A SMIDGIN OF SOAR

SOAR (Smalltalk On A RISC) was a research effort begun in 1982 as a follow on to the the RISC II work at Berkeley. SOAR was an attempt to extend RISC concepts into an area that was thought to be unsuitable for RISC, that of object oriented programming environments.

SOAR was like a RISC in that;

1) It was truly a reduced instruction set machine with only 22 instructions
2) It was a word oriented machine that did not support arbitrary shifts (although byte-oriented operations were possible using byte insert and extract instructions)
3) It had a simple Load/Store memory access architecture.
4) It had a simple 3 stage pipeline.
5) It was a Multiple Register Set (MRS) architecture utilizing overlapped register windows as in RISC I and RISC II. (It is debateable if this is a true RISC trait)

SOAR had some non-RISC traits also;

1) SOAR was a tagged architecture. The upper 4 bits of object pointers were tag bits. Operand tags were examined in real time as operands flowed through the pipeline. If the type of either operand were other than small integer, the machine would take a "tag trap". These traps were invisible to the programmer.
2) All the major machine state was shadowed in order to support the traps.
3) The instruction set include a load and store multiple to facilitate fast context switching and to reduce the cost of overflows. These were the only multicycle instructions beside Load and Store, and in retrospect were a mistake.

SOAR Software:

Direct Compilation:

Unlike most Smalltalk machines prior to SOAR that were bytecode interpreters, SOAR relied on a compiler that directly compiled Smalltalk into SOAR object code. I don't remember for sure what the code expansion factor over bytecodes was, but a figure of 3.3:1 comes to mind.

Using direct compilation rather than bytecode interpretation made supporting the debugging environment difficult.

OT less:

In the Smalltalk-80 virtual machine, all references to an object go through a level of indirection implemented as the Object Table. However SOAR is an OT less system where Object Pointers contain the direct address of the referenced object. One of the problems with this implementation is that the primitive *become* becomes an expensive operation (*become* swaps all the instance pointers of its receiver and its argument) as in principle one now has to look through the entire system for pointers to the two objects. All five *grow* methods use *become*.

A solution proposed by David Wallace was to move all of the variable fields out of growable objects into arrays (this eliminates 95%) and to implement a technique called "Lazy Become". Lazy Become is a good example of the interdisciplinary nature of the SOAR system design. Lazy Become leveraged the virtual memory system. Only objects resident in pages currently in core are fixed up. When a new page is read in, it would be scanned for all references to objects that had been the subjecs of a become, and such references fixed up. Then overnight, the system would scan through all currently invalid memory pages looking for objects to fix up. Of course Lazy Become would have to continue to operate in the debugging environment.

Method Cache:

In order to boost performance, SOAR supported self modifying code in the form of an "in-line method cache". What that basically amounted to was that the pointer to the object that actually executed the message the last time the message was sent was stored in the sending code. This helped to minimize class traversal that might occur each time a message looked for an implementer. It is very similar to a MMU TLB where instead of minimizing page table traversal class hierarchy traversal is minimized, and instead of being physically located in the TLB, the pointers are distributed in memory with the sending code. Simulations indicated that method caching had a 95% "hit" rate. (The idea was originally suggested by Peter Deutsch)

The Method cache plus overlapping register windows significantly reduced the call/return overhead in Smalltalk (which consumed nearly 50% of the Dorado, and 40% of Apple Smalltalk)

I don't recall ever having any serious discussions as to how method caching might complicate cache memory design of future versions of the machine. Especially on board cache memory.

Generation Scavenging

SOAR was the first Smalltalk implementation to support generation scavenging for automatic storage reclamation. Up until SOAR, reference counting was the standard technique for managing memory but it used a significant amount of the CPU (11% of the Dorado I think). On average, 12 words of data are freed and must be reclaimed per 100 Smalltalk-80 virtual machine bytecodes executed.

Generation Scavenging is based on the observation that objects either die young or live forever. Objects are "placed" into four different generations based on age and only new objects are reclaimed. If a pointer to a new object is stored into a memory location within an old object, the SOAR hardware would, by examining the generation tags of the object pointers, take a trap.

Virtual Memory

SOAR supported existing virtual memory hardware through the technique of "offline reorganization". Since pages dwarf objects, offline (overnight) utility programs would be used to group objects frequently used together into pages.

Traditional Languages

Unlike bytecode interpretive hardware, SOAR could execute compiled C or PASCAL code simply by setting a bit in the instruction opcode field which disabled tag checking.

REGISTER WINDOWS

The window size was less than in the RISC machines although there were more of them. This was due to the following ovservations;

a) Method sends tend to be more deeply nested, resulting in many "procedure calls"
b) Because operations in Smalltalk are performed by sends, any data manipulation can be turned into a procedure call. Hence no local registers are included in SOAR register windows.
c) Smalltalk methods use few temporary variables

A window size of 8 registers was found to hold all the arguments, temporaries, and fixed size elements needed by 93.4% of method contexts (activation records).

A register file containing 8 windows was found to be adequate.

THE COMPILER

The compiler was a 3 pass affair. Pass 1 generated 3 address intermediate code, Pass 2 did some simple peephole optimization, and Pass 3 was a register allocation pass.

An interesting aspect of the compiler was that the symbol table functioned as both a compile-time and run-time data structure. As a compile-time structure, if kept track of method temporaries and parameters, block parameters, class instance specifications, and the class hierarchy with asociated methods and metaclasses. As a run-time structure, it was used by the method lookup routines in order to find the proper method to which a message should be sent.

An interesting idea for an efficient Smalltalk compiler that was intended to augment the standard compiler was called Classy by its originators, Jim Larus and Will Bush. The idea was to take as input Smalltalk methods where the classes of the method's variables and arguments are all declared. These methods are then transformed, with the assistance of a library of transformations, called Rewrites, into conventional Smalltalk that has sends eliminated and class-specific operations put in place of more general ones.

Classy seems like a good way of improving kernel performance as long as the transformed methods can be considered correct and atomic (debugging them would pose some problems).

THE DEBUGGER

The debugger for SOAR proved to be a challenge for a number of reasons.

In Smalltalk-80 the virtual machine hides the underlying machine from the system, hence interrupts can be handled by having the interpreter poll at every bytecode (which is what Smalltalk-80 does). For SOAR however, before invoking the debugger it is necessary to recognize whether the CPU is busy executing primitives, runtime support, or actual Smalltalk code. One idea is to have a companion processor handle user interrupts and let the SOAR CPU simply poll a flag via a transparent trap at backward branches.

Another more serious difficulty is maintaining the context chain. In Smalltalk-80, block contexts are all objects that are linked in a oneway list by sender pointers. In SOAR, most contexts are not treated as objects. This would be ok if contexts always exhibited LIFO behavior as they could simply be placed on a stack. But Smalltalk contexts do not always exhibit LIFO behavior.

Access to a context's local variables also has to be treated differently. In Smalltalk-80, local variables are full fledged objects so that inspectors can be created for them. In SOAR however, local variables may be used only as temporaries within the method and, therefore, may show up only in a register within the context window frame.

Functions such as informing the user which routine is running, and single stepping also have to be specially handled in SOAR.

SOAR PERFORMANCE

With a cycle time of 550ns (8 times slower than the Dorado) some very questionable "micro benchmarks" showed performance of from .5 Dorado to over 5x a Dorado.

SOAR CPU DESIGN

While SOAR may have a minimal number of instructions, the tag checking, register shadowing, and trapping hardware proved to be nontrivial. All of the possible system exceptions and traps had to be prioritized which added to the fairly sizeable control PLA's. Control signals necessary for trap determination also led to some critical path problems.

Load and Store multiple proved to be a mistake in that their complexity and unusual nature did not warrant the design complexity increase (although the silicon area increase was actually quite small). In my first pass of the SOAR microarchitecture I froze the pipeline while the load or store multiple occurred, Joan Pendleton subsequently changed this to have special NOP instructons flow through the pipe, which is more natural. Load and Store multiplies had to be reexecuted if a system exception (such as a page fault) occured during their execution.

SOAR HARDWARE STATUS

A 35,000 transistor 3 micron NMOS implementation of SOAR was fabricated. The chip ran some diagnostics but that was about as far as it was taken. A board was designed to surround the SOAR chip and sit in a SUN on the multibus. The board was fabricated, but I do not know if it was ever fully debugged.

IN RETROSPECT

I believe that the SOAR approach to Smalltalk is still a viable one and could achieve significantly better performance than any implementation existing today. SOAR was never fully demonstrated in a decent technology (say 1.5 micron CMOS @ 15 MHZ). The problem with SOARs lack of "success" had a number of causes:

a) Many of the people working on the project had no deep understanding of what Smalltalk was or how it worked. There was no interactively useable Smalltalk engine at Berkeley during the entire SOAR project.
b) Doing a direct compiled RISC based implementation of Smalltalk is *hard* (see below).
c) Smalltalk never has caught outside of academia on so there was little technology transfer or enthusiasm from industry. This in turn negatively effected the level of continuing work at Berkeley.

By *hard*, I mean SOAR was a very large long term progect to be undertaken by the Berkeley CS and EE departments. The design of SOAR ,as compared to a RISC I or RISC II, meant the design of a *system*, a complete *programming environment.* In SOAR the distincion between hardware and software became even more fuzzy, and thus the design effort became very multidisciplinary involving a large number of faculty and graduate students over a long period of time. Much learning was done on how to manage a large program that significantly exceeded the lifetime of the average Masters student. Maintaining continuity and reasonably scheduling tasks became significant challenges. The following disciplines were involved, and input from all of them could affect design in any other area;

1) silcon design
2) debugger design
3) virtual memory design
4) performance analysis
5) compiler design
6) board level design
7) instruction set design
8) silicon and system simulation

Implementers of Smalltalk continue to figure out clever tricks for implementing Smalltalk on standard architectures. One can argue that clever tricks coupled with generally increasing horsepower obviates the need for an architecture like SOAR. But it does seem that much effort is focused on how to get Smalltalk to execute efficiently on existing hardware platforms rather than on improving the language itself.

SOURCES:

1) Proceedings of CS292R, "SMALLTALK ON A RISC", Architectural Investigatons April, 1983. (Approximately 18 authors)
2) The Architecture of SOAR: Smalltalk on a RISC
David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson
3) Preliminary SOAR Architecture Results of CS292R, Winter 1983
Mike Klein, Pete Foley, and Dain Samples

Pete Foley
6/10/88