Appeared in ASPLOS-V, October 12-15, 1992.

Efficient Data Breakpoints

Robert Wahbe*
Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

Abstract

Breakpoints are user-specified rules that trigger debugging actions when certain conditions arise in an executing program. To support source-level debugging, programmers should be able to specify breakpoint conditions in terms of programming language control and data abstractions. Support for breakpoints specified in terms of control conditions, known as *control breakpoints*, is ubiquitous. The analogous *data breakpoint*, a breakpoint specified in terms of a data condition, is difficult to implement efficiently and has only limited support in most current debuggers. A number of authors have speculated that efficient data breakpoints require hardware support.

In this paper we examine hardware and software strategies for implementing data breakpoints. We use a simulation experiment to estimate the performance of four representative implementations. We conclude that while hardware-based solutions are able to deliver the best overall performance, they are expensive and can simultaneously support only a limited number of breakpoints. In contrast, a software solution based on modifying the code of the program being debugged to monitor *all* instructions that might affect the data breakpoint condition is simple and portable, and provides for any number of breakpoints. Further, we show that its expected performance is acceptable for most debugging applications.

*This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD) under grants MDA972-92-J-1028 and N00039-88-C-0292, and by a grant from the State of California MICRO program. The content of this paper does not necessarily reflect the position or the policy of the government. The author's email address is rwahbe@cs.Berkeley.EDU.

1 Introduction

Breakpoints are user-specified rules that trigger debugging actions when certain conditions arise in an executing program. To support source-level debugging, programmers should be able to specify breakpoint conditions in terms of programming language control and data abstractions. Support for breakpoints specified in terms of control conditions, known as *control breakpoints*, is ubiquitous. An example control breakpoint is to print the value of a variable whenever control enters and exits a certain function. The analogous *data breakpoint*, a breakpoint specified in terms of a data condition, is difficult to implement efficiently and has only limited support in most current debuggers. An example data breakpoint suspends execution whenever a certain object is modified. Such a breakpoint would help identify pointer uses that are inadvertently modifying an otherwise unrelated data structure.

Interpreted environments can provide data breakpoints by enhancing the evaluator. However, for reasons of compatibility and efficiency, many programming tasks require that code be debugged in compiled form. For example, interpreted ANSI C using the Saber-C system runs approximately 200 times slower than compiled code [KLP88]. The semantics of a source language may differ in its compiler and interpreter implementations. Compilers for popular languages like C, C++, and FORTRAN are more widely available than interpreters, providing further incentive for supporting efficient data breakpoints in compiled environments.

Several authors have speculated that providing efficient data breakpoints requires hardware support [Kes90, Joh82, MCL89, CL87]. One of the crucial issues is how to implement a *write monitor service* efficiently. A write monitor service provides a notification to interested clients each time the program writes to a distinguished region of memory. The notification may occur after the write has succeeded, distinguishing write monitors from write barriers.

Proposed approaches to providing write monitors fall into three categories. We list each approach briefly; they are described in detail in Section 3.

- **Specialized Hardware.** One hardware approach is to provide monitoring support directly in the processor. Another hardware solution is to use an external analyzer to monitor bus activity. Because the functionality and performance of external analyzers depend so heavily on the details of the target architecture, we do not consider external analyzers further; we mention them here only for completeness.
- **Virtual Memory.** The virtual memory system can be used to monitor writes. All pages on which the monitored object resides are write protected. The breakpoint condition is checked from the virtual memory fault handler.
- **Software.** Software approaches modify, either before execution begins or dynamically at run-time, the program being debugged to check the target location of all writes that might affect the data breakpoint condition.

The above approaches differ significantly in their implementation cost, portability, and performance. To date, there has been very little discussion in the literature of their relative merits. Even though performance is a key issue in evaluating a strategy, we have been unable to find detailed data on any of the approaches examined in this paper.

Data breakpoints represent an important debugging service. Given well-established trends in architecture, does doing without specialized hardware support mean doing without efficient data breakpoints? If not, what approaches are preferable and under what conditions? This paper addresses these issues.

The remainder of the paper is organized as follows. Section 2 frames the problem and defines our terminology. Section 3 provides a description and qualitative analysis for each of the approaches outlined above. Sections 4, 5, 6, and 7 detail the simulation experiment we used to assess the performance of four representative implementations. Section 8 presents the results from our experiments, and Section 9 summarizes our conclusions.

2 Framework

It is sufficient for our purposes to identify only three components of a debugging system: the *debuggee*, the *debugger*, and the *write monitor service (WMS)*.[1] The debuggee is the program being debugged. The debugger is the high-level application with which the programmer interacts. The write monitor service, discussed below, provides the low-level support necessary for data breakpoints.

[1] These components do not necessarily fall along process or address space boundaries.

Before we define the write monitor service's interface, we first define some terms. A *write monitor* is a descriptor that specifies a contiguous region of memory. For convenience, we use this term to refer to both the descriptor and the memory it describes. A write monitor is *active* if the WMS has guaranteed that clients will be notified of all writes that affect the write monitor. The region of memory specified by an active write monitor is said to be *monitored.* Any machine instruction that writes to memory is a *write instruction.* A write instruction that writes to one or more write monitors is a *monitor hit*; otherwise, it is a *monitor miss.* There is a single *monitor notification* for each monitor hit.

A *monitor session* characterizes the write monitor activity with respect to one run of the program. For example, a high-level description of a monitor session might be: "Monitor all heap objects allocated by a particular function". We rely heavily on the concept of a monitor session when we attempt to assess an approach's performance.

The interface to a write monitor service is quite simple. While particular implementations might require slightly different interfaces, a similar set of services must be provided by any system. The interface consists of the following functions (`BA` = Beginning Address, `EA` = Ending Address):

- `InstallMonitor (BA, EA)`
  Installs a new write monitor.
- `RemoveMonitor (BA, EA)`
  Removes an existing write monitor.
- `MonitorNotification (BA, EA, PC)`
  Called by the monitor system for each monitor hit. `PC` is the program counter of the monitor hit.

3 Approaches

This section briefly describes and analyzes each of the approaches mentioned in the introduction. Note that system calls, shared memory, and objects stored in registers pose additional problems for all approaches.

3.1 Specialized Hardware

A small number of processors provide direct support for write monitors, including the Intel i386 [Int86] and the MIPS R4000 [KH92]. Typically, specialized registers, called *monitor registers*, are used to specify the region of memory to be monitored. A hardware trap is generated when a write occurs to a monitored region of memory. The monitor registers are part of an address space's state and require operating system support. The biggest disadvantage of this approach is the extremely limited functionality that is provided by existing or proposed hardware. No widely-used chip today supports more than four concurrent write monitors [Int86, KH92, Joh82].

3.2 Virtual Memory

A conceptually elegant approach to implementing write monitors is to use the virtual memory system [Bea83, SS91, Dig]. When a write monitor is installed, the WMS protects all pages the monitor resides on. The WMS can register a fault handler, allowing it to detect monitor hits when the debuggee attempts to write to a protected page. The WMS must arrange for execution to continue while insuring that the page is protected for subsequent writes. This may be accomplished by unprotecting the necessary pages, single-stepping the program, and reprotecting the pages. An alternative is for the WMS to emulate the faulting instruction. A disadvantage of using virtual memory is a potential lack of portability. Current operating systems have varying levels of support for user-level virtual memory services [AL91].

3.3 Software

Software approaches modify the debuggee to check the target location of all write instructions [Kes90, DMS84, HJ92]. This may be done before execution begins, in a way that supports all possible write monitors, or at runtime as write monitors are installed and removed. A hybrid approach might be used, such as leaving space between functions or strategically placing "nop" instructions, to make dynamic modification simpler [Kes90]. Which approach one employs depends on the language being monitored and the performance penalty of executing unused monitor code. In type-unsafe languages such as C, where almost any write instruction could corrupt memory, dynamic modification is less appropriate. If the modifications are done at compile time, detecting monitor hits in standard libraries requires that special versions of the libraries be used.[2]

Two methods can be used to transfer control to the WMS support code responsible for detecting monitor hits. The first is to replace the write instruction with a trap instruction. This method is used by the UNIX debuggers gdb and dbx. A user-level trap handler determines if the corresponding write instruction is a monitor hit. Control breakpoints are typically implemented in a similar fashion, thus minimizing the need for new debugging mechanisms. Using traps in this way requires the WMS to be integrated with the operating system signal facility. We refer to this approach as *trap patching*.

[2] This is similar to the **gprof** facility [GKM82], which also requires special versions of the libraries.

The second approach, which we call *code patching*, is to transfer control directly via either an inline check or a function call. Direct control transfer has the advantage of being operating system independent.

3.4 Discussion

Both virtual memory- and software-based approaches must maintain a mapping between virtual addresses and active write monitors. If these data structures reside in the debuggee's address space, two issues must be addressed. First, the mapping must be protected against corruption. Second, the debuggee's behavior might be perturbed.

One solution to protecting WMS data structures for virtual memory-based approaches is to have the WMS dynamically unprotect and protect the necessary data pages. An alternative is to assume that these data structures are shared with a program executing in a different address space, possibly the debugger, with read-only access for the debuggee, and read-write access for the debugger. Updates could be performed via interprocess communication. In the UNIX environment all signals may be intercepted via the `ptrace` system call, allowing the structures to be maintained in a separate address space, but imposing considerable overhead due to context switching on each fault.

Because software-based approaches check every write instruction, no additional mechanism and very little overhead is required to insure the integrity of WMS data. For both virtual memory- and software-based approaches, we conjecture that the impact of maintaining a small amount of read-only data in the debuggee's address space will be acceptable for debugging.

4 Experiment Overview

Section 3 identified and discussed a number of strategies for implementing a WMS. We now turn to the performance characteristics of implementations based on four strategies: specialized processor support, virtual memory, trap patching, and code patching. To carry out this study, we devised a simulation experiment, depicted in Figure 1 and described in detail in the next three sections. The experiment consists of two logical phases. Phase 1 generates a program event trace. In phase 2, the simulator uses that trace and a description of the objects to be monitored to output detailed data about program behavior with respect to the monitored objects. Using simple analytical models, we combine this data with operating system and hardware timing information to estimate the overhead imposed by the WMS implementation under study. Logically, phase 1 is done once per program, phase 2 once per monitor session.

Appeared in ASPLOS-V, October 12-15, 1992.

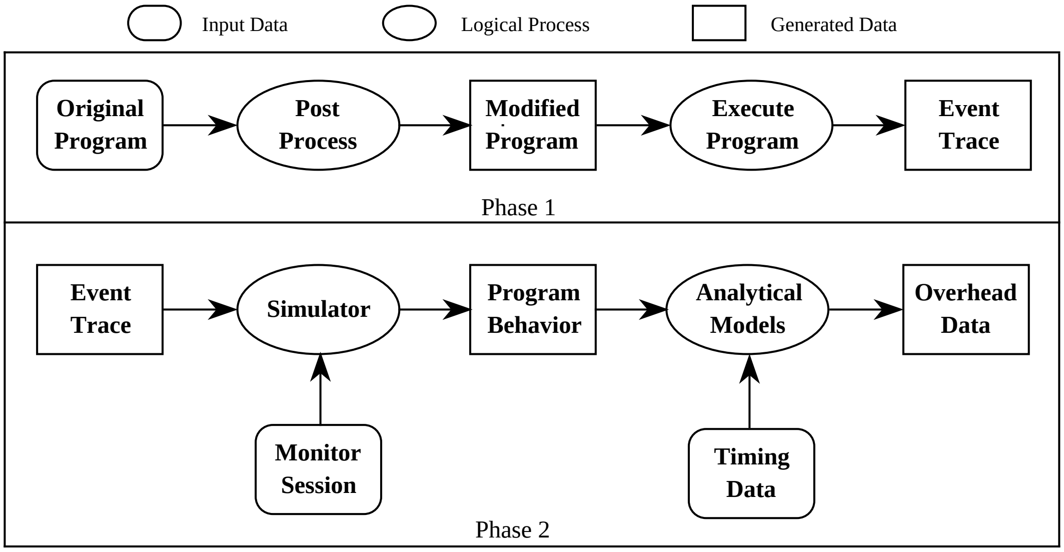

Figure 1: Experiment Overview

We chose simulation for a number of reasons. We are interested in comparing the approaches, and no easily-available system had all the necessary architecture and operating system services. Comparing prototype implementations across different platforms would be difficult. Secondly, we are interested in how page size affects the performance of strategies based on virtual memory protection, and a simulator allows us to change the page size easily. As will be described in the next section, we ran the experiment on a large number of monitor sessions. For a number of the test programs, running a WMS prototype implementation for each monitor session would be impractical.

5 Monitor Sessions

To assess the performance of each approach, we estimate the overhead incurred on a wide variety of monitor sessions for five C programs. For the purposes of this study, the goal in defining a monitor session is to reflect a "typical" debugging scenario. Ideally, these monitor sessions would be replays of actual debugging runs from a wide variety of programmers tackling a wide variety of problems. We compromise by defining a large number of *program-independent* monitor sessions that together provide evidence for how a particular approach will perform in practice.

We study the following types of monitor sessions:[3]

- *OneLocalAuto* Monitors a single local automatic variable. All instantiations of the variable belong to the same monitor session.
- *AllLocalInFunc* Monitors all local variables of a single function, including local static variables. All instantiations of the local variables belong to the same monitor session.
- *OneGlobalStatic* Monitors a single global static variable.
- *OneHeap* Monitors a single heap object.[4]
- *AllHeapInFunc* Monitors all heap objects created by a function f and any other functions executing in the dynamic context of f.[4]

Except for *AllHeapInFunc*, the above monitor session types are somewhat obvious. We wanted a monitor session type that reflected monitoring related heap objects, for example, all nodes in a particular data structure. *AllHeapInFunc* was chosen based on the assumption that each function, ideally, exports one or more abstractions, and that heap objects allocated by it and functions on its behalf have a reasonable chance of being logically related. For any program there are bound to be abstraction levels, i.e., functions, that are at the wrong granularity with respect to debugging. For example, monitoring all heap objects allocated on behalf of the top-level (e.g., `main`) function is an unlikely debugging request. Our expectation is that, for sufficiently large programs, a significant number of functions do create

[3] The monitor sessions are specified in high-level terms but they can easily be translated into the operations `InstallMonitor` and `RemoveMonitor` defined in Section 2.

[4] Heap objects whose size is changed via a call to `realloc` are considered to be the same object.

and manipulate related heap objects that a programmer might want to monitor.

6 Programs

We use the following C programs for our analysis:

- **GCC v1.4** An ANSI C compiler provided by the Free Software Foundation. Input was the 811 line GCC source file *rtl.c*.
- **CommonTEX v2.9 (CTEX)** An implementation of the TEX document processing system. Input was a document producing four pages of text and complex mathematical equations.
- **Spice v3c1** Computer-aided circuit analyzer. Transient analysis for a simple differential pair circuit was computed for 20ns at 5ns intervals.
- **QCD** Quantum chromodynamic simulation [Uni88]. Input was the test simulation provided in the distribution.
- **BPS** Bayesian problem solver using a tree search to arrange 8 numbers on a 3x3 grid into ascending order by sliding them in Manhattan directions using the empty grid element [HM89]. Input was an arbitrary initial grid configuration.

All programs were compiled with GCC v1.4, with the "-g" option, on a SPARCstation 2. No variables were allocated to registers. Non-ANSI features of GCC, such as inline declarations, were disabled.

For each program, the assembly code was post-processed so that at run-time a program event trace was generated. The trace consisted of the following three events and their arguments (`BA` = Beginning Address, `EA` = Ending Address):

- `InstallMonitorEvent [ObjectDesc, BA, EA]` `ObjectDesc` identifies the program object corresponding to the write monitor. This is used by the simulator to determine which write monitors are active in the current monitor session.
- `RemoveMonitorEvent [ObjectDesc, BA, EA]`
- `WriteEvent [BA, EA]`

The event trace is independent of any particular monitor session. `InstallMonitorEvent` and `RemoveMonitorEvent` events are generated for every program object that is of interest to one of our monitor sessions. Write monitors for automatic variables are installed and removed on function boundaries. System calls, standard libraries, and implicit writes (e.g., register spilling) do not appear in the trace.

7 Analytical Models

The analytical models used in our experiment are necessarily architecture and operating system dependent. For example, an accurate model must take into account whether hardware services are directly accessible by user programs. Ideally, we could base our models on an existing platform, but no readily available system meets our requirements. Our solution is to use a popular workstation, the SPARCstation 2 running SunOS 4.1.1, and estimate the cost of non-existent services in terms of existing ones.

The SPARCstation 2 does not provide any hardware monitoring registers. We logically extend the SPARCstation 2 by assuming that there are enough monitor registers for the monitor sessions that we are interested in. Our hypothetical monitor registers are readable and writable by user programs. While executing in the kernel we assume that the monitor registers are disabled, avoiding any operating system security problems. Monitor hits result in a trap which may be caught by user-level handlers via the SunOS signal mechanism. The time for a monitor hit trap is estimated to be the same as that of a virtual memory write fault for a resident page.

Our experiment relies on the SPARCstation 2 in three ways. First, it defines an architecture and a set of operating system services. Second, we use it to obtain the timing data required by our models. Third, it serves as the compilation target for our benchmark programs.

7.1 Models

We now present the analytical models used for calculating monitor session overhead. Each model consists of equations for calculating the overhead incurred installing monitors ($InstallMonitor_{ov}$), removing active monitors ($RemoveMonitor_{ov}$), handling monitor hits ($MonitorHit_{ov}$), and handling monitor misses ($MonitorMiss_{ov}$). The total overhead for a particular monitor session is simply their sum.

As stated in Section 4, the models combine simulator-generated information about a program's run-time behavior with timing data for the important primitives necessary to implement the approach. A program's run-time behavior is captured by a number of *counting variables*, which are subscripted with σ. Timing data is captured by *timing variables*, which are subscripted with τ. The models ignore secondary effects such as cache behavior, pipeline stalls, and virtual memory paging behavior.

Figure 2 lists a number of timing and counting variables used in more than one model. Three of the four implementations that we investigate must maintain a software mapping between virtual addresses and active write monitors. These approaches rely on es-

The following timing variables are used repeatedly:

$SoftwareLookup_\tau$:	Time to determine if a virtual address range intersects an active write monitor.
$SoftwareUpdate_\tau$:	Time to update the mapping when a write monitor is installed or removed.

The following counting variables are used repeatedly:

$MonitorHit_\sigma$:	The number of monitor hits.
$MonitorMiss_\sigma$:	The number of monitor misses.
$InstallMonitor_\sigma$:	The number of write monitors installed.
$RemoveMonitor_\sigma$:	The number of write monitors removed.

Figure 2: Shared timing and counting variables.

sentially the same software data structures, though small optimizations might be possible for certain platforms or certain implementations. $SoftwareLookup_\tau$ and $SoftwareUpdate_\tau$ are used to characterize this overhead. The models uniformly assume that these support data structures reside in the debuggee's address space. For convenience in further discussion, the section heading provides a name and abbreviation for each approach.

7.1.1 *NativeHardware* (*NH*)

In *NativeHardware* a monitor hit triggers a monitor register fault. The monitor hardware is accessible to user programs and we assume the cost to update it can be safely ignored. The model for *NativeHardware* is shown in Figure 3.

7.1.2 *VirtualMemory* (*VM*)

In *VirtualMemory* a monitor hit triggers a write fault. In addition to software lookup costs, *VirtualMemory* incurs additional overhead continuing past the faulting instruction. Execution is continued by unprotecting the appropriate page(s), *emulating* the faulting instruction, reprotecting the page(s), and arranging for execution to continue after the faulting instruction. Monitor misses which write to a page containing an active write monitor must go through the same process. Installing or removing a write monitor might require the permissions of the page(s) on which the monitor resides to be changed. The WMS-support data structures reside in the debuggee's address space. Whenever a write monitor is installed or removed, the appropriate page[5] of this data structure must be unprotected, updated, and reprotected. The model for *VirtualMemory* is shown in Figure 4.

[5] The model assumes that each update affects only one virtual memory page.

7.1.3 *TrapPatch* (*TP*)

TrapPatch, at compile time, replaces all write instructions with trap instructions. In the trap handler, as in *VirtualMemory*, the faulting instruction is emulated, and execution is continued after the faulting instruction. The model for *TrapPatch* is shown in Figure 5.

7.1.4 *CodePatch* (*CP*)

CodePatch, at compile time, patches the assembly code so that the target of every write instruction is checked. The check is done in a subroutine with the target address passed via an available register. The model for *CodePatch* is shown in Figure 6.

8 Results

For each benchmark program, we discovered all instances of the monitor session types described in Section 5. For example, there was a monitor session of type OneLocalAuto for each local automatic variable found in a program. Monitor sessions that had no monitor hits were discarded under the assumption that they are unlikely candidates during debugging. Table 1 shows, for each of the programs, the type and number of monitor sessions studied. In addition, Table 1 shows base program execution times. These times were obtained using the UNIX `time` command on the SPARCstation 2 described in Appendix A. This timing method had a resolution of 20 milliseconds.

The timing variable data for the SPARCstation 2 is shown in Table 2. A detailed discussion of how these numbers were obtained is provided in Appendix A.

Table 3 lists mean counting variable data for all monitor sessions of a program. The differences between $InstallMonitor_\sigma$ and $RemoveMonitor_\sigma$ are too small to warrant space in the table, and similarly for $VMProtect_\sigma$

The following timing variable is unique to *NativeHardware*:

$NHFaultHandler_\tau$: Time required to receive a user-level monitor register fault and continue execution.

NativeHardware is defined as follows:

$$\begin{aligned}
MonitorHit_{ov} &= MonitorHit_\sigma * NHFaultHandler_\tau \\
MonitorMiss_{ov} &= 0 \\
InstallMonitor_{ov} &= 0 \\
RemoveMonitor_{ov} &= 0
\end{aligned}$$

Figure 3: Analytical model for *NativeHardware*.

The following timing and counting variables are unique to *VirtualMemory*:

$VMFaultHandler_\tau$: Time required to receive a user-level write fault, emulate the faulting instruction, and continue execution.
$VMProtect_\tau$: Time required to protect a page of virtual memory.
$VMUnprotect_\tau$: Time required to unprotect a page of virtual memory.
$VMActivePageMiss_\sigma$: Number of monitor misses that wrote to a page containing an active write monitor.
$VMProtect_\sigma$: Number of times the count of active write monitors on a page changed from zero to one.
$VMUnprotect_\sigma$: Number of times the count of active write monitors on a page changed from one to zero.

VirtualMemory is defined as follows:

$$\begin{aligned}
MonitorHit_{ov} &= MonitorHit_\sigma * (VMFaultHandler_\tau + SoftwareLookup_\tau) \\
MonitorMiss_{ov} &= VMActivePageMiss_\sigma * (VMFaultHandler_\tau + SoftwareLookup_\tau) \\
InstallMonitor_{ov} &= InstallMonitor_\sigma * (VMUnprotect_\tau + SoftwareUpdate_\tau + VMProtect_\tau) \ + \\
&\quad (VMProtect_\sigma * VMProtect_\tau) \\
RemoveMonitor_{ov} &= RemoveMonitor_\sigma * (VMUnprotect_\tau + SoftwareUpdate_\tau + VMProtect_\tau) \ + \\
&\quad (VMUnprotect_\sigma * VMUnprotect_\tau)
\end{aligned}$$

Figure 4: Analytical model for *VirtualMemory*.

The following timing variable is unique to *TrapPatch*:

$TPFaultHandler_\tau$: Time required to receive a user-level trap fault, emulate the faulting instruction, and continue execution.

TrapPatch is defined as follows:

$$\begin{aligned}
MonitorHit_{ov} &= MonitorHit_\sigma * (TPFaultHandler_\tau + SoftwareLookup_\tau) \\
MonitorMiss_{ov} &= MonitorMiss_\sigma * (TPFaultHandler_\tau + SoftwareLookup_\tau) \\
InstallMonitor_{ov} &= InstallMonitor_\sigma * SoftwareUpdate_\tau \\
RemoveMonitor_{ov} &= RemoveMonitor_\sigma * SoftwareUpdate_\tau
\end{aligned}$$

Figure 5: Analytical model for *TrapPatch*.

CodePatch is defined as follows:

$$
\begin{aligned}
MonitorHit_{ov} &= MonitorHit_{\sigma} * SoftwareLookup_{\tau} \\
MonitorMiss_{ov} &= MonitorMiss_{\sigma} * SoftwareLookup_{\tau} \\
InstallMonitor_{ov} &= InstallMonitor_{\sigma} * SoftwareUpdate_{\tau} \\
RemoveMonitor_{ov} &= RemoveMonitor_{\sigma} * SoftwareUpdate_{\tau}
\end{aligned}
$$

Figure 6: Analytical model for *CodePatch*.

and $VMUnprotect_{\sigma}$. We include this summary of counting variable data so that readers wanting to change one of our models to reflect different requirements can estimate the results. Further, this data reveals the relative frequency of the various implementation primitives.

Using the models from Section 7 and the above data, we have calculated, for each program and each approach, the overhead of all monitor sessions listed in Table 1. We normalize overhead results to the base execution time of the program and refer to this as the *relative overhead.* Table 4 provides statistics about the relative overhead of all monitor sessions studied for each benchmark program and approach. For *NativeHardware*, *VirtualMemory-4K*, and *VirtualMemory-8K*, this data contains a number of extreme points. To help understand its distribution, Table 4 lists the 90th and 98th percentiles.

To help visualize some of the important results from Table 4, we have provided three graphs. The maximum and 90th percentile relative overhead over all the monitor sessions for each program is graphed in Figure 7 and Figure 8, respectively. Figure 9 graphs the mean for monitor sessions whose relative overhead is between the 10th and 90th percentiles.

As mentioned above, the relative overhead data for *NativeHardware* and *VirtualMemory* contain a number of extreme points. For *NativeHardware*, since all overhead is due to handling monitor hits, monitor sessions with high overhead correspond to frequently updated program objects. While each program had a unique set of these hot spots, the majority of expensive sessions for *NativeHardware* monitored induction variables and functions that allocated large numbers of heap objects. For *VirtualMemory*, the majority of expensive sessions monitored local variables, often for functions toward the root of the call graph.

Another important facet of the performance data is a breakdown of where the time was spent. This provides insight into how changes to the cost of various services might affect the results presented here. For each program we calculated the mean, over all monitor sessions, of the percentage of time taken by each of the operations corresponding to our timing variables. We present here a summary of that data. For *NativeHardware*, as can easily be predicted from our model, 100% of the overhead was due to *NHFaultHandler*. For *VirtualMemory-4K*, *VMFaultHandler* contributed between 86% and 97% of the total overhead. The results for *VirtualMemory-8K* were similar. For *TrapPatch*, *TPFaultHandler* consistently accounted for 97% of the overhead. Finally, *SoftwareLookup* accounted for between 98% and 99% of the total overhead for *CodePatch*.

A final note concerns the space requirements of *CodePatch*. For each write instruction, *CodePatch* must insert a call to a WMS routine responsible for detecting monitor hits. For the SPARC architecture this requires a minimum of two additional instructions. Using the percentage of write instructions found in each benchmark program we estimated the code expansion for *CodePatch*. We found that only a modest increase of between 12% and 15% is expected.

9 Conclusion

Of the approaches we studied, *NativeHardware* delivered the best overall performance. *CodePatch* was significantly more efficient than the other two approaches and, for the most demanding monitor sessions, provided better performance than even *NativeHardware*. *CodePatch* exhibited extremely low variance, a desirable user interface characteristic. For a large number of monitor sessions *VirtualMemory* was unacceptably slow. *TrapPatch* shares with *CodePatch* a low variance but was unacceptably slow for most debugging applications.

While specialized processor support is attractive, it has the overwhelming disadvantage that support for only a small number of simultaneous write monitors can be expected. Consider monitoring a large central data structure with thousands of constituent elements. Recall that no existing processor could have supported all of the monitor sessions used in our experiment. Further, given the encouraging performance estimate for code patching, expensive monitoring hardware will be difficult to justify.

Code patching is the most likely choice for providing efficient data breakpoints. It is simple, portable, and provides for any number of simultaneous write moni-

Program	**OneLocal Auto**	**AllLocal InFunc**	**OneGlobal Static**	**OneHeap**	**AllHeap InFunc**	**Execution Time (ms)**
GCC	2328	493	347	323	138	3900
CTEX	583	157	230	0	0	1067
Spice	989	161	32	416	68	833
QCD	145	21	19	0	0	2900
BPS	193	54	12	4184	33	1100

Table 1: Base program execution time in milliseconds and type and number of monitor sessions studied. Does not include monitor sessions that had no monitor hits.

Timing Variable	**Time** (μs)
$SoftwareUpdate_{\tau}$	22
$SoftwareLookup_{\tau}$	2.75
$NHFaultHandler_{\tau}$	131
$VMFaultHandler_{\tau}$	561
$VMProtectPage_{\tau}$	80
$VMUnprotectPage_{\tau}$	299
$TPFaultHandler_{\tau}$	102

Table 2: Timing variable data in microseconds.

				VM-4K		**VM-8K**	
Program	*Install/Remove* $Monitor_{\sigma}$	*Monitor* Hit_{σ}	*Monitor* $Miss_{\sigma}$	$VMProtect_{\sigma}$/ $VMUnprotect_{\sigma}$	*VMActive* $PageMiss_{\sigma}$	$VMProtect_{\sigma}$/ $VMUnprotect_{\sigma}$	*VMActive* $PageMiss_{\sigma}$
GCC	937	2231	3185039	416	32223	414	53500
CTEX	916	2141	1459769	543	35551	542	37924
Spice	98	1323	508071	55	21022	54	32119
QCD	4645	31120	3305221	2921	835091	2920	835091
BPS	37	583	559202	21	3701	21	5137

Table 3: For each program, mean counting variable data over all monitor sessions studied for that program.

Program	**Statistic**		**NH**		**VM-4K**		**VM-8K**		**TP**		**CP**	
GCC	Min	Max	0	10.45	0	102.76	0	287.90	85.61	87.94	2.25	4.58
	T-Mean	Mean	.01	.07	2.48	5.21	3.16	8.29	85.61	85.62	2.25	2.26
	90%	98%	.09	.62	15.31	37.08	17.37	37.09	85.63	85.69	2.27	2.33
CTEX	Min	Max	0	29.30	0	339.88	0	343.64	143.52	146.17	3.77	6.42
	T-Mean	Mean	.07	.26	11.77	20.78	13.03	22.05	143.53	143.56	3.78	3.81
	90%	98%	.49	2.24	48.93	116.66	48.93	117.86	143.58	143.96	3.83	4.21
Spice	Min	Max	0	27.87	0	213.52	0	223.33	64.06	65.05	1.68	2.68
	T-Mean	Mean	.01	.21	7.15	15.24	11.94	22.75	64.06	64.06	1.68	1.69
	90%	98%	.16	1.19	53.55	118.56	72.34	215.32	64.07	64.09	1.69	1.72
QCD	Min	Max	0	61.98	0	636.44	0	636.44	120.51	123.19	3.16	5.84
	T-Mean	Mean	.36	1.41	158.99	170.05	158.99	170.05	120.53	120.58	3.19	3.23
	90%	98%	2.56	15.11	459.63	636.44	459.63	636.44	120.65	120.88	3.31	3.53
BPS	Min	Max	0	28.16	0	158.96	0	158.96	53.31	53.99	1.40	2.09
	T-Mean	Mean	0	.07	.56	2.23	1.02	2.97	53.31	53.31	1.40	1.40
	90%	98%	.02	.14	2.31	14.30	4.45	18.98	53.31	53.32	1.40	1.41

Table 4: Relative Overhead Statistics. T-Mean refers to mean of monitor sessions whose relative overhead is between the 10th and 90th percentiles. 90% and 98% refer to the 90th and 98th percentiles, respectively.

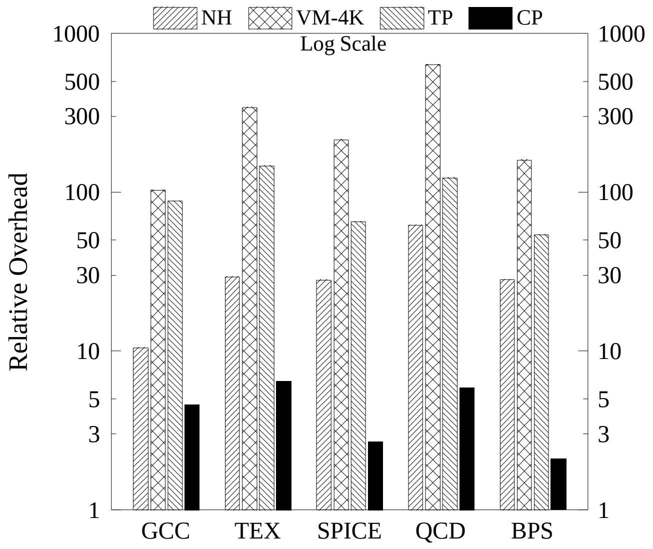

Figure 7: Maximum relative overhead over all monitor sessions.

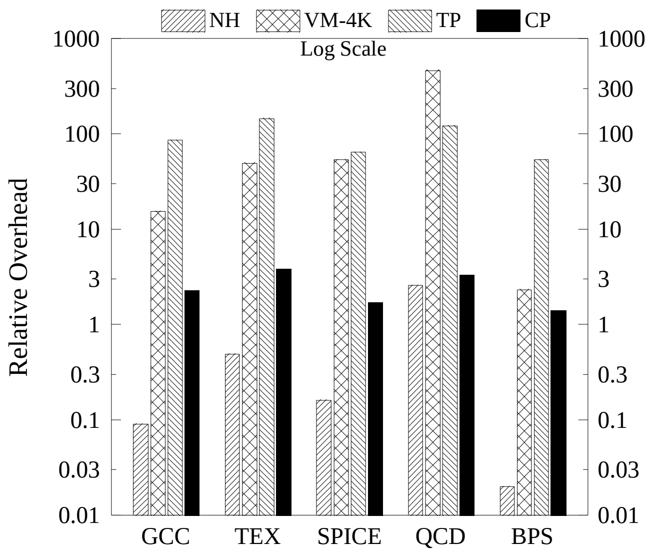

Figure 8: 90th percentile relative overhead over all monitor sessions.

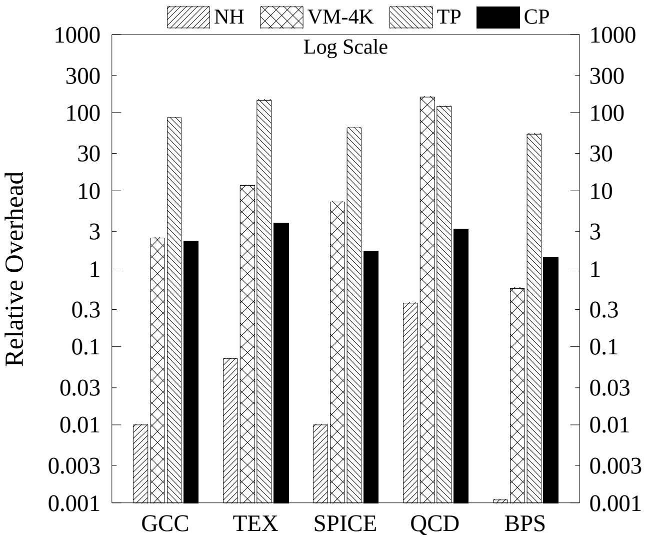

Figure 9: Mean relative overhead over all monitor sessions whose relative overhead is between the 10th and 90th percentiles.

tors. For performance reasons, code patching requires that data structures mapping virtual addresses to write monitors be maintained in the debuggee's address space. Because all write instructions are already being checked, however, protecting this data requires no additional mechanism and very little overhead. Our expectation is that the impact of having a small amount of read-only WMS data in the debuggee's address space will be acceptable.

As described and studied, the performance of code patching is acceptable. Our approach was to uniformly check the target location of every write instruction. A simple optimization reduces the overhead for candidate instructions inside loops. A preliminary check *outside* the loop may be applied for write instructions whose target is a loop-invariant memory range. If the preliminary check determines that the instruction will be a monitor hit, the loop body can be dynamically patched so that each iteration correctly results in a monitor notification. Our expectation is that this and other optimizations will significantly reduce the overhead of code patching.

This work was motivated by the need for an efficient WMS in the context of the Ensemble software development environment being built at the University of California, Berkeley. Ensemble will provide a number of novel services for managing the dynamic behavior of programs, including a sophisticated high-level debugging system called QEI.[6] An implementation of a WMS based on code patching is underway. As this technology

[6] QEI is a Latin abbreviation for the phrase "which was to be found out."

is demonstrated, our hope is that data breakpoints will be routinely supported in future debuggers.

10 Acknowledgements

I would like to thank Susan Graham, my research advisor, for her direction and support. Asawaree Kalavade took part in early discussions and helped implement a prototype of the simulator. I would also like to thank David Bacon, John Hauser, Peter B. Kessler, Steve Lucco, Bill Maddox, Ethan Munson, John Pasalis, Vern Paxson, Oliver Sharp, Ken Stanley, and Tim Wagner, all of whom provided their time and expertise.

References

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, California, April 1991. Appeared as SIGPLAN Notices 26(4).

[Bea83] Bert Beander. Vax debug: an interactive, symbolic, multilingual debugger. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 173–179, August 1983. Appeared as SIGPLAN Notices 18(8).

[CL87] T.A. Cargill and B.N. Locanthi. Cheap hardware support for software debugging and profiling. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–83, October 1987. Appeared as SIGPLAN Notices 22(10).

[Dig] Digital Equipment Corporation. *Ultrix v4.2 Ptrace Manual Page.*

[DMS84] Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz. Viewing a programming environment as a single tool. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 49–56, May 1984. Appeared as SIGPLAN Notices 19(5).

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall E. McKusick. Gprof: a call graph execution profiler. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982. Appeared as SIGPLAN Notices 17(6).

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1992 Usenix Winter Conference*, pages 125–136, 1992.

[HM89] O. Hanson and A. Mayer. Heuristic search as evidential reasoning. In *Proceedings of the Fifth Workshop on Uncertainty in AI*, August 1989.

[Int86] Intel Corporation, Santa Clara, California. *Intel 80386 Programmer's Reference Manual*, 1986.

[Joh82] Mark Scott Johnson. Some requirements for architectural support of software debugging. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 140–148, April 1982. Appeared as SIGPLAN Notices 17(4).

[Kes90] Peter B. Kessler. Fast breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 78–84, White Plains, New York, June 1990. Appeared as SIGPLAN Notices 25(6).

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC ARCHITECTURE.* Prentice Hall, New Jersey, 1992.

[KLP88] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C: an interpreter-based programming environment for the C language. In *Proceedings of the 1988 Usenix Summer Conference*, pages 161–171, San Francisco, CA, June 1988.

[MCL89] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, April 1989. Appeared as SIGPLAN Notices 24(Special Issue).

[SS91] Mark Sullivan and Michael Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 171–180, September 1991.

[Uni88] University of Illinois at Urbana-Champaign. The perfect club benchmark: Effective performance evaluation of supercomputers, November 1988.

A Timing Data

This section provides high-level pseudo-code and where necessary a short discussion for the timing data shown in Table 2. All tests were executed three times and their mean taken on an unloaded 32 megabyte 40MHz SPARCstation 2, running SunOS 4.1.1.

The code fragments below use the following data and functions:

- **WorkingSet** Two megabytes of data pages consisting of every other page of a contiguous memory region.
- **WorkingMonitorSet** 100 non-overlapping write monitors with random size and location allocated from a 2 megabyte contiguous memory region.
- **Protect(Pages, Mode)** Changes virtual memory protection for **Pages** to **Mode** via the `mprotect` system call. For each page of **Pages** an access of type **Mode** is performed.
- **TimerOn(), TimerOff()** Multiple calls to **TimerOn** and **TimerOff** are cumulative. Timing is done via the system call `getrusage` and has a resolution of 10 milliseconds on the SPARCstation 2. All results include both system and user time.
- **Read(VMPage)/Write(VMPage)** Reads/writes the first word of page **VMPage**.
- **RandYesReplace(Set), RandNoReplace(Set)** Chooses a random element from **Set** with replacement in the case of **RandYesReplace** and without replacement in the case of **RandNoReplace**. The random values are precomputed so that this operation is a simple array lookup.
- **RandInit(Set)** Used in conjunction with **RandNoReplace**. Logically initializes **RandNoReplace** state so that all elements are available from **Set**.

A.1 *NHFaultHandler*$_\tau$

```
Main:
 Protect(WorkingSet, Read)
 TimerOn()
 Iterate
   VMPage ← RandYesReplace(WorkingSet)
   Write(VMPage) /* Causes write fault. */
 TimerOff()

/* Invoked by operating system upon write fault. */
NHFaultHandler (FaultingAddr, FaultingInstr):
 SkipInstruction(FaultingInstr)
```

A.2 *VMFaultHandler*$_\tau$

```
Main:
 Protect(WorkingSet, Read)
 TimerOn()
 Iterate
   VMPage ← RandYesReplace(WorkingSet)
   Write(VMPage)  /* Causes write fault. */
 TimerOff()

/* Invoked by operating system upon write fault. */
VMFaultHandler(FaultingAddr, FaultingInstr):
 Protect(Page(FaultingAddr), ReadWrite)
 Protect(Page(FaultingAddr), Read)
 SkipInstruction(FaultingInstr)
```

A.3 *VMProtect*$_\tau$ / *VMUnprotect*$_\tau$

The numbers for *VMProtect*$_\tau$ and *VMUnprotect*$_\tau$ differ significantly. Unprotecting a page, unlike protecting it, does not require that the hardware mapping be updated synchronously with respect to the `mprotect` system call. Rather, the mapping may be updated lazily in response to a write fault on the page. This lazy updating, we conjecture, is adding to the time of *VMUnprotect*$_\tau$. Our efforts to verify this have not been successful. Because *VMFaultHandler* dominates the overhead for *VirtualMemory* (see Section 8), this idiosyncrasy does not significantly affect our results.

A.3.1 *VMProtect*$_\tau$

```
Main:
 Iterate
   Protect(WorkingSet, ReadWrite)
   RandInit(WorkingSet)
   TimerOn()
   Iterate Cardinality(WorkingSet)
     VMPage ← RandNoReplace(WorkingSet)
     Protect(VMPage, Read)
   TimerOff()
```

A.3.2 *VMUnprotect*$_\tau$

```
Main:
 Iterate
   Protect(WorkingSet, Read)
   RandInit(WorkingSet)
   TimerOn()
   Iterate Cardinality(WorkingSet)
     VMPage ← RandNoReplace(WorkingSet)
     Protect(VMPage, ReadWrite)
   TimerOff()
```

A.4 *TPFaultHandler*$_\tau$

```
Main:
 TimerOn()
 Iterate
   ExecuteTrapInstr() /* Causes trap. */
```

```
 TimerOff()

/* Invoked by operating system upon trap fault. */
TPFaultHandler (FaultingInstr):
 SkipInstruction(FaultingInstr)
```

A.5 $SoftwareUpdate_\tau$ / $SoftwareLookup_\tau$

Numbers for $SoftwareUpdate_\tau$ and $SoftwareLookup_\tau$ required us to design and implement a data structure on which to base our timing information. For each page that has an active write monitor we maintain a bitmap; each bit corresponds to a word of memory.[7] Using the page number as a key, the bitmaps are stored in a hash table.

A.5.1 $SoftwareUpdate_\tau$

```
Main:
 TimerOn()
 Iterate
    RandInit(WorkingMonitorSet)
    Iterate Cardinality(WorkingMonitorSet)
       Monitor ← RandNoReplace(WorkingMonitorSet)
       InstallMonitor(Monitor)
    RandInit(WorkingMonitorSet)
    Iterate Cardinality(WorkingMonitorSet)
       Monitor ← RandNoReplace(WorkingMonitorSet)
       RemoveMonitor(Monitor)
 TimerOff()
```

A.5.2 $SoftwareLookup_\tau$

```
Main:
 InstallMonitor(WorkingMonitorSet)
 TimerOn()
 Iterate
    Addr ← RandomAddr()
    LookupMonitor(Addr)
 TimerOff()
```

[7] This restricts write monitors to word-aligned boundaries. Higher-level clients can easily compensate for this restriction.