Low-Overhead Interactive Debugging via Dynamic Instrumentation with DISE[*]

Marc L. Corliss E Christopher Lewis Amir Roth
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
{*mcorliss,lewis,amir*}*@cis.upenn.edu*

Abstract

Breakpoints, watchpoints, and conditional variants of both are essential debugging primitives, but their natural implementations often degrade performance significantly. Slowdown arises because the debugger—the tool implementing the breakpoint/watchpoint interface—is implemented in a process separate from the debugged application. Since the debugger evaluates the watchpoint expressions and conditional predicates to determine whether to invoke the user, a debugging session typically requires many expensive application-debugger context switches, resulting in slowdowns of 40,000 times or more in current commercial and open-source debuggers!

In this paper, we present an effective and efficient implementation of (conditional) breakpoints and watchpoints that uses DISE to dynamically embed debugger logic into the running application. DISE (dynamic instruction stream editing) is a previously-proposed, programmable hardware facility for dynamically customizing applications by transforming the instruction stream as it is decoded. DISE embedding preserves the logical separation of application and debugger—instructions are added dynamically and transparently, existing application code and data are not statically modified—and has little startup cost. Cycle-level simulation on the SPEC 2000 integer benchmarks shows that the DISE approach eliminates all unnecessary context switching, typically limits debugging overhead to 25% or less for a wide range of watchpoints, and outperforms alternative implementations.

1 Introduction

Bugs (programming errors) are an unfortunate but inevitable part of the application development cycle, and debugging, the identification and repair of these errors, is a major enterprise. Although tools exist to automatically pinpoint the sources of certain classes of errors (*e.g.*, memory leaks), in general there is no substitute for interactive debugging. A user employs a *debugger* to observe a bug as it develops in order to trace it to its origin; a debugger allows a user to control the execution of an application and inspect/manipulate its state.

Since only a small subset of an application's instructions and data may be pertinent to a particular bug, debuggers typically present users with two abstractions. A *control breakpoint* (often called simply a *breakpoint*) specifies that debugging-session control should transfer to the user when the application executes a specified instruction. Similarly, a *data breakpoint* (often called a *watchpoint*) transfers control to the user when the value of a user-specified expression changes. Most debuggers allow a breakpoint or watchpoint to be made *conditional* so that control is only transferred to the user if the breakpoint/watchpoint criterion is met and a user-specified predicate is true. Breakpoints, watchpoints, and conditionals reduce the frequency of user-application interaction, easing the intellectual burden on users and accelerating the debugging process.

Unfortunately, the natural implementation of breakpoints and watchpoints can be unacceptably inefficient. For safety and simplicity, the debugger and the application it controls typically reside in different processes [18]. The breakpoint and watchpoint logic resides in the debugger, necessitating an application-debugger context switch to determine whether session control should be transferred to the user. If control does ultimately transfer to the user, the overhead of the switch becomes irrelevant. Most application-debugger context switches, however, are not masked by user interaction, and their cost is perceived as additional application latency, resulting in substantial overhead (*e.g.*, as high as 40,000 times slowdown in current commercial and open-source debuggers).

The natural solution to this problem is to inject a subset of debugger logic into the application itself [2, 14, 22, 24], obviating all (or most) unmasked application-debugger context switches. Unfortunately, this approach has major deficiencies. It is cumbersome for the debugger implementor because it requires the debugger to perform static code transformation (including register scavenging, register re-allocation, and branch retargeting). It is inefficient because the transformation process contributes to the perceived latency of the debugging session; and the transformed code is bloated, degrading instruction cache performance (although this overhead is certainly lower than that arising from unmasked context switches). Most importantly, injecting debugger code and data into an application violates the separation of application and debugger, allowing a buggy application to corrupt debugger structures or the debugger to perturb application behavior (*e.g.*, by changing the stack-frame layout), potentially resulting in dreaded "heisenbugs." Alternatively, the hardware itself may be augmented to perform some subset of the debugger's duties. The challenge here is in defining support that is both efficient and sufficiently flexible to allow arbitrarily complex and many watchpoints and conditions.

In this paper, we show that DISE (dynamic instruction stream editing) [7] may be used to implement interactive breakpoints and watchpoints, conditional and otherwise, without the above shortcomings. DISE is a proposed general-purpose hardware mechanism for dynamically customizing applications (*e.g.*, for profiling, security checking, buffer overflow detection, and code decompression [6, 7, 8, 9]). Traditionally, these customizations have been implemented either statically via expensive but flexible binary rewriting or dynamically via cheap but rigid custom hardware widgets. DISE is a hybrid that marries the flexibility of software with the low overhead of hardware. It is a hardware widget that dynamically rewrites the fetched instruction stream, feeding the execution engine an instruction stream with modified or added

[*]This work was funded in part by NSF grant CCR-03-11199. Amir Roth is supported by NSF CAREER Award CCR-0238203, and E Lewis is supported by NSF CAREER Award CCF-0347290.

functionality.

A DISE-based debugger implements breakpoints and watchpoints as productions (simple rewriting rules) that add instructions to the application as it executes. Watchpoint productions insert instructions that check whether the values of watched expressions change. Conditional breakpoint/watchpoint productions insert instructions that check whether a certain predicate is true. The inserted instruction sequences trap and initiate a context switch to the debugger only when the user is to be invoked, avoiding all unmasked context switches that are perceived as latency. The cost of DISE breakpoints and watchpoints is the bandwidth cost of the added instructions. To be sure, this cost increases with the density of breakpoints and the complexity of conditionals, but it remains comfortably lower (by many orders of magnitude) than the cost of any implementation that involves even a minimal amount of unmasked context switching. DISE also provides features (*e.g.*, a private register space) that enforce the separation of application and debugger despite the fact that it dynamically intermingles code from each.

The focus of this paper is on interactive debugging and on the breakpoint/watchpoint interface presented to the user by existing interactive debuggers. However, DISE is not specific to debugging—certainly not to *interactive* debugging—and the same techniques we describe can also efficiently implement other debugging interfaces: non-interactive ones like Purify [13] and Valgrind [19] and programmatic ones like iWatcher [25]. In this context, we make three main contributions: (i) We show that debugging primitives can be implemented non-intrusively and efficiently using *general-purpose* (*i.e.*, not debugging-specific) hardware support. (ii) We discuss and evaluate performance tradeoffs of different yet semantically equivalent uses of DISE; this is important in understanding how to best exploit the features of DISE. And (iii) we demonstrate the generality of DISE by applying it to a domain dissimilar to that of prior studies.

The next section reviews existing approaches to implementing breakpoints and watchpoints. Section 3 summarizes the DISE facility, and Section 4 describes its use in implementing efficient breakpoints and watchpoints. Section 5 compares the performance of a DISE-based approach to existing implementations.

2 Debugger Implementations

A debugging session consists of three principals: the *application* to be debugged, the *user*, and the *debugger* which serves as a mediator between the two. The user is the slowest party. Breakpoints, watchpoints, and conditionals reduce the frequency of *user transitions*—transitions from the debugger to the user and back—and can dramatically accelerate the debugging process. Conversely, they increase the frequency of *debugger transitions*—transitions from the application to the debugger and back. Debugger transitions that are not masked by corresponding user transitions are perceived as additional application latency.

When user-transition frequency is low (typically a user's goal), the aggregate latency of debugger transitions can dominate execution time. As a consequence, breakpoint/watchpoint implementations can be evaluated by the number of *spurious* (unmasked) debugger transitions they generate. The more spurious transitions, the greater the perceived overhead. There are three types of spurious transitions. *Spurious address transitions* are transitions to the debugger that occur even though watched data is not written, or equivalently no instruction tagged as a breakpoint is executed. *Spurious value transitions* apply to watchpoints only and occur when a variable in a watched expression is updated but the value of the expression is unchanged. The most common cause for this is a *silent store*, which is a store that overwrites a value with the same value [16]. *Spurious predicate transitions* apply to conditional breakpoints and watchpoints and occur when the associated predicate evaluates to false.

Below we summarize well-established implementation techniques employed by widely-used debuggers. Techniques still under active researched are discussed in Section 6.

Single-stepping vs. trap-handling. The naïve breakpoint and watchpoint implementation relies on *single-stepping*. The application transfers control to the debugger after every instruction (or source-level statement), and checks whether any of the currently active breakpoints or watchpoint criteria are satisfied before single-stepping to the next instruction. Single-stepping is terribly inefficient, causing many spurious address transitions. Unfortunately, even debuggers that support superior implementations (see below) often resort to it. For example, Microsoft's Visual Studio 6.0 debugger uses single-stepping when watching global variables.

Trap handling is an attractive alternative that avoids many spurious address transitions. The debugger registers a trap handler with the operating system and configures either the application or the processor to generate a trap when an instruction (datum) at a particular address is executed (written). The fast breakpoint and watchpoint techniques that are implemented in modern debuggers all use this approach. Note, while there are straightforward mechanisms for trapping on address-based events, there are no such mechanisms for trapping on events related to values. As a result, trap handling solutions only reduce spurious address transitions. There are no currently used debugger techniques that eliminate spurious value and predicate transitions.

Breakpoint techniques. The standard trap handling solution for breakpoints uses static binary transformation to temporarily replace intended breakpoint instructions with explicit trapping instructions [18]. This implementation has excellent performance characteristics. It induces no spurious address transitions and it degrades application performance only when the breakpoint is encountered. Alternatively, some architectures (*e.g.*, x86, IA-64, PowerPC) provide breakpoint registers. The debugger loads these registers with the addresses of intended breakpoints; the processor traps when an instruction whose PC matches one of these addresses is about to commit. Breakpoint registers are convenient, but typically there are only a few of them. If the number of breakpoints required is larger than the number of hardware registers, the previous techniques are used for the remainder.

Watchpoint techniques. Hardware registers can also be used to implement watchpoints. The debugger loads these with the addresses of the variables in the watched expression, and the processor traps on a store to any of these addresses. For example, GNU's gdb 5.3.90 supports hardware watchpoint registers on Linux/x86 (notice that when setting some watchpoints, gdb prints the message "Hardware watchpoint 1"). Again, the drawback of hardware watchpoint registers is their limited number. IA-32 has four and these also serve as breakpoint registers, IA-64 also has four, PowerPC has one, and some architectures like SPARC and Alpha have none. While four watchpoint registers may sometimes suffice, the user may wish to watch multiple expressions, multiple distinct pieces of data appearing (*e.g.,* in a complex expression or representing a linked data structure), or a single large piece of data like a structure or an array. IA-64 addresses the latter shortcoming by allowing low-order bits to be ignored during matching, letting a single register watch a larger memory segment. However, this is not a general solution.

If the number of watched addresses exceeds the number of hardware watchpoint registers, the virtual memory system can be harnessed to generate traps on writes to certain addresses [1]. Here, the debugger uses an interface like `mprotect()` to remove the write permissions from the page on which the watched address resides. The virtual memory implementation can be used to watch an unlimited number of addresses, but at the cost spurious address transitions. Spatial data locality makes it likely that frequently written non-watched data resides on the same page as watched data.

Virtual memory and hardware registers can easily implement watchpoints provided that all addresses referenced by the watched expression can be statically calculated by the debugger. Addresses generated by indirection (*e.g.,* pointer dereferences or dynamically indexed array elements) cannot be statically determined. To watch an indirect expression `*p`, the debugger could watch the base address `p` then update the `*p` watch condition whenever the value of `p` changes. However, we know of no commercial debuggers that actually implement this. Instead, they resort to (highly inefficient) single stepping. In gdb, for example, a request to watch a pointer variable `p` elicits the message "Hardware watchpoint." A similar request to watch `*p` yields the message "Watchpoint."

3 DISE

Dynamic instruction stream editing (DISE) is a recently-proposed facility for implementing *application customization functions* (ACFs) like profiling, security checking, and dynamic code decompression. Complete descriptions of DISE are available elsewhere [7]. Here we give an overview, with an emphasis on those features most useful for debugging.

Overview. Traditional ACF implementations either embed code into the application's static executable via binary rewriting, or provide ACF functionality on custom (potentially programmable) hardware. The two approaches have complementary sets of advantages and drawbacks. Software ACFs are flexible but inconvenient, may have unintended interactions with the application, and degrade application performance by "stealing" both pipeline bandwidth and instruction cache capacity from it. Hardware ACFs have little or no performance overhead, but are rigid.

DISE is a hybrid: a hardware widget that performs binary rewriting. In contrast with static rewriting, DISE rewrites the dynamic instruction stream rather than the static executable. A DISE user specifies an ACF as a set of *productions* (rewriting rules). At runtime, the *DISE engine* takes an unmodified application instruction stream produced by the fetch unit, inspects and potentially rewrites each instruction, and feeds the execution engine a new instruction stream enhanced with ACF functionality. DISE's position between fetch and execution means that its ACFs have no "static" cost—they do not occupy instruction cache space, and there is no startup latency involved with actually rewriting the executable—yet can modify application behavior, not just observe it.

Mechanically, the DISE engine is similar to facilities in IA-32 processors for expanding CISC instructions to RISC microinstruction sequences [11, 12]. However, while the latter replaces coarse-grain instructions with fine-grain instructions

```
T.OPCLASS==load & T.RS==sp        ldq r4, 32(sp)
  ⇒ addq T.RS1, 8, dr0            ...becomes...
     T.OP T.RD, T.IMM(dr0)        addq sp, 8, dr0
                                  ldq r4, 32(dr0)

            (a)                          (b)
```

Figure 1. Production example (a) and its use (b).

that provide the same functionality, DISE logically replaces instructions with instructions of the same granularity that provide different functionality. System-wise, the DISE engine is wrapped in two layers of abstraction. A physical *DISE controller* virtualizes the engine's internal format and capacity. The operating system restricts access to the controller to enforce a simple safety policy: applications can create productions to apply to their own code streams without restriction, but only "trusted" entities may create/modify productions that act on other applications.

Basic feature set. DISE's basic functionality is *instruction pattern matching and parameterized instruction sequence replacement*. A pattern may specify any aspect of a single instruction: PC, opcode, register, *etc.* An instruction that matches a pattern (called a *trigger*) is replaced by the corresponding replacement sequence. Replacement sequences are parameterized, *i.e.*, they are templates in which some instruction fields are literal and others are instantiated using fields from the replaced trigger.

Figure 1 shows a contrived DISE production that adds eight bytes to the address of every load that uses the stack pointer as its base address (all our examples use an Alpha-like assembly language; the right-most operand names the target). In part (a), the production specifies the pattern to include all loads whose base address is the stack pointer. The replacement sequence consists of two parameterized instructions. Here, **T.OP**, **T.RD**, **T.RS1**, and **T.IMM** are template directives to replace the corresponding holes with the opcode, first source register, destination register, and immediate field, respectively, from the trigger. Part (b) of the figure shows how a particular load is expanded by this production.

A pattern specification considers only one instruction, so DISE can only perform "peephole" transformations. However, many ACFs have efficient peephole formulations, and DISE has several features that facilitate the orchestration of global behavior from peephole expansions. One such feature is evident in Figure 1: **dr0** is a DISE register accessible only to replacement instructions. The DISE registers can store temporary values within a replacement sequence or communicate values from one dynamic replacement sequence to a future one. They give ACFs fast local and global storage without forcing them to save/restore or reserve application registers.

DISE control flow. A second feature that simplifies ACF implementation is subtle and useful enough to merit a more detailed discussion. DISE replacement sequences support a replacement-internal program counter, DISEPC. In a DISE-enabled pipeline, instructions are associated with a ⟨PC:DISEPC⟩ pair, where PC is the PC of the trigger and DISEPC is the index of the replacement instruction within its sequence (0 for unexpanded instructions). The DISEPC exists in the microarchitecture only and serves two functions: (i) it makes replacement sequences precise from an interrupt standpoint and (ii) it enables control flow within replacement sequences. DISE replacement sequences can be full-fledged miniature programs, complete with function calls.

Replacement sequences contain two kinds of control transfers. Conventional control transfers change the PC to some ⟨newPC:0⟩. DISE control transfers change the DISEPC only, *i.e.*, to ⟨samePC:newDISEPC⟩. DISE does not support jumps to ⟨newPC:nonzeroDISEPC⟩, preserving the abstraction that expansions are self-contained within individual instructions.

DISE control transfers (*i.e.*, those that change DISEPC) are implemented differently than conventional control transfers. Because replacement sequences are not fetched, DISE control transfers are not predicted. Similarly, the DISE engine expands replacement sequences in full, with no knowledge of DISE control flow. Since DISE control transfers are all effectively "predicted not-taken," they are implemented using the mis-prediction recovery mechanism: the pipeline is flushed after the mis-predicting instruction and fetch resumes at ⟨samePC:newDISEPC⟩. The fetch engine does not recognize the DISEPC annotation and fetches the instruction at samePC. However, the DISE engine does and begins expanding the instruction at newDISEPC. The bottom line is that DISE control transfers offer the functionality of control flow, but they incur performance penalties in the form of pipeline flushes per taken branch. This is an important consideration when designing replacement sequences, as we will see in the next section.

DISE also allows replacement sequences to call functions. The semantics of a DISE function call appear strange at first, but they are consistent with DISE's overall relationship with the application and have a simple implementation. A DISE function call at ⟨PC:DISEPC⟩ to ⟨newPC:0⟩ saves the return address ⟨PC:DISEPC+1⟩, triggers a pipeline flush, and restarts fetch at ⟨newPC:0⟩. The called function is composed of conventional instructions which are fetched from instruction memory and whose branches are predicted. The function returns to ⟨PC:DISEPC+1⟩, triggering a second flush and returning to the replacement sequence from which it was called. DISE itself is disabled within the body of a function called from within replacement sequences. This again preserves the notion that replacement sequences are self-contained within application instructions and prevents bottomless recursion. DISE calls are useful for implementing replacement sequences with complex control flow. By embedding complex control flow within a call, a pipeline flush is incurred only on call and return rather than on every taken branch. Three new instructions are available to DISE-called functions: **d_mfr** (DISE move from register), **d_mtr** (DISE move to register), and **d_ret** (DISE return). The first two allow the called function to access DISE registers (analogously to **mfc0** and **mtc0** in MIPS), and **d_ret** returns from a DISE-called function and re-enables DISE expansion. These instructions may only be executed by instructions called from a DISE production, so non-DISE code is unable to access DISE registers.

4 Debugging with DISE

The high cost of watchpoints and conditional breakpoints in conventional debuggers is primarily due to the fact that the application and debugger reside in separate processes. Reducing the overhead of these vital primitives in a significant way requires embedding pieces of the debugger—address matching and condition-testing logic—into the application itself. As obviously beneficial as this approach is, existing debuggers do not use it because it is cumbersome (requiring register scavenging, register re-allocation, and branch retargeting), inefficient (due to code bloat), dangerous (because a buggy application may corrupt debugger state), and intrusive (because the extra debugger code may perturb application behavior, *e.g.*, by changing stack-frame layout). DISE-based implementations realize the benefits of injecting debugger logic into the application, without these problems.

In this discussion, it is important to remember that the DISE productions are automatically generated by the debugger (using templates) in response to the user's setting of breakpoints and watchpoints. We are not relying on the user to manually program the correct productions, so the debugging session is vulnerable to production errors to the same extent that it is vulnerable to errors in any other part of the debugger.

4.1 Breakpoints

There is little need to use DISE to implement unconditional breakpoints, because the static binary-transformation implementation is straightforward enough and performs well [18]. Nevertheless, there are two ways of doing so.

The first way parallels the binary transformation approach. The breakpoint instruction is replaced with a DISE *codeword*, an instruction with a reserved opcode whose only purpose is to match a DISE pattern and trigger an expansion. The replacement sequence consists of a trapping instruction followed by the original instruction. This implementation is actually more efficient than the conventional rewriting one, because it does not require a three step procedure—restore original instruction, single-step, re-install trapping instruction—to restart the application. The second way parallels the hardware breakpoint register mechanism. Here, the replacement sequence is the same, but it is triggered by a PC pattern specification rather than by a DISE codeword. As with breakpoint registers, the latter approach only supports a small, fixed number of breakpoint addresses.

4.2 Watchpoints

The DISE watchpoint implementation parallels single-stepping. In DISE, watchpoint productions match and replace stores. The replacement sequence varies in length depending on the number and complexity of the watched expressions. A naïve production for watching a single static (at least within the current scope) variable consists of the five instructions appearing in Figure 2a: (i) the original store (**T.INST**), (ii) a load of the watched variable from a statically-calculated address stored in DISE register **dar**, (iii) a comparison of the previous value stored in DISE register **dpv** and the current value **dr1**, (iv) a DISE branch (distinguished by the **d_** prefix) that skips one instruction if the values match, and (v) a trap. The sequence branches over the trap if the expression does not change in value.

Optimization I: Conditional trap. The preceding production works correctly but often performs poorly. Recall, DISE branches by flushing, refetching the original program instruction and re-expanding starting at a new DISEPC. This implies a pipeline flush on every store that does not change the value of the watched expression. As stores typically make up about 10-15% of the dynamic instruction stream, this is a costly proposition.

This cost can be overcome using simple ISA support: a *conditional trap* (**ctrap**) instruction. With this extension, the new production (Figure 2b) has one fewer instruction because the branch and trap are fused, but its primary advantage is that it avoids frequent flushing. Note, the conditional trap instruction is not necessarily an extension to the processor's "conventional" ISA—it doesn't provide any significant advantages to conventionally fetched code—but rather only to the DISE

```
T.OPCLASS==store
  ⇒ T.INST
    ldq dr1, 0(dar)
    cmpeq dr1, dpv, dr1
    d_bne dr1, +1
    trap
```
(a)

```
T.OPCLASS==store
  ⇒ T.INST
    ldq dr1, 0(dar)
    cmpeq dr1, dpv, dr1
    ctrap dr1
```
(b)

```
T.OPCLASS==store
  ⇒ T.INST
    lda dr1, T.IMM(T.RS1)
    bic dr1, 7, dr1
    cmpeq dr1, dar, dr1
    d_beq dr1, +1
    d_call dhdlr
```
(c)

```
T.OPCLASS==store
  ⇒ T.INST
    lda dr1, T.IMM(T.RS1)
    bic dr1, 7, dr1
    cmpeq dr1, dar, dr1
    d_ccall dr1, dhdlr
```
(d)

```
# prolog / save regs t0-t3
. . .
lda t0, glob_ptr
ldq t1, 0(t0) # get watch addr
ldq t2, 8(t0) # get old val
ldq t1, 0(t1) # get current val
cmpeq t2, t1, t2 # change?
bne t2, skip # no, continue
stq t1, 8(t0) # yes, update cur val
trap # and trap to debugger
skip:
# epilog / restore regs
. . .
d_ret
```
(e)

```
T.OPCLASS==store
  ⇒ lda dr1, T.IMM(T.RS1)
    srl dr1,11,dr2
    subq dr2, dseq, dr2
    beq dr2, error
    T.INST
    bic dr1, 7, dr1
    cmpeq dr1, dar, dr1
    d_ccall dr1, dhdlr
```
(f)

Figure 2. Example implementations of single watchpoint in DISE. (a) Naïve, (b) optimized (I), (c) optimized (II), (d) optimized (III), (e) Sample expression evaluation function, and (f) memory-protecting production.

ISA which is much more easily extended because its interface is virtual.

Optimization II: Address match gating. Another source of inefficiency in the naïve production is the load of the watched variable. Loads are expensive because data cache access is high latency and low bandwidth. Replacing every store with a replacement sequence that includes a load—or multiple loads if multiple expressions or complex expressions are watched—increases load port contention and may degrade performance.

The solution to this problem mirrors the virtual-memory and hardware-register techniques. Rather than always re-evaluating the watched expression, the replacement sequence first examines the store address. The expression is only re-evaluated if the store address matches a watched address. The expression re-evaluation is performed in a separate handler routine rather than inlined into the production. The new production (Figure 2c) uses a DISE call to jump to the handler routine. Load contention is reduced and performance improved because an expensive load is replaced by a cheaper address comparison.

Unless care is taken, address matching can miss "partial" read/write overlaps, *e.g.*, a long (4-byte) store to the lower half of a watched quad (8-byte) variable. Therefore, when the sizes of the watched and stored data differ, the address of the smaller must be aligned with that of the larger (via logical-bit-clear **bic** in Figure 2c). For instance, when watching a byte and storing a quad, the watched address is quad aligned. Conversely, when watching a quad and storing a byte, the store address is quad aligned.

Optimization III: Conditional call. The observant reader will notice a familiar problem with the production in Figure 2c. Like the first production, this production contains intra-sequence control flow, which will result in a pipeline flush! Similar problems call for similar solutions. Here we replace the load-bypassing branch with a *DISE conditional call* (**d_ccall**) to a debugger-generated function which evaluates the watched expression. With this formulation, the replacement sequence for a store (Figure 2d) is: (i) the original store, (ii) an ALU operation that re-constructs the store address from the base address register and immediate, (iii) potentially a logical-bit-clear operation to align either the store address or the watched address, (iv) a comparison of this address to the watched address which is stored in DISE register **dar**, and (v) a conditional call to a debugger-generated function (the address of which is in DISE register **dhndlr**) which is rarely taken.

Debugger-generated function. The final watchpoint implementation (above) requires the debugger to dynamically generate a function and add it into the application's text segment. Full-blown linking is not necessary, since the function is only called from within replacement sequences and does not call any application functions. The debugger encodes the address of this function into a dedicated DISE register.

The particulars of this function are somewhat unusual. First, since calls to it are not set up by the application's compiler, the function cannot use the normal calling convention; instead it treats all registers as callee-saved. Second, recall that DISE itself is disabled inside the function. A benefit of this restriction is that it allows us to be certain that the DISE registers will persist across function calls. In addition to the debugger-generated function, the debugger appends a number of values to the application's static data segment. This new memory region contains address(es) referenced by watched expressions and the expressions' current values. When evaluating expressions, the debugger-generated function statically indexes this region to access base addresses. Figure 2e shows the function that accompanies the DISE productions of Figures 2c or 2d.

Protecting the debugger's embedded data. By adding a temporary copy of some of its own data to the debugged application's virtual address space, the debugger makes this data vulnerable to corruption by a buggy application. Often this is not a problem, as the debugger's data region is small and the application itself naturally contains no pointers into it. However, the DISE mechanism can be used to provide an extra layer of protection. Specifically, the same productions that test store addresses against watched addresses can also test them against the debugger's own data region (similar to software-based fault isolation [23]). The production in Figure 2f augments that in Figure 2d to branch to an error handler if the store refers to an address in the debugger's 2KB data segment (the 21 high order bits of which are specified by the DISE register **dseg**). DISE-based protection has the virtue that all executed code (even that which is unavailable to the debugger, *e.g.*, dynamically generated code or shared libraries) is monitored. In general, the watchpoint productions may be combined with any other DISE productions, allowing, *e.g.*, compressed [8] or profiled [6] code to be debugged.

Multithreading DISE function calls. A taken DISE function call requires two pipeline flushes, one on the call itself and one on the return. The conditional call instruction means this cost is incurred only when a watched address is written, but the aggregate cost can be high for frequently written watchpoints. We can eliminate this cost by adapting a technique that was previously proposed to reduce the cost of short exception-handling routines like TLB miss handlers, avoiding the pipeline flushes that implement program/exception-handler/program serialization by executing the exception handler on another thread context in a multithreaded processor [26]. We simply execute the body of a DISE-called function on a separate thread. The mechanics of the technique are quite similar to, and actually simpler than, those of the previously proposed mechanism. Modified retirement logic provides global in-order retirement for the now-segmented main application thread and exception-handler/function-body thread. Unlike the previously proposed scheme, which must support precise application-handler communication via the exception registers, a DISE function body only communicates with the application thread via memory. This means that correct data dependences can be established by a simple extension to the function thread's store queue pointer; register renaming is not modified. iWatcher uses a similar technique to reduce the cost of its function calls [25].

Watching multiple addresses. Our optimized replacement sequence matches the current store's address to the watched variable's address as a preliminary test that avoids the potentially more expensive expression value test. For scalar, single-address expressions, the watched address is stored in a DISE register. In general, a user will set watchpoints on multiple or complex expressions that require comparisons of the current store's address to multiple watched addresses.

There are efficient ways of implementing multiple matches. If there are fewer watched addresses than available DISE registers, serial comparison is used. Otherwise, if the watched addresses are in a small range—for instance, the user may be watching a structure, all the elements in a small array, or several nearby variables—the replacement sequence checks the store's address against the upper and lower bounds of the region rather than individual addresses in it. Finally, if the number of watched addresses is both large and sparse, the debugger sets up a watched address bitmap similar to a Bloom filter [3]—in which zeros indicate definite negatives, and ones indicate only probable positives—in its static data region and hashes each store address into this bitmap. The last technique may trigger some spurious calls to the debugger-generated function, but these should be compensated for by the simplified address checking sequence. In general, because the replacement sequence is a piece of software, it may use any address comparison strategy whatsoever.

Pattern matching optimizations. In addition to replacement sequence optimizations, the debugger may also modify the watchpoint pattern specification to trigger on only a subset of the stores. For example, if all of the watched variables are either in the static data segment or the heap, the debugger can choose not to expand stores to the stack by specifying two patterns: a pattern for stores to the stack which expands to the original store, and a pattern for stores in general which expands to the watchpoint replacement sequence. DISE semantics dictate that the most specific pattern overrides all other applicable patterns.

The same technique cannot be used if only stack variables are watched because the stack can be accessed indirectly through registers other than the stack pointer. However, the debugger may choose to activate and deactivate the watchpoint expansion when the program enters or leaves the corresponding function's scope. The debugger can set an efficient hook to the scope entry and exit points by setting breakpoints on the function's first and last instructions.

4.3 Conditionals

With support for DISE calls to debugger-generated functions, implementing conditional watchpoints is trivial. The condition itself is compiled into the debugger-generated function and guards the trap. The conditional breakpoint implementation is somewhat trickier than the conditional watchpoint case. For conditional breakpoints—which do not require cheap address tests to bypass the more expensive condition test—it often makes sense to compile the condition into the replacement sequence directly. In this case, one or two dedicated DISE registers are used as temporaries to evaluate the conditional expression from auxiliary information in the debugger's static data area.

4.4 Discussion

DISE is a less cumbersome, less intrusive, safer, and better performing form of binary transformation. With the help of DISE, the debugger does not need to modify the application binary, except in two well-defined and simple ways, *i.e.*, appending a dynamically-generated function and small data region to the application's text and data segments, respectively. All would-be modifications to existing code are performed via DISE productions, transparent to the application itself, leaving the application statically unchanged and unbloated. DISE is also far less intrusive than any approaches that transform the static image of the program. It does not change the placement of any code or data, and it does not impact many hardware performance counters. Finally, the DISE mechanism itself can also be used to ensure that debugger structures embedded in the application are protected from buggy applications.

On the other hand, hardware-assisted debugging (*e.g.*, via watchpoint registers) can have nil overhead if there are no spurious value or predicate transitions, that is if all breakpoints and watchpoints are unconditional and if all writes to watched addresses also change values of the corresponding expressions. While DISE's overhead is not zero in this scenario, it is low as we will see in the next section. DISE's advantage over hardware-assisted debugging manifests if writes to watched addresses do not always change values of watched expressions or if conditional breakpoints and watchpoints are used, in which case DISE's small, constant overhead will be smaller than that resulting from expensive spurious transitions.

5 Experimental Evaluation

We use cycle-level simulation to measure the overhead of the DISE implementation of watchpoints and compare it to the overhead of four existing watchpoint implementations: source statement single-stepping, trap handling based on the virtual memory system, trap handling based on hardware watchpoint registers, and static code transformation via binary rewriting. Our experiments focus on conditional and unconditional watchpoints. Unconditional breakpoints have a widely-used "ideal" implementation via static binary transformation. Conditional breakpoints exhibit cross-implementation performance trends relative to unconditional breakpoints that are similar to the trends exhibited by conditional watchpoints relative to unconditional ones. We experiment with several different kinds of watchpoints—scalar, array, and complex expression—and also compare the mechanisms based on their effectiveness at supporting multiple watchpoints. Finally, we perform a sensitivity analysis on the DISE implementation,

Table 1. Benchmark summary.

	function	instructions	IPC	store density
bzip2	generateMTFValues	1828109152	2.45	19.8%
crafty	InitializeAttackBoards	18546482	2.39	10.8%
gcc	regclass	18016384	1.90	9.68%
mcf	write_circs	1847332	0.33	16.2%
twolf	uloop	2336334	1.87	13.7%
vortex	BMT_TraverseSets	205690692	2.25	17.6%

Table 2. Watchpoint write frequency (per 100K stores).

	HOT	WARM1	WARM2	COLD	INDIRECT	RANGE
bzip2	24805.7	193.4	∼0	0	24805.7	193.4
crafty	6531.4	3308.4	6.7	.4	6531.4	72.8
gcc	454.8	223.7	.2	.1	454.8	8197.9
mcf	11229.8	1168.4	215.4	0	11229.8	0
twolf	1467.4	227.5	101.4	80.8	1467.4	250.6
vortex	7290.3	27.6	27.6	∼0	7290.3	.4

evaluate the benefit using multithreading to lower the DISE overhead, and measure the cost of using DISE to protect debugger structures embedded in the application.

Simulator. Our performance simulator is built using SimpleScalar's Alpha AXP ISA and system call definition modules [5]. We model a dynamically-scheduled 4-way superscalar processor with a 12-stage pipeline, 128-entry re-order buffer, and 80 reservation stations. The simulated processor has an 8K entry hybrid branch predictor, 2K-entry BTB, and uses intelligent load speculation. The on-chip memory system is composed of 32KB 2-way set-associative instruction and data caches, 64-entry 4-way set-associative instruction and data TLBs, and a 1MB, 4-way set associative L2. Main memory has 100 cycle access latency and is sufficiently large that paging is never necessary. The memory bus is 32 bytes wide and operates at $\frac{1}{4}$ processor frequency. The DISE engine is modestly configured (32-entry pattern table and a 512-instruction 2-way set-associative replacement table [7]). The simulator extracts all nops from the dynamic instruction stream at no simulated cost.

Benchmarks and watchpoints. We perform our experiments on the SPEC2000 integer benchmarks, which we compiled for the Alpha EV6 using GNU gcc 3.2 with the debugging-appropriate optimization flags *-O0 -g* (although the topic of debugging optimized code is an interesting one, it is beyond the scope of the present work). For each benchmark, we use the GNU *gprof* profiler to identify a statically large and long running function, which we simulate in its entirety. For each benchmark, Table 1 gives properties of the functions we use.

We use a combination of source-code inspection and profiling to select six watchpoints for each benchmark. The first four watchpoints are scalar variables (two heap and two locals) whose written-to frequency ranges from frequently (HOT) to occasionally (WARM1/WARM2) to rarely (COLD). The fifth is a dereference (INDIRECT), and the sixth is a non-scalar, like a structure or an array (RANGE). INDIRECT actually refers to the same storage as HOT, but through a pointer. The use of multiple watchpoints allows us to measure debugging overhead under a range of conditions, but they consume presentation space. To compensate, we show only a representative subset of the SPEC2000 benchmarks. Table 2 shows the written-to frequency of each watchpoint, normalized by the total number of stores.

Experimental methodology. Presenting results that can be meaningfully interpreted and easily compared requires that we: (i) simulate the same number of instructions for each experiment, (ii) factor "user latency" out of our measurements, and (iii) realistically model the cost of debugger transitions and the debugger itself even though our simulator executes only user-level code.

We satisfy these requirements by modeling user transitions and their accompanying debugger transitions (*i.e.*, non-spurious debugger transitions) as having zero cost. We model the cost of spurious debugger transitions by flushing the pipeline and stalling for 100,000 cycles. This figure is a conservative estimate of the actual cost as measured in existing debugger implementations. Using the IA-32 cycle-level timer—via the `rdtsc` instruction—we measure the round-trip debugger transition latency for two debuggers: GNU's gdb 5.3.90 under Linux and Microsoft's Visual Studio 6.0 under Windows XP. On a 3 GHz Pentium 4, this latency is 290,000 cycles for gdb and 513,000 cycles for Visual Studio.

5.1 Unconditional Watchpoints

Figure 3 measures debugging overhead—execution time relative to an undebugged application—for a single unconditional watchpoint.

DISE. The DISE implementation dynamically inserts three or four instructions (depending on the data sizes of the watchpoint and store instruction) after every store, regardless of its address. While these increase the dynamic instruction count by as much as a factor of three, performance overhead rarely exceeds 25%. The added instructions are all ALU instructions and they do not add to the application's critical path. Nevertheless, overhead can be high for frequently written watchpoints (*e.g.*, HOT/*bzip2* and HOT/*vortex*), requiring frequent flushing when the expression-evaluation function is called. HOT/*mcf* is also frequently updated, but its cost is masked by the memory latency which dominates this benchmark.

Single stepping. Single-stepping is clearly the worst performing implementation, producing slowdown factors of 6,000 to 40,000 times in many cases. These figures are consistent with the observed performance behavior of real debuggers.

Virtual memory. The virtual memory implementation has virtually no overhead for some watchpoints (*e.g.*, COLD/*bzip2*), but for many others (*e.g.*, COLD/*twolf* and COLD/*vortex*) its overhead can be quite high, sometimes equaling the slowdown of single stepping (*e.g.*, WARM1/*bzip2*). This erratic behavior is due to the coarse (page) granularity of the address-matching mechanism and the frequency with which unwatched addresses that reside on the same page as a watched address are written. If most writes to the page are to the watched address, perceived overhead is low. Conversely, if a watched address shares a page with unwatched, frequently-written addresses, many spurious address transitions will result and overhead will be high. Certainly, page size can impact the number of spurious transitions, with smaller pages producing fewer. Our page size is 4KB, on the small end for real systems. Our experiments (not shown) indicate that reasonable overhead is achieved for these watchpoints only for impractically small page sizes (*e.g.*, 128 bytes).

Finally, notice that there is no virtual memory experiment for the INDIRECT watchpoint. The debugger cannot statically determine what pages to write-protect for a watchpoint expression containing pointer dereferences because the value of the pointer may change during execution. It is possible to watch the pointer itself and dynamically update the page

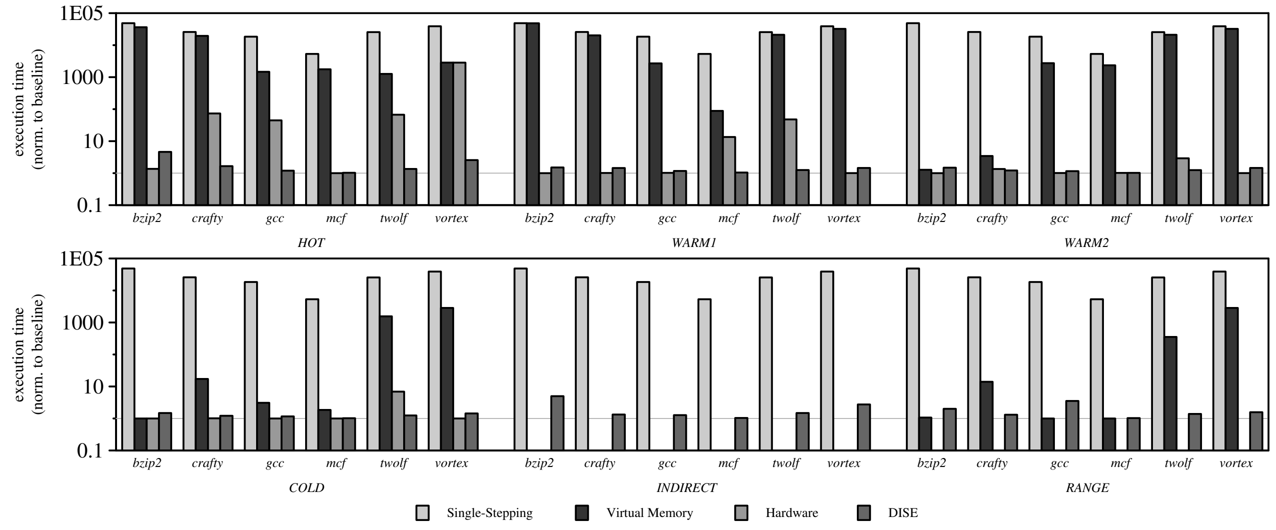

Figure 3. Comparison of four unconditional watchpoint implementations.

protection for the datum to which it points (in which case the overhead would be similar to the HOT case), but we are aware of no debugger that does this.

Hardware watchpoints. Unlike virtual memory watchpoints, hardware register watchpoints are quad-granularity and only result in spurious address transitions when a partial quad is watched and a different part of the same quad is written. Unfortunately, hardware watchpoints are still susceptible to spurious value transitions caused by silent stores. If these occur with any frequency, performance can be significantly impacted. For example, in all HOT benchmarks—save *bzip2*—50% or more of all stores to the watched address do not change the data value, resulting in significant perceived slowdowns. This is a realistic scenario, because silent stores are common [16] and watchpoints are appropriate for determining exactly where such data are actually changed.

Like virtual memory, there is no hardware register experiment for the INDIRECT watchpoint, because a debugger cannot statically determine the address to monitor. In contrast with virtual memory, there is also no experiment for the large watchpoint RANGE. Hardware registers are principally used to watch scalars. For non-scalars like structures and arrays, real debuggers resort to using virtual memory or single-stepping. Some hardware watchpoint register implementations (*e.g.*, IA-64) allow larger segments of memory to be monitored by masking low-order bits during address comparison, but this may result in spurious address transitions.

Static transformation. Watchpoints may also be implemented via binary rewriting [22, 24]. In Section 4 we argued that this approach is cumbersome, intrusive, and dangerous; but it is also inefficient, both in terms of the startup cost to perform the transformation and the instruction cache cost, which we illustrate here. Figure 5 gives the COLD watchpoint overhead of a binary-rewriting-based watchpoint implementation in which the code of Figure 2c is statically inlined at every store (*i.e.*, no static optimization is performed). Note that this graph does not include the additional overhead of performing the static transformation. We examine COLD watchpoints because they represent a common usage scenario and highlight the difference between DISE and binary rewriting. Both presented implementations have comparable performance (ignor-

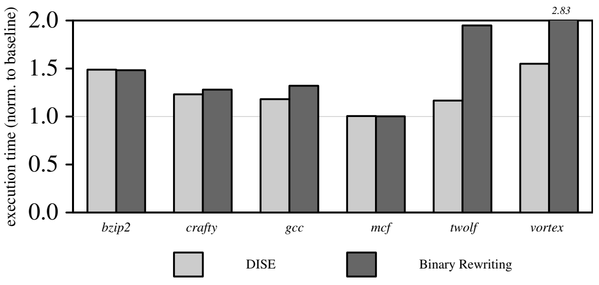

Figure 5. Comparison to binary rewriting.

ing static transformation cost) for benchmarks with small instruction memory footprints (*e.g.*, *bzip2*, *crafty*, and *mcf*). For larger programs (*e.g.*, *gcc*, *twolf*, and *vortex*) the additional instructions in the static image degrade instruction cache performance considerably. We exclude figures for binary-rewriting-based implementations from our other graphs because these results are governed by code size and instruction cache performance, *not* watchpoint characteristics.

Summary. For single, unconditional watchpoints, DISE has low overhead, generally 0–25%. It also significantly outperforms virtual memory on all indirect watchpoints and direct ones that share pages with unwatched, frequently-written data. It outperforms hardware debugging registers for large and indirect watchpoints and watchpoints with a non-negligible number of silent stores. It is comparable to a binary rewriting implementation for codes with small instruction working sets and superior otherwise.

5.2 Conditional Watchpoints

The performance benefits of DISE are even more pronounced for conditional watchpoints. Of the four implementations, it is the only one that can avoid spurious predicate transitions by evaluating conditions in the application itself. Figure 4 compares the overheads of the four implementations on single, conditional watchpoints. Aside from the condition, these are the same watchpoints used in the previous experiment. To model a realistic condition which significantly reduces the number of user transitions, our predicate compares the value

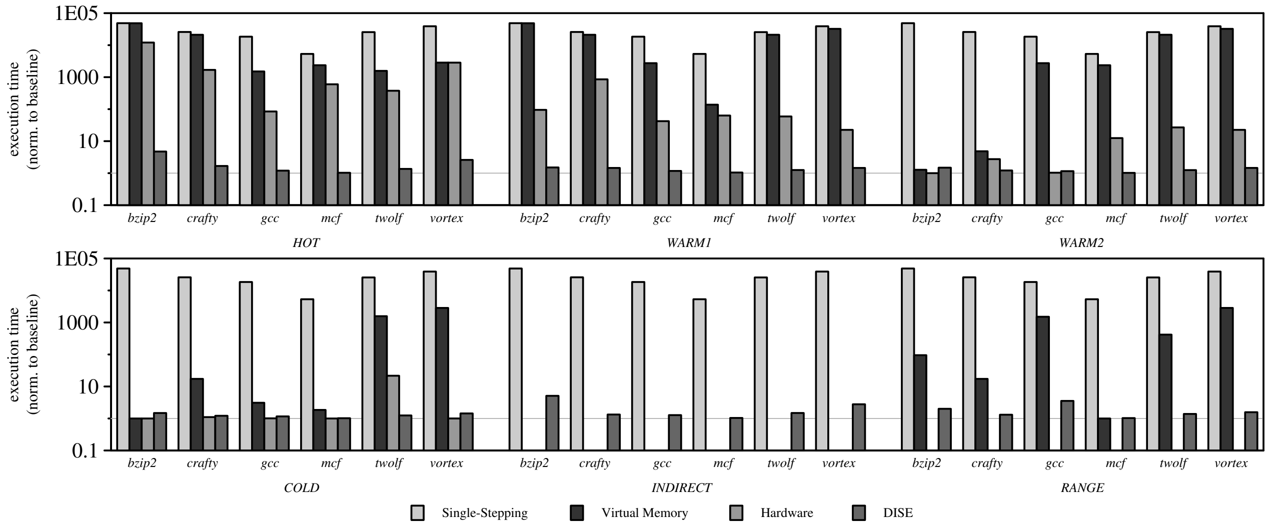

Figure 4. Comparison of four conditional watchpoint implementations.

of the watched expression to a constant it never matches.

The use of conditionals does not change DISE's relative advantage over single-stepping. Conditional or not, single-stepping incurs a debugger transition on (approximately) every store while DISE incurs transitions only when they lead to user transitions.

DISE's overhead as compared to that of the virtual-memory and hardware-register implementations depends on the frequency with which the watchpoint address is written. DISE adds a small fixed amount of overhead per store, regardless of its address. Virtual memory (modulo false address positives) and hardware registers add a much higher cost, but one that is proportional to the number of writes to the watched address. For infrequently-written watchpoints (*e.g.*, COLD/*bzip2* and COLD/*gcc*), they trigger few spurious predicate transitions and slightly outperform DISE. HOT watchpoints and many WARM watchpoints trigger many spurious predicate transitions, making DISE's constant low overhead seem insignificant by comparison.

We can compute the rough store frequency crossover point from the ratio of the cycle cost of DISE replacement sequence to the cycle cost of a debugger transition. Let us assume that DISE watchpoints add one cycle per store and that a debugger transition costs 100,000 cycles. Hardware registers and virtual memory will have lower overheads on conditional watchpoints whose addresses are written to by fewer than one of every 100,000 stores (less than 1 in Table 2). Otherwise, DISE will have the advantage. From Figure 4 it is clear that DISE is always competitive with and usually superior to the alternative implementations of conditional watchpoints.

5.3 Number of Watchpoints

With respect to performance, DISE faces strong competition in only limited scenarios. For unconditional scalar watchpoints (admittedly the most common kind) it may be outperformed, albeit not significantly so, by a hardware register mechanism. However, even here DISE has an advantage in that it can easily support multiple watchpoints with constant low overhead, while the hardware mechanism is limited by the number of watchpoint registers.

In Figure 6, we vary the number of watchpoints for both DISE and a hardware register mechanism. The hardware mechanism uses virtual memory for every watchpoint after the fourth. For DISE, we examine three replacement sequence implementations. *Serial-Address-Match* matches each address serially. *Bytewise-Bloom* hashes store addresses to bytes in a 2KB array, similar to a Bloom filter [3]; a byte value of 1 indicates a probable match and triggers a DISE function call; false positives impact performance but not correctness. *Bitwise-Bloom* hashes quad addresses to bits, increasing effective array size by a factor of eight. This results in fewer false positives, but requires two extra bit-manipulation operations to access the array.

As long as it can use hardware registers and not fall back to virtual memory—*i.e.*, there are four or fewer watchpoints—a hardware mechanism will often slightly outperform any DISE implementation. Again, a large number of silent stores on any of the watchpoints can change this dynamic, as is seen for *vortex* at four watchpoints. Once virtual memory must be used, however, a single watchpoint that occupies the same page as unwatched, frequently-written data will cause spurious address transitions to spike along with overhead. With multiple watchpoints, the probability that such a watchpoint is included in the set is high. For 5, 8 and 16 watchpoints, all three DISE implementations outperform the hardware mechanism by at least three orders of magnitude.

Note that our experimental methodology, which discounts user transitions and their accompanying debugger transitions, results in some anomalous-looking—but not actually anomalous—virtual memory results. When going from five watchpoints to eight on *gcc*, the slowdown drops from 4127 to 4088. One of the three new watchpoints resides on the same page as the fifth watchpoint. With only five watchpoints, writes to this address trigger spurious address transitions. When this address is watched, the transitions triggered by its writes are no longer considered spurious, and they are assigned no cost in our experiment.

Across the DISE implementations, the dominant effect is the efficiency of the replacement sequence. For one or two watchpoints, *Serial-Address-Matching*—which avoids costly loads—is the best approach. However, the length of this replacement sequence increases linearly with the number of watchpoints. Despite the fact that it contains a load, the constant length Bloom filter replacement sequences are more effi-

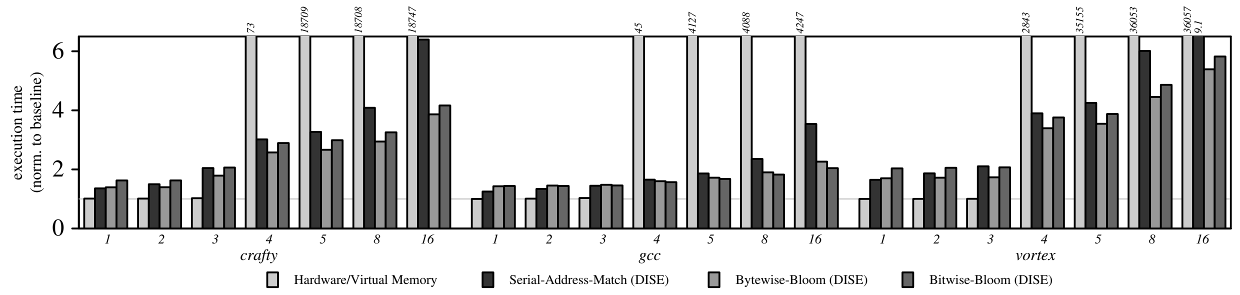

Figure 6. Impact of number of watchpoints.

cient for three or more watchpoints. The bytewise Bloom filter performs better then the bitwise version in almost all cases, as the shorter replacement sequence compensates for the cost of a few additional false positives (each of which incurs two pipeline flushes and the execution of a short function). The exception is *gcc* where the bitwise filter outperforms the bytewise one for three or more watchpoints. Here false positives dominate. For 16 watchpoints, the bytewise filter incurs 30,000 false positives (with 40,000 true hits) as compared to 100 false positives for the bitwise filter. The important point is that for a (relatively) large number of watchpoints *any* DISE approach is superior to a hardware-register/virtual-memory combination. Furthermore, the DISE implementations are much less data dependent (*i.e.*, they have good *and* predictable performance).

5.4 Implementation Effects

Below we evaluate the performance impact of several variations of the DISE watchpoint implementation.

Varying ISA support. Our DISE experiments to this point use a replacement sequence that contains a cheap address check and a conditional DISE call (see Section 4). Here we investigate the impact of the conditional call instruction and the call itself.

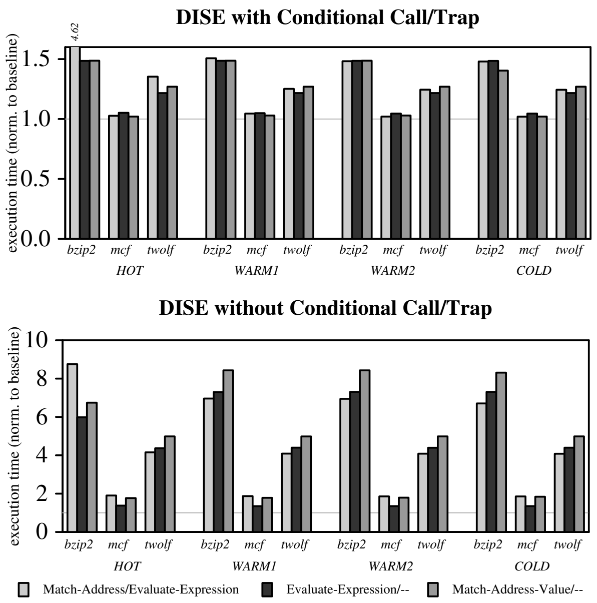

Figure 7. Alternate DISE implementations.

Figure 7 shows the unconditional watchpoint overheads of six different versions of the DISE replacement sequence/function combination. The six versions are divided into two groups. In the top group, conditional calls and traps are used to avoid common-case pipeline flushes. In the bottom group, these instructions are not available and the same functionality is instead implemented using a combination of conditional branch and unconditional call/trap, which elicits flushes in the common case. Within each group, three alternative implementations are presented. *Match-Address/Evaluate-Expression* matches addresses in the replacement sequence and calls a function to re-evaluate the expression on a match (Figures 2c and d). This has been our default. *Evaluate-Expression/–* evaluates the expression in the replacement sequence directly (Figures 2a and b), forgoing the address match and obviating the need for a function call. *Match-Address-Value/–* matches the store's address to the watched address and its value to the previous value of the expression. This is tantamount to evaluating the expression without the cost of a load if (i) the watched expression is a scalar, and (ii) the data size of the watched scalar and the store are the same (*e.g.*, both quads or both bytes).

Not surprisingly, the unavailability of conditional calls and traps (bottom graph) results in considerably higher overhead, regardless of the replacement sequence/function organization. The lesson is clear: intra-replacement-sequence control transfers should be avoided even at the expense of executing more instructions.

When conditional calls and traps are available (bottom graph) and the number of pipeline flushes is kept to a minimum, second order effects can be observed. For instance, with frequent flushing, the *Evaluate-Expression/–* implementation often has the highest overhead even though it executes the fewest additional instructions. The key is that one of the added instructions is a load, and load bandwidth is often highly contended. *Match-Address-Value/–* often has the lowest overhead, requiring neither pipeline flushes nor replacement sequence loads. Unfortunately, this implementation can only be used in select cases.

There are exceptions, however, arising from the trade-off between the cost of a load and the cost of an address-match-induced function call. For instance, for frequently-written watchpoints, *Evaluate-Expression/–* (despite its load) can be more efficient than *Match-Address/Evaluate-Expression*. This is the case for HOT/*bzip2*, which under *Match-Address/Evaluate-Expression* triggers a function call on 25% of all stores, resulting in a slowdown factor of 4.62. For watchpoints written this frequently, direct expression evaluation in the replacement sequence is a better alternative.

Except in extreme cases like the one described above, DISE implementations are not particularly sensitive to the frequency

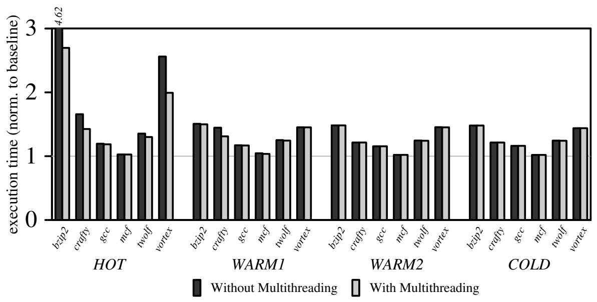

Figure 8. DISE overhead with multithreading.

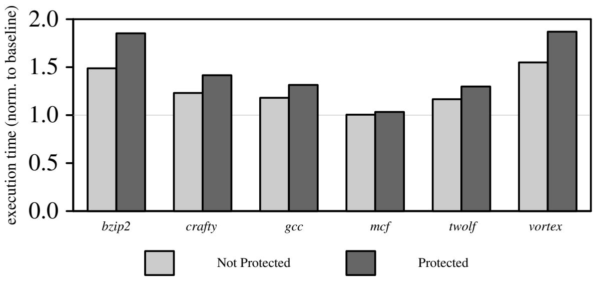

Figure 9. Cost of protecting debugger structures.

with which a variable in a watched expression is updated. Again, DISE is a form of ultra-lightweight single-stepping. Like single-stepping, it has roughly constant overhead. The difference is that this overhead is quite low. It is important to note that the overhead of all of these alternatives (even the worst among them) is orders of magnitude lower than worst-case overhead for each of the other non-DISE implementations.

Exploiting multithreading. A major component of the overhead of DISE comes from the pipeline flushes necessary to call and return from a function from within a replacement sequence. Figure 8 shows the benefit to DISE of the multithreading optimization described in Section 4. Watchpoints with relatively little overhead (*e.g.*, most of the WARM and COLD ones) benefit very little, because the flushing cost is already a minor performance factor. The HOT watchpoints that have many address matches (resulting in many function calls) naturally exhibit a significant reduction in the overall overhead (by nearly a factor of two for *bzip2*).

Protecting debugger structures. A virtue of a DISE-based implementation of watchpoints is that debugger logic is *dynamically* embedded into the running program, preserving the logical separation of the application and debugger. Unfortunately, an errant program can corrupt the debugger data structures (*e.g.*, the Bloom filter). Using DISE, we can naturally solve this problem by checking the legality of addresses referenced by all store instructions (as described in Section 4). Figure 9 plots the overheads of watching a COLD expression with and without protecting debugger data structures. We evaluate COLD expressions in order to illustrate the maximum additional cost, for the overhead of hotter watchpoints would mask the additional address-checking overhead. Nevertheless, the protection contributes only a modest additional overhead.

6 Related Work

Several lines of research relate to ours.

Embedding debugging logic into the application. The high cost of context switching that results from keeping the debugger and debugged application in separate processes has been observed numerous times. Several systems (propose to) move some debugging logic into the debugged application's process to reduce the number of context-switches. In Parasight [2], the debugger shares an address space with a shared-memory parallel application. Kessler moves debugger logic into a serial application to reduce the cost of conditional breakpoints [14]. An intended conditional breakpoint is replaced with a jump to a custom code snippet that evaluates the condition before trapping to the debugger or jumping back to the application's original control path. Wahbe *et al.* extend this work to include watchpoints [22, 24] by replacing stores with calls to an address matching routine. Unlike Kessler's system, which uses simple in-place rewriting, Wahbe's requires—and benefits from—wholesale re-compilation in order to prune unnecessary calls. Re-compilation cost is high for large applications, but individual functions may be re-compiled on demand using just-in-time infrastructures like DELI [10] or Dyninst [4].

Our implementation follows the embedding approach, but has several important advantages. These derive from DISE's advantages as compared to traditional static rewriting. First, DISE is much less intrusive than static rewriting. The presence of DISE registers means that there is no need to scavenge registers from the application, and the fact that instructions are expanded after fetch means that there is no need to retarget application branches around inserted code and that inserted code does not reduce effective instruction cache capacity; rewriting systems that insert code out-of-line using "trampolines" [14] eliminate the need to retarget branches but still require register scavenging and expand the instruction footprint. Similarly, watchpoints and breakpoints can be enabled and disabled quickly by activating and de-activating the proper DISE productions, without modifying the executable. Non-intrusiveness begets safety. Because they primarily use different register and PC spaces, the application and debugger are less likely to interfere with each other than they would if combined statically. The DISE mechanism can also be used to ensure that application stores do not corrupt debugger structures.

Reducing context-switch cost. An alternative to eliminating context-switches is to reduce their cost. Thekkath and Levy propose hardware modifications that allow traps to vector directly into user code [21].

Valgrind. Valgrind is a popular tool that has been applied to profiling and debugging x86 programs [19]. Valgrind is a basic-block interpreter/dynamic compiler with an instrumentation interface similar to those supplied by static rewriting packages like Atom [20], EEL [15], and Etch [17]. Non-interactive (and we suppose interactive) debugging features can be implemented in Valgrind by registering the appropriate instrumentation functions. Unlike a conventional interactive debugger, Valgrind forces the user to write debugging code. Unlike our DISE implementation, its performance is quite poor [25]. Even without instrumentation it induces slowdown factors of four; basic instrumentation—like instruction counting—can increase this factor to 25. In addition, the Valgrind runtime system perturbs much of the processor state, including registers, caches, and hardware performance monitors.

iWatcher. iWatcher [25] is a recently-proposed hardware-assisted debugger. There are two aspects to iWatcher. The first is a programming interface for registering with the processor pairs of "interesting" memory regions and fixed-interface callback functions; when a program writes to (or reads from) a registered memory region, the processor arranges for the registered function to be called with arguments describing the access supplied by the hardware. The second is hardware support for efficiently executing this interface, including a hierarchal implementation of a memory region tracking table and an adaptation of thread-level speculation for serializing the function call within the execution of the program at low cost (our multithreading technique is a lighter-weight version of this). Our work relates primarily to the implementation aspect. Here, while iWatcher relies primarily on "hardware," *i.e.*, tables and comparators, DISE provides the same support using what is in effect lightweight software, *i.e.*, injected instructions. We could easily replace the iWatcher implementation with DISE—(almost) anything one can do in hardware can also be done in software—with comparable performance. The iWatcher implementation would have a slight performance advantage for infrequently-modified watched regions as DISE's instruction overhead (while low) may still be noticeable. For more frequently-modified watched regions, the DISE implementation would have an advantage because DISE can prune many spurious value and predicate transitions without making a function call whereas iWatcher cannot. DISE has the additional advantage of not being debugging specific.

7 Conclusions

The conventional implementation of debuggers—as processes separate from the debugged application—makes the implementation of breakpoints and watchpoints costly. The typical debugging session will contain many expensive application-debugger context switches that do not ultimately transfer session control to the user but are necessary to evaluate expressions and predicates in the debugger. These can slow down the application by factors of 40,000 or more.

In this paper, we propose to avoid expensive and unnecessary context-switching by embedding the debugger's breakpoint, watchpoint, and conditional logic into the application itself. Most debuggers avoid this approach because it is practically cumbersome, has the high initial overhead of analyzing and transforming the application, may introduce "heisenbugs," and is potentially unsafe. The novel aspect of our proposal is that we perform the embedding without these problems using DISE (dynamic instruction stream editor), an ultra-lightweight hardware facility that transforms an application's dynamic instruction stream rather than its static image. We find that for most watchpoints and all conditional breakpoints and watchpoints, DISE's performance advantage is significant. Its slowdown versus undebugged code is usually less than 25% and is always modest, while that of a conventional debugger can be four orders of magnitude worse.

References

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proc. of 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Apr. 1991.

[2] Z. Aral, I. Gertner, and G. Schaffer. Efficient debugging primitives for multiprocessors. In *Proc. of 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 87–95, Apr. 1989.

[3] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *CACM*, 13(7):422–426, Jul. 1970.

[4] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Intl. J. of High Performance Computing Applications*, 14(4):317–329, 2000.

[5] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin–Madison Computer Sciences Department, 1997.

[6] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: Dynamic instruction stream editing. Technical Report MS-CIS-02-24, University of Pennsylvania, Jul. 2002.

[7] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *Proc. 30th Intl. Symp. on Computer Architecture*, Jun. 2003.

[8] M. L. Corliss, E. C. Lewis, and A. Roth. A DISE implementation of dynamic code decompression. In *Proc. of Conf. on Languages, Compilers, and Tools for Embedded Systems*, pages 232–243, Jun. 2003.

[9] M. L. Corliss, E. C. Lewis, and A. Roth. Using DISE to protect return addresses from attack. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Oct. 2004.

[10] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A new run-time control point. In *Proc. of 35th Intl. Symp. on Microarchitecture*, pages 257–268, Nov. 2002.

[11] K. Diefendorf. K7 challenges Intel. *Microprocessor Report*, 12(14), November 1998.

[12] P. Glaskowsky. Pentium 4 (partially) previewed. *Microprocessor Report*, 14(8), August 2000.

[13] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors in C and C++ programs. In *Proc. of Winter 1992 USENIX Conf.*, pages 125–138, Jan. 1992.

[14] P. B. Kessler. Fast breakpoints: Design and implementation. In *Proc. of Conf. on Programming Language Design and Implementation*, pages 78–84, Jun. 1990.

[15] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proc. of 1995 ACM SIGPLAN Conf. on Programming Languages Design and Implementation*, June 1995.

[16] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proc. 27th Intl. Symp. on Computer Architecture*, pages 182–191, Jun. 2000.

[17] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proc. of USENIX Windows NT Workshop*, August 1997.

[18] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architectures*. John Wiley and Sons, 1996.

[19] J. Seward. Valgrind – A GPL'd system for debugging and profiling x86-linux programs. Web Page: *http://valgrind.kde.org*.

[20] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1994.

[21] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *Proc. of 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, Oct. 1994.

[22] R. Wahbe. Efficient data breakpoints. In *Proc. of 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 200–212, Oct. 1992.

[23] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of 14th ACM Symp. on Operating Systems Principles*, December 1993.

[24] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: Design and implementation. In *Proc. of Conf. on Programming Language Design and Implementation*, pages 1–12, Jun. 1993.

[25] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proc. 31st Intl. Symp. on Computer Architecture*, pages 224–235, Jun. 2004.

[26] C. Zilles, J. Emer, and G. Sohi. The use of multithreading for exception handling. In *Proc. 32nd Intl. Symp. on Microarchitecture*, pages 219–229, Nov. 1999.