the least-recently-used ones. The SOAR system takes this approach a step further and compiles all methods from bytecodes into SOAR instructions, discarding the bytecodes. Where caching is done for space efficiency, compiling everything simplifies and speeds up the system for a moderate cost in space.

We considered compiling Smalltalk directly to SOAR, avoiding bytecodes entirely, but we did not take this path for two reasons. First, the virtual machine is the semantic definition of the language. This implies that a correct bytecode compiler and correct runtime system functions, constitute a correct Smalltalk implementation. Second, the standard virtual image is a bytecode-based image; to produce a SOAR-based native image those bytecodes must be translated to SOAR instructions. Rather than develop two separate compilers, one for taking bytecodes to SOAR and another for compiling Smalltalk to SOAR, we implemented one bytecode compiler with two implementations: a C version for image conversion, and a Smalltalk version for use with the converted image.² We have discussed image conversion in a previous paper [3].

The basic task of the compiler is to translate stack-oriented bytecodes into RISC-style loads, stores, and other register-based instructions. It does this by assigning Smalltalk variables and stack locations to registers and memory locations, and then simulating at compile time the bytecode stack operations on a symbolic stack, converting the operations into SOAR instructions. The symbolic stack is used to remember value sources and operations; when a value destination is encountered the code to load, compute, and store the value is generated. If the Smalltalk variables A and B are assigned to registers, for example, a push of A and a pop-and-store into B is translated into a register-to-register move; the symbolic stack is used to remember the source A until the destination B is encountered.

²Debugging Smalltalk programs in a converted image, in which the standard virtual image debugger does not work, is addressed in Lee's report [6].

In a more complete example, Figure 1 presents the contents of the symbolic stack and the resulting emitted code as the four bytecodes to evaluate ‘C _ A + B’ are scanned in order. The angle brackets indicate the bytecode being scanned and the current top of the symbolic stack. The variables are local variables assigned to registers. After the ‘+’ operator has been scanned the stack contains one entry, the result of the operation. The operation is symbolic and so are its arguments.

The Smalltalk bytecodes perform the same compilation function as any stack-based intermediate language such as, for example, UCODE [7]. Bytecodes are unlike UCODE in that, as we will discuss later, they restrict the compiler writer, particularly in implementing optimizations that would result in code motion or that require type information to perform. For example, common subexpression elimination is exceptionally difficult to do in Smalltalk.

Given the restrictive semantics of the Smalltalk bytecodes and the simple architecture of SOAR, the compiler falls or stands on its success in mapping Smalltalk variables to registers and memory.

BYTECODES	SYMBOLIC STACK	EMITTED CODE
>pushVar: A	>1:A	-
pushVar: B	2:-	
send: +		
popIntoVar: C		

pushVar: A	1:A	-
>pushVar: B	>2:B	
send: +		
popIntoVar: C		

pushVar: A	>1:plus A B	-
pushVar: B	2:-	
>send: +		
popIntoVar: C		

pushVar: A	1:-	add rA,rB,rC
pushVar: B	2:-	
send: +		
>popIntoVar: C		

Figure 1: An Example of Bytecode-Based Compilation

3. SOAR Register Windows

One feature of the Berkeley RISC architectures is a register file of overlapping register windows, each window corresponding to a procedure activation frame. The windows are allocated on procedure call in a stack discipline, using the registers in the window overlap to pass parameters. The advantage of register windows is fast procedure call and return, avoiding the saving and restoring of registers. Tests with benchmarks indicate that SOAR Smalltalk would be 46% slower without them.³

It is crucial that the size of these register windows be chosen wisely. If a method requires more storage than can fit in a window, the extra values are spilled to memory, slowing down the procedure call and the method's execution as well as increasing its code size. If windows are too small, an excessive number of methods will spill, degrading the performance of the whole system. On the other hand, if windows are too large, registers will be wasted and fewer windows can be accommodated on the processor chip.

For SOAR, the goal was to have 90% to 95% of all activation frames fit in SOAR windows. Preliminary studies [9] indicated that a window size of 16 registers with 8-register overlap was best. The SOAR windows are smaller than those of other RISC designs such as RISC II, which has 32, because Smalltalk methods are correspondingly smaller than procedures in more traditional imperative-style languages.

An alternative would have been to use variable-size windows at some expense in hardware (see, for example, Katevenis' dissertation [10]). For SOAR, we concluded that variable-size windows would not result in significant enough improvement to offset the additional hardware complexity.

The SOAR hardware divides each window into two identical sets of 8 registers, a high set (15-8) and a low set (7-0). Each set contains 6 general purpose registers and 2 dedicated registers used for return addresses (15 and 7) and return values (14 and 6).

The currently executing method receives its parameters and stores its local variables in the high register set. It sets up parameters for any methods that it calls in the low set. Unlike other RISC designs, the SOAR architecture has no registers dedicated solely to local use -- all are shared between two activation records. This allows the compiler flexibility in register allocation; registers can be used for arguments or local values depending on what is appropriate. Temporaries that must persist through calls to other methods ('retained' temporaries) are put in free high registers. 'Transitory' temporaries (those whose life spans do not cross procedure calls, such as intermediate results from compiler-generated expressions) can be put in the lows. Figure 2 presents this categorization pictorially.

Allocating variables and temporaries to registers is complicated by the fact that it is possible to write a Smalltalk method whose variables and temporaries will not all fit in a register window. When more registers are required than are available, spilling is necessary. There are two rules used to determine what and when to spill to memory. The first rule of assignment by category specifies that entire categories of variables are spilled -- if not all of the arguments fit in the registers, for example, all are spilled. The second rule of permanent assignment means that a variable cannot be moved once it has been allocated a location -- if, for example, a local variable has been put in a

³ All results quoted in this paper are found in either Ungar's dissertation [4] or Bush's report [8], unless stated otherwise.

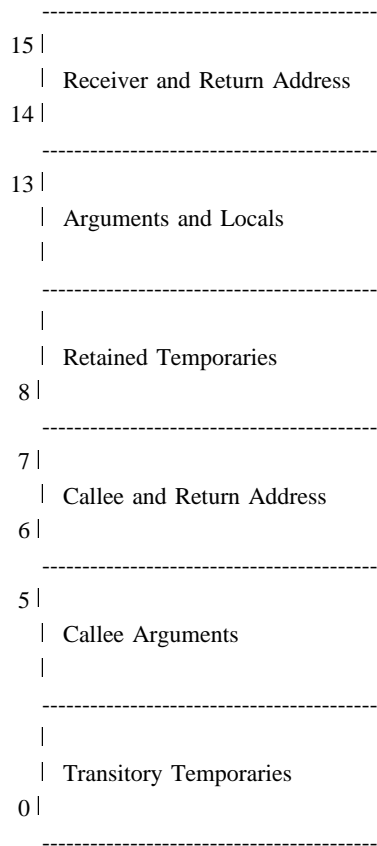


Figure 2: Register Allocation in a Register Window

register, it will not later be spilled to make room for a temporary. Neither of these rules result in minimal register usage or minimal memory traffic, but they are reasonable and simple. Since a major goal of the SOAR architectural design was to minimize spills at a reasonable cost, their infrequent occurrence justified an easily modifiable, simple spilling strategy.

The details of our allocation and assignment strategy are straightforward. The compiler assigns up to four arguments and locals to a method's high registers, and the remaining high registers are used for retained temporaries. If there are more arguments than will fit in four registers, all arguments are put in memory. If the arguments will fit in four registers, but the local variables will not fit with the arguments, all of the locals are stored in memory. Transitory temporaries can be stored in the lows. Retained temporaries are stored in registers if any are available at that point in the computation, and in memory otherwise.

A nice benefit of the above rules is that for most Smalltalk methods, code can be generated in one pass over the bytecodes by making some initial assumptions about the method's register requirements. If the assumptions turn out to be false, a second pass can be made with the new information. Only 9% of all methods in the Smalltalk-80 system require the second pass.⁴

Spills can be implemented in one of two ways. First, space can be allocated from a common spill pool, or a separate spill object can be allocated for each activation frame that spills. The former has complications for garbage collection and processes, and the latter eats up a register in each spilling frame. Both techniques have been tried in SOAR; neither has shown itself to be clearly superior over the other.

4. The SOAR Architecture Simplifies the Compiler

Register windows and spills complicate the compiler; SOAR architectural support for Smalltalk has in the main simplified it. An important architectural feature allows the compiler to generate standard arithmetic and comparison instructions in spite of the fact that the operations may be on non-integer objects.

Since Smalltalk is polymorphic and variable types are not known at compile time, it is never safe to generate integer instructions without runtime type checking. The virtual machine requires a dynamic method lookup on each operator, including '+'. Implementing this lookup naively is very expensive. Studies at Apple [11] and Hewlett-Packard [12] have shown that in fact most arithmetic operations in Smalltalk involve only integers. SOAR takes advantage of this fact by assuming that all simple arithmetic operations (plus, minus, comparisons, etc.) will be performed only on integers. The compiler thus treats all such operations as if they were on integers, and generates integer code. If, say, an add instruction is initiated on two objects, and one or both of them turn out to be non-integers, the hardware will trap and transfer to a handler that will look up the correct method for the intended operation. When the execution of the looked-up method is complete, the trap handler will return to the instruction immediately following the one that caused the trap.

It would not be difficult for the compiler to generate code to test explicitly for integer tags, but it would slow down system performance by an estimated 26% and would increase image size by 15%.

The tag mechanism is also used to provide hardware assistance for garbage collection, thus further simplifying the task of code generation. SOAR uses a generation scavenging scheme, dividing memory into regions of old and new objects, and uses generation tags on pointers to detect when pointers to new objects are stored into old ones. The tag check is performed in the hardware, and a trap handler records the necessary information if a trap is taken. However, benchmark results indicate that tagged stores are so rare that the compiler could in fact generate explicit tests and only slow the system about 1% and expand the image 2%.

The compiler also takes advantage of hardware that maps registers to memory addresses, and thus allows pointers to registers. Since a Smalltalk program can access any object in memory, and activation records are just another kind of object, it is necessary to handle references to them. Unfortunately, some activation records exist as on-chip register windows. The pointer to register feature permits the compiler to ignore complications that would otherwise be caused by overt references to activation records. Once again, however, it turns out that checking for pointers to

⁴The Smalltalk and the C versions of the compiler differ in this regard: the C version effectively makes two passes over each method. However, speed was not as critical for the C version as for the Smalltalk version.

activation records is not as severe a problem as it was initially feared to be. The instances requiring such a check do not occur often, and Ungar concluded that the feature could be removed with only a 3% performance penalty.

5. Bytecode Compilation Restricts the Compiler

Both language and pragmatic considerations limited the optimizations we could perform.

First, Smalltalk is a polymorphic language. That is, the same 'A+B' expression within a Smalltalk method could on one instantiation add two integers and on the next concatenate two strings. In the latter case it is not true that 'A+B' equals 'B+A'. Furthermore, which method is invoked is, by definition, a function of the leftmost operand ('A' in our example). Together these facts preclude almost all expression evaluation optimization.

Second, the fact that activation frames are full-fledged Smalltalk objects requires a relatively straightforward mapping between bytecode frames and compiled ones, eliminating optimizations such as method integration.

Third, our pragmatic goal was to bring up a working Smalltalk-80 image for the SOAR processor. This made us fundamentally conservative in our approach to the language. Correctness, as defined by the bytecodes and the image, was more important than efficiency. Because of this we were loath to change the image (including the standard Smalltalk-80 compiler) or take advantage of ambiguities in the language specification. We were not in the normal position of a compiler writer.

There is one optimization that has been employed successfully by Deutsch and Schiffman and by the SOAR system. Although the language permits polymorphic expressions, the fact of the matter is that 96% of the time the method invoked by a particular message send operation does not change. Thus, for example, if 'A+B' added complex numbers the last time it was evaluated, it is highly likely that it will be used to add complex numbers the next time it is evaluated. Using this observation we cache the last method called. When it is called again, we merely check that the type of the leftmost operand this call matches the type of the leftmost operand from the last call. If they do not match the standard method lookup mechanism is re-invoked. Ungar estimated that removing this caching strategy would slow down our system by 33%.

6. Results

Three types of performance measurements were made with the SOAR compiler: compilation speed, code expansion, and register window utilization.

Since the SOAR compiler is an extra stage added to the bytecode compiler, it can only slow compilation down. However, on a Dorado (equivalent in performance to SOAR) the compiler is reasonably fast both objectively and subjectively. Results from compiling the entire Smalltalk-80 image of 4770 methods indicate that it adds a mean time of 50 milliseconds to a method's total compilation time. This lengthens compilation by 70%. Subjectively, the compiler does not intrude on system use since it is usually invoked interactively on one method at a time. A mean total compilation time of about 120 milliseconds per method does not noticeably affect response time.

One of the virtues of bytecodes is their compactness. A major concern at the start of the project was an anticipated explosion in method size that could result from moving to four-byte word-sized instructions. Preliminary estimates indicated potential expansion of up to 1000% [13]. Fortunately, observed expansion with the actual implementation is considerably lower. For the

Smalltalk-80 image the expansion factor for methods⁵ is 3.78 -- one byte of bytecode method expands to an average 3.78 SOAR bytes. These expansion results compare favorably to the expansion factor of 5.03 reported by Deutsch and Schiffman [5] for their comparable translator with in-line caching.

To put these averages in perspective, in a 1,240,000 byte Smalltalk-80 image there are 155,025 bytes of virtual machine bytecodes, which expand to 573,593 bytes of SOAR instructions. For current workstations with memories in the four to eight megabyte range, this increase of roughly 420,000 bytes for SOAR instructions is acceptable.

A major goal of the SOAR design was to have over 90% of the methods fit all their storage into registers. Static results from the compiler support the current window size. Only 9% of all methods spill. Examining total register requirements also supports the 16 register window. These numbers are expressed in terms of the number of high registers used, because all method-specific allocation is done in the high registers.⁶

# of high registers	methods using no more than that #
2	29%
4	65%
8	92%
16	99%
32	100%

Dynamic results from benchmarks confirm the static results, and more emphatically endorse the chosen window size. Dynamically, less than 3% of methods spill. The above figures show that if we increased the register window set size to 32 from 8, all methods in the Smalltalk-80 system would fit in register windows, and spilling could effectively be eliminated. However, since less than 3% of all calls require spilling, even if we could (somehow) keep the same number of register windows on the chip, increasing the window size from 16 to 64 registers would not be a cost effective use of chip area, given current VLSI technology.

Static results also indicate that the current spill rules are reasonable.

category	mean size	maximum size
arguments	0.83	13
local variables	0.68	19
retained temporaries	0.82	10
total registers	2.32	28
spill area	0.45	22

Argument, local variable, and temporary demands are small on the average, but in the worst case all exceed the window size.

⁵The method lengths include literals and method headers.

⁶These numbers include the 2 special receiver and return address registers. Note that the percentage of methods that spill is higher than the percentage of methods that require more than 8 registers because the registers are divided up into two separate regions which are not treated as one continuous space.

7. Conclusions

The Smalltalk-80 language and its bytecode representation restrict the conventional optimizations available to the SOAR compiler. Nevertheless, the compiler generates efficient code, primarily due to register windows, integer tags and traps, and in-line method caches. Experience with the compiler has verified the architectural design decision to use a 16-register fully overlapped window. Several features supported by the SOAR hardware could easily be performed by the compiler at marginal increased time and space costs.

8. Acknowledgments

This project was sponsored by Defense Advance Research Projects Agency (DoD) ARPA Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00034-K-0251. It was also sponsored by Defense Advance Research Projects Agency (DoD) Arpa Order No. 4871, monitored by Naval Electronic Systems Command under contract N00039-84-C-0089. Dain Samples is supported in part by an AT&T Bell Laboratories Scholarship. Glenn Krasner and Xerox PARC provided support in the development of the compiler.

References

- [1] 'The Design and Evaluation of a High Performance Smalltalk System'; David Michael Ungar; *U.C. Berkeley Computer Science Division Report UCB/CSD 86/287*, March 1986.
- [2] 'Architecture of SOAR: Smalltalk on a RISC'; David Ungar, Ricki Blau, Peter Foley, A. Dain Samples, David Patterson; 11th Annual International Symposium on Computer Architecture, Ann Arbor, Michigan, June 1984.
- [3] 'SOAR: Smalltalk Without Bytecodes'; A. Dain Samples, David Ungar, Paul Hilfinger; *CACM Object-Oriented Programming Systems, Languages and Applications Proceedings*, November 1986, pp 107-118.
- [4] 'What Price Smalltalk?'; David Ungar and David Patterson; *IEEE Computer*, January 1987, pp 67-74.
- [5] 'Efficient Implementation of the Smalltalk-80 System'; L. Peter Deutsch, Allan M. Schiffman; *11th ACM Symposium on Principles of Programming Languages*, January 1984, pp 297-302.
- [6] 'The Design of a Debugger for SOAR'; Peter K. Lee; Master's Report, U.C. Berkeley, 1984.
- [7] 'Machine Independent Pascal Code Optimization'; D.R. Perkins and R.L. Sites; *CCC SIG-PLAN* 14,8, August 1979, pp 201-207.
- [8] 'Smalltalk-80 to SOAR Code'; William R. Bush; *U.C. Berkeley Computer Science Division Report UCB/CSD 86/297*, June 1986.
- [9] 'Register Windows for SOAR'; John Blakken; *Smalltalk On A RISC -- Proceedings of CS929R*, April 1983, pp 126-140.
- [10] 'Reduced Instruction Set Computer Architectures for VLSI'; Manolis G.H. Katevenis; *U.C. Berkeley Computer Science Division Report UCB/CSD 83/141*, October 1983.

- [11] 'An MC68000-Based Smalltalk-80 System'; Richard Meyers and David Casseres; in *Smalltalk-80 Bits of History, Words of Advice*; Glenn Krasner, Editor; Addison-Wesley, 1983, pp 175-187.
- [12] 'The Analysis of the Smalltalk-80 System at Hewlett-Packard'; Joseph R. Falcone; in *Smalltalk-80 Bits of History, Words of Advice*; Glenn Krasner, Editor; Addison-Wesley, 1983, pp 207-237.
- [13] 'Implementing a Smalltalk Compiler'; Wayne Citrin, Carl Ponder; *Smalltalk On A RISC -- Proceedings of CS929R*, April 1983, pp 167-185.