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

Qin ZHAO[2], **Saman AMARASINGHE**[1], **Rodric M. RABBAH**[1], **Larry RUDOLPH**[1], and **Weng Fai WONG**[2]
[1] CSAIL, Massachusetts Institute of Technology
[2] Soc, National University of Singapore

***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. The latter is a notoriously difficult category of bugs to diagnose and repair especially in pointer-heavy applications. The debugging challenges will increase with the arrival of multicore processors which require explicit parallelization of the user code to get any performance gains. Parallelization in turn can lead to more data debugging issues such as the detection of data races between threads. This paper leverages the increasing efficiency of runtime binary interpreters to provide a new concept of Efficient Debugging using Dynamic Instrumentation, or 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 the low associated overheads. EDDI provides a scalable and extensible debugging framework that can substantially increase the feature set of standard off the shelf debuggers.**

***Index Terms*— Debuggers, watchpoints, dynamic instrumentation**

I. Introduction

APPLICATION debugging is an inevitable part of any software development cycle. Software debuggers often run as separate processes that attach to the end user application, and then trace through runtime events to detect execution anomalies or interrupt execution at programmable breakpoints. Some anomalies result from dereferencing null pointers or executing illegal branch instructions, and when detected, the user can inspect the code at the site of the anomaly, and trace back in the program stack to derive more clues. Breakpoints on the other hand are user programmable predicates that specify when to interrupt the execution of a program. An instruction breakpoint allows the user to pause execution at specific instructions when a specified set of conditions is met. A data breakpoint, or *watchpoint*, pauses executions when an update to a specific memory location is encountered. Watchpoints are especially helpful in discovering data corruption bugs that arise from out-of-bounds or buffer overflow bugs in C and C++ programs. This class of errors is notoriously difficult to discover and diagnose without watchpoints.

Many architectures provide some support for debugging in order to reduce the overheads of implementing the debugging feature set in software. In particular, data breakpoints are very expensive to implement without hardware support since they require watching all updates to memory. On every data write, a check is made against the set of addresses that are of interest to the end user (the watchlist). As such, execution can slow down significantly in the absence of hardware support. The GNU Project Debugger (GDB) [6] on x86 architectures uses the four available debugging registers to accelerate the watchpoint debugging feature. This often results in imperceptible or acceptable slowdowns although support for a watchlist of more than a handful of memory locations is prohibitively expensive and thus not realistic. For example, using a simple program, we observed that in GDB version 5.3, performing a watchpoint with hardware assistance forcibly disabled resulted in a 11,000 times slowdown.

Program debuggers will play an increasingly important role in software development. With the ever increasing size and complexity of software, and the advent of multicore and hence mass parallel programming, developers are facing an ever increasing challenge in debugging. This bodes poorly for diagnosing data corruption errors that might arise from bad pointers, buffer overflows, or data races. The feature sets offered by most existing standalone debuggers are either not sufficiently rich, or exhibit poor overhead scalability. There are however more advanced debuggers that can manage the performance penalties via static program analysis and advanced compilation [4], [18], [19]. The drawback to these techniques is that they require additional compilation steps, and generally cannot apply to precompiled binaries or dynamically linked code. These factors may impede widespread user acceptance.

This paper offers a new approach: *Efficient Debugging using Dynamic Instrumentation* (EDDI). EDDI leverages the advances in binary instrumentation and code manipulation tools [10], [9], [1], [2] to provide an efficient debugging framework that can substantially increase the feature set of standard off the shelf debuggers. As an example, the paper describes an implementation of EDDI using GDB that can monitor more than a million data watchpoints, with a slow down of less than 3x compared to native execution. This ability to monitor such a large number of memory locations allows for significant versatility in defining a wide range of watchlists. For example, a user can choose to watch (i) entire data structures such as records or arrays, (ii) objects allocated from specific call sites, and/or (iii) objects of a specific size or type. Furthermore, EDDI runs on off-the-shelf x86 processors running Linux, on dynamically linked, stripped binary executables without a need for application's source code.

We believe this is the first paper to demonstrate the feasibility of using a dynamic binary instrumentor in an interactive debugging environment. EDDI provides a more efficient interactive debugger that is considerably more powerful than existing debuggers. In contrast, prior works that use dynamic instrumentation for program analysis and bug discovery [14], [20] are not yet well suited for interactive debugging. For example, tools such as MemCheck [14] which can detect uninitialized memory read, writes to unallocated memory, and other memory use errors, can incur slowdowns between 10-30x, and are more suitable for regression testing than interactive debugging.

EDDI uses on-demand dynamic instrumentation to augment traditional debuggers. It uses a set of optimizations to lower the cost of checking for runtime anomalies and breakpoint conditions, as well as heuristics to reduce the frequency of the checks altogether. Section II describes the framework, and Sections III and IV details the implementation and optimization of EDDI for use as a general purpose watchpoint debugger. It allows the end user to efficiently watch updates to individual memory cells, or any range of addressable memory in general. Section V presents an evaluation of EDDI using several SPEC2000 benchmarks. It documents the time and space overhead associated with EDDI. This is followed by the conclusion.

II. The EDDI Interactive Debugging Environment

EDDI is on-demand binary instrumentation and code manipulation that aim to reduce the overhead associated with application debugging. It receives commands from a standard debugger, and then instruments the user application to implement those commands. For example, to set a data breakpoint and then watch for updates to that memory location, EDDI instruments the stores to memory and checks on every write if the address written matches the address watched. EDDI uses a set of optimizations and heuristics to reduce the instrumentation and runtime overhead of checking breakpoint conditions or predicates.

An overview of the EDDI interactive debugging environment is shown in Figure 1. It consists of three components.

- The user application is interpreted using an off the shelf binary instrumentation and code manipulation system. We use DynamoRIO [2], although Pin [9] and other systems are also plausible.
- The debugger is a separate process. It provides traditional debugging functionality. We use GDB for that purpose.
- The front-end works as the interface between the user, and the debugger and interpreter layer. Programmers uses 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 the interpreter. The front-end also consolidates the code manipulation carried out by EDDI against the code mapping assumed by the debugger.

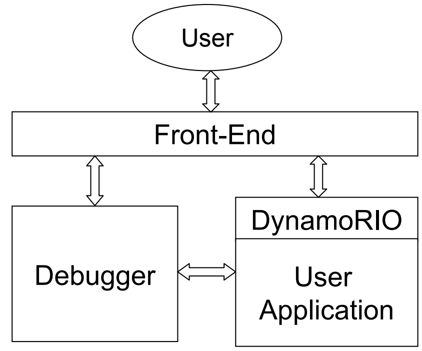

Fig. 1. The EDDI Debugging Infrastructure.

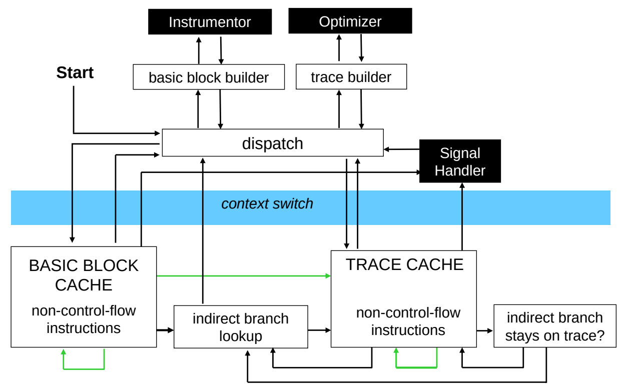

Fig. 2. DynamoRIO dynamic binary rewriting framework with EDDI extensions.

A. *DynamoRIO and extensions for EDDI*

DynamoRIO is a transparent runtime code manipulation system that can execute and manipulate real world applications running on IA-32 hardware. DynamoRIO works under both Linux and Windows. Figure 2 shows the main components of DynamoRIO, and highlights the added modules for EDDI. When an application is running with DynamoRIO, it is copied into a thread-private code cache one basic block at a time, and then runs directly from the code cache. When some basic blocks on a common path become "hot", they are stitched together to form a single-entry multi-exits trace, and promoted to a trace cache. The basic block and trace caches are collectively called the code cache.

EDDI adds an on-demand instrumentor to instrument basic blocks before they are stored into the code cache, and an optimizer to exploit instrumentation redundancy and reduce overhead. Section IV focuses in detail on the set of optimizations that EDDI performs. DynamoRIO uses thread-private code caches, and this allows for instrumenting threads differently when necessary. The signal handler in DynamoRIO is modified to intercept and process all the signals before relaying them to and from the user application.

B. *Instruction Mapping*

An application debugged with EDDI is heavily manipulated such that the instruction sequences and code layout can differ substantially from the original native user code. Thus in order to maintain transparency, it is necessary to maintain and preserve a mapping from the instructions in the code cache to the native application instructions. This translation process is akin to the mapping between binary code and source code in traditional debuggers. For instance, when a user sets an instruction breakpoint at specific program location, the front-end will need to map the location to the address of the corresponding instruction in the code cache, and relay that location to the debugger which then can interrupt execution at the remapped instruction address. This gives the user the illusion of debugging their native code, whereas the debugger is in fact operating in a different context. When the debugger pauses execution, the front-end recalls the instructions in the native application to display, and the user remains unaware of the binary instrumentation and runtime optimizations that are underway.

EDDI maintains a hash table of `(pc, tag)` key-value pairs to translate instruction addresses from the application code to the code cache. The `pc` (program counter) is the address of a native instruction, and the `tag` is the index of the code fragment that instruction is mapped into. The mapping provides a very lightweight mechanism to implement instruction breakpoints. When a user requests a new breakpoint, the code fragment containing the instruction is removed from the code cache (if it is cached), and the breakpoint is triggered the next time the code fragment is placed in the code cache. This approach allows us to implement a software control breakpoint without any support of hardware or trap instructions. In effect it boot-straps on the code caching mechanisms inherent to binary manipulation systems.

Translation in the reverse direction is also simple. This is necessary for example when a data breakpoint or watchpoint is triggered. Because instrumentation and optimization are deterministic operations, EDDI can simulate the instrumentation and optimization process on a code fragment, and identify the original application instruction corresponding to each instruction in the fragment. In order to reconstruct a trace, EDDI maintains a record of the basic blocks that make up a trace region.

C. Coordination and Communication

The front-end parses all user inputs commands, whether to the application or the debugger. The front-end then communicate with the other two components to execute the commands. We use the standard GDB/MI (machine independent) interface to communicate between the front-end and GDB. Because GDB is unaware of the changes made by DynamoRIO to the user application, the front-end has to mediate GDB's interaction with the user application.

The communication between the front-end and the application is implemented using UNIX signals and inter-processor message queues. The front-end generates a signal when it wants to communicate with the application. The signal is caught by EDDI, and delayed until the current code fragment completes its execution (i.e., executes a branch instruction). This avoids a mapping between the code cache and the native instruction sequence. Afterward, the application and front-end can exchange messages.

III. Software Watchpoint

To demonstrate the effectiveness of EDDI, we implemented an important debugging facility, namely software watchpoints, within EDDI. This watchpoint facility is more flexible than hardware watchpoints – more data locations can be watched, and any size of watch ranges is allowed. There are many issues in the design of software watchpoints. These include scalability issues as the number of watchpoints is increased, the memory footprint, runtime overhead, support for 64-bit and multi-threading. In this section, we will discuss some of the insights we gained in these issues gained from our implementation of software watchpoints. In the following section, we will discuss how we optimized the implementation.

A. Memory Checks

In order to monitor all memory references, whenever the application code is copied into code cache, the instrumentor inserts watchpoint checking code before every memory references. In order to support any number of watchpoints with arbitrary sizes, the overhead of checking must scale the with the number of watchpoints. We associate each memory byte location with a tag to indicate if that it is being watched. Before each memory reference is executed, this tag is checked. There are two advantages of this approach. First, it is easy to set a watchpoint. We only need set the associated tag of memory location being watched, instead of modifying the code injected by the dynamic instrumentor. Secondly, the runtime overhead does not increase dramatically as the number of watchpoints is increased although the tag lookup overhead may increase a little.

In particular, the injected code performs the following major operations:

1) Save the registers that will be used or affected in the checking process.
2) Calculate the reference address and lookup the associated tag.
3) Checks if the tag is set to 'watched', and trap if it is.
4) Otherwise, the saved registers are restored.

Figure 3 is an example of simple checking code injected for the memory store instruction `mov esi -> [eax, ebx]`.

B. Tag Lookup Table

There is a trade-off between memory usage and the efficiency of the watchpoint tag lookup. To tag, or potentially tag, every or any memory location, the most straight forward approach is to split the address space into equal halves [3]. In this approach, the tag corresponding to a memory location can easily be obtained by a simple addition of a constant offset to the memory location. However, it is also the most space consuming approach. Another approach is to use a bit map to represent the whole addressible space. For example, Wahbe et. al. [19] suggested the association of a one-bit tag to every 4 bytes. This incurs only a 3% overhead. However on byte-addressible architectures like the x86, the required space is increased to 12.5% of total address space [12]. If we need to distinguish between reads and writes, then two bit tags are

```
  mov  %ecx -> [ECX_slot]
  mov  %edx -> [EDX_slot]
  mov  %eax -> [EAX_slot]
  seto [OF_slot + 3]
  lahf
  mov  %eax -> [AF_slot]
  mov  [EAX_slot]   -> %eax
  lea  [%eax, %ebx] -> %ecx
  mov  %ecx -> %edx
  shr  %ecx, $20 -> %ecx
  cmp  L1_table[%ecx, 4], $0
  je   LABEL_RESTORE
       mov   L1_table[%ecx, 4] -> %ecx
       and   %edx, $0xfff -> %edx
       testb $0xAA, [%ecx, %edx]
       jz    LABEL_RESTORE
             int3
LABEL_RESTORE:
  mov  [AF_slot] -> %eax
  add  [OF_slot], $0x7f000000 -> [OF_slot]
  sahf
  mov  [EAX_slot] -> %eax
  mov  [EDX_slot] -> %edx
  mov  [ECX_slot] -> %ecx
```

Fig. 3. An example of the checking code that is injected by EDDI.

needed. In this case, 25%, or 1G byte memory for a 32-bit address space is needed. Worse, such mappings do not scale to 64-bit architectures.

Instead of using continous memory for the tags, EDDI uses a two-level hierarchical table approach. Given a memory address, the n most significant bits are used to lookup the first level table, while the rest of the bits determines the tag's position in the second level table. To reduce the memory usage and lookup overhead, the entry of the first level table is set to `null` if there is no memory location in that region being watched. The sizes of the first level and the second level table are not necessarily the same as the size of a page of operating system. This approach results in a significantly smaller memory footprint especially in 'normal' use. For instance, a first level table with 2^{20} entries uses 4 Mbytes. Assuming that each byte of memory is associated with a one-byte tag for ease of lookup, we will need 4 Kbytes for each of the 1M second level tables. If we set aside another 4 Mbyte of memory for use in the second level, we would allow the user to set byte-size watchpoints in up to 1024 pages. If every byte in the second level table is set, then the user can effectively set over 4 million watchpoints.

To extend to a 64-bit system, more levels in the tag table would be needed. Meanwhile, since in practice only a relatively small amount of memory is needed, it is possible to monitor memory allocations, and design a hash function to reduce the number of tag table levels.

There is one problem with this approach: an unaligned memory reference can cross table boundaries, *i.e* four tags for a 4-byte memory reference could be stored in two consecutive second level tables. To solve this problem, the spare bits of the one-byte tag scheme are used. A one-byte tag stores information not only for the corresponding byte, but also the three bytes following it, *i.e.*, eight bits are used to indicate if four bytes are being watched for load or store. So the last byte in the table contains information about the first three bytes in the next region. It is also possible to use bit-maps in the second level to reduce the memory requirement. However such approach would make the lookup more complicated as additional operations such as shifts is required. Our approach uses a less compaq representation for an easier array indexing in the lookup process.

The two-bit tag to a byte location mapping allows us to perform an important optimization – the checking of multiple locations in a single comparison. For example, a four byte tag would hold the information for 16 bytes. In a single 32-bit width comparison, we can check the status of 16 byte locations. If comparison is done using 64-bit operands, even more lcoations can be checked in a single comparison.

C. Runtime Overhead

The naive software watchpoint implementation described above suffers from significant runtime overhead. In the next section, we will describe important optimizations to drastically reduce this overhead. However, to get a better understanding of what should be optimized, we need to uunderstand the overhead that we are dealing with. In the instrumented code, we can find three major overheads in each check:

- The overhead for saving and restoring registers so as to context switch to the checking code, O_S,
- The overhead in finding the corresponding tag, O_L, and
- The overhead in the checking O_C.

The total overhead in the naive implementation is therefore $(O_S + O_L + O_C) \times N$ where N is the number of memory references. As we shall see, the overhead can be reduced by decreasing any element in this formula. This is the subject of the next section. Although we did not consider cache locality in our design, we benefit our tag table lookup should exhibit good locality of reference.

D. Other Issues

With the trend of towards 64-bit multi-core processors, our implementation of software watchpoints should scale to multi-threaded or 64-bits applications. The extension for a 64-bit architecture has been discussed above. For multi-threaded applications, the lookup table can be made either global or thread-specific. This choice will depend on the user's preference. Meanwhile, because DynamoRIO employs a thread-private code cache, instrumentation can be performed on specific threads or on all threads.

Because DynamoRIO can only monitor and instrument the code executed in user mode, our software watchpoint cannot detect memory reference on watchpoints in kernel mode. However, by taking advantage of DynamoRIO being able to identify the system call instruction, *e.g* `int80`, we insert functions before and after the system call to discover its tyep, and predict if any watchpoint will be accessed. For instance, if a buffer containing watchpoints is used for read or write system call, we might trap it depending on the length.

IV. OPTIMIZATIONS

We have discussed the lookup overhead and memory usage tradeoffs involved in the design of the tag lookup table. In this section, we shall focus on optimizing the checking code injected into the user's application. Some of these optimizations reduce running time at the expense of using more memory. We perform optimization at two places – when the instrumentor inserts checking code into basic blocks, and when code fragments are upgraded into the trace cache.

A. Basic Block Optimizations

1) Context Switch Reduction (CSR): The naive checking code in Figure 3 consists of 13 instructions for context switches: six instructions for three resgisters stealing and restore, and seven for `eflags` save and restore. To reduce these overhead, a register liveness analysis is performed in each basic block. This analysis tries to determine registers that the checking code can use safely without saving and restoring. This reduces the overhead O_S.

2) Group Checks (GC): We can also reduce the total number of context switches by grouping checks together within a single context switch. Two consecutive memory reference checks can be grouped together if there is no instruction between them that affects the address calculation of the latter's memory reference, i.e. the register used by two memory references is not updated by the instructions in between. To be more aggressive, we can relax the condition such that if the update of the register for the address calculation can be statically computed, the checking can be aggregated. For instance, the checks for three consective `push` instructions can be grouped together since we know the way the stack register changes.

3) Merged Checks (MC): When the locations referenced by different instructions are the same or within close proximity of one another, we can merge the checks for these memory reference into a single check. As mentioned before, using a 4-byte tag for a single byte memory, we can check sixteen adjacent bytes of memory in a single comparison. At the cost of increased memory usage, we can reduce the total number of checks significantly.

B. Trace Optimizations (TO)

In DynamoRIO "hot" basic blocks are stitched together to form traces. So the code in trace are more frequently executed, and it is profitable to perform more aggressive optimizations on traces. By taking advantage of the single-entry, multi-exits property of a trace, the following optimizations can be performed.

1) Redundant Check Removal: In a trace, with the exception of the first basic block, a basic block cannot be executed unless the basic blocks preceding it are executed. Therefore a check is redundant in a basic block if an identical check has been performed in its predecessor blocks and can be safely removed.

2) Loop Invariant Check Hoisting: When executing code in a loop, it is often the case that the same location is referenced in different iterations. As in the case of traditional loop-invariant code motion, checks for loop invariant memory references can be moved outside the loop. In DynamoRIO, often a trace is a frequent path in a loop, and its last exit corresponds to the back-edge of the loop. For such traces, we can scan the code to see if there is any memory reference that does not change its address in the trace, *i.e.*, the registers used for address calculation are not changed. Checks for these memory references can be moved up to beyond the entry of trace so that the checks are only executed before entering the trace. It is possible that execution may exit earlier in a trace. This can cause a false alarm if checks at the entry discover a location being watched is to be accessed in the loop body but execution does not reach there. Instead of trapping immediately, a bit for the varible is set to 1 when the basic block is entered. The original lookup and check inside the loop is replaced with a single bit check that traps if the bit is set.

C. Page Protection based Instrumentation (PI) Mode

The optimizations described above are based on full instrumentation, *i.e.*, instrumenting every memory reference. Another optimization strategy we explored in EDDI is to instrument only those memory references that might potentially reference the watched data. We refer to this strategy as the *page protection based instrumentation* (PI) mode of EDDI, as oppose to the *full instrumentation* (FI) mode described in the preceding sections.

In PI mode, we make use of the operating system's page protection mechanism to help identify memory references that may access a watchpoint. When a watchpoint is set, we first make a copy of the pages containing that watchpoint. The access right of the original page is then set to be not accessible. During the program execution, if an instruction references one of this page, a signal will be raised. A signal handler catches that signal, and then subsequently replaces the code fragments containing that instruction with a new one with additional checking and reference redirection code before that instruction.

Here we use an example to explain how this is done. Suppose we want to set a watchpoint on data at address `0x020040` with a length of 64 bytes. Besides updating tag lookup table, EDDI will allocate a new page. Let's assume that this new page's address starts at `0x050000`. EDDI copies the data from [`0x020000`, `0x020fff`] to [`0x050000`, `0x050fff`], and updates a corresponding entry in an offset table to `0x030000`. This offset table is a table with 2^{20} entries for storing the redirection offsets of every pages. The page at `0x020000` is then set to be non-accessible. If an instruction, I, tries to load 4 bytes from `0x020010` for the first time, a `SIGSEGV` signal will be raised. EDDI's signal handler receives this signal, and locates the code fragment containing I and I itself. EDDI then builds a new code fragment that is identical as the old one except that a function call (a context switch is needed here for transparency) is inserted right before I. By means of a DynamoRIO utility function, the old fragment is replaced with new one, and execution is resumed at the function call. The function call calculates the reference address and check if it is watched and if redirection is needed. In our case, it is [0x020010, 0x020013]. There is no watchpoint triggering but a redirection is needed. A sequence of instructions are dynamically generated by EDDI and executed instead of I. The new generated code stub works the same as I, except for referencing 0x050010 and not 0x020010. At the end of this code stub, a branch instruction jumps to the instruction immediately after I, so that execution may continue. Hereafter, whenever this code fragment is executed again, the function before I will be executed first, checking the reference target, and making decision if to execute I or the redirection code stub. The rewriting process may be performed again if I tries to reference another protected page and a different set of redirection stub code with a different offset is required.

Beside the user code, we also insert function call before and after system call in case of protected page being referenced by kernel code. We maintain a list of system calls that may access the user's pages. If a system call is on this list, EDDI will first restore the original access right of those pages, and update the data from shadow pages. After the system call, EDDI will redo the watchpoints again.

By doing so, in PI mode, EDDI incurs an overhead only for instructions referencing pages that contain watchpoints, not *all* memory references as is the case of the FI mode. Our experiments show that this can significantly reduce overheads.

D. Watchpoint Scheduling

Because DynamoRIO does not change the data layout, the hardware watchpoint facility provided by architecture and made available via the debugger can still be used. By scheduling hardware and software watchpoints together, EDDI can provide a flexible watchpoint facility that does not compromise on the overhead. EDDI first runs the user application under DynamoRIO without any instrumentation, using hardware watchpoint first wherever possible. When hardware watchpoints are exhausted – this is caused by either having too many watchpoints or attempting to watch a wide range of memory – EDDI can switch to PI mode, clone and set those pages having watchpoints as non-accessible. Meanwhile, some watchpoints can still be selected to be hardware watchpoints. This will help reduce the overhead further. When it is no longer beneficial to pay for the overhead of PI mode, we can flushes the code cache and starts instrumenting the code in FI mode as execution continues.

V. Experiment

In this section, we evaluated the performance of our EDDI-based software watchpoint. We ran the experiments on a 3.06 GHz Intel Pentium 4 processor with 1G bytes of RAM. The operating system is Linux Fedora Core 1. We used the SPEC CPU2000 integer [16] benchmarks and their reference input workloads. All benchmarks were compiled with gcc 3.3 using -O3 flag.

```
int main()
{
  int i = 0;
  int local_val = 0;
  int *local_p  = &local_val;
  int *heap_p   = (int *)malloc(sizeof(int));

  for(i = 0; i < 100000000; i++) {
    if((i % 1000) >= 2)  // set the ratio
      *local_p = 1;
    else
      *heap_p  = 1;
  }
  return 0;
}
```

Fig. 5. Simple synthetic benchmark.

A. Basic Performance Results

We first evaluated the performance of the different instrumentation and optimization techniques without any watchpoints. We used a lookup table with 2^{20} entries in first level table. Each first level entry points to a second level tag table for 4 Kbytes of memory – the default page size of Linux. The performance measurements include naive instrumentation (Naive), and full instrumentation (FI) with the context switch reduction (CSR), group check (GC), merged checks (MC), and trace optimization (TO).

The performance result is nomalized to that of native execution and shown in Figure 4.

As expected, naive instrumentation has a significant overhead, averaging a 16 times slowdown in performance. However, the relative slowdown is benchmark dependent. For example, the overhead of naive instrumentation in 181.mcf is 27% compared to native execution. That is because 181.mcf is an application with poor cache locality, and there is probably a good deal of overlap between the checking and its cache misses. In contrast, benchmarks such as 186.crafty, 252.eon, and 255.vortex are applications with significantly higher cache hit ratios, so they suffered from substantial slowdown from executing the checking code.

The second set of bars in Figure 4 shows that CSR is a very effective optimization that effectively halves the overhead. That is reasonable since in naive instrumentation more than half of the injected checking instructions are used for context switching. When GC and MC optimizations are added, the overhead is halved further. These two optimizations are especially effective for the 186.crafty and 252.eon benchmarks. Memory references in these benchmarks have good temporal and spatial localities. On the other hand, 181.mcf benefits little from either the GC or MC optimizations. Performance improvement from the TO optimization was disappointing. This can be attributed to the good job done by the basic block optimizations.

The last set of bars represents the performance of the page protection based instrumentation (PI) mode of EDDI. In this case, because there are no watchpoints set, no instrumentation is injected and the slowdown is due purely to DynamoRIO.

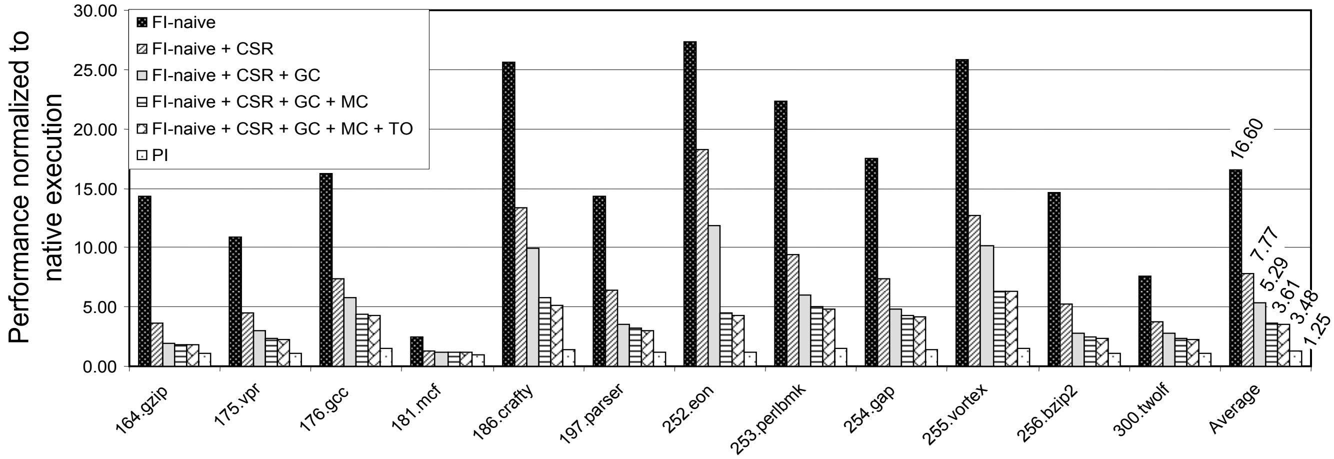

Fig. 4. Performance of EDDI under different optimizations for full instrumentation and page protection based instrumentation with no watchpoint set.

TABLE I

PERFORMANCE SCALING OF SYNTHETIC BENCHMARKS WITH RESPECT TO NUMBER OF WATCHPOINT ACCESSES. NATIVE EXECUTION TIME OF THE BENCHMARK IS 1.171 SECS.

Ratio of watchpt acc.	FI time (sec)	No. of sigsegv	No. of redirects	PI time (sec)
0.00%	1.876	1	1	1.221
0.10%	1.874	2	100,001	2.108
0.20%	1.811	2	200,001	2.939
0.30%	1.813	2	300,001	3.791
0.40%	1.814	2	400,001	4.674
0.50%	1.820	2	500,001	5.541
0.60%	1.823	2	600,001	6.433
0.70%	1.821	2	700,001	7.272
0.80%	1.823	2	800,001	8.171
0.90%	1.825	2	900,001	9.028
1.00%	1.847	2	1,000,001	9.904
1.10%	1.823	2	1,100,001	10.819
1.20%	1.837	2	1,200,001	11.678

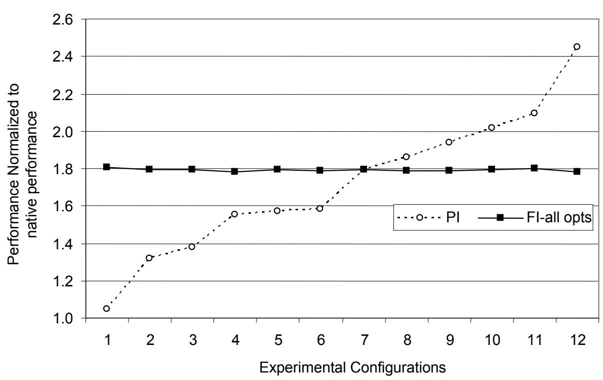

Fig. 6. Performance scaling for 164.gzip.

B. Scalability

In this section, we evaluate how performance changes when watchpoints are set. To this end, we first used a synthetic application so we can fine-tune the frequency of accessing the watchpoints. We then studied the performance changes on several benchmarks with different watchpoints setups. Since we do not have a model of how humans actually go about setting watchpoints, we used a somewhat contrived method. In our experiments, the way we set watchpoints is to intercept calls to the `malloc()` function, if we want to watch a

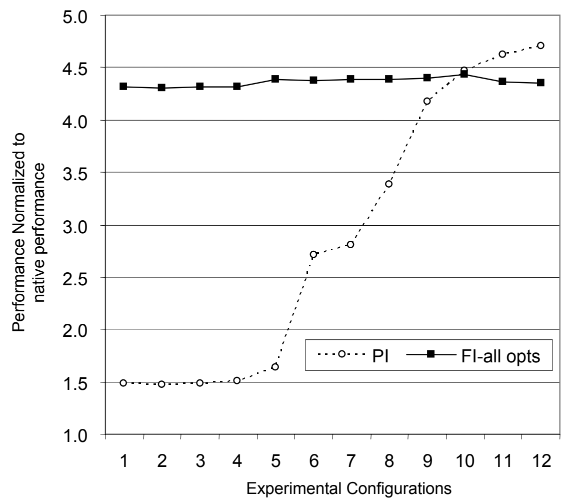

Fig. 7. Performance scaling for 176.gcc.

TABLE II

CONFIGURATIONS FOR `164.gzip` AND `176.gcc` BENCHMARKS USED IN OUR SCALABILITY EXPERIMENTS.

Config	164.gzip		176.gcc	
	No. of SIGSEGV	No. of redirects	No. of SIGSEGV	No. of redirects
1	22	858	31	135
2	33	2,835	36	1,022
3	35	12,159	49	1,556
4	42	60,852	100	3,736
5	42	117,930	962	642,775
6	42	183,838	1,312	6,458,137
7	45	580,135	1,448	6,809,815
8	45	1,063,054	107	10,512,472
9	45	1,679,386	433	14,750,028
10	45	2,176,154	455	16,326,327
11	45	2,787,984	1,459	16,983,414
12	45	5,682,609	1,555	17,348,450

malloc'ed block, we allocate that memory from a memory pool. All pages allocated from this pool are set as 'watched'. This allows us to set watchpoints easily at runtime for our experimental purpose, without any user intervention.

In the synthetic application, shown in Figure 5, we first allocate a block of memory and add it to the watchlist. Then using a loop, we reference either variables on the stack or variables in the allocated block. The choice is based on a tunable parameter so that we can study the relationship between

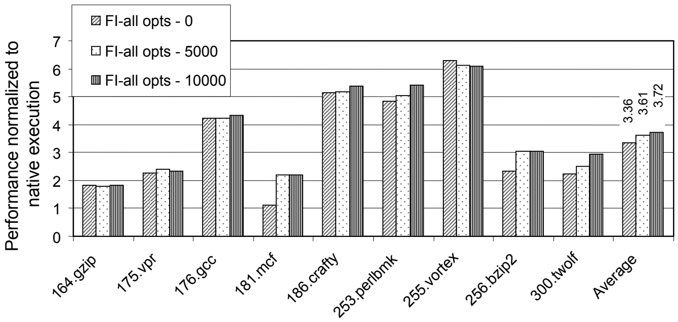

Fig. 8. Worst case performance of full instrumentation in EDDI.

performance and the frequency of accessing a watchpoint. The result is shown in Table I. The performance of the FI mode remains constant even as the frequency of referencing the watchpoints increases. We think this is because the second level check is simple and has relatively little performance impact compared to context switching and the first level check. In contrast, the performance of the PI mode degrades almost linearly with the increase in watchpoint references. From Table I, we can conclude every 10^5 reference redirection would result in a 0.9 second slowdown.

We repeated the scalability experiments, this time on two of the SPECint benchmarks, namely `164.gzip` and `176.gcc`. To evaluate the performance scaling in these benchmarks, we measured performance using different experimental configurations of watchpoints, i.e. different allocated memory blocks are watched. As one can see from Table II, the general trend in our experiments is to increase the number of `SIGSEGV` faults and page accesses. However, the exact details of how that can be achieved are benchmark dependent. In Table II, we listed the the number of `SIGSEGV` raised and the number of reference redirection performed for two of our benchmarks. These two factors is the main determinant of performance. The number of `SIGSEGV` raised indicates the number of *static* instructions in the application referencing protected pages, while the number of redirections gives the *dynamic* number of instructions referencing the protected pages.

The results are shown in Figures 6 and 7. The observations made in the synthetic benchmarks remained valid. From the experiments, we observe that performance of the FI mode is insensitive to the number of times a page containing a watchpoint is referenced. The opposite is true for PI mode operation. The main reason for this is that the cost of a data reference redirection is non-trivial and if more such redirections are performed because pages containing a watchpoint is accessed more often, then the overhead increases accordingly. Note that the number of `SIGSEGV` raised also affects the performance, but as evident in Table II, that number is much smaller than the number of reference redirections, and therefore the impact on performance is less pronounced.

We also compare the performance of the FI mode running under the best and worst case scenarios. Trivially, the best case is the case where there are no watchpoints. To evaluate the worst case, we watched (at most) 5,000, and 10,000 allocated memory blocks, or 1024 pages in total. Three benchmarks were ignored in the evaluation because `197.parser` allocated only very huge memory, and 254.gap does not request memory allocation, `252.eon` only allocates one 16 bytes blocks. We evaluated the rest. We did not evaluate the worst case for the PI mode since it is obviously going to be quite disasterous. As shown in Figure 8, we again conclude that the performance of FI is stable.

Our experiments verify our idea for watchpoint scheduling: one should start with hardware debug registers, then followed by PI mode, and when accesses to pages containing watchpoints increases, EDDI should switch over to the FI mode of operation. Although the current implementation of EDDI do not perform this scheduling, we see no technical difficulty that will prevent us from implementing it. The experimental results certainly point us in that direction.

VI. Related Work

A. Software Watchpoint

Watchpoint is an important debugging facility that helps users identify data corruption bugs. This importance has been given due recognition in the form of hardware debug registers for watching memory locations implemented in almost all state-of-the-art processors. There has also been several proposals in the past on how to implement software watchpoints. They can be generally classified as the following.

- **Using page protection:** pages containing watched addresses are protected from being written to and/or read from. Any attempt to reference a data location residing in these pages is trapped, and the reference address is checked against the watchpoints. Note that EDDI's PI mode goes further by optimizing on false sharing.
- **Through trap patching:** each store and/or load instruction is replaced with a trap instruction to trigger checking.
- **Through code patching:** the watchpoint checking code is inlined before each store and/or load instruction. This is the equivalent to the naive FI mode of EDDI.

Wahbe [18] and Roberts [13] 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 watchpoint [19] using code patching and static analysis. However, their work cannot be used on shared libraries. Copperman and Thomas [4] used a post-load to insert checks into executable to solve the shared library issue. Keppel [8] suggested using checkpointing to identify the short period of execution that updates the watchpoint, then re-execute that period with additional checks.

LIFT [12] is a dynamic instrumentation based information flow tracing system with a small runtime overhead. It is possible to apply the ideas given in this paper to extend their work so as to perform software watchpoint. However, LIFT makes use of a key advantage in their implementation framework that may not hold in general. LIFT performs dynamic translation to run 32-bit applications on a 64-bit system. This allows them to use up to eight extra registers available on the 64-bit x86 system that cannot be used by their 32-bit benchmark applications [12]. This completely avoids the significant overhead of context switching. We feel that EDDI's assumption of no 'free' registers is more congruent with how such a tool will be employed. Because the Intel

IA32 architecture only has eight general purpose registers, the overhead for register stealing is usually unavoidable.

B. Memory Debugging

There are many software, hardware or hybrid approaches proposed to detect memory bugs. Purify [7] and MemCheck [14] are two widely used software tools for discovering memory usage problems. HeapMon [15] is hardware/software approach for detecting memory bugs with a helper thread. Hardware such as SafeMem [11], iWatcher [21], and MemTracker [17] have also been proposed for detecting inappropriate usage of memory with low overhead. For example, SafeMem exploits the use of ECC (Error-Correcting Code) for detecting memory leaks and memory corruption with low overhead. But this approach relies on a write-allocate cache policy, *i.e.*, the block is loaded on a write miss. One important drawback of these hardware approaches is that it requires customized, fixed-functionality hardware.

There have been several hardware proposals to extend current hardware support for debugging. DISE [5] (Dynamic Instruction Steam Editing) is a general hardware mechanism for interactive debugging. It adds dynamic instructions for checking memory references into the instruction stream during execution. Beside requiring customized hardware, it does not scale well as the number of checks increases when more watchpoints are set. Without the optimizations found in EDDI, for example, this would cause a substantial runtime overhead.

VII. Conclusion

This paper presents EDDI, an efficient debugging framework that uses on-demand dynamic instrumentation and runtime optimizations to accelerate traditional debugging features. This paper emphasizes the use of EDDI to implement a versatile data watchpoint facility. It allows users to set orders of magnitude more watchpoints than is practical using off the shelf debuggers today. EDDI does not rely on any specialized hardware, and is evaluated in this paper using several SPEC2000 benchmarks running on a Pentium 4 machine.

The new capabilities facilitated by EDDI can make application developers more productive. In the case of data corruption bugs that occurs in pointer codes, or from buffer overflow errors, EDDI allows for tracking millions of memory locations for updates, with an average $3\times$ slowdown. A user can, for example, set data breakpoints for entire ranges of memory, or restrict watchpoints to objects allocated from specific call sites, or of a certain type. Though not yet implemented, it is also conceivable that instead of trapping, user supplied code can be executed on watchpoints. Such features extend the repertoire of features available in standard debuggers, and creates new debugging capabilities that are otherwise prohibitively expensive for interactive use. We believe that EDDI is also practical for detecting data races and data corruption bugs in parallel programs that will run on multicore architectures.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12. ACM Press, 2000.

[2] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004. http://www.cag.csail.mit.edu/rio/.

[3] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. *iscc*, 0:749–754, 2006.

[4] M. Copperman and J. Thomas. Poor man's watchpoints. *SIGPLAN Not.*, 30(1):37–44, 1995.

[5] M. L. Corliss, E. C. Lewis, and A. Roth. Low-overhead interactive debugging via dynamic instrumentation with dise. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 303–314, Washington, DC, USA, 2005. IEEE Computer Society.

[6] GNU/FSF. *GDB: The GNU Project Debugger*.

[7] R. Hastings and B. Joyce. Purify: fast detection of memory leaks and access errors. In *Proceedings of the Winter Usen x Conference*, 1992.

[8] D. Keppel. Fast data breakpoints. Technical Report TR-93-04-06, University of Washington, 1993.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[10] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004. http://valgrind.org/.

[11] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[12] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Micro 39: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[13] P. E. Roberts. Implementation and evaluation of data breakpoint schemes in an interactive debugger.

[14] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, 2005.

[15] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev.*, 50(2/3):261–275, 2006.

[16] SPEC CPU2000 Benchmark Suite, 2000. http://www.spec.org/osg/cpu2000/.

[17] G. Venkataramani, B. Roemer, M. Prvulovic, and Y. Solihin. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[18] R. Wahbe. Efficient data breakpoints. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 200–212, New York, NY, USA, 1992. ACM Press.

[19] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: design and implementation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1993. ACM Press.

[20] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 272–281, New York, NY, USA, 2006. ACM Press.

[21] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architectural support for software debugging. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 224, Washington, DC, USA, 2004. IEEE Computer Society.