How to do a million watchpoints: Efficient Debugging using Dynamic Instrumentation

Qin Zhao[1], Rodric Rabbah[2], Saman Amarasinghe[3],
Larry Rudolph[4], and Weng-Fai Wong[1]

[1]National University of Singapore, [2]IBM Research,
[3]Massachusetts Institute of Technology, [4]VMware, Inc.

Abstract. Application debugging is a tedious but inevitable chore in any software development project. An effective debugger can make programmers more productive by allowing them to pause execution and inspect the state of the process, or monitor writes to memory to detect data corruption. This paper introduces the new concept of *Efficient Debugging using Dynamic Instrumentation* (EDDI). The paper demonstrates for the first time the feasibility of using dynamic instrumentation on-demand to accelerate software debuggers, especially when the available hardware support is lacking or inadequate. As an example, EDDI can simultaneously monitor millions of memory locations without crippling the host processing platform. It does this in software and hence provides a portable debugging environment. It is also well suited for interactive debugging because of its low overhead. EDDI provides a scalable and extensible debugging framework that can substantially increase the feature set of current debuggers.

1 Introduction

Application debugging is an inevitable part of any software development cycle. It is increasingly important in modern day programming practices because of the growing complexity of software and hardware.

Software debuggers often run as separate processes that attach to user applications and trace through runtime events to detect execution anomalies. It is often that case that runtime errors arise because a program's memory is somehow corrupted. Common examples include out-of-bounds accesses and buffer overflow bugs which lead to null-pointer exceptions or the execution of illegal branch instructions. Other errors include the use of uninitialized variables, and data races in the case of shared-memory multi-threaded applications. All of these errors are notoriously difficult to discover and diagnose because it is often not clear when the memory corruption occurred, and which instructions were responsible.

A debugger allows the programmer to inspect the code at the site of an anomaly and trace back in the program stack to derive more clues about the cause of the problem. A particularly useful debugging feature that helps with memory corruption bugs is the data breakpoint, also known as the *watchpoint*.

A watchpoint pauses program execution when an update to a specific memory location is encountered. Watchpoints are similar to instruction breakpoints that allow the user to pause execution at specific instructions.

1.1 Challenges faced by current approaches

Data breakpoints are very expensive to implement without architectural support because they require watching all updates to memory: every write (store) to memory triggers a lookup of the store's address in the *watchlist*. The watchlist consists of all "watched" memory locations that are of interest to user.

The GNU Debugger (GDB) [1] on x86 architectures uses four debugging registers to accelerate the watchpoint debugging feature. The hardware-assist leads to imperceptible or acceptable slowdowns. With hardware-assist forcibly disabled, we observed that even a simple program slows down by a factor of a thousand when a single watchpoint is set.

Hardware-assist has its limitations however. In case of GDB, when the watchlist consists of more than handful of addresses, GDB is forced into a single-step mode that scans a linked list of breakpoints and watchpoints following the execution of every instruction. The performance quickly deteriorates and the cost becomes prohibitively expensive for interactive debugging. As a result, a large number of watchpoints is generally not practical and their use, while potentially very helpful, remains quite limited in scope.

Furthermore, the feature sets offered by most existing standalone debuggers are either not sufficiently rich, or exhibit high overhead and poor scalability for practical and ubiquitous use. There are some advanced debuggers that can manage the performance penalties with static program analysis and advanced compilation [2–4], but they require additional compilation steps, and generally cannot apply to precompiled binaries or dynamically linked code. These factors may impede their adoption by programmers.

1.2 A new and practical alternative

This paper contributes a new approach to debugging, with a primary emphasis on debugging with watchpoints. We leverage advances in binary instrumentation and code manipulation tools [5–8] to provide an effective and efficient debugging framework that can substantially increase the feature set of standard off-the-shelf debuggers.

We present *Efficient Debugging using Dynamic Instrumentation* (EDDI). We demonstrate how to lower the cost and frequency of checking for runtime anomalies and breakpoint conditions using a unique combination of on-demand dynamic instrumentation and a set of simple and carefully engineered optimizations and heuristics. We believe this is the first paper to demonstrate the feasibility of using a dynamic binary instrumentor in an interactive debugging environment.

The paper describes our prototype implementation of EDDI using a state-of-the-art binary instrumentation infrastructure, and an off-the-shelf debugger,

namely GDB. The prototype inherits the properties of the binary instrumentor to run on off-the-shelf IA32 processors. It can handle dynamically linked and stripped binaries without the need for application source code or recompilation.

In the context of debugging using watchpoints, we demonstrate that we can monitor more than a million data locations, with a slowdown of less than 3x compared to native execution. The low overhead makes EDDI practical for interactive debugging. This is in contrast to existing tools that use dynamic instrumentation for program analysis and bug discovery [9, 10] but suffer from high overheads. For example, MemCheck [9] – which can detect uninitialized memory reads, writes to unallocated memory, and other memory use errors – can incur slowdowns between 10x and 30x compared to native execution. Such tools are better suited for regression testing than interactive debugging.

The ability to monitor a large number of memory locations significantly broadens the scope of debugging with watchpoints, and allows for a lot of versatility in defining a wide range of watchlists. For example, a user can choose to watch (1) objects of a specific size or type, (2) objects allocated from specific call sites, (3) entire data structures (e.g., arrays, records, graphs), and (4) reads from addresses written by specific instructions. This new ability leads to potential uses in constraint and consistency checks on heap data structures, data race detection in multi-threaded programs, taint analysis for security analysis, and many other scenarios. We highlight and evaluate a few practical debugging scenarios to concretely demonstrate some of the new debugging capabilities afforded by EDDI. Specifically, we dynamically watch the return addresses of all functions, and break if an instruction modifies a return address. This scenario is useful for detecting malicious attacks that attempt to hijack a program's execution. Second, we identify all static instructions that reference heap objects of a particular types. This particular use scenario can be useful for dynamic pointer analysis. Lastly, we use EDDI to discover all runtime read-accesses from uninitialized heap location, and similarly, we use watchpoints to detect buffer overflow errors.

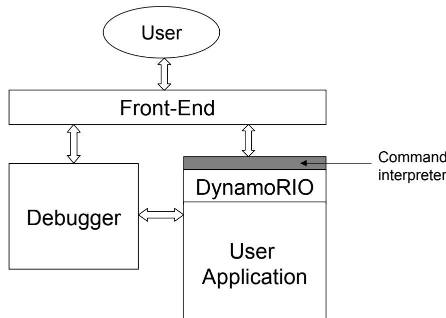
The contributions of the paper are summarized as follows:

- We designed and engineered EDDI, the first on-demand accelerated debugger using binary instrumentation.
- We demonstrate that EDDI provides an efficient and scalable implementation of an important debugging facility, namely data breakpoints (watchpoints).
- We show that EDDI is practical for interactive debugging, and its ability to monitor millions of memory locations provides new debugging capabilities.

2 Interactive Debugging with EDDI

Our goal is to substantially reduce the overhead associated with application debugging so that it is possible to implement new debugging capabilities that can substantially ease the burden on users when they are tracking down programming errors. Our approach with EDDI is to use dynamic instrumentation with

Fig. 1. The EDDI interactive debugging infrastructure.



an off-the-shelf debugger to provide on-demand acceleration and efficient execution under a debugger. An interactive debugger with EDDI consists of four components as illustrated in Figure 1.

The first component is the user application that is interpreted using a binary instrumentation and code manipulation system. We use DynamoRIO [8], although Pin [6] or other systems can also be used. DynamoRIO is a transparent runtime code manipulation system that can execute and manipulate real world applications running on IA-32 hardware. It works under both Linux and Windows. When an application is running with DynamoRIO, it is copied into a thread-private code cache one basicblock at a time, and then runs directly from the code cache. When some basicblocks on a common path become "hot", they are stitched together to form a single-entry multiple-exits trace, and promoted to a trace cache. The basicblock and trace caches are collectively called the *code cache*. DynamoRIO uses thread-private code caches, and this allows for tailoring the instrumentation per thread when necessary. We modify the signal handler in DynamoRIO to intercept and process all runtime signals before relaying them to and from the user application.

The second component is the debugger. It runs as a separate process, and provides typical debugging functionality. We use GDB as-is for this purpose.

The third component is the front-end. It functions as the interface between the user, the debugger, and the instrumentation layer. Programmers use the front-end to relay commands to the debugger, and the debugger relays output back to the user through the front-end. Some commands are directly relayed to a command interpreter that translates the commands into actions for DynamoRIO. The front-end also consolidates the code manipulation carried out by EDDI against the code mapping assumed by the debugger.

The command interpreter is the fourth component. It receives commands from the front-end, and then collaborates with the debugger to instrument the user application to implement the appropriate commands. For example, to set a data breakpoint and then watch for updates to that memory location, EDDI instruments store instructions in the program to check if the address being writ-

ten to matches the address being watched[1]. EDDI uses a set of optimizations and heuristics to reduce the instrumentation and runtime overhead of checking breakpoint conditions and predicates.

3 Efficient Debugging using Dynamic Instrumentation: Software Watchpoint

We believe that binary instrumentation can significantly improve the performance of standard debuggers. In turn, this can lead to richer debugging features that can improve programmer productivity. In this paper, our goal is an efficient and scalable implementation of software watchpoints. We believe this work enables new and potentially very powerful debugging capabilities, with likely applicability in other use scenarios as well.

Watchpoints essentially require monitoring every memory read (load) and write (store), and comparing the addresses against a list of watched addresses. A basic monitoring approach using dynamic binary instrumentation adds new instructions before every memory reference to perform the following tasks: (1) determine the memory address referenced by the load or store, (2) check if address is watched, and (3) raise a trap if the address is watched. We refer to this scheme as *full instrumentation* (FI). A naïve implementation of this scheme adds more than 10 instructions per static memory reference, and can degrade performance by an average of 15x compared to native execution.

We refined this approach in two ways. First, we implemented a set of optimizations to substantially reduce the monitoring overhead as is detailed in Section 4. Second, we used a coarse-grained monitoring scheme that first operates at page granularity before switching to a more fine-grained monitoring mode that inspects individual memory references. We call this coarse-grained scheme *partial instrumentation* (PI).

The PI scheme is tuned for the common case: the majority of memory references do not access watched locations. It focuses primarily on instructions that may reference watched locations, and uses the page protection mechanisms in the operating system to help identify those instructions. In this scheme, pages that contain watched data are protected, and any instructions that try to access these pages trigger a signal. The signal handler checks if the referenced address is watched and takes appropriate action. The runtime overhead for PI is highly dependent on the number of references to the protected pages because signal handling involves expensive operating system mechanisms. As with the FI scheme, we lower the PI overhead by taking advantage of dynamic instrumentation as is described in Section 5.

In addition to the monitoring schemes, we pay particular attention to the design of the *watchlist*. The watchlist is the data structure that records the watchpoints. Since the watchlist is accessed on every memory reference, it is

[1] EDDI will first attempt to use any available hardware-assist resources before falling back on a more general software-only approach.

important to design an efficient watchpoint organization and to implement a fast and scalable query mechanism with reasonable space-time properties.

A linked-list watchlist organization is not practical since the time for a query scales linearly with the number of watchpoints. An alternate strategy is to use shadow tags [4, 11–13] to represent each byte in memory. If a tag is set, then the corresponding byte in memory is watched. In this scenario, a watchpoint query requires loading the appropriate tag and checking its status. The query cost for this approach is constant and independent of the number of watchpoints.

In our work, we designed a new shadow tag based watchlists for efficient tag lookup and with good scalability. The basic idea is to use an on-demand watchlist. Each application memory page is associated with an equal sized shadow page when necessary. The status of a byte in an application page is indicated by the tag at the same offset in the associated shadow page (if the tag is zero, the byte is not watched). We use a byte-long tag for byte-addressable memory. The extra bits in each tag encode the status of adjacent bytes[2].

A *lookup table* maintains the mapping between the application pages and their shadow pages. On 64-bit architectures, a hierarchical lookup table is required, although an optimized one-level table is feasible via careful memory allocation for shadow tags.

4 Optimizations for Full Instrumentation (FI)

The full instrumentation scheme inserts instructions to check the watchlist before every memory reference. Figure 2 shows the necessary instrumentation for a single x86 store instruction `mov esi -> [eax, ebx]`. The instrumentation performs the following tasks before the store:

1. Save any registers that are used or side-effected by the watchlist query.
2. Calculate the reference address and lookup its associated tag.
3. Checks if the tag is set to "watched", and trap if it is.
4. Otherwise, restore the saved registers and continue execution.

In this example, the lookup table stores the displacement between the shadow page and the accessed data. There are 20 new instructions in total. Instructions 1–6 and 16–20 save and restore the execution context. Instruction 7 obtains the effective address in register `ecx`. Then the lookup table entry is identified and checked by instructions 8–10. Instructions 11–14 check the tag found in the shadow pages. If the lookup table entry is null, the tag check is skipped.

The naïve instrumentation described above suffers from a significant runtime overhead. We implemented and applied a series of optimizations to systematically and drastically reduce that overhead. We group the optimizations into three categories: previously published, watchlist-specific, and local analysis.

[2] Special care is required to handle two adjacent memory addresses that span two pages. Due to space limitations, we do not describe the encoding in any more detail.

Fig. 2. Example instrumentation code.

```
01: mov  %ecx → [ECX_slot]     ! Steal registers
02: mov  %eax → [EAX_slot]
03: seto [OF_slot + 3]         ! Save oflag
04: lahf                       ! Save eflags
05: mov  %eax → [AF_slot]
06: mov  [EAX_slot]   → %eax   ! Restore eax
07: lea  [%eax, %ebx] → %ecx   ! Get address
    ! Compute table index
08: shr  %ecx, $12 → %ecx      ! Shift right
09: cmp  table[%ecx, 4], $0    ! Check entry
10: je   SKIP_CHECK
    ! Check if tag is set to 'watched'
11: add   %eax, table[%ecx, 4] → %eax
12: testb $0xAA, [%eax, %ebx]
13: jz    AFTER_TRAP
14: trap                       ! Watchpoint trap
AFTER_TRAP:
15: sub   %eax, table[%ecx, 4] → %eax
    ! Restore all
SKIP_CHECK:
16: mov  [AF_slot] → %eax
17: add  [OF_slot], $0x7f000000 → [OF_slot]
18: sahf
19: mov  [EAX_slot] → %eax
20: mov  [ECX_slot] → %ecx
```

1. Previously published optimizations. We applied two optimizations published by Qin et al. [12].

- *Context Switch Reduction* (CSR) performs register liveness analysis in each basicblock to identify registers that can be safely used without requiring a register spill and restore.
- *Group Checks* (GC) consolidates two consecutive memory reference checks into a single check if there are no intervening instructions that affect the address calculation of the second memory reference.

2. Watchlist-specific optimizations. The following two optimizations take advantage of the watchlist design.

- *Merged Checks* (MC) aims to exploit the locality of memory references. For example, two instructions in the same basicblock may access different members of the same object (e.g., `mov 0 -> [%eax + 4]` followed by `mov 0 -> [%eax + 8]`). In this case, a single shadow tag lookup is initiated before the first reference. If the tag is zero, then neither location is watched. Otherwise, the tag lookup is carried out for each reference individually.
- *Stack Displacement* (STK) aims to reduce the watchlist query for stack accesses that use the stack pointer. This optimization elides the mapping step through the lookup table. This is achieved through a simple precomputation step. When a thread is initialized, we allocate a shadow memory for the application stack, and calculate the constant offset between the stack and its corresponding shadow memory. Subsequently, when an instruction that

accesses the stack via the stack pointer is encountered, the instrumentation code directly calculates the displacement to the shadow tag without going through the lookup table.

3. Local optimizations.

– *Local Variables Check Elimination* (LVE) eliminates tag checks on local scalar variables referenced via the stack pointer (e.g., `mov 0 -> [esp + 20]`) since they are amenable to static analysis.

There are many other optimization that can further reduce the monitoring overhead. Our purpose is not to be exhaustive but rather to demonstrate that online memory reference monitoring can be achieved with reasonable overhead.

5 Partial Instrumentation (PI)

In addition to the fine-grained instrumentation approach, we rely on a coarse grained partial instrumentation technique to further manage the runtime overhead associated with monitoring memory updates. With PI, we only instrument memory references that may reference watched locations. Partial instrumentation optimizes for the common cases where references do not access watched locations.

PI relies on the operating system page protection mechanism to switch between a coarse-grained mode and a fine-grained mode that checks memory instructions with greater scrutiny. When a watchpoint is set, we make a twin copy of the pages containing that watchpoint. The access rights of the original pages are then set to be protected. During the program execution, if an instruction I references an address on the watched pages, a `SIGSEGV` signal is raised. Our signal handler catches that signal, and subsequently replaces the code fragments containing the instruction I with a new code fragment that includes additional instrumentation. The instrumentation serves dual roles. First, it performs any necessary tag checks. Second, it performs *reference redirection* so that the instruction I accesses the copied pages instead of the protected pages, hence avoiding future `SIGSEGV` signals from the same instruction.

PI is suitable for many situations, especially when monitoring data accesses to the heap. For example, consider the case where a user wants to monitor all references to objects allocated from a specific call site. With PI, we can (1) allocate memory from a special pool of protected pages, (2) update the lookup table, and (3) return the allocated address to the application. Meanwhile, we allocate the twin page from another pool such that the difference between the corresponding locations of a data object in both pools is a constant D_p. This simplifies the redirection code. Figure 3 shows an example of reference redirection for the x86 instruction `mov 0 --> [%eax + 0x10]`, where D_p is 0x30000.

As is the case with full instrumentation, it is possible to apply more optimization to further reduce the overhead associated with partial instrumentation.

Fig. 3. An example of reference redirection.

```
    mov %ecx → [ECX_SLOT]       ! steal ecx
    lea [%eax+0x10] → %ecx      ! calculate address
    ...                         ! save eflags
    shr %ecx, 20 → %ecx         ! right shift
    cmp table[%ecx], $0         ! check table entry
    je  LABEL_ORIG
    ...                         ! check tag status
    ...                         ! restore eflags and ecx
    mov 0 → [%eax + 0x030010]   ! redirected reference
    jmp LABEL_NEXT
LABEL_ORIG:
    ...                         ! restore eflags and ecx
    mov 0 → [%eax+0x10]         ! access original location
LABEL_NEXT:
    ...                         ! continue execution
```

6 Evaluation and Results

We measured the performance overhead associated with EDDI in the context of software watchpoints. We conducted a set of experiments to quantify the impact of our optimizations for full and partial instrumentation. We also designed some experiments to showcase various debugging scenarios that make use of the large number of watchpoints that can be monitored with EDDI.

We ran all of our experiments on a 2.66 GHz Intel Core 2 processor with 2 GBytes of RAM. The operating system is Linux Fedora Core 4. We used the full SPEC CPU2000 [14] benchmark suite and the reference input workloads. All benchmarks were compiled with GCC 4.0 using `-O3`. We used a shadow page lookup table with 2^{20} entries. We used shadow pages that are 4 KBytes in size to match the default Linux page size.

Results for FI. Figure 4 compares the native performance of each benchmark to the same benchmark run with DynamoRIO and the full instrumentation scheme (along with its accompanying optimizations). In this set of experiments, there are no user-defined watchpoints (i.e., all shadow tag bits are zero). We are simply measuring the instrumentation overhead attributed to the monitoring of memory accesses and watchlist queries.

We report the results for full instrumentation with common optimizations (CSR and GC), merged checks (MC), stack displacement (STK), and local variable check elimination (LVE). The performance results are normalized to native execution, and hence a value greater than one indicates a slowdown, with smaller numbers reflecting lower runtime overhead.

The common optimizations reduced the runtime overhead from an average slowdown of 15x in the unoptimized instrumentation scheme (data not shown), to an average of 5x. The addition of MC reduced the overhead further. MC is especially effective on benchmarks with good temporal and spatial locality (e.g., `252.eon`). Performance improvement due to STK were mixed. This particular optimization reduces the opportunities for merged checks. The LVE optimization improved performance significantly because it removes all checks for scalar (i.e.,

Fig. 4. Impact of optimizations on FI.

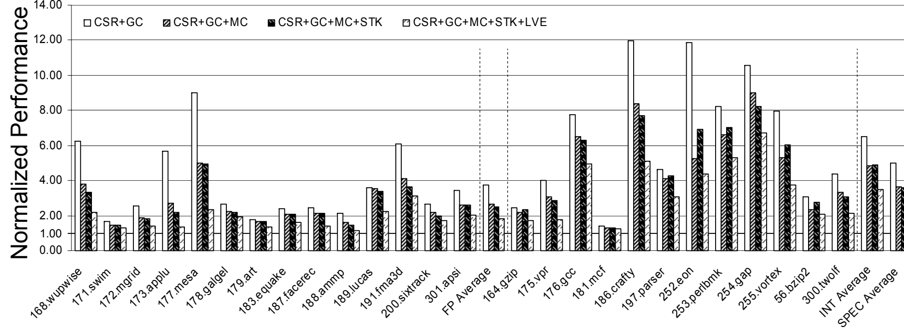
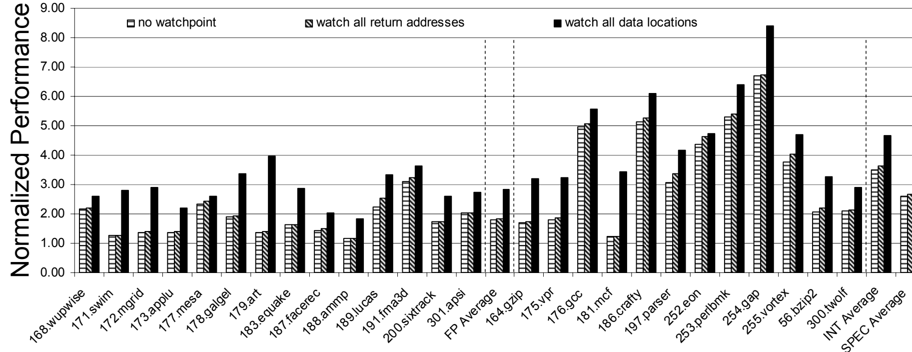


Fig. 5. Performance of FI with different watchlist configurations.



non-array) local variables. Overall, FI with all of our optimizations incurs an average slowdown of 2.59x compared to native execution.

Next, we measured our system performance using three watchlist configurations: no watchpoints[3], watch all function return targets, and watch all data locations. In an actual debugging scenario, data breakpoints that are triggered alert the user and the debugger interrupts execution and waits for additional commands from the user. In our experiments, however, when a watchpoint was triggered, the instruction that accessed the watched location was merely marked and execution continued. Figure 5 shows the results normalized to native execution time. As expected, the runtime overhead increased as the number of watchpoints increased. The low additional overhead that is observed when watching all return targets is due to the efficacy of the STK optimization. In the worst case, the average overhead is 3.68x when all memory locations are watched.

Results for PI. We studied the performance implications of the partial instrumentation scheme using several SPEC INT benchmarks. Two benchmarks were ignored: `197.parser` allocates a large memory pool at the start of execution, and `254.gap` requests a chunk of memory only to free it immediately without using it.

[3] This set of data point corresponds to the fourth bar in each set of bars from Figure 4.

Table 1. Runtime overhead using PI.

Benchmarks	Native (sec)	PI (sec)	Overhead PI / Native	No. of Redirects	Watched Objects	No. of SIGSEGV
164.gzip	24.379	26.189	1.074	1.45×10^{8}	20423	45
175.vpr	48.346	50.605	1.047	1.04×10^{6}	10000	76
176.gcc	7.980	10.026	1.256	1.51×10^{6}	1	22
181.mcf	53.243	80.659	1.515	1.08×10^{10}	1	468
186.crafty	58.777	249.777	4.250	2.77×10^{8}	37	443
252.eon	35.961	41.470	1.153	3.50×10^{1}	1	8
253.perlbmk	9.101	24.489	2.691	7.69×10^{7}	1	249
255.vortex	29.940	47.582	1.589	1.65×10^{9}	10	219
256.bzip	27.598	50.837	1.842	9.01×10^{9}	7	541
300.twolf	155.725	161.442	1.037	2.78×10^{5}	1100	470

We selected watchpoints by intercepting allocation requests and randomly deciding to watch the allocated objects. Objects that are watched were allocated from a designated memory pool with protected pages. This approach allowed us to easily define watchpoints at runtime without any user intervention. The results are summarized in Table 1. The column labeled "Watched Objects" lists the number of watched heap objects. It is worthy to note that these objects are not necessarily uniform (i.e., they may vary in type and size).

The second column in Table 1 reports the native execution time, and the third column shows the execution time when using EDDI and partial instrumentation. The last column reports the number of `SIGSEGV` signals that were caught (due to accesses to watched regions), and the column labeled "No. of Redirects" reports the number of subsequent reference redirections that occurred. The overhead in PI is mainly due to the redirection mechanism. The benchmarks `181.mcf`, `175.vpr`, and `256.bzip` have 10^9 or more redirections with slowdowns ranging from 51% to 84%. The two benchmarks with the highest overhead, `186.crafty` and `253.perlbmk`, execute a large number of string operations that require a relatively more heavyweight redirection mechanism. The benchmarks `176.gcc` and `252.eon` incur a large DynamoRIO overhead (not shown). The added PI overhead is these two benchmarks and the remaining three is less than 7%.

In general, partial outperforms full instrumentation. The former optimizes for the common cases and only instruments a small number of static instructions.

Space Overhead. The size of the shadow memory depends on the number of watchpoints. Since we use a demand-driven watchlist, we expect the shadow memory footprint to be typically small. In the worst case when all memory locations are watched, FI doubles the total memory footprint since every application page requires a shadow page. The PI scheme needs additional memory since it also clones the application pages. Hence the total memory footprint grows threefold when watching all heap data.

There is also some instruction memory overhead since the watchlist monitoring adds instrumentation code. In practice, this is usually small compared to the size of data memory. Note that these are virtual memory overheads, and our results show that the spatial footprints remain manageable. We do not expect the space overhead to be a serious concern on 64 bit processors.

Debugging Scenarios. We believe that EDDI can be a useful tool for understanding program behavior and discovering software bugs. Here we highlight several debugging scenarios that demonstrate how a user may use the large number of watchpoints afforded by EDDI.

- We used EDDI to discover all the instructions that access the return addresses of functions. To do this, a watchpoint is automatically set on the return address of a function when it is called. This watchpoint is cleared when the function returns. Note that such a task is nearly impossible to achieve in a standard debugger like GDB. The middle set of bars in Figure 5 show the expected performance for the SPEC benchmarks. Interestingly, we found that besides the return instructions, there are several functions such as `setjmp` that also read the return address.
- We also used EDDI to perform a kind of dynamic pointer analysis. In particular, using `181.mcf` as an example, we watched all 33,112 instances of the `node` data type and identified the 468 static instructions that referenced these `node` objects 1.08×10^{10} times during execution.
- A common programming error is the use of uninitialized variables. We used EDDI to detect such occurrences by replacing calls to `calloc` with `malloc` in `181.mcf`. Unlike `calloc`, `malloc` does not initialize the allocated memory. We used EDDI to discover reads to uninitialized values by marking all allocated memory as watched. Once an object is initialized, it is removed from the watchlist. EDDI reported all reads to uninitialized memory locations correctly. As an example, the first uninitialized read in `181.mcf` occurs 0.001 seconds from the start of execution. EDDI reports the error in 0.037 seconds, and overall, the instrumented execution is 83% slower using PI and 250% slower using FI.
- The last use scenario that we investigated was inspired by software security attacks. We used EDDI on a set of benchmarks for buffer overflow attacks [15]. By placing watchpoints on buffer boundaries, EDDI successfully identified all offending instructions. By setting watchpoints on key locations that include return addresses, stack frame pointers, and function pointers, many kinds of intrusion schemes can be detected. Furthermore, because EDDI monitors all instructions executed in user mode, it will discover any buffer overflows that occur in shared libraries as well.

The above use-cases are not exhaustive. The ability to watch large and variable sized regions of memory efficiently is very powerful. We believe that EDDI affords new capabilities and potentially new approaches for debugging.

7 Related Work

Application Debugging. There are many software, hardware and hybrid techniques for application debugging, especially for detecting memory bugs. Hardware schemes such as SafeMem [16], iWatcher [17], and MemTracker [18] are geared toward low overhead detection mechanisms of inappropriate memory uses.

DISE [19] (Dynamic Instruction Steam Editing) is a general hardware mechanism for interactive debugging. HeapMon [20] is a hardware/software approach for detecting memory bugs with a helper thread. An important drawback of these techniques is that they require specialized hardware.

There are many software tools use dynamic instrumentation to discovery memory related bugs. For example, Purify [21] and MemCheck [9] are two widely used software tools for discovering memory problems. However, their significant overhead make them unsuitable for interactive debugging purposes. Tdb [22] is a tool that integrates dynamic instrumentation with GDB. However, the paper only describes how to handle the code mapping between the application code and the code cache, and uses code breakpoints as a demonstration vehicle. In contrast, EDDI encompasses instruction and data breakpoints and can monitor memory operations efficiently.

Software Watchpoint. Watchpoint is an important debugging facility that helps users identify data corruption bugs. Almost all state-of-the-art processors provide limited hardware watchpoints facilities. There has also been several proposals in the past on how to implement software watchpoints. They can be generally classified as three approaches: page protection, trap patching, and code patching. Wahbe [3] and Roberts [23] both compared the above implementation strategies, and made the same conclusion that code patching has the lowest overhead. Wahbe then proposed an efficient implementation of software watchpoints [4] using code patching and static analysis. However, that work could not be used on shared libraries. Copperman and Thomas [2] extended the work to use a post-loading technique to insert checks into an executable and solve the shared library issue. EDDI can monitor all memory accesses efficiently. In addition, EDDI's optimized page protection strategy outperforms the code patching approach in certain situations.

Another interesting approach proposed by Keppel [24] is to use checkpointing to discover data updates, but we did not find any implementation details or any results on this work. There is a published patch to MemCheck in Valgrind [25] to perform watchpointing. However, the watchpoints are organized in a linked-list and performance scales very poorly. This clearly was not designed to operate at the scale that we envision for EDDI.

Shadow Memory. Shadowing memory is important for efficient software watchpoints. Cheng et al. [11] suggested splitting the address space with half of it used for shadow memory. This approach simplifies the address calculation for locating the tag in the shadow memory to a single add. However, this is also the most space consuming proposal. Wahbe et al. [4] suggested a bit-map approach that associates a one-bit tag to every 4 bytes. On byte-addressable architectures like the x86, the required space overhead is 12.5%, or 512 MBytes for a 32-bit address space [12]. Unfortunately, on 64-bit architectures, this scheme requires 2^{61} contiguous bytes. MemCheck [13] use a two-level table to organize the shadow memory that is similar to our approach. However, our approach is more flexible, and allows for a more efficient tag lookup.

8 Conclusion

The state of debugging technology has largely remained unchanged for more than a decade, and many programmers still rely on the "debug by printing" approach to track down and resolve software bugs. As software becomes larger and more complex, new approaches are required to improve programmer productivity. We believe that the union of traditional debuggers and binary rewriting systems can bring about many new debugging features and techniques. Our contribution in this paper is to show the viability of such an approach.

We presented EDDI, a debugging framework that uses on-demand dynamic instrumentation and runtime optimizations to accelerate and extend features found in traditional debuggers. We showed how EDDI can be used to implement a data watchpoint facility that allows users to set orders of magnitude more watchpoints than is practical today. EDDI does not rely on any specialized hardware, and is evaluated in this paper using several SPEC2000 benchmarks running on an Intel Core 2 machine. The results show that the average overhead is 3x, which is low enough to make EDDI practical for interactive debugging. Besides a large number of variable sized watchpoints, EDDI also provides dynamic event handling capability and customized trigger actions. We highlighted several practical uses of EDDI in debugging, program analysis and security. Combined with the orthogonal effort of 'reversible' debugging [26], we believe EDDI can contribute to powerful new ways of debugging software.

References

1. GNU/FSF: GDB: The GNU Debugger.
2. Copperman, M., Thomas, J.: Poor Man's Watchpoints. SIGPLAN Not. 30 (1995) 37–44.
3. Wahbe, R.: Efficient Data Breakpoints. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (1992) 200–212.
4. Wahbe, R., Lucco, S., Graham, S.L.: Practical Data Breakpoints: Design and Implementation. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (1993) 1–12.
5. Nethercote, N.: Dynamic Binary Analysis and Instrumentation. PhD thesis, University of Cambridge (2004) http://valgrind.org/.
6. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (2005) 190–200.
7. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: A Transparent Dynamic Optimization System. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (2000) 1–12.
8. Bruening, D.: Efficient, Transparent, and Comprehensive Runtime Code Manipulation. PhD thesis, Massachusetts Institute of Technology (2004) http://www.cag.csail.mit.edu/rio/.

9. Seward, J., Nethercote, N.: Using Valgrind to Detect Undefined Value Errors With Bit-Precision. In: Proceedings of the USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2005) 17–30.
10. Zhang, X., Gupta, N., Gupta, R.: Locating Faults Through Automated Predicate Switching. In: Proceeding of the 28th International Conference on Software Engineering (2006) 272–281.
11. Cheng, W., Zhao, Q., Yu, B., Hiroshige, S.: Tainttrace: Efficient Flow Tracing With Dynamic Binary Rewriting. In: Proceedings of the 11th IEEE Symposium on Computers and Communications (2006) 749–754.
12. Qin, F., Wang, C., Li, Z., Kim, H., Zhou, Y., Wu, Y.: LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In: Proceedings of the 39th Annual International Symposium on Microarchitecture (2006) 135–148.
13. Nethercote, N., Seward, J.: How to Shadow Every Byte of Memory Used by a Program. In: Proceedings of the 3rd International Conference on Virtual Execution Environments (2007) 65–74.
14. Standard Performance Evaluation Corp.: SPEC CPU2000 Benchmark Suite (2000) http://www.spec.org/osg/cpu2000/.
15. Wilander, J., Kamkar, M.: A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In: Proceedings of the 10th Network and Distributed System Security Symposium (2003) 149–162.
16. Qin, F., Lu, S., Zhou, Y.: SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In: Proceedings of the International Symposium on High Performance Computer Architecture (2005) 291–302.
17. Zhou, P., Qin, F., Liu, W., Zhou, Y., Torrellas, J.: iWatcher: Efficient Architectural Support for Software Debugging. In: Proceedings of the 31th International Symposium on Computer Architecture (2004), 224–237.
18. Venkataramani, G., Roemer, B., Prvulovic, M., Solihin, Y.: MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In: Proceedings of the International Symposium on High Performance Computer Architecture (2007) 273–284.
19. Corliss, M.L., Lewis, E.C., Roth, A.: Low-Overhead Interactive Debugging via Dynamic Instrumentation with DISE. In: Proceedings of the International Symposium on High Performance Computer Architecture (2005) 303–314.
20. Shetty, R., Kharbutli, M., Solihin, Y., Prvulovic, M.: HeapMon: a Helper-Thread Approach to Programmable, Automatic, and Low-Overhead Memory Bug Detection. IBM Journal of Research and Development 50 (2006) 261–275.
21. Hastings, R., Joyce, B.: Purify: Fast Detection of Memory Leaks and Access Errors. In: Proceedings of the Winter USENIX Conference (1992) 125–136.
22. Kumar, N., Childers, B.R., Soffa, M.L.: Tdb: A Source-Level Debugger for Dynamically Translated Programs. In: Proceedings of the 6th international symposium on Automated analysis-driven debugging (2005) 123–132.
23. Roberts, P.E.: Implementation And Evaluation Of Data Breakpoint Schemes In An Interactive Debugger. Master's thesis, University of Utah (1996) http://www.cs.utah.edu/flux/papers/perobert_abstract.html.
24. Keppel, D.: Fast Data Breakpoints. TR-93-04-06, University of Washington (1993).
25. Walsh, R.: Patches for Valgrind (2005) http://valgrind.org/downloads/variants.html.
26. The GNU GDB Project: GDB and Reversible Debugging (2006) http://sourceware.org/gdb/news/reversible.html.