

Fast Data Breakpoints

David Keppel

3 May 1990, revised 14 April 1993

UW CSE TR# 93-04-06

Abstract

Debuggers allow a user to halt a program and examine its state. The debugger stops execution when some user-specified condition is satisfied: *Code breakpoints* halt program execution when a particular instruction is executed. *Data breakpoints* halt program execution when a variable is referenced. Code breakpoints are supported directly in hardware on most machines and are fast. Data breakpoints, however, are notoriously slow. This note describes how data breakpoints can be made fast.

1 Introduction

Debuggers provide commands that allow the programmer to halt a program and examine its state. The most common command is a *code breakpoint*, which halts the program when execution reaches a certain instruction. Some debuggers also support *conditional breakpoints* which evaluate an expression whenever a certain instruction is reached and halt the program only if the expression evaluates to true [Kes90]. Some debuggers provide *data breakpoints*, which halt the program whenever a variable is referenced; *conditional data breakpoints* halt the program when a variable is referenced and an expression evaluates to true.

Data breakpoints (and conditional data breakpoints) are notoriously slow. Using data breakpoints typically slows execution by several orders of magnitude, which makes execution too slow to allow debugging of even small programs. Data breakpoints are slow because debuggers typically implement them by executing one instruction, evaluating the condition,

then executing one more instruction, then reevaluating the condition, and so on. Single-stepping a program generally involves at least one trap to the operating system, and, typically, half a dozen protection domain crossings.

Fast conditional breakpoints save many protection domain crossings by evaluating some debugger code in the context of the debuggee. Fast conditional breakpoints suggest a way that data breakpoints can be implemented more efficiently [Kes90, Wah92]: Each load and store instruction is patched with a jump to code that performs a test. The debugger is restarted if an “interesting” address is accessed. Although this mechanism is faster, it still requires patching, and thus slowing, of every memory reference instruction.

Here we suggest using page protection mechanisms to patch memory reference instructions *lazily*. The page that holds the interesting variable is access-protected so that accesses to the page cause a trap. An accessible copy of the page is placed elsewhere in the address space. If the variable spans pages, then all of those pages are access-protected and remapped. When a memory reference instruction accesses a protected page and traps, the instruction is then patched with a jump to code that evaluates whether the “interesting” variable is being accessed. This implementation of data breakpoints has two features: First, instructions are patched only when they reference “close” to the interesting variable; most patching is thus avoided. Second, instructions are patched only once. Thus, each memory reference instruction causes at most one trap.

The remainder of this note is organized as follows: Section 2 provides background on other mechanisms for performing data breakpoints. Section 3 gives more detail on the design presented here. Section 4 illustrates the idea with an example. Section 5 considers related issues, such as setting multiple data

Please address correspondence to the author at pardo@cs.washington.edu or at the University of Washington, Department of CS&E, FR-35; Seattle, Washington 98195

breakpoints, code patching issues, and the lifetime of variables and the resulting effects on data breakpoints. Finally, Section 6 summarizes and concludes.

2 Background

There are many ways of implementing data breakpoints. Here we consider some of them:

- **NATIVE HARDWARE:** Some processors, such as the Intel 80386, have registers that can be set to cause a trap when certain memory locations are referenced. Other systems, such as the Connection Machine CM-5, have bits associated with each word of memory;¹ the bits can be set to cause a trap when a particular word is referenced [Cor91]. Although hardware is typically fastest [Wah92], it also the most machine-dependent. Further, machines such as the 80386 can only check a small number of breakpoints: it is not possible to set breakpoints simultaneously on 10 locations.
- **SIMULATION:** The program is run on a virtual machine. For each instruction that references memory, the simulation executes code that checks the *effective address*, which is the data address being referenced. If the effective address matches a breakpoint address, simulation stops and the debugger is restarted. Simulators are typically 5 to 50 times slower than native hardware [Bed89, CK93], plus the cost of checking memory operations.
- **SINGLE STEPPING:** Each instruction is checked before it is executed. If the instruction references memory and the effective address matches a breakpoint, the user is notified. Else, the instruction is executed, then the debugger examines the next instruction and so on. Data breakpoints implemented this way are typically *slower* than simulation because single-stepping typically requires many protection boundary crossings.
- **TRAP PATCH:** Each memory reference instruction is replaced with a trap to the debugger. This

¹The CM-5 bits are actually associated with each 32 bytes (4 words) of memory. Either distinct variables must be segregated in to different 32-byte regions, or the memory bits are used to implement virtual memory with very small “pages” [RHL⁺93] and, thus, infrequent traps to the operating system.

improves over single stepping because typically only about 30% of instructions are memory reference instructions, and only a third of those are writes.

- **VIRTUAL MEMORY:** Page(s) with breakpoint variables are reference-protected. References to other pages run at full speed. References to protected pages cause a trap to the debugger, which evaluates the condition.² If execution continues, either the program is single-stepped with pages unprotected, or the debugger simulates the instruction. **VIRTUAL MEMORY** improves over **TRAP PATCH** because only some of the memory reference instructions cause traps. However, some memory reference instructions can still trap frequently.
- **CONDITION SEARCHING:** A *search breakpoint* is inserted so that execution stops when the program is half done. When the search breakpoint is reached, the condition is checked. If the condition has not yet been reached, the program state is checkpointed and execution continues. If the condition has been reached, execution is restarted from an earlier checkpoint, with a new search breakpoint set between the previous and current search breakpoints. **CONDITION SEARCHING** can run the program at or near full speed while it is making forward progress [WM89]. Using a binary search means that if interrupts and checkpointing halves the execution speed, the overall slowdown will be at most a factor of four.

The scheme as described above is essentially impossible to implement, because it is generally hard to decide where to put search breakpoints in order to stop “half way” between two search points. It is also impractical because half the program is executed before any breakpoint checks are performed. Thus, a better implementation periodically interrupts the program and checks the condition.³ Initially, interrupts and checkpoints are performed at small intervals, with the interval growing if the program runs for a long time without backing up. When the program is backed up and restarted from an earlier checkpoint, the interrupt/checkpoint interval is reduced. Eventually, the interval is small enough

²Alternatively, the trap handler may evaluate the conditional instead of the debugger [Wah92].

³In some circumstances, it may be preferable to instrument the code so that it performs polling.

that interrupt/checkpoint overhead is very high cost, but the total number of instructions to be executed before a breakpoint is so small that the real execution time is still small.

CONDITION SEARCHING only works for certain kinds of data breakpoints, those in which a condition goes true and stays true for the remainder of the program.

- **CODE PATCH:** Each instruction that references memory is patched with a jump to a new code fragment, called a *displaced handler*. The displaced handler first checks the data breakpoint condition and halts the program if necessary. Otherwise, the displaced handler simulates the displaced instruction and jumps to the logical successor of the displaced instruction. Code patching suffers from three problems: First, it is potentially hard to find all memory reference instructions. Second, it may be expensive to patch all such instructions. Finally, the cost of testing is paid by all memory reference instructions, even though most instructions don't reference the interesting memory.

3 Patch On Trap

This note proposes a scheme, PATCH ON TRAP, based on lazy code patching. PATCH ON TRAP incorporates ideas from CODE PATCH and VIRTUAL MEMORY. There are several observations about each scheme: First, CODE PATCH has high startup for finding and patching memory reference instructions. Second, code patch has an overhead for each memory reference, whether or not it references “interesting” memory. Third, virtual memory has low overhead on references that miss “interesting” memory, but a high overhead per execution for instructions that reference data on the same pages as breakpoint variables.

PATCH ON TRAP uses virtual memory primitives [AL91] to perform patching lazily: memory reference instructions are patched only if they actually reference a protected page. Like CODE PATCH, the reference instruction is replaced with a jump instruction to a displaced handler that evaluates the conditional and simulates the displaced instruction. Unlike CODE PATCH, the displaced instruction cannot be simulated in a straightforward way, because the breakpoint page is reference-protected. Therefore, the reference-protected page is remapped to an

otherwise unused address and memory references are transliterated by the displaced handler.

Note that in principle, write breakpoints need only write-protect the page, so that reads can run without trapping and patching. However, if virtual aliasing is a problem, then both reads and writes must be displaced.

PATCH ON TRAP proceeds in three steps: First, the page with the breakpoint variable is reference-protected and remapped to a new part of the address space. The program is then started. The second step happens when the program references a protected page: The debugger is invoked and it generates a displaced handler and patches the memory reference instruction with a jump to the displaced handler. The program is then restarted. The third step happens each time the patched instruction is executed: execution jumps off to the displaced handler, which checks the memory reference. If it satisfies the breakpoint condition, then the debugger is restarted. If the condition is false, then execution continues. The code displaced handler simulates the displaced instruction by checking the effective address. If the reference would go to the protected page it is instead redirected to the page at its new mapping address. Otherwise the memory reference simply completes.

4 An Example

This section walks through an example to make the discussion more concrete. Suppose we want to trace memory location `0x53` and stop when it reaches the value `777`. Since the breakpoint condition depends on changes to the traced memory location, only writes need to be patched and redirected. If the breakpoint condition were instead to break whenever the value `777` is *read* from the location, then both reads and writes would need to be redirected.

The first step proceeds as follows: `0x53` is on page zero, so that page is remapped as read-only and a writable copy is mapped at some other location, say, page 8, starting at location `0x8000`. `0x53` is therefore “shadowed” at `0x8053`. The first step is then done and the program is (re)started.

Suppose the instruction `store r1, 4(r0)` causes a protection fault. The instruction is then patched with a jump to a new displaced handler, shown in Figure 4. The trapping instruction is restarted, but now it executes a branch to the displaced handler.

When the displaced handler is invoked, it checks

```

// Handler for "store r1, 4(r0)",
// checks writes to address 0x53,
// stops if value to write is 777.

if PAGE(4+r0) == 0 then
  // Translate page 0 address
  // to a page 8 address.
  tmp = 4 + r0 + 0x8000;
  if 4+r0 == 0x53 and r1 == 777 then
    breakpoint();
  endif
  *tmp = r1
else
  // Any page other than page 0.
  *(4+r0) = r1
endif
jump to next instruction

```

Figure 1: Example displaced handler

the effective address: If it goes to a page other than page zero, then the write simply completes. If the write goes to page 0, the effective address and conditional are checked. If the effective address is `0x53`, and if the new value is `777`, then the displaced handler causes a breakpoint. If no breakpoint is needed, the write is transliterated to page 8. Execution then continues at the instruction following the displaced instruction.

Although the above handler is more expensive than the original `store` instruction, it is probably a dozen instructions in the common case.⁴ Further, only *some* memory reference instructions are patched; most run at full speed.

When the breakpoint is deleted, the displaced instruction is replaced, clobbering the branch instruction. The displaced handler is deallocated, page zero is mapped as read/write, and the page 8 duplicate is removed.

⁴This estimate ignores condition codes. However, many RISC processors lack condition codes and some RISC processors can update condition codes cheaply. Further, condition codes rarely span memory reference instructions, and thus can be clobbered freely. Finally, even if saving and restoring condition codes is “expensive” doing so is still typically cheap compared to alternative implementations of data breakpoints.

5 Other Issues

5.1 Multiple Breakpoints

There are at least two ways to implement breakpoints on multiple locations. One alternative is to have each displaced handler check each possible breakpoint address. However, each handler then checks for all possible writes, even if the particular instruction only tends to write to one breakpoint page.

Another alternative is to apply lazy breakpoints to displaced handlers. Each handler checks breakpoints for one page. If the reference is off that page, the handler blindly performs the write. The write may fault, causing the displaced handler’s write instruction to be patched with a branch to a new handler. This recursive application of breakpoints builds a *displaced handler chain*, with each handler in the chain handling one page.

When a breakpoint is removed, several schemes are possible. One is to simply remove all handlers for the application instruction and let it rebuild the displaced handler chain. Another alternative is to splice the particular handler out of the chain. The latter approach is probably preferred where handlers are inserted and deleted automatically by the debugger (when, e.g., watching stack frames).

5.2 Limits On Insertion

Code patch only works in some cases. For example, displacing a memory reference instruction must free up enough space that a jump instruction can be put there. In general, this requires help from the compiler. There are many other cases that need careful attention [Kes90].

5.3 Stack Variables

Variables that allocated on the stack pose several challenges. First, the debugger must ensure that breakpoints to the variable have the same lifetime as the variable itself. Second, each time the variable is allocated it may appear at a new location. Finally, there may need to be multiple breakpoints if there can be several simultaneous invocations of the function.

For variables that are purely local, code patch is probably the preferred implementation, since there are a small number of write instructions (limited by the procedure size) and the code patching need be performed only once. Lazy patching performs badly

because breakpoints must be added and deleted at each call and return, and because page protections must be changed frequently.

When stack variables are exported to called procedures, lazy patching again becomes a reasonable choice because it is easier to patch memory references lazily.

If lazy patching is used, it is probably important to allocate each stack frame (at least each that uses a breakpoint) to use disjoint pages. Otherwise, references to local variables in a called routine will be very likely to suffer protection faults, code patching, etc.

5.4 Breakpoint Lifetimes

Global and function-allocated variables have a well-defined lifetime. It is harder to set breakpoints on heap-allocated variables because they can be allocated and freed at arbitrary times and the memory can be reused for another variable.

If the variable has a well-defined destructor, then setting a breakpoint on the variable can also set an *update point* in the destructor. When the object is freed, the update point removes the breakpoint.

Some allocators free objects implicitly so that, e.g., freeing the root of a graph frees the entire graph [Han90]. In these cases, the debugger may need some “deep” understanding of the memory allocator in order to determine when the object is being freed.

5.5 Named Breakpoints

The discussion so far has assumed a straightforward mapping from variable name to memory address. This is not always the case in the face of e.g., garbage collection.

6 Summary

Data breakpoints are useful, but notoriously slow. This note sketches out the technique PATCH ON TRAP that patches memory reference instructions and uses lazy code patching to perform the patching on demand. Lazy code patching reduces overhead compared to other code patching techniques, because only some instructions are patched. It also reduces overhead because it checks accesses only to pages that are actually referenced. The PATCH ON TRAP technique also reduces overhead compared to trapping-based breakpoint schemes, because only some mem-

ory reference instructions cause traps, and those that do require traps cause at most one trap per breakpoint.

7 Acknowledgements

Thanks to Robert Bedichek for discussing these ideas.

References

- [AL91] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, page 96, 1991.
- [Bed89] Robert Bedichek. Some Efficient Architecture Simulation Techniques. *Winter '90 USENIX Conference*, pages 53–63, 26 October, 1989.
- [CK93] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report (in preparation), Sun Microsystems Laboratories, Inc. and University of Washington, 1993.
- [Cor91] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary, 1991.
- [Han90] David R. Hanson. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software—Practice and Experience*, 20(1), January 1990.
- [Kes90] Peter Kessler. Fast Breakpoints: Design and Implementation. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, 25(6):78–84, June 1990.
- [RHL+93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *ACM SIGMETRICS*, May 1993.

- [Wah92] Robert Wahbe. Efficient Data Breakpoints. *Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 200–212, October 1992.
- [WM89] Paul R. Wilson and Thomas G. Moher. Demonic Memories for Process Histories. *Proceedings of the ACM '89 Conference on Programming Language Design and Implementation (PLDI)*, pages 330–343, June 1989.