Fast Breakpoints: Design and Implementation

Peter B. Kessler
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

We have designed and implemented a fast breakpoint facility. Breakpoints are usually thought of as a feature of an interactive debugger, in which case the breakpoints need not be particularly fast. In our environment breakpoints are often used for non-interactive information gathering; for example, procedure call count and statement execution count profiling [Swinehart, et al.]. When used non-interactively, breakpoints should be as fast as possible, so as to perturb the execution of the program as little as possible. Even in interactive debuggers, a conditional breakpoint facility would benefit from breakpoints that could transfer to the evaluation of the condition rapidly, and continue expeditiously if the condition were not satisfied. Such conditional breakpoints could be used to check assertions, *etc.* Program advising could also make use of fast breakpoints [Teitelman]. Examples of advising include tracing, timing, and even animation, all of which should be part of an advanced programming environment.

We have ported the Cedar environment from a machine with microcode support for breakpoints [Lampson and Pier] to commercial platforms running C code [Atkinson, et al.]. Most of our ports run under the Unix* operating system, so one choice for implementing breakpoints for Cedar was to use the breakpoint facility provided by that system. The breakpoints provided by the Unix operating system are several orders of magnitude too slow (and also several process switches too complicated) for the applications we have in mind. So we designed a breakpoint system that was fast enough for our purposes.

Breakpoints for uni-processors running single threads of control used to be fast and simple to implement. This paper shows that breakpoints can still be fast, even with multiple threads of control on multi-processors. This paper describes problems in the design of a breakpoint package for modern computer architectures and programming styles, and our solutions to them for a particular architecture.

Previous Work

The original inspiration for the breakpoints presented here is the primeval method of patching code to alter its semantics [Gill]. This technique was apparently completely natural for programmers who were programming in assembler (or machine) language, and needed to modify their programs without re-assembling or relocating a lot of their code. User inserted breakpoints became a part of debugging systems early on [Stockham and Dennis]. Breakpoints in one form or another have been available in programming environments since that time [Evans and Darley] [Johnson].

Breakpoints are implemented by assembling the replacement code at some convenient place in memory and planting a branch to the replacement code in place of the original code. The replacement code can perform its operations and then either return to the main instruction stream after the breakpoint, or may completely alter the program flow. For example, an interactive debugger might be invoked from the breakpoint code to allow program state to be examined or altered. Breakpoints can also be used as assembler program editors in synchrony with high-level language editors to keep a running program up to date with source modifications [Fabry].

At some point, self-modifying code became taboo, and breakpoints came to be implemented by traps to the operating system, that would then transfer control to a

* Unix is a trademark of AT&T Bell Laboratories.

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

debugger [Klensk and Larson]. Traps to the operating system are unnecessarily expensive in our view, and we have tried to avoid them wherever possible.

In many ways, this work is similar in scope to the "scan-point" work in Parasight [Aral and Gertner]. That work focused on breakpoints in a multiprocessor setting. A main point for them was the off-loading of the code executed by the breakpoint to another processor in their multiprocessor. In contrast, though we are running a system with multiple light-weight threads in each address space, our concern is for speed and simplicity in the breakpoint mechanism: starting up a separate thread to execute the breakpoint code seemed like too much mechanism. If the code for the breakpoint wants to start up a separate thread, it can easily do that. For many applications of fast breakpoints (*e.g.*, counting procedure entries, *etc.*), the work of the breakpoint code is less than the work of forking a separate thread. For some applications, such as checking assertions, the breakpoint code should run in the same thread, so it can raise an error synchronously with respect to that thread.

This work should be contrasted with breakpoints that contact an interactive debugger, like the breakpoints in dbx [Sun Debugging]. In dbx, hitting a breakpoint causes a trap to the operating system, which then schedules the debugger to run. Examining the state of the thread that hit the breakpoint from the debugger goes through the operating system, and resuming execution of the thread is another context switch. In contrast, the breakpoints described here do not switch threads to execute the code in the breakpoint and have full access to the state of the computation that hit the breakpoint. Hardware assisted breakpoints, which have the advantage of being able to trap on memory accesses as well as instruction execution, also usually transfer to handlers inside the operating system [Pappas and Murray].

Discussion

What we wanted was fast breakpoints, with little or no intervention required from the operating system. We wanted to be able to plant breakpoints in code from any of our translators, without any special preparation of the programs. The code to be executed in the breakpoint is ordinarily already available, either supplied by the programmer [Knuth], or referenced from a library of such routines (*e.g.*, profiling tools, assertion checkers, *etc.*). The purpose of the breakpoint is to transfer control to this code. The interface we provide maintains a registry of closures (each of which is a procedure and some private data), to be called when execution of a thread reaches a particular instruction. At the lowest levels, the breakpoint address is specified by a program counter value, though at higher levels one can request breakpoints at source locations, procedure entry points, *etc.*

Note that we are interested in breakpoints that execute quickly once planted. The speed with which we can plant (and clear) breakpoints is a secondary consideration. If our only use of breakpoints were to contact an interactive debugger, speed would not be an issue. Instead, we use breakpoints to augment the normal processing of a program, and so would like breakpoints that execute as quickly as possible.

The basic technique for planting a breakpoint is to assemble the code to call the registered closure and then replace the instruction at the breakpoint address with a branch to the newly assembled breakpoint code. But it is not that easy. Since we are invoking a closure, we must save any global state accessible to the closure before invocation, and restore that state afterwards. Since the code necessary to invoke the closure safely is larger than the instruction at the breakpoint address, we assemble the code elsewhere in memory and use a sufficiently small branch instruction to transfer to the breakpoint code. Since we are replacing an instruction at the breakpoint address, we must arrange for a copy of that instruction to be executed after the call to the closure. Relocating the instruction from the breakpoint address may involve some transformation of the instruction, *e.g.*, if it is a pc-relative branch, or if it references pc-relative data. After the execution of the displaced instruction control must be returned to the instruction stream following the breakpoint address. Figure 1 shows the instruction stream and the breakpoint code when the breakpoint has been planted. By operating at the machine language level we can plant breakpoints in code produced from any source. The specifics of the breakpoint (the closure and the breakpoint address) are "compiled into" the breakpoint code, so there is no need for auxiliary data structures.

First we discuss some restrictions on when this technique is applicable. These conditions are the basic requirements of the breakpoint package on the underlying runtime system. Next, we discuss how modern programming languages and computer architectures have complicated the task of producing a breakpoint package. These difficulties are addressed in general, and then the implementation of our breakpoint package for a particular popular architecture is given.

Restrictions and Complications

Instruction memory must be writable. Since the basic breakpoint operation is to replace an instruction at the breakpoint address, we must be able to write instruction

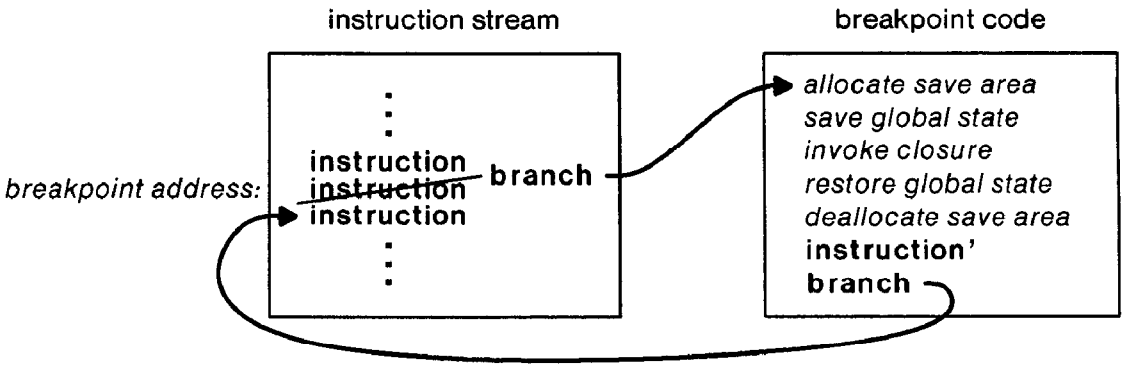

Figure 1. General form of a fast breakpoint, in which code is assembled out of line to invoke a closure, and the instruction at the breakpoint address is replaced by a branch to the auxiliary code.

memory. Since we are setting breakpoints in a multi-processing environment, writing the new instruction must be atomic, at least with respect to the other processes executing at that address. We must also allocate some writable instruction space in which to assemble the breakpoint code. The space for the breakpoint code must be within the branch distance of the instruction written to the breakpoint address. These restrictions might be a source of trouble in some runtime systems or architectures.

The branch planted at the breakpoint address may not be the same size as the instruction it is replacing. If the branch is smaller than the instruction it is replacing, no-op's may be inserted along with the branch. If the branch is larger than the instruction at the breakpoint address, one might have to move the instruction at the breakpoint address and some of the following instructions to have room for the branch instruction. One might not be able to replace a shorter instruction, *e.g.*, if control transfers to any of the other instructions that need to be moved (or if one cannot determine that no such transfers occur). The branch instruction might violate some scheduling constraints that were ensured by the compiler. Any of these problems would disallow the planting of a breakpoint at that address.

The breakpoint code must include code to save and restore some amount of global state around the invocation of the closure to ensure that the displaced instruction executes with the registers, *etc.*, in the same state they would be in if the instruction were not displaced by the breakpoint. Saving state involves allocating space to hold the state and restoring state includes deallocating that space, so the breakpoint code must be able to allocate data space.

Many modern machine architectures do not transfer control directly on executing a branch instruction but delay the branch, executing the instruction after the branch before the transfer takes effect [Sun SPARC] [Kane]. The instruction after the branch is said to be in the "delay slot" of the branch. Since we wish to replace a single instruction with a branch, but not execute the instruction which would be the delay slot of that branch, this would be a problem on such architectures. It might be possible to replace the instruction after the inserted branch with a no-op to avoid the problem of the delayed branch. However, that solution opens up the problem, discussed above, of atomically writing the replacement instructions, and also the problem of transfers to the instruction after the breakpoint address, as in the case of variable-sized branch instructions discussed above.

One problem with branching to the breakpoint code is that the branch instructions may not have sufficient range to transfer control from the breakpoint address to the breakpoint code. Similarly, if the instruction at the breakpoint address has pc-relative operands, it may not be possible to relocate the instruction into the breakpoint code.

Once a breakpoint has served its purpose, it must be removed. Disabling a breakpoint is easy. The branch instruction that was planted at the breakpoint address is replaced by the instruction that was displaced to plant the branch. The same atomicity requirements apply to the replacement as apply for planting the breakpoint in the first place. Thereafter, no new threads of control will branch to the breakpoint code. Then there is the problem of freeing, or reusing, the space occupied by the breakpoint code.

In a uni-processing environment, when the branch is removed the space occupied by the breakpoint code can be freed, because the program counter is obviously not in the breakpoint code. In a multi-processing environment we have a garbage collection problem. Since many threads may be executing in the breakpoint code

(or in the closure that is invoked from it) when we replace the instruction at the breakpoint address we cannot free the space occupied by the breakpoint code. We need to trace all the threads to see if any of them has a frame that is executing in, or expects to return to, the breakpoint code. This is somewhat simpler than the general garbage collection problem, since all we must find are the program counters and return addresses in the stacks of the threads. It should be possible to present this problem to a general garbage collector [Demers, et al.], though we have not done that yet. Note that reference counting the breakpoint code space is not possible, since program counters do not maintain reference counts on the code they are causing to be executed. We could have the first few instructions of the breakpoint code increment a counter, and have the last few instructions decrement the counter, but there would still be a window of several instructions before the counter was incremented, and after the counter was decremented, where the reference count would be low. Also there is a problem in that changes to the counter have to be monitored, which might require even more instructions in the breakpoint code, and might increase the window in which the counts would be low.

Details of the implementation for the SPARC

Below we describe the details of an implementation of a fast breakpoint package for the SPARC* architecture. We have implemented breakpoints in a multi-threaded, shared address space, programming environment with dynamically loaded program modules. However, we could repackage our system to be a C library, with breakpoint setting and clearing invoked either by the program itself (*e.g.*, for performance monitoring), or via an interactive debugger (*e.g.*, for assertion checking or for conditional breakpoints).

We do not have trouble replacing the instruction at the breakpoint address. Since, on the SPARC, all instructions are word-sized, we never have any problem with branch instructions being larger than the instruction they are replacing. All instructions are word-aligned, and word-sized word-aligned writes are atomic, so the atomicity constraint was a non-problem. The Cedar programming environment dynamically loads code and leaves that code writable. Branch distances and relocation of displaced instructions are not problems for us since the SPARC instructions have generous displacement fields and our dynamic loader leaves convenient holes between dynamically loaded modules. Other runtime environments might not be so generous. If the loader did not leave holes between modules, we could instead request space from the storage allocator at convenient address intervals. (We do such allocations anyway, since the loader does not leave large enough holes for some breakpoint applications.) Most C programs occupy a small enough amount of code space that heap-allocated storage would be within reach of the branches available in the SPARC instruction set.

The SPARC architecture has an unfortunate amount of global state to save. (This amount of global state has little or nothing to do with the overlapping register windows. We are not saving any non-local register windows. They are not accessible to the displaced instruction, nor during the execution of the registered closure, so they need not be saved or restored.) In the worst case, we have to save the window of registers visible to the displaced instruction, the global registers, the floating point registers, and a small collection of status registers (the condition codes, the Y register, and the floating point status register). It seems reasonable to allow a breakpoint closure to declare (at registration time) that is does not alter certain parts of the global state of the machine (*e.g.*, the floating point registers). If it is known that a closure does not affect some part of the global state, the breakpoint code can be assembled to avoid saving and restoring that state. Such declarations make a substantial difference in the time to execute the breakpoint. Similarly, if a breakpoint does not require a full closure call, that can be arranged when the breakpoint is planted, for an additional savings during breakpoint execution.

Saving all the state on a SPARC takes a large number of instructions. For this reason, we choose not to inline expand the saves and restores, but rather encapsulate them in procedures. (This is possible in our environment, since we have control over the runtime libraries. An alternative would be to construct these procedures in memory as the first breakpoint was planted.) However, calling a procedure damages part of the global state (the return address register in particular). Before we can call the state-saving procedure, we have to allocate space to save the state. These reasons, and the desire to avoid saving and restoring the in and local registers (as seen by the displaced instruction), recommend the SPARC **save** and **restore** instructions. The **save** instruction acquires a new register window and allocates a memory stack frame. The **restore** instruction releases the register window and deallocates the memory stack frame. (The extra frame has a deleterious effect on programs that examine stack traces, *e.g.*, debuggers and profilers, since during the execution of the breakpoint the supplementary register window and memory stack frame have to be dealt with. To avoid this difficulty, the breakpoint code can set up

* SPARC is a trademark of Sun Microsystems, Inc.

```
save    %sp, -sizeof(saveArea), %sp    -- push window and allocate save area.
call    saveRegisters                  -- saveRegisters(&saveArea.registers)
add     %sp, registersOffset, %o0
sethi   %hi(closureData), %o0          -- closureProc(closureData)
call    closureProc
or      %o0, %lo(closureData), %o0
call    restoreRegisters               -- restoreRegisters(&saveArea.registers)
add     %sp, registersOffset, %o0
restore %sp, +sizeof(saveArea), %sp    -- pop window and deallocate save area.
instruction'                           -- displaced (and relocated) instruction.
ba,a    breakpointAddress + 4          -- return to instruction stream.
```

Figure 2. Patch for an ordinary SPARC instruction.

the return address register of the new window to point back to the breakpoint address, so the frame holding the global state looks as if it were called from the breakpoint address. Saving the global state in an ordinary memory frame also makes the saved state available to debuggers during the execution of the breakpoint.) Allocating a new frame also avoids some trouble with transforming a leaf procedure (*i.e.*, one that does not call any procedures) into a procedure that does further calls (*e.g.*, to the register save and restore procedures, and to the closure). Note that on the SPARC the procedure call and return instructions are sufficiently fast that calling procedures from the breakpoint code is not a major part of the cost of executing a breakpoint. The code for a SPARC breakpoint code is shown in Figure 2 (except for the setting up of the return address register for the debugger). Each breakpoint address has its own corresponding breakpoint code.

We are expecting to develop several applications that use self-modifying code with this style of assembling patches to the main instruction stream. Therefore, it is desirable that the patches identify what address they are patching. Patches are identified simply by prefixing the breakpoint code with a word containing the breakpoint address. It is important to keep multiple breakpoints (or other patches) from affecting the same instruction address, because if the breakpoints were cleared other than in a last-in first-out manner, the original intent of the program would be lost.

All of the transfers of control on the SPARC are delayed, so it would appear that we have a problem with the instruction in the delay slot of the inserted branch to the breakpoint code. However, the SPARC conditional branch instructions include an "annul" bit that suppresses the effect of the delay slot instruction. With the annul bit set, the branch-always instruction **(ba,a)** suppresses the execution of the instruction in the delay slot. Thus, it is the perfect instruction for the branch to the breakpoint code.

For instruction sets with only delayed branch instruction, we have investigated the use of trap instructions to transfer control through the operating system. A sufficiently flexible operating system would allow us to register a handler for a particular trap. There are several disadvantages to using traps, not the least of which is that handling the trap through the operating system will invariably take more time than a simple transfer within a thread. In the instruction sets we have examined, the trap instructions cannot carry enough information to identify directly the breakpoint but will instead involve some lookup of the breakpoint code address, which will further delay the execution of the breakpoint.

The SPARC instruction set does offer some other problems generic to delayed transfers. Consider the problem of setting a breakpoint on a delayed control transfer instruction. We can use the usual branch to get to the breakpoint code. When the displaced instruction executes, the instruction from its delay slot must also be in the displaced delay slot in the breakpoint code. This is easily arranged, if the instruction can be relocated to the breakpoint code. Now consider the problem of setting a breakpoint on an instruction in a delay slot. It would appear that we could just put the branch in the delay slot, except that the SPARC architecture manual says that a delayed branch may not appear in the delay slot of a delayed conditional branch. We had designed an approach that essentially plants two breakpoints: the first is on the conditional branch, thus transforming it to an unconditional branch; the second is for the breakpoint we wanted to set originally, which is now legal since it is not in the delay slot of a conditional branch. That technique works on a uni-processor system, but on a multi-processor there is a race where some of the processors see the illegal instruction sequence in which the conditional branch has a branch in its delay slot. In the present system we do not allow breakpoints to be planted in delay slots.

Machine	Package	Microseconds
SPARCstation1	dbx	24300
Sun 4/280	dbx	23300
Dorado	Celtics	33.0
SPARCstation1	Shepherd	24.4
Sun 4/280	Shepherd	17.4
SPARCstation1	Shepherd	11.2*
Sun 4/280	Shepherd	9.7*

* not saving floating point registers

Table 1. A comparison of breakpoint times.

A special case of instructions with delay slots are call instructions. If we used our usual technique for breakpoints on delayed branches, the call instruction (and thus the return address) would be in the breakpoint code. For several reasons we want the return address to be in the real instruction stream not the breakpoint code. For example, profilers sometimes build dynamic call graphs by tracing return addresses [Graham, et al.]. Some compilers also encode information about the call (*e.g.*, the number or address of expected return arguments) at the return address. These are compiler and runtime designs that we cannot hope to anticipate. Our solution is to use a call instruction instead of a branch instruction to get to the breakpoint code. The call instruction correctly sets the return address. Then, instead of executing the displaced call from the breakpoint code we simply branch to the first instruction of the called procedure. There are similar wrinkles for indirect calls and indirect returns (where the transfer address is in a register, as opposed to being a constant in the instruction stream). Each of these cases is distinguished as the breakpoint is planted, and distinct breakpoint code is generated for each case.

Results and Applications

Our results are encouraging. We implement breakpoints with only a few dozen instructions. These breakpoints are 1000 times faster than the breakpoints available in the conventional Unix debuggers [Sun Debugging]. Most of the time outside of the invoked closure is spent saving and restoring global state. We have tried the experiment of declaring breakpoint closures that do not affect parts of the global state, and have been able to reduce the save and restore times by more than one half. Further improvements are possible, but the returns are diminishing. Table 1 compares our technique (identified as "Shepherd" because it uses the **ba,a** instruction) with breakpoint times on several machines and breakpoint packages. We note that fast breakpoints have been in use in Cedar on Dorados (3 MIPS personal workstations) for several years and are fast enough for our applications. We have given up microcode support by moving off Dorados and yet the relative speed of breakpoints to processor speed has remained relatively constant. So we have reason to believe that our breakpoints will be fast enough on commercially available processors.

We have already used this breakpoint package as the basis for several different debugging tools, and feel that the mechanism is suitable for several other applications. Other members of our lab have used the breakpoint facility to contact our interactive debugger for ordinary program debugging. Another application of fast breakpoints implements data breakpoints: *e.g.*, by planting a breakpoint on every store instruction in a procedure and checking which of them writes to a particular memory location. (Hardware assistance might be a better solution for this application [Pappas and Murray].) We have also discussed using breakpoints to transfer control to a processor simulator, either for debugging or to experiment with changes to the semantics of a machine operation. This breakpoint facility is fast enough to use for procedure call count profiling (or statement execution count profiling) without the need to recompile code. Algorithm animation, again without the need to recompile code, could make use of these breakpoints. These are examples of procedure (or statement, or instruction) advising, that, along with the traditional uses of advised procedures, could be implemented using the techniques described in this paper.

Future work includes building novel applications on top of the breakpoint facility, and implementing breakpoints on a variety of instruction sets.

Acknowledgments

I thank my colleagues in the Xerox PARC Computer Science Laboratory who contributed ideas to the breakpoint package presented here and who also built the rest of the programming environment in which it runs. Thanks especially to Andy Litman, who implemented the data breakpoints based on these ideas while I was putting the rest of the package together. Thanks also to the program committee for their many good suggestions for improving the presentation of the paper.

References

[Aral and Gertner]
Z. Aral, and I. Gertner, "High-Level Debugging in Parasight", in Proceedings of the *Workshop on Parallel and Distributed Debugging*, SIGPLAN Notices Volume 24, Number 1, January 1989.

[Atkinson, et al.]
R. Atkinson, A. Demers, C. Hauser, C. Jacobi, P. Kessler, M. Weiser, "Experiences Creating a Portable Cedar", in *Proceedings of the SIGPLAN'89 Conference on Programming Design and Implementation*, SIGPLAN Notices Vol. 24, No. 7, July 1989.

[Demers, et al.]
A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, S. Shenker, "Combining Generational and Conservative Garbage Collection: Framework and Implementations", in *Proceedings of the 17th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1990.

[Evans and Darley]
T. G. Evans and D. L. Darley, "On-line Debugging Techniques: A Survey", in *Proceedings of the Fall Joint Computer Conference*, 1966.

[Fabry]
R. S. Fabry, "MADBUG - A MAD Debugging System", in *The Compatible Time-Sharing System. A Programmer's Guide, Second Edition*, MIT Press, Cambridge MA, 1965.

[Gill]
S. Gill, "The Diagnosis of Mistakes in Programmes on the EDSAC", in *Proceedings of the Royal Society*, Series A, Vol. 206, pp. 538-554.

[Graham, et al.]
S. Graham, P. Kessler, M. McKusick, "gprof: A Call Graph Execution Profiler", in *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, SIGPLAN Notices Vol. 17, No. 2, June 1982.

[Johnson]
M. S. Johnson, *An Annotated Software Debugging Bibliography*, Hewlett-Packard Laboratories, March 1982.

[Kane]
G. Kane, *MIPS RISC Architecture*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1988.

[Klensk and Larson]
T. J. Klensk and L. E. Larson, *IBM Technical Disclosure Bulletin*, Vol. 15, No. 4, pp 1248-50, September 1972.

[Knuth]
D. E. Knuth, *The Art of Computer Programming*, Vol. 1, p. 189, Addison-Wesley, 1968.

[Lampson and Pier]
B. Lampson and K. Pier, "A Processor for High-Performance Personal Computer", in *SIGARCH/IEEE Proceedings of the 7th Symposium on Computer Architecture*, La Baule, May 1980.

[Pappas and Murray]
C. H. Pappas and W. H. Murray III, *80386 Microprocessor Handbook*, Osborne McGraw-Hill, Berkeley, CA, 1988.

[Stockham and Dennis]
T. G. Stockham and J. B. Dennis, "FLIT - Flexowriter Interrogation Tape: A Symbolic Utility Program for the TX-0", Memo 5001-23, Department of Electrical Engineering, MIT, July 1960.

[Sun Debugging]
Sun Microsystems, Inc., *Debugging Tools*, Part No. 800-1775-10, May 1988.

[Sun SPARC]
Sun Microsystems, Inc., *The SPARC Architecture Manual*, Part No. 800-1399-03, June 1987.

[Swinehart, et al.]
D. Swinehart, P. Zellweger, R. Beach, R. Hagmann, "A Structural View of the Cedar Programming Environment", *Transactions on Programming Languages and Systems*, Vol. 8, No. 4, October 1986.

[Teitelman]
W. Teitelman, *Interlisp Reference Manual*, Xerox Corporation, Palo Alto, CA, Oct. 1978.