Practical Data Breakpoints: Design and Implementation*

Robert Wahbe† Steven Lucco† Susan L. Graham†

Computer Science Division, 571 Evans Hall
UC Berkeley, Berkeley CA, 94720

Abstract

A data breakpoint associates debugging actions with programmer-specified conditions on the memory state of an executing program. Data breakpoints provide a means for discovering program bugs that are tedious or impossible to isolate using control breakpoints alone. In practice, programmers rarely use data breakpoints, because they are either unimplemented or prohibitively slow in available debugging software. In this paper, we present the design and implementation of a practical data breakpoint facility.

A data breakpoint facility must monitor all memory updates performed by the program being debugged. We implemented and evaluated two complementary techniques for reducing the overhead of monitoring memory updates. First, we checked write instructions by inserting checking code directly into the program being debugged. The checks use a *segmented bitmap* data structure that minimizes address lookup complexity. Second, we developed data flow algorithms that eliminate checks on some classes of write instructions but may increase the complexity of the remaining checks.

We evaluated these techniques on the SPARC using the SPEC benchmarks. Checking each write instruction using a segmented bitmap achieved an average overhead of 42%. This overhead is *independent* of the number of breakpoints in use. Data flow analysis eliminated an average of 79% of the dynamic write checks. For scientific programs such the NAS kernels, analysis reduced write checks by a factor of ten or more. On the SPARC these optimizations reduced the average overhead to 25%.

1 Introduction

Breakpoints are user-specified rules that associate debugging actions with *break conditions* that arise during program execution. *Control breakpoints* specify the break condition in terms of the program's control flow, for example `stop on call to function main`. *Data breakpoints* specify the break condition in terms of the program's memory state, for example `stop when field` *f* `of structure` *s* `is modified`.

Data breakpoints support debugging tasks such as `print the value of field` *f* `of structure` *s* `every time it is updated`. To perform this task using only control breakpoints, the programmer must find all statements in the program that *might* update *f*. In the presence of pointers or reference parameters, this search is both tedious and error-prone.

At present, efficient data breakpoint facilities are not available to programmers. Using efficient runtime data structures and ideas from compiler optimization, we have developed and evaluated several methods for providing practical data breakpoints.

The key difference between control break conditions and data break conditions is in the complexity of the mapping from the break condition to the set of program instructions that can trigger the condition. We call an instruction that *may* trigger a break condi-

*This research was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under grant MDA972-92-J-1028 and contract DABT63-92-C-0026. The content of the paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

†Email: {rwahbe, lucco, graham}@cs.berkeley.edu

ACM-SIGPLAN-PLDI-6/93/Albuquerque, N.M.
© 1993 ACM 0-89791-598-4/93/0006/0001...$1.50

tion an *unsafe* instruction. For control break conditions, the mapping to unsafe instructions is one-to-one, or one-to-few in the presence of function inlining. Hence, debuggers can guarantee detection of each control break condition by monitoring a handful of unsafe instructions.

In contrast, the mapping from data break conditions to unsafe instructions is one-to-many, because the same memory location can be updated by write instructions scattered throughout a program. A debugger must either watch a memory location corresponding to a data breakpoint, or it must check each write instruction in the program that might update that memory location.

We say a write instruction is *known* if static analysis can resolve its target address. A debugger can determine the set of known instructions by inspecting target addresses prior to execution. Programs written in languages that include pointers or permit runtime type violations (e.g. out-of-bounds array indices) generally contain a large number of *unknown* write instructions. If any data breakpoint is active, *all* unknown write instructions must be considered unsafe, and must be checked at runtime.

Implementation Strategies Some commercially available processors provide direct support for data breakpoints. Examples include the Intel i386 [10], the MIPS R4000 [12], and the SPARC[16]. Special-purpose hardware can monitor memory efficiently. Unfortunately, the hardware approach inherently limits the number of data words simultaneously monitored. The Intel i386 can monitor four words; the MIPS R4000 and the SPARC can only monitor a single word.

The UNIX debuggers **gdb** [17] and **dbx** [14, 18] provide data breakpoints. Both systems conservatively assume all instructions are unsafe. The possible side-effects of each instruction are checked through dynamically inserted trap instructions. Due to context switch and trap costs, this approach incurs very high overhead. We measured the overhead of **dbx** to be a factor of 85,000, independent of the program being debugged.

VAX DEBUG provides data breakpoints using virtual memory page protection [3]. Like **gdb** and **dbx**, VAX DEBUG assumes that all instructions are unsafe. However, rather than check each instruction, VAX DEBUG protects each virtual memory page containing data that is part of a data break condition.

Magpie is a programming environment for Pascal that allows debugging actions to be associated with variable updates [6]. This functionality is implemented by inserting checks during compilation. Magpie does not support monitoring of heap objects.

To this date, no performance information has been reported for Magpie or for VAX DEBUG. Several authors have speculated that efficient data breakpoints require special-purpose hardware [4, 11, 15].

To quantify the differences among data breakpoint implementation strategies, Wahbe [19] compared facilities based on specialized processor support, virtual memory page protection, checking the destination address of machine instructions via an operating system trap, and checking the destination address of machine instructions via a procedure call. Virtual memory and the use of traps were shown to be too slow for practical use. Special-purpose hardware, while efficient, could not support all test cases due to limits on the number of memory words simultaneously monitored. Checking each write instruction via a procedure call emerged as the most promising method for realizing efficient data breakpoints. Its most significant disadvantage was the expected overhead of between 209% and 642%.

In this paper, we adopt a code patching approach to checking write instructions and significantly reduce its overhead. We investigate two complementary approaches to reducing the overhead of write checks. First, we develop an efficient data structure, the *segmented bitmap*, for checking whether an individual target address is part of a data breakpoint condition. We compare several techniques for optimizing these checks.

Second, we investigate the additional performance benefit of eliminating write checks through compile-time analysis. Our strategy for eliminating write checks has three components. First, we use a symbol table matching algorithm to find as many known write instructions as possible. We check those instructions only when the variables to which they write occur in break conditions. Second, we use data flow analysis techniques to remove checks on write instructions within loops. We replace such checks by checks that execute only once, on entry to the loop. Third, we support these loop optimizations by using data structures that provide efficient checks on contiguous *ranges* of memory locations. We demonstrate that, at the cost of considerable implementation complexity, these techniques dramatically diminish the dynamic count of checked write instructions.

The remainder of this paper has the following structure. Section 2 describes a *monitored region service* abstraction that defines the specific functionality provided by our system and outlines its implementation. Section 3 describes the segmented bitmap data structure, its optimization through caching and inlining, and its overhead for the SPEC benchmarks. Section 4 provides the details of our write-check elimination algorithms, and evaluates their effectiveness. Finally, Section 5 draws some conclusions.

```
st %o0,[%fp-20]         ! Write instruction
sub %fp,20,%g5          ! Address passed via %g5
call check_1_word,0     ! Write check procedure
nop                     ! Delay slot
```

Figure 1: Simplified example of a write check.

2 Monitored Region Service

We encapsulate the core functionality needed to implement data breakpoints within an abstraction called a *monitored region service (MRS)*. A monitored region service detects writes to contiguous regions of memory called *monitored regions*. To simplify the implementation, monitored regions are assumed to be word aligned and non-overlapping. The interface to the monitored region service consists of the following three functions:

```
CreateMonitoredRegion (MonitoredRegion)
DeleteMonitoredRegion (MonitoredRegion)
NotificationCallBack  (TargetAddress, Size)
```

It is the responsibility of the debugger to map source language names used in the break conditions to monitored regions, and to create and delete monitored regions as necessary.

If the target of a write instruction intersects a monitored region, there is a *monitor hit*, otherwise there is a *monitor miss*. The function `NotificationCallBack` is used to notify MRS clients, such as the debugger, of monitor hits.

The monitored region service does not monitor writes to registers or memory updates due to system calls. Because registers cannot be aliased, detecting writes to registers is straightforward and incurs negligible runtime overhead. A debugging system can detect memory updates due to system calls by replacing the library portion of each system call with an equivalent call that reports memory updates to the debugger.

2.1 Implementation

We now outline our basic approach to implementing a monitored region service. In the next two sections we will show how this straightforward implementation can be optimized.

Our system consists of a program analysis tool and a runtime library. The analysis tool acts as an extra processing stage between the compiler and the assembler, patching each write instruction with a function call that checks the target address to detect monitor hits. An example SPARC assembly code fragment, generated by our MRS implementation, is shown in Figure 1. In this example, the write instruction is a one word store to memory (`st`), and the target address is `%fp-20`. The runtime *monitor library* contains the data structures necessary to check whether a target address represents a monitor hit.

Since our simple MRS implementation does not monitor indirect jump instructions during program execution, it must be prepared for arbitrary control transfers. Indirect control transfers pose two potential problems for the MRS. First, as a result of data corruption, control might be transferred directly to a write instruction. To insure that all monitor hits are detected, checks are placed after, rather than before, the write instruction. Second, the program might jump into the middle of a monitor library routine. This situation is detected by maintaining a `check-in-progress` flag which is set on entry to a monitor routine and cleared on exit. The above scheme does not prevent the routine from referencing an invalid address before checking the flag. However, the MRS may use an operating system signal to detect this situation gracefully.

The write check implementations described in Section 3 and Section 4 reserve a minimum of three registers for the monitored region service. One register holds the `check-in-progress` flag described above. The MRS uses a second register to hold a global `disabled` flag. The MRS sets this flag whenever no data breakpoints are active. The code for each write check branches around the body of the check when the `disabled` flag is set, reducing runtime overhead. Finally, a third register is used by write checks to hold the target address of the write instruction.

For efficiency, the monitor library data structures are maintained in the address space of the program being debugged. The MRS creates monitored regions as necessary to insure the integrity of these data structures.

3 Write Check Implementations

In this section, we describe our basic SPARC implementation of a monitored region service, and present several possible optimizations for reducing the overhead of checking write instructions. On the SPARC, all write instructions update either one or two memory words. Because the two word write instructions are suitably aligned, one-word and two-word checks incur identical overhead for the implementation techniques that we tested. We will discuss only single-word write checks.

The basic operation that a write check performs is to determine whether a write instruction's target address references a monitored region. We call this operation *address lookup*. We found that the best strategy for implementing efficient address lookup is to minimize the average number of memory accesses required. The write checks tested in Wahbe's pilot study of data breakpoint implementations used a hash table for address lookup [19]. This data structure uses memory efficiently, consuming space proportional to the number of monitored regions. However, it requires several memory accesses for each address lookup. We tested this data structure using the SPEC benchmarks and verified that the write check overhead generally matched the 209% to 642% reported in the previous study for a different set of benchmarks.

The overhead of hash table address lookups is due mostly to the memory accesses performed in matching a target address against a list of hash table entries. The write checks described in this section use a *segmented bitmap* data structure to implement address lookup. This data structure uses one bit to represent each word allocated by the program being debugged. Each bit indicates whether or not the corresponding word is monitored. Hence a segmented bitmap consumes more space than a hash table - roughly 3% of the total memory used by the program. However, it incurs at most two memory accesses per address lookup.

Further, these memory accesses are more likely to be cached than the hash table memory accesses. Since one bitmap word represents 32 words of memory allocated by the program being debugged, and since cache lines on the SPARC contain 32 bytes, any lookup of an address within 512 bytes of a recently checked address is very likely to require only cached memory accesses.

Conceptually, the bitmap contains one bit for every word of addressable memory. To reduce its space overhead, we organize the bitmap into segments of size SEGMENT-SIZE. Segments are allocated lazily in response to monitored region installations. Right shifting the target address by log_2(SEGMENT-SIZE) bits yields its *segment number*. Segments are accessed via a *segment table*, which is an array of *segment pointers*, indexed by segment number. All segment pointers are initialized to a single zeroed bitmap segment. Figure 2 depicts a segmented bitmap.

Breaking the bitmap into segments requires an extra load instruction to index into the segment table. However, the target of this load instruction is very likely to be cached, again assuming spatial locality among checked addresses.

3.1 Reserving Registers for the Monitored Region Service

On modern RISC architectures the naive compilation typically used during debugging requires only a subset of the available registers. We can take advantage of this situation by reserving registers for use by the MRS. Optimizations based on reserved registers are less applicable to architectures, such as the i386, which have small register sets [10]. Also, as compilation for debugging incorporates more sophisticated register allocation, there will be a tradeoff between freeing an additional register for the compiler, and reserving that register for the MRS. On the SPARC we found two techniques that can benefit from reserved registers.

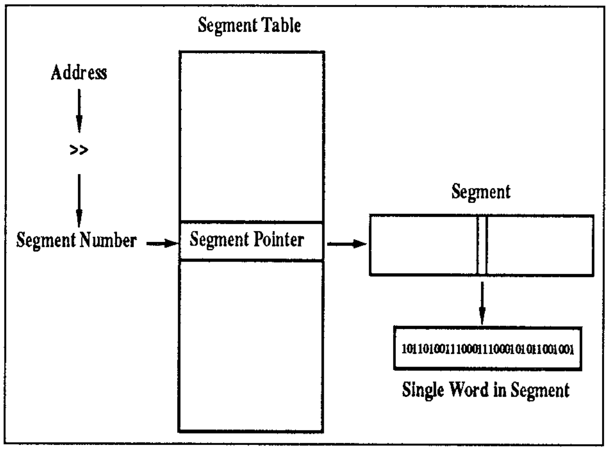

Figure 2: A Segmented Bitmap

The segmented bitmap code requires three registers to hold intermediate values during address lookup. The MRS can use reserved registers to avoid pushing a register window. The MRS can use a fourth register to hold the base address of the segmented bitmap table. Although this value is constant, it is too large to be an immediate assembly operand and therefore placing it in a register requires extra register instructions during address lookup.

A more sophisticated technique, called *segment caching*, uses registers to cache the results of previous write checks on the same bitmap segment. To support caching, each write instruction is assigned a *write type*. The goal of the write type is to identify groups of write instructions which are likely to exhibit spatial locality. For C programs, we used the write types BSS, STACK, and HEAP. All target addresses computed using the frame or stack pointer were assigned type STACK. Writes with constant target addresses were assigned type BSS; the remaining writes were assigned type HEAP. For FORTRAN, in addition to the above types, we also used the type BSS-VAR. The Sun FORTRAN compiler used a simple idiom to calculate related BSS accesses. By recognizing this idiom we were able to increase the effective cache hit rate.

For each write type we maintain a *segment cache*. The segment cache holds the segment number of the last checked segment to have no monitored regions. By keeping the segment cache in a reserved register, we can check the cache in four register instructions.

To support segment caching we must be able to determine efficiently whether a bitmap segment contains any monitored regions. We do this by maintaining, for each bitmap segment, a boolean flag `unmonitored`, indicating whether the segment has any monitored regions. By suitably aligning the bitmap segments, we can store the unmonitored flag in the unused low order bit of the corresponding segment pointer.

In addition to supporting segment caching, the unmonitored flag saves the address calculation and load of the correct bitmap segment when the segment contains no monitored regions. To support efficient creation and deletion of monitored regions in the presence of the unmonitored flag, an auxiliary data structure maintains a count of monitored regions for each bitmap segment.

The algorithm for maintaining the segment caches is as follows:

```
if SegmentNum(TargetAddress) = SegmentCache
  Done
elsif SegmentNotMonitored(TargetAddress)) then
  SegmentCache ← SegmentNum(TargetAddress)
elsif MonitorHit(TargetAddress)
  NotificationCallBack(TargetAddress, Size)
endif
```

Note that the segment cache is only updated if there is a cache miss and the new segment contains no monitored regions.

While larger segments improve segment cache locality, smaller segments reduce the number of *full lookups.* Full lookups are monitor misses that require checking the cache, the unmonitored flag, and finally the appropriate bitmap segment word. A full lookup occurs for target addresses whose bitmap segment contains monitored regions; the impact of full lookups is discussed in Section 3.3. Segment size also determines the number of segments in the segment table. For a 2^{32} byte address space, a 128 word segment size requires 1 million segments. While the segments themselves are allocated lazily, the segment table is not. Thus, to decide segment size, one must consider tradeoffs among segment cache locality, the expected number of full lookups, and the size of the segment table.

To limit table size, we restricted our choice of segment sizes to 128 words or greater. Figure 3 graphs segment cache locality as a function of segment size. Segment sizes greater than 128 words did not offer enough gain in cache locality to justify the possible increase in full lookups. Hence, all experiments reported in Section 3.3 were performed with a 128 word segment size.

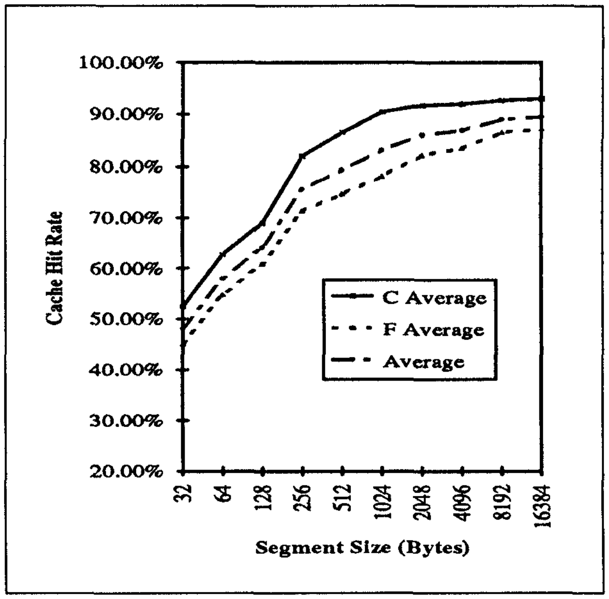

Figure 3: Segment Cache Locality

3.2 Inlining

In addition to reserving registers, we evaluated the impact of inlining write checks. On the SPARC, inlining eliminates as many as six instructions. Section 3.3 demonstrates, however, that inlining can increase overhead due to instruction cache misses. To evaluate the effectiveness of inlining, we compared inlined and non-inlined versions for both simple bitmap lookup and segment cached implementations. For segment caching, the four instructions necessary to check the cache are always inlined. The non-inlined version makes a procedure call when there is a segment cache miss.

3.3 Evaluation

Table 1 presents monitored region service overhead for the following write check implementations:

Bitmap. Address lookup executed via procedure call.

BitmapInline. An inlined version of *Bitmap.*

BitmapInlineRegisters. An inlined version of *Bitmap* that makes use of reserved registers to avoid spilling and the recalculation of address constants.

Cache. Segment caching implementation using four segment caches. On a cache miss, lookup is executed via a procedure call.

CacheInline. An inlined version of *Cache.*

In addition to the above implementations, we measured the overhead of branching around checks when

Programs	Disabled	Bitmap	Bitmap Inline	Bitmap Inline Registers	Cache	Cache Inline	σ
(C) 023.EQNTOTT	-3.2%	0.2%	-0.5%	-1.7%	-3.7%	-4.4%	2.3%
(C) 008.ESPRESSO	22.2%	70.4%	66.2%	40.4%	29.6%	22.2%	4.9%
(C) 001.GCC1.35	28.1%	75.4%	83.6%	63.1%	49.7%	53.3%	6.1%
(C) 022.LI	60.2%	128.5%	124.2%	94.8%	77.2%	62.3%	19.4%
(F) 015.DODUC	19.3%	58.6%	73.3%	45.2%	21.1%	37.8%	5.6%
(F) 042.FPPPP	33.8%	55.4%	68.7%	56.1%	41.2%	53.8%	3.3%
(F) 030.MATRIX300	7.5%	39.1%	31.8%	25.3%	15.4%	13.8%	1.1%
(F) 020.NASKER	9.2%	44.5%	40.0%	37.2%	17.2%	19.6%	1.6%
(F) 013.SPICE2G6	7.1%	30.9%	29.1%	25.1%	15.9%	15.7%	4.1%
(F) 047.TOMCATV	13.6%	44.7%	36.6%	32.5%	19.2%	27.8%	1.3%
C AVERAGE	26.8%	68.6%	68.4%	49.2%	38.2%	33.3%	8.2%
FORTRAN AVERAGE	15.1%	45.5%	46.6%	36.9%	21.7%	28.1%	2.8%
OVERALL AVERAGE	19.8%	54.8%	55.3%	41.8%	28.3%	30.2%	5.0%

Table 1: Monitored region service overhead for different write check implementations.

the `disabled` flag is set. The column labeled σ is explained below. All routines were carefully hand coded in SPARC assembly code. The standard libraries were not patched for these experiments; using **gprof** [8], we measured the percentage of time spent in library routines to be an average of 2.6% for C programs and 1.6% for FORTRAN programs, excluding the SPARC library routines for integer multiplication and division which do not update memory.

3.3.1 Cache Effects

Our measurements show a number of interesting anomalies. The most obvious are the negative overheads for 023.EQNTOTT. For a number of programs, the savings due to reserving registers is somewhat higher than we had estimated. Finally, while inlining the segment caching routine slightly improved the average overhead on C programs, inlining the simple bitmap lookup routine had essentially no effect.

We conjecture that these anomalies are due to cache effects. The SPARC used in our experiments has a direct-mapped combined instruction and data cache with 32 byte cache lines. Inserting write checks affects cache performance in two ways. First, because write checks increase the size of the code, the cache size is effectively reduced. Second, adding or moving instructions changes the alignment of code and data relative to cache line boundaries.

To evaluate the impact of caching we performed an experiment in which we inserted 2, 4, 8, 16, or 32 `nop` instructions before each write instruction. In the absence of cache effects, the overhead should be linearly dependent on the number of instructions inserted. We make the simplifying assumption that the effective cache size is also linearly reduced. For each program we performed a simple linear regression on the measured overhead for the different number of inserted `nop` instructions. Under the above assumptions, any deviation from the expected linear behavior must be caused by cache alignment effects. The last column of Table 1 shows the standard deviation of the differences between expected and observed overhead.

In two ways, these cache effects alter the conclusions we can draw from comparing different write check implementations. First, the ranking of a given approach could change given a different cache organization. Second, we must rely more heavily on average measurements over all SPEC benchmarks, as individual measurements may include anomalous overhead due to cache performance variation.

3.3.2 Inlining

Inlining had little effect on the measured overheads for our benchmark programs; overall, it slightly increased overhead. Because inlining dramatically changes the alignment characteristics of the program, the small differences observed for individual programs are not significant. We conclude that inlining write checks on the SPARC is not necessary.

3.3.3 Segment Caching

In contrast, segment caching did reduce the effective overhead of write checks. However, because full lookups may occur when monitored regions are present, the savings from segment caching is dependent on the debugging situation. If there are too many full lookups, the additional overhead incurred for checking the cache and unmonitored flag cancels the benefit of caching. To address this issue, we compared the cycle counts for *BitmapInlineRegisters* and *Cache*. *BitmapInlineRegisters* executes 12 register instructions and 2 loads. *Cache* executes 6 register instructions if there is a cache hit, 18 register instructions and 1 load if there is a cache miss, and 26 register instructions and 2 loads if there is a full lookup. Assuming that loads take between 2 - 8 cycles, the break-even point for C programs occurs when the percentage of write instructions requiring a full lookup is 24.3 - 44.0%. For FORTRAN programs, the break-even point is 16.4 - 36.7%.

Segment caching reduced overhead by an average of 13.5%. For short lived programs, we do not think the savings justify the scheme's variability. Consider that 14% overhead represents only about 50 seconds for a program that normally runs for 6 minutes.

For compute intensive applications, improving the naive compilation typically used during debugging would have more performance impact than reserving registers for segment caching [1, 2, 9]. In particular, register allocation would speed program execution, while reducing the number of write checks. Because register allocation targets scalar variables found on the stack and stack writes exhibited excellent locality in our tests, we conjecture that register allocation would reduce the effective hit rate of segment caching, narrowing the performance gap between simple bitmap lookup and segment caching.

4 Eliminating Write Checks

The implementation described in the previous section is simple but may incur significant overhead when the dynamic count of write instructions is large relative to the total instruction count. The optimizations described in this section concentrate on reducing this overhead by eliminating unnecessary write checks.

Write check elimination is based on dynamic insertion and deletion of write checks. For certain classes of write instructions, data flow analysis can determine runtime conditions under which a given write instruction is safe. For example, if a particular write instruction w can only write to a specific program variable $\mathbf{x}$, then the MRS need only check w while $\mathbf{x}$ is being monitored. Hence, the analysis tool can eliminate the check on w, and arrange for the MRS to re-insert the check at runtime upon creation of a monitored region that includes $\mathbf{x}$.

Similarly, the analysis tool can often determine that a particular write instruction w_l within a loop will update a contiguous range of memory locations. Given this information, it can arrange for the MRS to check, on loop entry, whether the range of memory locations to be updated intersects any monitored regions. If this range check succeeds, the MRS can dynamically reinsert the eliminated write check on w_l.

Kessler [13] describes a method for dynamically patching a running program. To insert a check before an instruction, the instruction is replaced with a branch to a *write check patch*. The write check patch, in addition to checking for a monitor hit, is responsible for executing the displaced instruction. At compile-time, for each write instruction, a write check patch is constructed. By having dedicated patches we insure that inserting checks is extremely efficient. On most architectures only a handful of instructions are required.

Unfortunately, the MRS must incur additional runtime overhead to support write check elimination. For example, both write check optimizations outlined above require that the MRS check all indirect jump instructions, to verify that the control flow graph used in data flow analysis matches the actual control flow of the program being debugged. Whether these additional sources of overhead cancel the benefit of eliminating write checks depends on both the target processor architecture and the type of program being debugged.

4.1 Analysis

To support write check elimination, we augmented our code patching and analysis tool to include a machine independent optimizer. As before, the analysis tool takes as input a sequence of SPARC assembly instructions. It then converts this sequence into an intermediate representation (IR) which is defined as a set of 3-address codes. In addition, the analysis tool converts symbol table entries (e.g. STAB) into IR form. The optimizer takes as input the IR, and converts it to static single assignment (SSA) form [5]. It then performs several IR transformations that eliminate checks on the target addresses of individual write instructions.

4.2 Symbol Table Pattern Matching

The first pass of the optimizer identifies known write instructions through *symbol table pattern matching*. The optimizer creates an expression DAG for each target address, matching the DAG against debugging symbol table entries. When the expression for a target address matches a symbol table expression, the optimizer eliminates the expression from the IR instruction sequence and replaces it with a *pseudo-operand*. For example, if the expression `%fp-20` matches a symbol table entry, the optimizer will replace all instances of %fp-20 with a unique variable name v. Instructions that read from %fp-20 are converted to IR move instructions with v as the source. Instructions that write into %fp-20 are converted into IR move instructions with v as the target.

This transformation has two benefits. First, it enables a substantial portion of the write checks to be eliminated, since there is no uncertainty about which variable is being written. Second, substituting pseudo-operands for target address expressions such as %fp-20 simplifies the recognition of induction variables, a necessary step in the loop optimizations described below.

During this first pass, the optimizer generates a table which is used at runtime to translate a symbol name x into a list of associated write instructions. When the monitored region containing x is created, each instruction w in this list must be patched to detect a monitor hit when w is executed. To support this procedure, the monitored region service interface exports two additional operations:

```
PreMonitor(symbol)
PostMonitor(symbol)
```

The **PreMonitor** operation performs this code patching procedure; the **PostMonitor** operation reverses it. When a break condition involving x is set, the debugger calls **PreMonitor** to patch the write instructions for x and then calls **CreateMonitoredRegion** on the memory region associated with x. The debugger must create a monitored region for x because x could be written through aliases as well as through instructions patched to detect a monitor hit directly.

To support this symbol table optimization we must check all *definitions* of registers that appear in symbol table expression DAGs. For example, whenever %fp is modified, we must ensure that it points to the correct stack frame. Hence, eliminating the *uses* of %fp will be profitable only if the number of uses of %fp in write instructions is greater than the dynamic count of its *definitions*. Further, to check whether %fp is reset to its correct value following a function return requires a pair of memory accesses to save and retrieve the correct %fp value. A dedicated register will not suffice, except for leaf procedures. Hence, checking a definition of %fp will be as expensive as checking two or three write instructions that use %fp. Further, to avoid still greater overhead (an extra load instruction for each %fp check), the MRS must reserve a register for this check. Because the %fp check reserves a register, the remaining write checks in the optimized implementation must push a register window. As a whole, the optimized implementation requires four dedicated registers.

Elimination of known write instructions also depends on the control flow of the program. For example, if the program erroneously jumps into the middle of a procedure, the %fp can contain an incorrect value. To prevent such an occurrence, we must also check all indirect jumps in the program being debugged, to ensure that they transfer control to legitimate targets.

Finally, this optimization requires compiler support for the correct treatment of exceptions. If an exception causes stack unwinding, the MRS must be notified so that it can unwind its stack of correct %fp values.

4.3 Loop Optimization

For many programs, writes performed in loops can dominate the dynamic write-count, even if loop writes make up only a small minority of the static write-count. Because of the importance of loop writes, the optimizer uses additional data flow analysis to eliminate checks for some of the writes found in loops.

The optimizer performs two loop-based optimizations: loop invariant check motion and monotonic write check elimination. First, the optimizer detects all loop invariant target addresses. It eliminates the checks for these loop invariant addresses and replaces them with write checks in a pre-header block that dominates all entrances to the loop. If one of these checks succeeds at runtime, the MRS will insert the eliminated write check within the loop.

Second, the optimizer detects write instructions that will generate a monotonic sequence of target addresses during the execution of a loop. We call such instructions *monotonic writes*. The optimizer replaces checks on monotonic writes with *range checks* in the loop pre-header. We use an efficient data structure to implement range checks. For ranges of 2^{25} bytes or less, the lookup requires at most three memory accesses. As with loop invariant checks, if a range check succeeds at runtime, the MRS will dynamically restore the eliminated write check.

To detect monotonic writes, the optimizer determines the *monotonic variables* for each loop. The value of each monotonic variable must increase or decrease monotonically during the execution of the loop.

4.3.1 Assert Definitions

To support loop optimization, the post-processor converts the SPARC condition code and conditional branch instructions into IR *assert* statements. An assert statement has the form

```
DEST1,DEST2 := ASSERT_OP SRC1,SRC2
```

where ASSERT_OP is one of the relational operators. In an assert statement, DEST1 is the same operand as

SRC1, and DEST2 is the same as SRC2. The role of the assert statement is to update the data flow information about DEST1 and DEST2 to reflect the condition code setting. The purpose of this re-definition is to determine precisely, for each use of a variable, the symbolic lower and upper bounds of the value of the variable.

4.3.2 Bound Propagation

The optimizer uses a single *bound propagation* algorithm to detect both loop invariant and monotonic writes. Bound propagation is performed once per loop. Loop nests are processed from inner to outer loops, so that checks moved out of inner loops can become candidates for further optimization. The loop being processed is called the *current loop*. Following bound propagation, the optimizer processes each write instruction in the current loop, replacing the checks on all *bounded writes* with checks in the current loop's pre-header. A bounded write is a write instruction whose target address has both an upper bound and a lower bound.

To detect bounded writes, the optimizer tags each SSA variable with *bounds* (L,U), where L represents the lower bound on the variable, and U the upper bound. L can have one of five values: L_C, L_{LI}, L_M, L_A, or $\perp$. L_C represents a lower bound derived from constants. A variable tagged with L_C is either a constant or derived from an expression DAG containing only constants. L_{LI} designates a lower bound derived from loop invariants or constants. L_M designates a lower bound derived from monotonic variables, loop invariants or constants. L_A designates a lower bound derived from assert statements, monotonic variables, loop invariants or constants. Finally, $\perp$ designates that the variable has no known lower bound. Similarly, U can have the values U_C, U_{LI}, U_M, U_A, or $\perp$.

The possible values for L are totally ordered according to the usefulness of the bounds these values represent: $L_C > L_{LI} > L_M > L_A > \perp$. For example, a bound derived only from constants or loop invariants (L_{LI}) is more useful than a bound derived from monotonic variables, constants, and loop invariants (L_M). The latter type of bound requires a range check in the pre-header of the current loop, while the former requires only a standard write check. The values for U are ordered analogously.

Before bound propagation begins, the bounds of all SSA variables are initialized. Constants and variables with constant values have bounds (L_C, U_C). Loop invariant variables have bounds (L_{LI}, U_{LI}). Members of monotonic groups have bounds $(L_M, \perp)$ or $(\perp, U_M)$ depending on whether their direction is increasing (L_M) or decreasing (U_M).

```
Def = statements that define variables
while ($Def \neq \emptyset$)
  changed = false
  remove S from Def
  new_lower_bound =
      max(LowerBound(Dest(S)),
        ComputeLowerBound(Operands(S)))
  new_upper_bound =
      max(UpperBound(Dest(S)),
        ComputeUpperBound(Operands(S)))
  if (LowerBound(Dest(S)) $\neq$ new_lower_bound)
      changed = true
      LowerBound(Dest(S)) = new_lower_bound
  if (UpperBound(Dest(S)) $\neq$ new_upper_bound)
      changed = true
      UpperBound(Dest(S)) = new_upper_bound
  if (changed)
      add all statements using Dest(S) to Def
```

Figure 4: Bounds propagation algorithm.

After this initialization step, bound propagation proceeds using the algorithm shown in Figure 4. Here $Dest(S)$ denotes the destination operand for statement S. *LowerBound* and *UpperBound* select the appropriate component of the bounds associated with the variables.

This algorithm iterates to a fixed-point. It places all statements defining SSA variables into a set Def. It then processes every statement S in Def, computing bounds for $Dest(S)$. If the bounds for $Dest(S)$ change, then all statements using $Dest(S)$ are added to Def. By using the max operator to combine the bounds on $Dest(S)$ with the bounds computed from the source operands of S, the algorithm propagates the computed bounds only when they are more useful than the current bounds for $Dest(S)$.

The *ComputeLowerBound* and *ComputeUpperBound* functions depend on the type of S. For example, the ADD and SHIFT statements require only the simple conjunction rule $l = \min(l\_src1, l\_src2)$ to compute the a new lower bound l from the two source operand lower bounds l_src1 and l_src2.

4.4 Generation of Checks

Once bound propagation has completed, the optimizer visits each write instruction in the current loop. If the target address a of a write instruction w is a bounded value, then the optimizer can replace the check on a with a check in the loop pre-header. The particular optimization performed depends on the symbolic bounds for a. Let a have bounds (l,u). Then if $l \geq L_{LI}$ and $u \geq U_{LI}$, a is loop invariant and the optimizer can re-

Program	Checks Eliminated				Checks Generated		Runtime Overhead	
	Symbol	LI	Range	Total	LI	Range	Full	Sym
(C) 023.EQNTOTT	71.9%	0.0%	0.6%	72.5%	0.0%	0.0%	0.5%	4.0%
(C) 008.ESPRESSO	23.1%	19.5%	15.4%	58.0%	0.9%	7.4%	27.8%	39.9%
(C) 001.GCC1.35	49.0%	1.3%	1.8%	52.1%	0.0%	0.8%	80.4%	109.2%
(C) 022.LI	75.9%	0.0%	0.0%	75.9%	0.0%	0.0%	89.2%	156.4%
(F) 015.DODUC	84.7%	0.1%	10.6%	95.4%	0.1%	4.6%	3.1%	80.8%
(F) 042.FPPPP	70.4%	0.0%	10.8%	81.2%	0.0%	0.0%	11.9%	39.5%
(F) 030.MATRIX300	51.7%	0.0%	48.3%	100.0%	0.2%	0.2%	0.4%	18.8%
(F) 020.NASKER	42.6%	17.3%	34.5%	94.4%	0.1%	0.2%	13.9%	26.9%
(F) 013.SPICE2G6	77.7%	0.2%	1.0%	78.9%	0.0%	0.4%	11.4%	34.4%
(F) 047.TOMCATV	70.4%	0.0%	10.8%	81.2%	0.0%	0.0%	8.2%	40.6%
C AVERAGE	55.0%	5.2%	4.5%	64.6%	0.2%	2.1%	49.5%	77.4%
FORTRAN AVERAGE	66.3%	2.9%	19.3%	88.5%	0.1%	0.9%	8.1%	40.2%
OVERALL AVERAGE	61.7%	3.8%	13.4%	79.0%	0.1%	1.4%	24.7%	55.1%

Table 2: Results of write check elimination.

place the check on a with a standard write check in the pre-header of the current loop. If $l = L_M$ and $u \geq U_A$ or $u = U_M$ and $l \geq L_A$, then a is derived from a monotonic variable and the optimizer can replace the check on a with a range check in the loop pre-header.

To generate code for the moved checks, the optimizer walks the expression DAG for a, generating statements until it reaches loop invariant or constant operands. For monotonic write check elimination, the optimizer walks the DAG twice, generating code for the lower bound and then the upper bound. If the write check for a ever succeeds during program execution, the monitored region service dynamically restores the eliminated write check inside the loop.

4.5 Implementation Complexities

In performing these transformations, the optimizer must take into account possible aliases that might affect the value of a. As the optimizer generates code, it maintains an *alias list* of all memory operands encountered while walking the expression DAG for a. The optimizer precedes the range check generated for a with a sequence of statements that create monitored regions for each address on the alias list. At all exits to the current loop, the optimizer inserts a code sequence that deletes these monitored regions. Thus, alias detection, like symbol table optimization, requires verification of program control flow. Further, it requires compiler support for notification of exceptions, as the MRS may need to delete monitored regions when an exception transfers control outside of a loop.

4.5.1 Overflow

For range checks, the optimizer must also guard against *overflow*. Overflow occurs when the monotonic variable is incremented or decremented to a value that is not in the domain of the variable's type. For example, a 16-byte signed integer might be incremented past $2^{15} - 1$, yielding a non-monotonic sequence of values. Detecting overflow requires the compiler to provide type information. The optimized code would use this type information to verify the type consistency of each sub-expression leading to a loop-optimized address.

4.5.2 Reserved Registers

The MRS implementation that uses both symbol table and loop optimization reserves five registers. The extra register beyond what is needed to support symbol table optimization is used to hold one of the two bounds computed by the range check code.

4.6 Evaluation

We evaluated the symbol table and loop optimizations in two ways. First, we provide detailed dynamic count data for write checks eliminated as a result of optimizations. Second, we measured the runtime overhead of the monitored region service for symbol table optimization and for symbol table optimization combined with loop optimizations.

4.6.1 Dynamic Write Check Counts

To compare the counts of write checks executed with and without optimization, we measured the number of checks that optimization was able to eliminate while still insuring that all monitor hits are detected. We also measured the number of dynamic write checks executed in loop pre-headers generated as a result of monotonic variable and loop invariant optimizations. Under the headings "Checks Eliminated" and "Checks Generated," Table 2 reports these results as percentages of total write instructions executed.

For seven of ten programs our optimizations were able to eliminate more than 75% of the checks. 001.GCC1.35 and 008.ESPRESSO have the lowest percentage of checks eliminated. Both programs make extensive use of C's `register` declaration. During debugging, the C compiler keeps `register` declared variables in registers. Because registers are not aliased, these declarations reduce both the need and the opportunity for optimization.

4.6.2 Expected Performance

We now turn to the expected overhead of the monitored region service. This overhead includes all uneliminated write checks and loop pre-header checks generated as a result of loop optimization. In addition, as stated earlier, the MRS must check all indirect jumps and definitions of the `%fp`.

The column "Sym" in Table 2 shows the effect of using symbol table optimization on the SPEC benchmark. Comparing the overheads from Section 3 reported in Table 1, we observe that for some programs, checking every write instruction incurs less overhead than the analysis based implementations. This is due to the added overhead of checking `%fp` definitions and control flow.

The column "Full" in Table 2 reports the overhead of monitoring the SPEC benchmarks with checks eliminated through both symbol table and loop optimization. These measurements are optimistic in that our implementation does not check for either overflow or aliases. Since all work performed for these checks is done only on loop entry or exit, overhead for this check will be insignificant for loops that have large iteration spaces.

The scientific programs in the benchmark suite gain the most from loop optimization. For these programs, the costs of checking control flow and symbol table register definitions are subsumed by the benefit of eliminating most of the write checks. However, for some system codes such as 001.GCC1.35, these costs dominate, and checking every write instruction emerges as the better choice.

5 Conclusion

Among the data breakpoint implementation methods we studied on the SPARC architecture, we believe that the best method is to check all write instructions using a segmented bitmap, reserving registers to hold intermediate values during address lookup. This implementation choice has several advantages. First, its overhead is independent of the number and distribution of monitored regions. Second, its average overhead on the SPEC benchmarks is 42%, which is small when compared to the cost of using unoptimized code for debugging. Finally, this choice simplifies both the monitor library and the assembly language analysis tool. The monitor library need not initialize and maintain data structures that support fast lookup on address ranges. The analysis tool can simply insert checks after every write instruction; it does not need to perform data flow analysis on the assembly language code.

As debugging systems evolve to support more sophisticated register allocation, the dynamic count of write instructions executed by a typical program should diminish relative to the total instruction count. This development will reduce the overhead of checking every write instruction. It may also dictate that freeing more registers for the compiler, rather than reserving them for write checks, will minimize the overhead of providing a monitored region service. This development will also decrease the importance of some write check elimination techniques. For example, an optimizing compiler will eliminate many of the same write instructions whose write checks can by eliminated through symbol table pattern matching. The majority of these instructions access local variables that, in optimized code, will reside in registers.

On processor architectures such as the i386, the dynamic count of write instructions will be far greater relative to the total instruction count than on RISC architectures such as the SPARC. Further, some applications of data breakpoints, such as detecting access anomalies [7] in parallel programs, require the monitoring of read instructions as well as write instructions. Since the dynamic count of read instructions is typically two to three times that of write instructions, the overhead of monitoring every read and write can be significant. The data flow analysis techniques outlined in Section 4 successfully address this problem by providing a means for eliminating checks on the majority of write instructions dynamically executed by the program. Straightforward extensions of these techniques will handle read instructions as well.

On the SPARC both implementation approaches yielded data breakpoint services whose overhead is low enough for practical use. In addition to supporting important debugging queries such as `stop when`

field *f* **of structure** *s* **is modified**, a practical data breakpoint service opens the door for higher level applications of data breakpoints. Data breakpoints can be combined with control breakpoints to support *fault isolation*. Using this technique, programmers can prevent a subset of their program's code from accessing a given data structure. For example, a programmer could detect corruption of library data structures such as those used by a memory allocator. Other applications of data breakpoints include access anomaly detection, data structure animation, checkpointing data for replayed execution, and support for runtime type checking. We are currently investigating several of these applications.

Acknowledgements

We wish to thank Oliver Sharp for his valuable comments on earlier drafts of this paper.

References

[1] A. Adl-Tabatabai and T. Gross. "Detection and Recovery of Endangered Variables Caused by Instruction Scheduling,". In *Programming Language Design and Implementation*, 1993.

[2] A. Adl-Tabatabai and T. Gross. "Evicted Variables and the Interaction of Global Register and Symbolic Debugging,". In *Principles of Programming Languages*, pages 371–383, 1993.

[3] B. 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).

[4] T. Cargill and B. 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).

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph,". *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] N. M. Delisle, D. E. Menicosy, and M. 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).

[7] A. Dinning and E. Schonberg. "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection,". In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 1990.

[8] S. L. Graham, P. B. Kessler, and M. K. McKusick. "An Execution Profiler for Modular Programs,". *Software–Practice & Experience*, 13:671–685, August 1983.

[9] J. L. Hennessy. "Symbolic Debugging of Optimized Code,". *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

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

[11] M. S. 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).

[12] G. Kane and J. Heinrich. *MIPS RISC ARCHITECTURE*. Prentice Hall, New Jersey, 1992.

[13] P. 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).

[14] M. A. Linton. "The Evolution of Dbx,". In *Proceedings of the 1990 Usenix Summer Conference*, pages 211–220, Anaheim, CA, June 1990.

[15] 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).

[16] Sparc International. *The Sparc Architecture Manual*. Prentice-Hall, Inc., Menlo Park, CA, version 8 edition, 1992.

[17] R. M. Stallman and R. H. Pesch. *A Guide to the GNU Source-Level Debugger*. Free Software Foundation, 4.01 revision 2.77 edition, January 1992.

[18] Sun Microsystems, Inc. *Programmer's Language Guide*, revision a edition, March 1990. Part Number: 800-3844-10.

[19] R. Wahbe. "Efficient Data Breakpoints,". In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–212, October 1992. Appeared as SIGPLAN Notices 27(9).