# Poor Man's Watchpoints

Max Copperman
Jeff Thomas

UCSC-CRL-94-17
April 26, 1994

Max Copperman
Board of Studies in Computer and Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064

Jeff Thomas
Kubota Pacific Computer, Inc.
2630 Walsh Avenue
Santa Clara, CA 95051-0905

This work largely supported by Kubota Pacific Computer, Inc.

## ABSTRACT

Bugs that result from corruption of program data can be very difficult to track down without specialized help from a debugger. If the debugger cannot help the user find the point at which data gets corrupted, the user may have a long iterative debugging task. If the debugger is able to stop execution of the program at the point where data gets corrupted, as with watchpoints (also known as data breakpoints), it may be a very simple task to find a data corruption bug. In this paper, we discuss a method of implementing watchpoints on a system without hardware watchpoint support. By instrumenting the program code to check memory accesses, and supplying an interface to the instrumentation in the debugger, we provide an efficient, general method of implementing watchpoints.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging — *debugging aids*; D.2.6 [Software Engineering]: Programming Environments; D.3.4 [Programming Languages]: Processors — *code generation, compilers*
General Terms: Algorithms, Languages
Additional Keywords and Phrases: debugging, watchpoints, post-loaders, instrumentation

# Introduction

There is a class of bugs that can be very time-consuming and frustrating to find: when an unidentified pointer has an incorrect value, and an assignment through that pointer modifies an arbitrary memory location. The difficulty stems from the fact that the symptom occurs at a subsequent read of that memory location, which may be arbitrarily far from the assignment that wrote to that location. Watchpoints aid considerably in finding bugs in this class, because watchpoints allow the user to get control at the point that the location is written to, that is, at the assignment through the pointer, allowing the user to identify the errant pointer. Watchpoints are also valuable because they enable a data-oriented debugging strategy in addition to a control-flow-oriented strategy. With conditional watchpoints, it is simple to arrange to break when a memory location is set to a particular value, without prior concern for which source statements are responsible for setting that location.

The most efficient implementation of watchpoints requires hardware support. It is becoming more common for machines to provide hardware support for watchpoints, but it is by no means universal; even otherwise state-of-the-art CPU's may be designed without watchpoint support. While they are efficient, hardware-supported watchpoints are not general, typically limiting the user to watching a few words of memory. This article describes an implementation of watchpoints in software that is both general and efficient enough to be useful.

Early implementations of watchpoints without hardware support were extremely slow because they required single-stepping the program and giving the debugger control after each instruction, or after each load or store. More recently there have been implementations in software that are reasonably efficient. Watchpoints can be implemented in software via

- virtual memory—protecting pages containing watched addresses,
- trap patching—replacing each store and/or load instructions with a trap instruction, and installing a trap handler that gives control to the debugger if the accessed location is being watched, and emulates the instruction otherwise,
- code patching—replacing each store and/or load instructions with an inline check or call to a function that gives control to the debugger if the accessed location is being watched, and subsequently executes or emulates the instruction.

The range of implementations is discussed in Wahbe [Wah92], who compared simulations of each type of watchpoint implementation. According to his simulations, code patching is the most efficient method of implementing watchpoints in software.

We implemented watchpoints via code patching on the Titan, a Mips R3000-based multiprocessor machine sold by Kubota Pacific Computer, Inc. (KPC). The R3000 is a RISC chip with a branch delay slot and a load delay slot, but no other characteristics that impact our watchpoint implementation.

## Design

We chose code patching because of its expected efficiency and because we could implement it quickly without support from the operating system or the compiler. In adding watchpoint support to KPC's debugger, we had a number of requirements:

- It must be possible to watch an arbitrary number of memory locations at any time.
- It must be possible to watch a contiguous range of locations without specifiying each location individually.
- The debugger user must be able to choose the addresses to watch without significant delay. This implies that the patch code must be independent of the address(es) being watched.
- The patched program must not be too large to run.
- The patched program must not be so slow that no-one would use the facility.
- The patched program must be as nearly as fast as the original when no addresses are being watched.

These requirements were met with the following design:

- There is a table of watched locations.
- The table consists of low/high address pairs, giving a set of contiguous ranges.
- The table is maintained by the debugger, and copied into the debuggee each time the debuggee is executed under debugger control.
- A subroutine `_do_watch` to test an accessed address against each range in the table is included in the program initialization module `crt0`, so that there is a single copy of this code.
- A call to `_do_watch` is patched into the executable prior to each memory access.
- A branch around the patched-in call is taken when no addresses are being watched.

This design is general, and results in programs that are within acceptable size and speed increases over the original.

## The Debuggee

A post-loader is a program that inserts instrumentation code into executables. A post-loader toolkit that simplifies the building of post-loaders was available at KPC, and simple post-loaders had been built to modify executables for performance measurements, simple optimizations, and to finesse simple portability issues. With this toolkit, it was a simple matter to build a special-purpose post-loader to insert a patch prior to each watchpoint patch target, that is, each store and/or load in the debuggee.[1]

[1] A flag passed to the post-loader designates loads, stores, or both as patch targets.

```
        beq     $fp, $0, target   ; branch around the patch if
                                  ;   no register save area
        nop                       ; branch delay
        sw      $ra, 0($fp)       ; save return address register
        sw      $a0, 4($fp)       ; save a register in which to pass
                                  ;   the address being accessed,
        jal     _do_watch         ; jump to _do_watch
        addiu   $a0, $sp, 68      ; compute the address being accessed
                                  ;   (dependent on the patch target)
        lw      $a0, 4($fp)       ; restore register
        lw      $ra, 0($fp)       ; restore return address register
        nop                       ; load delay
target: sw      $ra, 68($sp)      ; patch target (from whence the
                                  ;   address being accessed is computed)
```

Figure 0.1: The patch as applied to the first store in a routine's prologue (which saves the return address). The original store is the last instruction shown.

To keep the patch small, we put as much work as possible into `_do_watch`. The patch minimally had to load a register with the address of the memory location accessed by the patch target and call `_do_watch`. To do this without affecting the semantics of the original program, the previous contents of that register and the return-address register must be saved and restored.

Within `_do_watch`, more registers are used. A general mechanism to save registers was needed. On the Titan, the native compiler sets aside one register, `$fp`, for use by tools that need a scratch register (such as the profiler). When the user sets a watchpoint, the debugger sets `$fp` to point to a register save area in the debuggee's static data space, which allows us to save and restore as many registers as we need.[2] When no watchpoints are set, we don't want to suffer the overhead introduced by the inserted code. We therefore made the first instruction in the patch branch around the rest of the patch if `$fp` contains zero. In our implementation, including branch and load delays, each patch takes nine instructions. The patch code is shown in Figure 0.1. The patch code could be smaller in some circumstances, but that would require more complex analysis of the code. Using a simple, general patch allowed us to use a simple post-loader.

The additional registers needed by `_do_watch` are saved upon entry and restored on exit. `_do_watch` contains code to check the passed-in address against the address ranges stored in the watch table. If the target address is found to be within range, control passes through a publicly known location `__watch_break` at which the debugger can set a breakpoint. Pseudo-code for `_do_watch` is shown in Figure 0.2. In our tuned implementation, `_do_watch` has a

[2]Our implementation is safe for non-preemptive multi-threaded code. However, the use of a single static data area for saving registers is not safe in the presence of preemptive threads.

```
typedef struct {
  int start, end;
  } watch_table_entry;

_do_watch(watched_address) {
  < save registers >
  watch_table_entry cur = &watch_table;
  while (cur <= &last_watch_table_entry) {
    if (watched_address >= cur->start  && watched_address <= cur->end) {
      __watch_break: /* watchpoint hit: set breakpoint at this label */
      }
    else {
      cur++
      }
    }
  < restore registers >
  }
}
```

Figure 0.2: `_do_watch()`

constant 14 instructions plus 12 instructions for every address range being watched. With the nine instructions from the patch, a single watchpoint has a speed overhead of 35 instructions per patch target.

With this much in place, testing of patched code began. Unexpected things happened right away. The test program was:

```
int a = 4;

main () {
        a = a + 10;
        printf("a = %d\n",a);
        a += 42;
        printf("a = %d\n",a);
        }
```

The unpatched version prints:

```
a = 14
a = 56
```

The first time it was run after being patched it printed garbage. When the patched program was run under debugger control and single-stepped at the instruction level, it gave the right output. When the offending code (in `_do_prnt`) was run at full speed, it appeared that a conditional branch was not being taken, even though we verified that the branch conditions were true. After considerable investigation, we discovered a patch placed in a branch delay slot. The patch begins with a branch, so the patched code had a branch in a branch delay slot. It is not surprising that the patched code did not give the same results as the original code.

We had not entirely overlooked this possibility. In the case in which a load or store is in a branch delay slot, we placed the patch immediately prior to the branch, rather than placing it immediately prior to the load or store. However, for historical reasons, the post-loader treated a branch that is also a jump target specially, and the case of a load or store in the delay slot of such a branch was not handled properly the first time around.

With that problem out of the way, the test case ran properly. The facility was then used on a C compiler in an attempt to find a real bug. We eventually did find the bug, but first we had to fix our patch technology: once again, the patched program gave different results from the unpatched version. There was a load to a register followed by a store from the same register. In the unpatched version, the store was in the delay slot, so the store took place using the old contents of the register. Because the patch inserted instructions between the load and store, in the patched version the store was not in the delay slot, so the store took place using the new contents.

Memory accesses in load delay slots have the potential to involve read/write or write/write dependences on the load in whose delay slot they sit. Most of these dependences were eliminated by re-ordering the code. When the post-loader finds such a dependence, it moves the instruction in the delay slot above the load is whose delay slot it resided, and places a `nop` in that slot. It then inserts the patch without modifying the program's semantics.

We punted on one such case. Suppose we have the following sequence of instructions:

```
lw  $s0, 0($a0)
lw  $a0, 0($s0)
```

Although the order of these instructions can be reversed, they must remain adjacent; a patch cannot safely be inserted between them. If the post-loader ever comes across this case, it will complain and will not insert a patch. To date we have not found a program from which the Titan compiler generates this code.

There were other bugs in our watchpoint patch technology, but they were all of the 'we forgot to patch this instruction' flavor, and are of no great interest.

## The Debugger

With the post-loader correctly patching the code, one conceptual half of the job was done. The remaining work was to modify KPC's debugger `dbg` to let the user set watchpoints.

`dbg` uses the syntax `<cmd> at location`, where `<cmd>` is any sequence of dbg commands and `location` is an address, a line number, or a function name, to execute `<cmd>` at `location`. In the common case that the user wants to get control, `<cmd>` is `break`. The syntax `<cmd> in fcn` executes `<cmd>` at the end of function `fcn`'s prologue. We chose the syntax `<cmd> on address` and `<cmd> on low_address high_address` for watchpoints, to maintain consistency in the interface. Internally, `<cmd> on address` is expanded to `<cmd> on address address`, so we need only discuss the latter form of the command.

On receiving a watchpoint command, the debugger has to add an entry to the watch table and ensure that `<cmd>` is executed when the watchpoint is hit.

## Maintaining The Watch Table

The watch table is located in a special version of the C start-up routine `crt0`. It must be located in the debuggee's address space because the patch code that reads it is debuggee code. All data associated with the table are publicly known symbols so that they are available to the debugger. The watch table itself is an array of address ranges (`structs` containing two pointers), and has one page of storage allocated to it—we didn't want to think about how big it should be, so we made it big enough.[3]

When a watchpoint command is entered or enabled, the address range is appended to the debugger's watch table. When a command is disabled or canceled, the last range in the table is copied over the range that is no longer being watched. At each point at which the debuggee is about to execute, the debugger's watch table is copied into the debuggee's address space. If the table is empty, `$fp` is set to zero. If the table is not empty, `$fp` is set to point to the register save area and a breakpoint instruction is placed at `__watch_break`.

[3]The page size on the Titan is 16 Kbytes.

## Handling Watchpoint Commands

There are two keys to the design of the debugger's watchpoint command handling facility: 1) every watchpoint that is hit by `_do_watch` causes a breakpoint to be reached at location `__watch_break`, and 2) the user's command must be executed at the patch target (the instruction that is about to access the watched location). We allocate one breakpoint structure, `wp_bp`, at debugger initialization time and associate it with `__watch_break`. Every time the user enters a watchpoint command, we chain the command to `wp_bp`, along with the address range (in addition to modifying the debugger's watch table as described above).

When a breakpoint at `wp_bp` is hit, the debugger gets control at `__watch_break`. The watched location has been passed to `_do_watch` in `$a0` and the target instruction is three instructions after the return from `_do_watch`.[4] The debugger searches the commands chained to `wp_bp` for commands in whose address range the watched location falls. It chains them to a temporary breakpoint set at the target instruction. When execution of the debuggee is continued, the standard breakpoint mechanism will execute the user's commands at the target instruction. This breakpoint is removed before a subsequent execution of the debuggee because there is no guarantee that the next execution of the target instruction is an access of a watched location.

In addition to adding these features, we added code to modify the parser, to disable, enable, and cancel individual watchpoints, and to disable and enable watchpoints as a group. There was little of interest involved in this, but adding an informative message required some subtlety and uncovered some situations under which a watched location was accessed but our implementation did not take a breakpoint.

The standard breakpoint handler prints a message telling the user about the point at which the debuggee halted:

```
a.out stopped at `main:#15:t.o` (pc=0x400240)
```

Under the implementation as described so far, when the user got control, it appeared both to the user and to the debugger that a normal breakpoint had been reached. We added a facility to tell the user that a watchpoint was reached. When `wp_bp` is hit, the debugger constructs a message like one of the following:

```
Watchpoint: about to store 0x100 from $s0 into 0x7fdffd2c (= 0x37)
Watchpoint: about to load 0x37 from 0x7fdffd2c into $t6 (= 0x0)
```

The watched location (shown as `0x7fdffd2c`) is available as the parameter to `_do_watch`. Its contents (shown as `0x37`) can be found by looking in the debuggee's memory space. Finding the other information requires parsing the target instruction to determine the type of the instruction and the register used. [5]

---

[4] If the access of the watched location is in a branch slot, the user must get control at the branch instruction, although the access is at the subsequent instruction. The user always gets control three instructions after the return from `_do_watch`.

[5] The debugger must take into account that if the target instruction is a branch, the following instruction is the load or store of interest.

The second time a watchpoint was hit after we added this facility, the patched program failed with a bus error. We were watching an eight byte range of memory. The target program was in a loop doing something like `*p++ = *q++` where `p` and `q` were pointers to characters. For our message, we were always fetching a word from the watched location. The first watchpoint we hit, the contents were properly fetched. The second watchpoint we hit was at the same instruction, a load-byte instruction. But the second time through the loop, the address of the accessed location (pointed to by `p`) was odd. When the debugger tried to fetch a word from that odd address, it used a load-word instruction and got an addressing exception.

Our solution was to take into account the size of the operand, and fetch the right-sized value from the watched location. Considering alignment issues brought a problem to our attention. Suppose the user enters the command `break on 19`. The debugger duly enters the range 19,19 into the watch table. What happens on a store-word instruction into address 16? Clearly byte 19 is stored into, and the user's command should be executed. But 16 is passed to `_do_watch`, is not found to be in the range 19,19, and `_do_watch` returns without causing the debugger to get control.

For a load-byte instruction, the code in `_do_watch` is correct. But for a load-halfword instruction or a load-word instruction, `_do_watch` may miss some accesses that it should catch.

In our implementation, when the debugger enters the address range into the watch table, it masks the low two bits of the start address. Every range that `_do_watch` tests starts on the largest four-byte-aligned address less than or equal to the start address. Now rather than missing some accesses that should be found, `_do_watch` gives false positives. If the user is watching byte 19, `_do_watch` will give the debugger control on all accesses of byte 19 – but also on accesses of bytes 16, 17, and 18. The debugger maintains (an unmasked copy of) the start address, so once it has determined the type of the instruction, it can weed out the false positives.

Each false positive causes a context switch to the debugger. These can be eliminated by having three copies of `_do_watch`, one for byte accesses, one for halfword accesses, and one for word accesses. Each copy of `_do_watch` could mask the low address of the range appropriately. The post-loader could patch in a call to `_do_watch_byte` for load-byte and store-byte instructions, `_do_watch_hword` for load-halfword and store-halfword instructions, and `_do_watch_word` for load-word and store-word instructions. Each version of the `_do_watch` routine could pass control through the same location (`__watch_break`) on a hit (this would obviate the need for the debugger to maintain three separate breakpoints).

For convenience, we allow the access type to be part of the watchpoint command. If you want to watch stores into address A and loads from address B, you have to patch both loads and stores. But at a load from A, you need not be given control of the program, because the debugger can check the access type and 'continue' for you. Note that there could be one watch table and copy of `_do_watch` for stores and another for loads; if at some time only stores were being watched, the debugger need not place a breakpoint instruction in the 'load' version of `_do_watch`, and no context switch to the debugger would be taken for loads.

| Program | Version | locations watched | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 10 |
| linpack | unpatched | 1.00 | | | |
| | stores patched | 1.03 | 2.13 | 2.46 | 5.08 |
| | loads patched | 1.27 | 6.27 | 7.74 | 20.02 |
| | both patched | 1.28 | 7.24 | 9.06 | 23.67 |
| C compiler | unpatched | 1.00 | | | |
| | stores patched | 1.20 | 3.33 | 3.93 | 8.69 |
| | loads patched | 1.55 | 5.13 | 6.13 | 13.79 |
| | both patched | N/A | – | – | – |
| linker | unpatched | 1.00 | | | |
| | stores patched | 1.04 | 2.46 | 2.91 | 6.19 |
| | loads patched | 1.12 | 3.70 | 4.45 | 10.68 |
| | both patched | 1.20 | 5.12 | 6.31 | 15.86 |

Table 0.1: Normalized execution times for sample programs. Execution time for an unpatched version of linpack was 37.1 seconds, for the C compiler, 42.5 seconds, and for the linker, 9.8 seconds.

With this implementation in hand, we compared the speed and size of some patched programs with their unpatched ancestors.

## Results

Patched programs vary in their running speed relative to unpatched versions depending on the proportion of patched instructions (loads, stores, or both) to other instructions, depending on the number of patched instructions inside loops, and depending on instruction-cache behavior. If we let $m$ be the number of memory accesses (patch targets) in the program, and $n$ be the number of address ranges that are being watched, the run-time overhead of the watchpoint facility is $(12n + 23)m$ instructions (except in the case when $n = 0$, in which case the overhead is $2m$ instructions, due to the branch around the patch).

Table 0.1 shows the relative execution times of patched and unpatched versions of several programs. The C compiler could not be patched for both stores and loads because the code expansion caused a relative branch to become out of range. This problem could have been solved by breaking a large routine into several smaller routines, or by modifying the postloader to replace the relative branch with an absolute branch. Table 0.2 shows the sizes of the text section of patched and unpatched versions of these programs. Table 0.3 shows the time required to patch the sample programs. All programs have been compiled without optimization and with debugging information to resemble typical to-be-debugged code.

Programs slow down significantly; by about a factor of 3 in the typical case (watching stores to a single location). By comparison, taking context switches to check memory access

| Program | Unpatched size | Stores patched | Loads patched | Both patched |
|---|---|---|---|---|
| linpack | 1.0 | 2.22 | 2.64 | 3.87 |
| | 130992 | 291228 | 346308 | 506544 |
| C Compiler | 1.0 | 1.88 | 2.75 | N/A |
| | 2724352 | 5122780 | 7499644 | |
| linker | 1.0 | 1.88 | 2.44 | 3.31 |
| | 56272 | 105592 | 137092 | 186412 |

Table 0.2: Normalized text sizes, followed by actual text sizes (in bytes), for sample programs.

| Program | Unpatched size | Time to patch stores | Time to patch loads | Time to patch both |
|---|---|---|---|---|
| linpack | 130992 | 1.1 | 1.1 | 1.2 |
| C Compiler | 2724352 | 21.6 | 23.1 | N/A |
| linker | 56272 | 0.5 | 0.5 | 0.5 |

Table 0.3: Time required to patch sample programs (in seconds).

slows programs down by an extreme factor: Wahbe et al. [Wah93] measured the overhead for watchpoints using dbx at a factor of 85,000. Unlike the context-switch approach, our watchpoint facility is fast enough for everyday use. The overhead can be significantly reduced by a more complex patching technology—Wahbe et al. [Wah93] achieved an overhead of only 25 per cent, watching stores only.

## Finding a Free Register

The high-level watchpoint facility design is portable at least among RISC machines. The implementation is highly dependent on the particular architecture. On the Titan, we benefit from the convention that makes `$fp` available for instrumentation code. On other architectures or using a different compiler, there may not be a register available.

Our patch methodology uses a number of registers, each of which must be made available by 'spilling' it: saving and restoring its value around the inserted uses. To spill a register, the address of the spill location must be in a register, so one free register is necessary. Other registers can be spilled using the same base address, thus one free register can be used to bootstrap the spilling of an arbitrary number of registers. Finding the first register remains a problem; we suggest several possible alternatives:

- It may be possible to spill a register into a location relative to the `$pc`. An example of this would be adding a `nop` instruction where it can never be executed (as after the end of each routine), and spilling the register into that location. There may be operating system or architectural restrictions preventing use of this technique. For example, this technique cannot be used on most UNIX[6] systems, because the text section is not writable. It cannot be used on the Mips R3000 processor because `$pc` cannot be used in address calculations.
- Another possibility is to spill a register onto the stack. We deem this to be an undesirable option because it is more likely than the other options to change the behavior of the program. However, if none of the other options are feasible, a stack location can be used for the spill. Depending on the calling convention, this may require help from the compiler. For example, on the R3000 it is not possible to simply spill the value to a location beyond the top of the stack, because a trap handler may use the same space. If there is free space on the stack (or the compiler can be modified to provide it), the value can be spilled there. It is best to minimize the number of registers spilled onto the stack, so we recommend spilling one register onto the stack and using it to bootstrap the spilling of other registers into the data section.
- The best option is to spill registers into static data space. This is possible if there is a register that always points to a known location in the data section and space can be reserved in the data section: the space can be accessed relative to the fixed register. For example, many RISC processors have a register that is fixed to read-as-zero. If there is data space that can be accessed relative to the zero register and reserved for this purpose, registers can be spilled there. A number of platforms provide a fixed global-pointer (`$gp`) register. On such a platform, a register save area can be reserved at a fixed offset from `$gp`; as with our use of `$fp`, the 'bootstrap' is always available. Similarly, if there is a register that is "pseudo-fixed", that is, fixed per routine or compilation unit, spill locations can be reserved for each possible value of that register. However, this probably requires compiler modifications, where with a truly fixed register, data space can probably be reserved using the data declarations available in either a high-level or an assembly language.

In general, the problem of finding a free register requires the aid of the compiler, but on many architectures the existing conventions allow for a solution without compiler modifications.

## Enhancements

We would like to be able to patch the program from within the debugger, and to have the debugger run the patched executable, but keep its displays (especially disassembly) based on the original executable. This would require a translation between addresses in the original executable and addresses in the patched executable.

[6] UNIX is a trademark of Bell Laboratories.

We would like to allow symbol names in the expressions that specify the address range in a watch command.[7]

A minor modification to the implementation would generalize the functionality considerably: provide a functional interface to the watch table within the patched program, and when a watchpoint is hit, make a call through a public function pointer (initialized to point to a routine that simply returns). Then code could be linked into the debuggee and executed on watchpoint hits without using the debugger at all.[8]

## Related Work

Patching technology has been successfully used for some time for performance evaluation. Patching technology was used by Wahbe et al. to implement watchpoints on stores efficiently ([Wah93]). Patching technology has also been used to implement fast conditional breakpoints ([BK92], [Kes90]). When a breakpoint is set, the debugger patches in the code to test the condition, so that a context switch to the debugger only occurs when the condition is met. An interesting technical issue is where the code to test the condition comes from. Kessler [Kes90] requires the code to be previously compiled and available to the Cedar debugger. Brown [BK92] has built a mini-compiler into the Los Alamos debugger **ldb**.

## Summary

We have described a simple, reasonably efficient, extremely general watchpoint implementation that does not require hardware, operating system, or compiler support. A single watchpoint command can watch any contiguous range of locations. Up to 2048 watchpoint commands can be active at a time. The technique should be fairly easily adaptable to other platforms with RISC processors. A watched program runs significantly slower than an unwatched program, as shown in Table 0.1, but orders of magnitude faster than watchpoints implemented via regular context switches to check accesses. Faster software watchpoint implementations are possible but are more complex.

### Acknowledgments

This watchpoint facility is the brainchild of (in alphabetical order) Max Copperman, Bob Hood, Samir Patel, and Jeff Thomas. The post-loader was modified by Jeff Thomas;

[7] Allowing symbol names in watch commands introduces a problem. If a variable is residing in a register, accesses of that variable will not be trapped. (This is the case with hardware-supported and operating-system-supported watchpoint implementations as well.) This may confuse the naive user. We consider this unfortunate, because it makes watchpoints less useful for a data-oriented debugging strategy than we would like. However, it does not compromise the use of watchpoints to locate the point at which memory is getting trashed.

[8] I'd like to thank an anonymous reviewer for this insight.

the debugger was modified by Max Copperman. Debugging patched code while viewing unpatched code was Kung Hsu's idea. The work was largely funded by Kubota Pacific Computer, Inc.

Thanks are due to the referees, whose comments led to a vastly improved paper.

## References

[Wah92] R. Wahbe, "Efficient Data Breakpoints", *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Systems*, SIGPLAN Notices, Vol. 27, No. 9, pp. 200-212, September 1992.

[Wah93] R. Wahbe, S. Lucco, S. Graham, "Practical Data Breakpoints: Design and Implementation", *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, Vol. 28, No. 6, pp. 1-12, June 1993.

[BK92] J. Brown, R. Klamann, "The Application of Code Instrumentation Technology in the Los Alamos Debugger", Proceedings of the Supercomputer Debugging Workshop '92, October 1992.

[Kes90] P. Kessler, "Fast Breakpoints: Design and Implementation", Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, June 1990.

[Bro91] J. S. Brown, "The Los Alamos Debugger ldb", Proceedings of the Supercomputer Debugging Workshop '91, November 1991.

[CH91] B. Chase, R. Hood, "Debugging with Lightweight Instrumentation", Proceedings of the Supercomputer Debugging Workshop '91, November 1991.

[HJ92] R. Hastings, B. Joyce, "Fast Detection of Memory Leaks and Access Errors", Proceedings of the Winter Usenix Conference, pp. 1-12, January 1992.

[Kep93] D. Keppel, "Fast Data Breakpoints", University of Washington, Computer Science and Engineering Technical Report 93-06, April 1993. Available as UW-CSE-93-04-06.PS.Z via anonymous ftp from ftp.cs.washington.edu.

[LB92] J. R. Larus, T. Ball, "Rewriting Executable Files to Measure Program Behavior", University of Wisconsin-Madison, Computer Science Technical Report 1083, March 1992, to appear in Software Practice & Experience.