Efficient and Flexible Architectural Support for Dynamic Monitoring

Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu and Josep Torrellas
Department of Computer Science, University of Illinois at Urbana-Champaign
{yyzhou,pinzhou,fengqin,liuwei,torrellas}@cs.uiuc.edu

Recent impressive performance improvements in computer architecture have not led to significant gains in ease of debugging. Software debugging often relies on inserting run-time software checks. In many cases, however, it is hard to find the root cause of a bug. Moreover, program execution typically slows down significantly, often by 10-100 times.

To address this problem, this article introduces the *Intelligent Watcher (iWatcher)*, a novel architectural scheme to monitor dynamic execution automatically, flexibly and with minimal overhead. iWatcher associates program-specified monitoring functions with memory locations. When any such location is accessed, the monitoring function is automatically triggered with low overhead. To further reduce overhead and support rollback, iWatcher can optionally leverage Thread-Level Speculation (TLS). The iWatcher architecture can be used to detect various bugs, including buffer overflow, accessing freed locations, memory leaks, stack-smashing and value-invariant violations. To evaluate iWatcher, we use seven applications with various real and injected bugs. Our results show that iWatcher detects many more software bugs than Valgrind, a well-known open-source bug detector. Moreover, iWatcher only induces a 0.1-179% execution overhead, which is orders of magnitude less than Valgrind. Our sensitivity study shows that even with 20% of the dynamic loads monitored in a program, iWatcher adds only 72-182% overhead. Finally, TLS is effective at reducing overheads for programs with substantial monitoring.

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms: Design, Reliability

Additional Key Words and Phrases: Architectural Support, Software Debugging, Dynamic Monitoring, Thread-Level Speculation (TLS)

An earlier version [Zhou et al. 2004] of this article appears in the 31th ACM International Symposium on Computer Architecture (ISCA2004). This article expands on the following aspects: First, Section 6 is newly added to describe how to detect different types of bugs; Second, in Sections 7 and 8, five more applications with real bugs (introduced by the original programmers) are used in our evaluation. Using these real bugs demonstrates that the proposed iWatcher architecture and bug detection methods (Section 6) are useful for real-world software; Third, Section 8.4 evaluates the effect of the number of microthread contexts in the processor on iWatcher overhead. This makes the evaluation more complete and gives some guidance on how many resources are necessary; Finally, Section 9 discusses other uses of iWatcher, such as performance debugging, interactive debugging or detection of other types of bugs not included in our evaluation.

1. INTRODUCTION

1.1 Motivation

Despite costly efforts to improve software-development methodologies, software bugs in deployed codes continue to thrive, often accounting for as much as 40% of computer system failures [Marcus and Stern 2000]. Software bugs can crash systems, making services unavailable or, in the form of "silent" bugs, corrupt information or generate wrong outputs. According to NIST [National Institute of Standards and Technlogy (NIST), Department of Commerce. 2002], software bugs cost the U.S. economy an estimated $59.5 billion annually, or 0.6% of the GDP!

There are several approaches to debug codes. One approach is to perform checks statically. Examples of this approach include explicit model checking [Musuvathi et al. 2002; Stern and Dill 1995] and program analysis [Choi et al. 2002; Engler and Ashcraft 2003; Hallem et al. 2002]. Most static tools require significant involvement of the programmer to write specifications or annotate programs. In addition, most static tools are limited by aliasing problems and other compile-time limitations. This is especially the case for programs written in unsafe languages such as C or C++, the predominant programming languages in industry. As a result, many bugs often remain in programs even after aggressive static checking.

Another approach is to monitor execution dynamically, with instrumentation inserted in the code that monitors invariants and reports violations as errors. The strength of this approach is that the analysis is based on actual execution paths and accurate values of variables and aliasing information. Examples of dynamic monitors include Purify [Hastings and Joyce 1992], Valgrind [Nethercote and Seward 2003], Intel Thread Checker [KAI-Intel Corporation], DIDUCE [Hangal and Lam 2002], Eraser [Savage et al. 1997], CCured [Condit et al. 2003; Necula et al. 2002], and other tools [Austin et al. 1994; Cowan et al. 1998; Loginov et al. 2001; Patil and Fischer 1997; 1995].

Unfortunately, most dynamic checkers suffer from two limitations. First, they are often computationally expensive. One major reason is their large instrumentation cost [Hangal and Lam 2002; Savage et al. 1997]. Another reason is that dynamic checkers may end up instrumenting more places than necessary due to lack of accurate information at instrumentation time. As a result, some dynamic checkers slow down a program by 6-30 times [Hangal and Lam 2002; Savage et al. 1997], which makes such tools undesirable for production runs. Moreover, some timing-sensitive bugs may never occur with these slowdowns.

Second, most dynamic checkers rely on compilers or pre-processing tools to insert instrumentation and, therefore, are limited by imperfect variable disambiguation. Consequently, some accesses to a monitored location may be missed by the instrumentation tool. Because of this reason, some bugs are caught much later than when they actually occur, which makes it hard to find the root cause of the bug. The following C code gives a simple example.

```
int x, *p;
           /* assume invariant: x = 1 */
...
p = foo(); /* a bug: p points to x incorrectly */
*p = 5;    /* line A:  corruption of x */
...
```

```
InvariantCheck(x == 1);  /* line B */
z = Array[x];
...
```

While x is corrupted in line A, the bug is not detected until the invariant check at line B. Due to the difficulty of performing perfect pointer disambiguation, it may be hard for a dynamic checker to know that it needs to insert an invariant check after line A.

To assist software debugging, several processor architectures such as Intel x86 and Sun SPARC provide support for watchpoints to monitor several programmer-specified memory locations [Intel Corporation 2001; Kane and Heinrich 1992; SPARC International 1992; Wahbe et al. 1993]. When a watched memory location is accessed, the hardware triggers an exception that is handled by the debugger. It is then up to the programmer to manually check the program state. While watchpoints are a good starting point, they have several limitations. First, they do not support *low-overhead* checks on variable values *automatically*. Since exceptions are expensive, it would be very inefficient to use them for dynamic bug detection during production runs. Second, most architectures only support a handful of watchpoints (four in Intel x86). Therefore, it is hard to use watchpoints for dynamic monitoring in production runs, which requires efficiency and watching many memory locations.

1.2 Our Approach

This article introduces the *Intelligent Watcher (iWatcher)*, a novel architectural scheme to monitor dynamic execution *automatically*, *flexibly* and with *minimal overhead*. iWatcher associates program-specified monitoring functions with memory locations. When any such location is accessed, the monitoring function is automatically triggered with low overhead. To further reduce overhead and support rollback, iWatcher can optionally leverage Thread-Level Speculation (TLS). The main advantages of iWatcher are:

—It monitors *all* accesses to the watched memory locations. Consequently, it catches hard-to-find bugs such as updates through aliased pointers and stack-smashing attacks commonly exploited by viruses.

—It has low overhead because it (i) only monitors memory instructions that *truly* access the watched memory locations, and (ii) uses minimal-overhead hardware-supported triggering of monitoring functions.

—It is flexible in that it can support a wide range of checks, including program-specific checks. Moreover, iWatcher is language independent, cross-module and cross-developer.

—It can *optionally* leverage TLS to hide monitoring overhead and provide rollback support. Specifically, with TLS, a monitoring function is executed in parallel with the rest of the program, and the program can be rolled back if a bug is found.

In contrast, due to aliasing problems, it is very hard for software-only dynamic checkers to monitor all accesses to the watched memory locations and only those.

We evaluate iWatcher using seven buggy applications with various real and injected bugs including accessing freed locations, memory leaks, buffer overflow, value-invariant violations, and smashed stacks. iWatcher detects all the bugs evaluated in our experiments with only a 0.1-179% execution overhead. Overall, iWatcher's reasonably small overhead and ability to monitor many memory locations enable it to be used in both in-house testing and production runs. In contrast, a well-known open-source bug detector called Valgrind

induces orders of magnitude more overhead, and can only detect a subset of the bugs. Moreover, even with 20% of the dynamic loads monitored in a program, iWatcher only adds 72-182% overhead. We also show that TLS is effective at reducing overheads for programs with substantial monitoring. Finally, supporting four contexts in an SMT is enough to achieve the best performance in our experiments.

This article is organized as follows. Section 2 briefly describes some background. Sections 3, 4, and 5 describe iWatcher's functionality, architectural design, and advantages, respectively. Section 6 describes how to use iWatcher to detect various bugs. Sections 7 and 8 present the evaluation methodology and experimental results. Section 9 discusses some other uses of iWatcher. Section 10 discusses related work and Section 11 concludes.

2. BACKGROUND

2.1 Dynamic Execution Monitoring

Many methods have been proposed for dynamic code monitoring. The most commonly used ones are assertions, dynamic checkers, and watchpoints.

Assertions. Assertions are inserted by programmers to perform sanity checks at certain places. If the condition specified in an assertion is false, the program aborts. Assertions are one of the most commonly used methods for debugging. However, they can add significant overhead to program execution. Moreover, it is often hard to identify all the places where assertions should be placed.

Dynamic Checkers. Dynamic checkers are automated tools that detect common bugs at run time. For example, DIDUCE [Hangal and Lam 2002] automatically infers likely program invariants, and uses them to detect program bugs. Purify [Hastings and Joyce 1992] and Valgrind [Nethercote and Seward 2003] monitor memory accesses to detect memory leaks and some simple instances of memory corruption, such as freeing a buffer twice or reading an uninitialized memory location. StackGuard [Cowan et al. 1998] can detect some buffer overflow bugs, which have been a major cause of security attacks. Eraser [Savage et al. 1997] can detect data races by dynamically tracking the set of locks held during program execution. These tools usually use compilers or code-rewriting tools such as ATOM [Srivastava and Eustace 1994], EEL [Larus and Schnarr 1995] and Dyninst [Buck and Hollingsworth 2000] to instrument programs with checks.

While this approach is promising, dynamic checkers often suffer from the following limitations: (1) aliasing problems, especially in C or C++ programs, (2) high run-time overhead, (3) hard-coded bug detection functionality, (4) language specificity, and (5) difficulty to work with low-level code.

Hardware-Assisted Watchpoints. Hardware-assisted watchpoints [Intel Corporation 2001; Kane and Heinrich 1992; SPARC International 1992] use simple hardware support to watch a user-selected memory location. When a watched location is accessed by the program, an exception is handled by an interactive debugger such as gdb. Then, the state of the process can be examined by programmers using the debugger. The hardware support is provided through a few special debug registers. Watchpoints are designed to be used in an interactive debugger. For non-interactive execution monitoring, they are both inflexible and inefficient. They do not provide a way to associate an automatic check to the access of a watched location. Moreover, they require an expensive exception when a watched location is accessed. Finally, most architectures only support a few watchpoints (four in

Intel's x86).

2.2 Classifying Dynamic Monitoring Methods

We classify the dynamic monitoring methods into two categories:

—*Code-Controlled Monitoring (CCM).* Monitoring is performed only at special points in the program. Assertions and most dynamic checkers belong to CCM because they only check at assertions or instrumentation points.

—*Location-Controlled Monitoring (LCM).* Monitoring is associated directly with memory locations and therefore all accesses to such memory locations are monitored. Hardware-assisted watchpoints and iWatcher belong to this category.

LCM has two advantages over CCM: (1) LCM monitors *all* accesses to a watched memory location using all possible variable names or pointers, whereas CCM may miss some accesses because of pointer aliasing; (2) LCM monitors only those memory instructions that *truly* access a watched memory location, whereas CCM may need to instrument at many unnecessary points due to the lack of accurate information at instrumentation time. Therefore, LCM can be used to detect both invariant violations and illegal accesses to a memory location, whereas it may be difficult and too expensive for CCM to check for illegal accesses. The main advantage of CCM is that it does not require hardware support while LCM typically needs it.

2.3 Thread-Level Speculation (TLS)

TLS is an architectural technique for speculative parallelization of sequential programs [Cintra et al. 2000; Sohi et al. 1995; Steffan et al. 2000; Tsai et al. 1999]. TLS support can be built on a multithreaded architecture, such as simultaneous multithreading (SMT) or chip multiprocessor (CMP) machines. With TLS, the execution of a sequential program is divided into a sequence of *microthreads* (also called tasks, slices, or epochs). These microthreads are then executed speculatively in parallel, while special hardware detects violations of the program's sequential semantics. A violation results in squashing the incorrectly executed microthreads and re-executing them. To enable squash and re-execution, the memory state of each speculative microthread is typically buffered in caches or special buffers. When a microthread finishes its execution and becomes safe, it can commit. Committing a microthread implies merging its state with the safe memory. To guarantee sequential semantics, microthreads commit in order.

iWatcher can leverage TLS to reduce monitoring overhead and to support rollback and re-execution of a buggy code region [Prvulovic and Torrellas 2003]. For our design, we assume an SMT machine, and that the speculative memory state is buffered in caches. However, our iWatcher design can be easily ported to other TLS architectures.

If we use TLS in iWatcher, each cache line is tagged with the ID of the microthread to which the line belongs. Moreover, for each speculative microthread, the processor contains a copy of the initial state of the architectural registers. This copy is generated when the speculative microthread is spawned and is freed when the microthread commits. It is used in case the microthread needs to be rolled back.

The TLS mechanisms for in-cache state buffering and rollback can be reused to support incremental rollback and re-execution of the buggy code [Prvulovic and Torrellas 2003]. To do this, the basic TLS is modified slightly by postponing the commit time of a successful microthread. In the basic TLS, a microthread can commit when it completes and all

its predecessors have committed. We say that such a microthread is *ready*. To support the rollback of buggy code, a ready microthread commits only in one of two cases: when we need space in the cache and when the number of uncommitted microthreads exceeds a certain threshold. With this support, a ready but uncommitted microthread can still be asked to rollback. This feature can be used to support one of the iWatcher modes (Section 4.5).

3. IWATCHER FUNCTIONALITY

iWatcher provides high-flexibility and low-overhead dynamic execution monitoring. It associates program-specified monitoring functions with memory locations. When any such location is accessed, the monitoring function associated with it is automatically triggered and executed.

iWatcher provides two system calls to turn on and off monitoring on a memory location, namely *iWatcherOn* and *iWatcherOff*. These calls can be inserted in programs either automatically by an instrumentation tool or manually by programmers. The following is the *iWatcherOn* interface:

```
iWatcherOn(MemAddr, Length, WatchFlag, ReactMode,
   MonitorFunc, Param1, Param2, ... ParamN)
/* MemAddr: starting address of the memory region*/
/* Length: length of the memory region */
/* WatchFlag: types of accesses to be monitored */
/* ReactMode: reaction mode */
/* MonitorFunc: monitoring function */
/* Param1...ParamN: parameters of MonitorFunc */
```

If a program makes such a call, iWatcher associates monitoring function $MonitorFunc()$ with a memory region of $Length$ bytes starting at $MemAddr$. The $WatchFlag$ specifies what types of accesses to this memory region should be monitored. Its value can be "READONLY", "WRITEONLY", or "READWRITE", in which case the monitoring function is triggered on a read access, write access or both, respectively.

At a *triggering access* (an access to a monitored memory location), the hardware automatically initiates the monitoring function associated with this memory location. The architecture passes the values of $Param1$ through $ParamN$ to the monitoring function. In addition, it also passes information about the triggering access, including the program counter, the type of access (load or store; word, half-word, or byte access), reaction mode, and the memory location being accessed. It is the monitoring function's responsibility to perform the check.

A monitoring function can have side effects and can read and write variables without any restrictions. To avoid recursive triggering of monitoring functions, no memory access performed inside a monitoring function can trigger another monitoring function.

From the programmers' point of view, the execution of a monitoring function follows sequential semantics, just like a very lightweight exception handler (Section 4 describes why monitoring in iWatcher is very lightweight). The semantic order is: the triggering access, the monitoring function, and the rest of the program after the triggering access.

Upon successful completion of a monitoring function, the program continues normally. If the monitoring function fails (returns FALSE), different actions are taken depending on the $ReactMode$ parameter specified in iWatcherOn(). iWatcher supports three modes: $ReportMode$, $BreakMode$ and $RollbackMode$:

—$ReportMode$: The monitoring function reports the outcome of the check and lets the program continue. This mode can be used for profiling and error reporting without interfering with the execution of the program.

—$BreakMode$: The program pauses at the state right after the triggering access and control is passed to an exception handler. Users can potentially attach an interactive debugger, which can be used to find more information.

—$RollbackMode$: The program rolls back to the most recent checkpoint, typically much before the triggering access. This mode can be used to support replay of a code section to analyze an occurring bug [Prvulovic and Torrellas 2003], or to support transaction-based programming [Oplinger and Lam 2002].

A program can associate multiple monitoring functions with the same location. In this case, upon an access to the watched location, all monitoring functions are executed following sequential semantics according to their setup order.

When a program is no longer interested in monitoring a memory region, it turns off the monitoring using

```
iWatcherOff(MemAddr, Length, WatchFlag, MonitorFunc)
/* MemAddr: starting address of the watched region*/
/* Length: length of the watched region */
/* WatchFlag: types of accesses to be unmonitored */
/* MonitorFunc: the monitoring function */
```

After this operation, the $MonitorFunc$ associated with this memory region of $Length$ bytes starting at $MemAddr$ and $WatchFlag$ is deleted from the system. Other monitoring functions associated with this region are still in effect.

Besides using the iWatcherOff() call to turn off monitoring for a specified memory region, the program can also use a $MonitorFlag$ global switch that enables or disables monitoring on all watched locations. This switch is useful when monitoring overhead is a concern. When the switch is disabled, no location is watched and the overhead imposed is negligible.

Note that iWatcher only provides a very flexible mechanism for dynamic execution monitoring. It is not iWatcher's responsibility to ensure that a monitoring function is written correctly, just like an *assert(condition)* call cannot guarantee that the condition in the code is correct. Programmers can use invariant-inferring tools such as DIDUCE [Hangal and Lam 2002] and DAIKON [Ernst et al. 2000] to automatically insert iWatcherOn() and iWatcherOff() calls into programs.

With this support, we can rewrite the example of Section 1 using iWatcherOn() and iWatcherOff() operations. There is no need to insert the invariant check. iWatcherOn() is inserted at the very beginning of the program so that the system can continuously check x's value whenever and however the memory location is accessed. This way, the bug is caught at line A.

```
int x, *p;
           /* assume invariant: x = 1 */
iWatcherOn(&x, sizeof(int), READWRITE,
           BreakMode, &MonitorX, &x, 1);
...
p = foo(); /* a bug: p points to x incorrectly */
*p = 5;    /* line A: a triggering access */
```

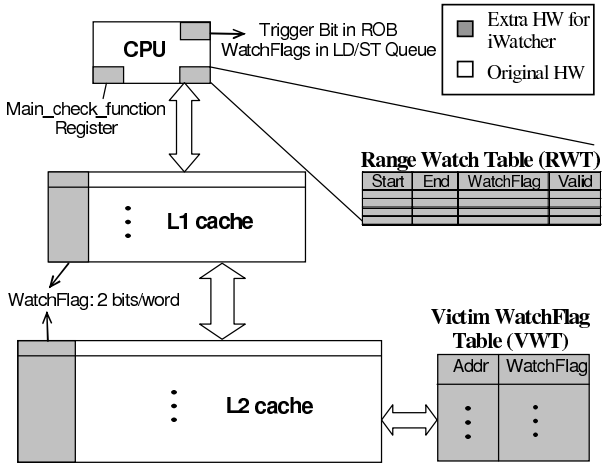


Fig. 1. iWatcher hardware architecture.

```
...
z = Array[x]; /* line B: a triggering access */
...
iWatcherOff(&x, sizeof(int), READWRITE, &MonitorX);

bool MonitorX(int *x, int value){
  return  (*x == value);
}
```

4. ARCHITECTURAL DESIGN OF IWATCHER

To implement the functionality described above, there are at least four challenges: (1) How to monitor a location? (2) How to detect a triggering access? (3) How to trigger a monitoring function? (4) How to support the three reaction modes? In this section, we first give an overview of the implementation and then show how it addresses these challenges.

4.1 Overview of the Implementation

iWatcher is implemented using a combination of hardware and software. Logically, it has four main parts. First, to detect triggering accesses on small monitored memory regions, we tag cache lines in both L1 and L2 caches with WatchFlags; to detect triggering accesses on large monitored memory regions, we use a small *Range Watch Table (RWT)*. Second, the hardware triggers monitoring functions on the fly and provides a special *Main_check_function* register to store the common entry point for all monitoring functions. Third, we use software to manage the associations between watched locations and monitoring functions. Finally, we *optionally* leverage TLS to reduce overheads.

Figure 1 gives an overview of the iWatcher hardware. Each L1 and L2 cache line is augmented with WatchFlags. They identify words belonging to small monitored memory regions. There are two WatchFlag bits per word in the line: a read-monitoring one and a write-monitoring one. If the read (write)-monitoring bit is set for a word, all loads (stores) to this word automatically trigger the corresponding monitoring function. The processor also has a Main_check_function register that holds the address of the *Main_check_function()*,

which is the common entry point to all program-specified monitoring functions. In addition, iWatcher also has a *Victim WatchFlag Table (VWT)*, which stores the WatchFlags for watched lines of small regions that have at some point been displaced from L2.

To detect accesses to large (multiple pages) monitored memory regions, iWatcher uses a set of registers organized in the RWT. Each RWT entry stores the start and end virtual addresses of a large region being monitored, plus two bits of WatchFlags and one valid bit. We will see that the RWT is used to prevent large monitored regions from overflowing the L2 cache and the VWT. The lines of these regions are not brought into the cache on an iWatcherOn() call. Moreover, the WatchFlags of these lines do not need to be set in the L1 or L2 cache unless the lines are also included in a small monitored region. When the RWT is full, additional large monitored regions are treated the same way as small regions.

The software component of iWatcher includes the iWatcherOn/Off() system calls, which set or remove associations of memory locations with monitoring functions. iWatcher uses a software table called *Check Table* to store detailed monitoring information for each watched memory location. The information stored includes MemAddr, Length, WatchFlag, ReactMode, MonitorFunc, and Parameters. Using software simplifies the hardware. An iWatcherOn/Off() call adds or removes the corresponding entry to or from the Check Table.

The iWatcher software also implements the Main_check_function() library call, whose starting address is stored in the Main_check_function register. When a triggering access occurs, the hardware sets the program counter to the address in this register. The Main_check_function() is responsible to call the program-specified monitoring function(s) associated with the accessed location. To do this, it needs to search the Check Table and find the corresponding function(s).

To reduce monitoring overhead, iWatcher can optionally leverage TLS to speculatively execute the main program in parallel with monitoring functions. Moreover, iWatcher can also leverage TLS to roll back the buggy code with low overhead, for subsequent replay.

While TLS was also used by Oplinger and Lam to hide overheads [Oplinger and Lam 2002], iWatcher uses a different TLS spawning mechanism. Specifically, iWatcher uses dynamic hardware spawning, which requires no code instrumentation. Oplinger and Lam, instead, insert thread-spawning instructions in the program statically. In general, their approach is less efficient and may hurt some conventional compiler optimizations. Many of the new issues that appear with dynamic hardware spawning are discussed in Sections 4.3 and 4.4.

4.2 Watching a Range of Addresses

When a program calls iWatcherOn() for a memory region equal or larger than *LargeRegion*, iWatcher tries to allocate an RWT entry for this region. If there is already an entry for this region in the RWT, iWatcherOn() sets the entry's WatchFlags to the logical OR of its old value and the WatchFlag argument of the call. If, instead, the region to be monitored is smaller than *LargeRegion*, iWatcher loads the watched memory lines into the L2 cache (if they are not already in L2). We do not explicitly load the lines into L1 to avoid unnecessarily polluting L1. As a line is loaded from memory, iWatcher accesses the VWT to read-in the old WatchFlags, if they exist there. Then, it sets the WatchFlag bits in the L2 line to be the logical OR of the WatchFlag argument of the call and the old WatchFlags. If the line is already present in L2 and possibly L1, iWatcher simply sets the WatchFlag bits in the line to the logical OR of the WatchFlag argument and the current WatchFlag. In all

cases, iWatcherOn() also adds the monitoring function to the Check Table.

When a program calls iWatcherOff(), iWatcher removes the corresponding monitoring function entry from the Check Table. Moreover, if the monitored region is large and there is a corresponding RWT entry, iWatcherOff() updates this RWT entry's WatchFlags. The new value of the WatchFlags is computed from the remaining monitoring functions associated with this memory region, according to the information in the Check Table. If there is no remaining monitoring function for this range, the RWT entry is invalidated. If, instead, the memory region is small, iWatcher finds all the lines of the region that are currently cached and updates their WatchFlags based on the remaining monitoring functions. iWatcher also updates (and, if appropriately removes) any corresponding VWT entries.

Caches and VWT are addressed by the physical addresses of watched memory regions. If there is no paging by the OS, the mapping between physical and virtual addresses is fixed for the whole program execution. In our prototype implementation, we assume that watched memory locations are pinned by the OS, so that the page mappings of a watched region do not change until the monitoring for this region is disabled using iWatcherOff().

Note that the purpose of using RWT for large regions is to reduce L2 pollution and VWT space consumption: lines from this region will only be cached when referenced (not during iWatcherOn()) and, since they will not set their WatchFlags in the cache, they will not use space in the VWT on cache eviction.

It is possible that iWatcherOn()/iWatcherOff() access some memory locations sometimes as part of a large region and sometimes as a small region. In this case, the iWatcherOn() or iWatcherOff() software handlers, as they add or remove entries to or from the Check Table, are responsible for ensuring the consistency between RWT entries and L2/VWT WatchFlags.

4.3 Detecting Triggering Accesses

iWatcher needs to identify those loads and stores that should trigger monitoring functions. A load or store is a triggering access if the accessed location is inside any large monitored region recorded in the RWT, or the WatchFlags of the accessed line in L1/L2 are set.

In practice, the process of detecting a triggering access is complicated by the fact that modern out-of-order processors introduce access reordering and pipelining. To help in this process, iWatcher augments each reorder buffer (ROB) entry with a *Trigger* bit, and each load-store queue entry with 2 bits that store WatchFlag information.

To keep the hardware reasonably simple, the execution of a monitoring function should only occur when a triggering load or store reaches the head of the ROB. At that point, the values of the architectural registers that need to be passed to the monitoring function are readily available. In addition, the memory system is consistent, as it contains the effect of all preceding stores. Moreover, there is no danger of mispredicted branches or exceptions, which could require the cancellation of an early-triggered monitoring function.

For a load or store, when the TLB is looked up early in the pipeline, the hardware also checks the RWT for a match. This introduces negligible visible delay. If there is a match, the access is a triggering one. If there is no match, the WatchFlags in the caches will be examined to determine if it is a triggering access.

A load typically accesses the memory system before reaching the head of the ROB. It is at that time that a triggering load will detect the set WatchFlags in the cache. Consequently, in our design, as a load reads the data from the cache into the load queue, it also reads the WatchFlag bits into the special storage provided in the load queue entry. In

addition, if the RWT or the WatchFlag bits indicate that the load is a triggering one, the Trigger bit associated with the load's ROB entry is set. When the load (or any instruction) finally reaches the head of the ROB and is about to retire, the hardware checks the Trigger bit. If it is set, the hardware triggers the corresponding monitoring function.

Stores present a special difficulty. A store is not sent to the memory system until it reaches the head of the ROB. At that point, it is retired immediately, but it may still cause a cache miss, in which case it may take a long time to actually complete. In iWatcher, this would mean that, for stores that do not hit in the RWT, the processor may have to wait a long time to know whether it is a triggering access. During that time, no subsequent instruction could be retired, as the processor may have to trigger a monitoring function. To reduce this delay as much as possible, we change the microarchitecture so that, as soon as a store address is resolved early in the ROB, a prefetch is issued to the memory system. Such prefetch brings the data into the cache, and the WatchFlag bits are read into the special storage in the store queue entry. If the RWT or the WatchFlag bits in the caches indicate that the store is a triggering one, the Trigger bit in the ROB entry is also set. With this support, the processor is much less likely to have to wait when the store reaches the head of the ROB. While issuing this prefetch may have implications for the memory consistency model supported in a multiprocessor environment, we consider the topic to be beyond the scope of this article.

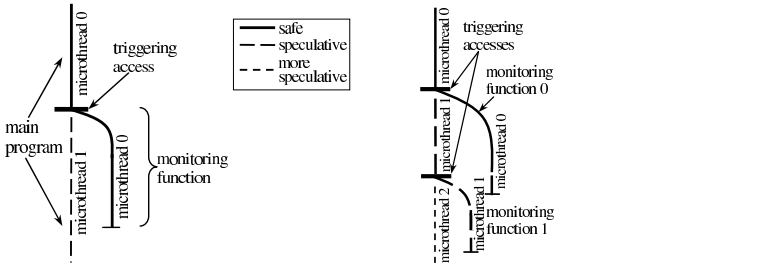
Note that bringing the WatchFlag information into the load-store queue entries enables correct operation for loads that get their data directly from the load-store queue. For example, if a store in the load-store queue has the read-monitoring WatchFlag bit set, then a load that reads from it will set its own Trigger bit.

4.4 Executing Monitoring Functions

When a triggering load or store is retired, the architectural registers and the program counter are automatically saved, and execution is redirected to the address in the Main_check_function register. After the monitoring function completes, execution resumes from the saved program counter.

As an optimization, we can leverage the TLS mechanism. Specifically, when a triggering load or store is retired, the iWatcher hardware automatically spawns a new microthread (denoted as microthread 1 in Figure 2(a)) to speculatively execute the rest of the program after the triggering access, while the current microthread (denoted as microthread 0 in Figure 2(a)) executes the monitoring function non-speculatively. To provide sequential semantics (the remainder of the program is semantically after the monitoring function), data dependencies are tracked by TLS and any violation of sequential semantics results in the squash of the speculative microthread (microthread 1).

Microthread 0 executes the monitoring function by starting from the address stored in the Main_check_function register. It is the responsibility of the Main_check_function() to find the monitoring function(s) associated with the triggering access and call all such function(s) one after another. Note that, although semantically, a monitoring function appears to programmers like a user-specified exception handler, the overhead of triggering a monitoring function is tiny with our hardware support. Indeed, while triggering an exception handler typically needs OS involvement, triggering a monitoring function in iWatcher is done completely in hardware: the hardware automatically fetches the first instruction from the Main_check_function(). iWatcher can skip the OS because monitoring functions are not related to any resource management in the system and, in addition, do not need to be



(a) *Executing a monitoring function.* (b) *Triggering a monitoring function from a speculative microthread.*

Fig. 2. Examples of monitoring function execution with TLS support.

executed in privileged mode. Moreover, the monitoring functions for a program are in the same address space as the monitored program. Therefore, a "bad" program cannot use iWatcher to mess up other programs.

Microthread 1 speculatively executes the continuation of the monitoring function, i.e., the remainder of the program after the triggering access. To avoid the overhead of flushing the pipeline, iWatcher dynamically changes the microthread ID of all the instructions currently in the pipeline from 0 to 1. Note that, it is possible that some un-retired load instructions after the triggering access may have already accessed the data in the cache and, as per TLS, already updated the microthread ID in the cache line to be 0. Since the microthread ID on these cache lines should now be 1, the hardware re-touches the cache lines that were read by these un-retired loads, correctly setting their microthread IDs to 1. There is no such problem for stores because they only update the microthread IDs in the cache at retirement. In our experiments, these retouches account for a very tiny fraction of all accesses, and practically always hit the L1 cache. So the performance impact is negligible.

It is possible that a speculative microthread issues a triggering access, as shown on Figure 2(b). In this case, a more speculative microthread (microthread 2) is spawned to execute the rest of the program, while the speculative microthread (microthread 1) enters the Main_check_function. Since microthread 2 is semantically after microthread 1, a violation of sequential semantics will result in the squash of microthread 2. In addition, if microthread 1 is squashed, microthread 2 is squashed as well. Finally, if microthread 1 completes while speculative, iWatcher does not commit it; it can only be committed after microthread 1 becomes safe.

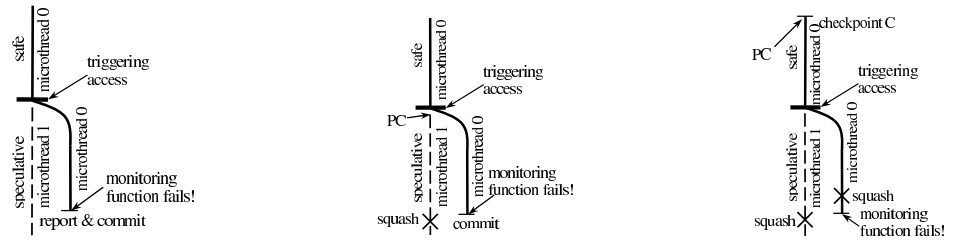
Note that, in a CMP-based iWatcher, microthreads should be allocated for cache affinity. In our Figure 2(a) example, speculative microthread 1 should be kept on the same CPU as the original program, while microthread 0 should be moved to a different CPU. This is because microthread 1 continues to execute the program and is likely to reuse cache state.

4.5 Different Reaction Modes

If a monitoring function fails, iWatcher takes different actions depending on the function's ReactMode. ReactMode can be *ReportMode*, *BreakMode*, and *RollbackMode*.

In *ReportMode*, the monitoring function reports the outcome of the check and lets the program continue. This mode is used for profiling and error reporting without interfering with the execution of the program.

In *BreakMode*, the program pauses at the state right after the triggering access, and



(a) *ReportMode.* (b) *BreakMode.* (c) *RollbackMode.*

Fig. 3. Different reaction modes supported by iWatcher with the TLS optimization.

control passes to an exception handler. Users can attach an interactive debugger, which can be used to find more information.

Finally, in *RollbackMode*, the program rolls back to a previous checkpoint, typically much earlier than the triggering access. This mode can be used to support the replay of a code section to analyze a bug, or to support transaction-based programming.

Figure 3 illustrates the three supported reaction modes with the TLS optimization. *ReportMode* is the simplest one. iWatcher treats it the same way as if the monitoring function had succeeded: microthread 0 commits and microthread 1 becomes safe. If the reaction mode is $BreakMode$, iWatcher commits microthread 0 but squashes microthread 1. The program state and the program counter (PC) of microthread 1 are restored to the state it had immediately after the triggering access (Figure 3(b)). The cache updates of microthread 1 are discarded. At this point, programmers can use an interactive debugger to analyze the bug.

If the reaction mode is $RollbackMode$, iWatcher squashes microthread 1 and also rolls back microthread 0 to a previous checkpoint (the checkpoint at PC in Figure 3(c)). iWatcher can use support similar to ReEnact [Prvulovic and Torrellas 2003] to provide this reaction mode.

4.6 Other Issues

Displacements and Cache Misses. When a watched line of small regions is about to be displaced from the L2 cache, its WatchFlags are saved in the VWT. The VWT is a small set-associative buffer. If the VWT needs to take an entry while full, it selects a victim entry to be evicted, and delivers an exception. The OS then turns on page protections for the pages that correspond to the WatchFlags to be evicted from the VWT. Future accesses to these pages will trigger page protection faults, which will enable the OS to insert their WatchFlags back into the VWT. However, in our experiments, we find that a 1024-entry VWT is never full. The reason is that the VWT only keeps the WatchFlags for watched lines of small regions that have at some point been displaced from L2.

On an L2 cache miss, as the line is read from memory, the VWT is checked for an address match. If there is a match, the WatchFlags for the line are copied to the destination location in the cache. We do not remove the WatchFlags from the VWT because the memory access may be speculative and be eventually undone. If there is no match, the WatchFlags for the loaded line are set to the default “un-watched” value. Note that this VWT lookup is performed in parallel with the memory read and, therefore, introduces negligible visible delay.

If TLS is used, speculative lines cannot be displaced from the L2. If space is needed in a cache set that only holds speculative lines, a speculative microthread is squashed to make room. More details can be found in [Prvulovic and Torrellas 2003].

Check Table Implementation. The Check Table is a software table. Our current implementation uses one entry for each watched region. The entries are sorted by start address. To speed-up Check Table lookup, we exploit memory access locality to reduce the number of table entries accessed during one search. A table entry contains all arguments of the iWatcherOn() call. If there are multiple monitoring functions associated with the same location, they are linked together. Since the Check Table is a pure software data structure, it is easy to change its implementation. For example, a possible implementation could be to organize it as a hash table. It can be hashed with the virtual address of the watched location.

5. ADVANTAGES OF IWATCHER

Based on the previous discussion, we can list the advantages of iWatcher. One of them is that it provides location-controlled monitoring. Therefore, *all* accesses to a watched memory location are monitored, including "disguised" accesses due to dangling pointers or wrong pointer manipulations.

Another advantage of iWatcher is its low overhead. iWatcher only monitors memory operations that truly access a watched memory location. Moreover, iWatcher uses hardware to trigger monitoring functions with minimal overhead. Finally, iWatcher optionally uses TLS to execute monitoring functions in parallel with the rest of the program, effectively hiding most of the monitoring overhead.

iWatcher is flexible and extensible. Programmers or automatic instrumentation tools can add monitoring functions. iWatcher is convenient even for manual instrumentation because programmers do not need to instrument every possible access to a watched memory location. Instead, they only need to insert an iWatcherOn() call for a location when they are interested in monitoring this location and an iWatcherOff() call when the monitoring is no longer needed. In between, all possible accesses to this location are automatically monitored. In addition, iWatcher supports three reaction modes, giving flexibility to the system.

iWatcher is cross-module and cross-developer. A watched location inserted by one module or one developer is automatically honored by all modules and all developers whenever the watched location is accessed.

iWatcher is language independent since it is supported directly in hardware. Programs written in any language, including C/C++, Java or other languages can use iWatcher. For the same reason, iWatcher can also support dynamic monitoring of low-level system software, including the operating system.

iWatcher can be used to detect illegal accesses to a memory location. For example, it can be used for security checks to prevent illegal accesses to some secured memory locations. In our experiments, we have used iWatcher to protect the return address in a program stack to detect stack-smashing attacks [Cowan et al. 1998; Frantzen and Shuey 2001; One 1996; Xu et al. 2002].

Table I summarizes the differences between iWatcher and the three related approaches discussed in Section 2.

Feature	Assertions	Hardware Watchpoints	Software-Only Dynamic Checkers		iWatcher
			Valgrind	Generic	
Hardware support	No	Simple	No	No	Modest. Optionally, TLS
Type of checks	Code-controlled	Location-controlled	Code-controlled	Code-controlled	Location-controlled
Reaction modes	Abort	Interrupt	Report or break	Report or break	Report, break or rollback
Programmer's effort	High	High	Low	Low	Moderate or low (automatic instrumentation)
Language dependent	No	No	No	Typically yes	No
Flexibility	Very flexible. Program specific	Inflexible. Support only a few watchpoints. Rely on programmers or debuggers for checks	Moderately flexible. Currently, it only supports limited types of checks	Moderately flexible	Very flexible. Program specific
Cross-module and cross-developer	No	Yes	Yes	No	Yes
Completeness	Hard to make sure all accesses are checked	Detects all accesses	Detects all accesses	May miss some accesses due to aliasing problems	Detects all accesses
Overhead	High	Low	Very high	Typically high	Low

Table I. Comparison of iWatcher to three other approaches. Completeness refers to whether an approach monitors all accesses to a watched memory location by construction. Examples of software-only dynamic checkers include Purify, DIDUCE, Eraser, etc.

6. BUG DETECTION

To detect bugs, programmers and some software debugging tools like DIDUCE [Hangal and Lam 2002] need to use specific monitoring functions. In this section, to demonstrate how to write monitoring functions for iWatcher, we describe some monitoring functions that are used in our experiments to detect various bugs, including buffer overflow, memory leaks, stack smashing, accessing freed memory, and invariant violations.

(1) Detecting Buffer Overflow (BO_check): To detect buffer overflow for both dynamic buffers and static arrays, some paddings are added at the two ends of each buffer. The padding areas are then monitored by iWatcher. The monitoring function simply reports any accesses to these padding areas as bugs. It may miss some bugs if the out-of-bound access does not hit the padding areas. When a dynamic buffer is deallocated, its paddings are also freed and the corresponding monitoring is turned off.

(2) Detecting Memory Leaks (ML_check): Memory leak bugs are tackled by monitoring all accesses to heap objects. Each heap access triggers the monitoring function, which updates the time-stamp associated with the accessed object. The monitoring is turned off when an object is deallocated. Periodically, the time-stamps are checked. The heap objects that have not been accessed for a long time are likely to be memory leaks. Those objects are then ranked based on their time-stamps.

(3) Detecting Accesses to Freed Locations (FREE_check): All unallocated memory space in the heap is monitored using iWatcher. Any access to this space triggers the monitoring function that reports it as a bug. When a memory region is allocated, the monitoring for this memory region is turned off. Of course, the monitor may miss bugs in some cases, such as a dangling pointer points to a reallocated location.

(4) Detecting Various Memory Bugs (COMBO_check): This is used to catch all the above three types of bugs. The monitoring function is combination of the above three functions.

(5) Detecting Stack Smashing (STACK_check): To catch stack smashing bugs that are commonly exploited by viruses to launch security attacks, iWatcher monitors every stack location that stores return addresses. More specifically, after entering a function, iWatcherOn() is called on the return address location, and before the function returns, iWatcherOff() is called to turn off the monitoring to this location. The monitoring function, if triggered, simply reports any access as a bug.

(6) Detecting Invariant Violations (IV_check): To detect an invariant violation, the specific variable needs to be monitored. The monitoring function checks if the variable value satisfies the program-specific invariant.

The first five are general checks. The monitoring can be fully automated using a tool to insert the monitors into any programs. The last check is a program-specific monitoring, requiring specific knowledge about the program semantics.

Monitoring dynamic objects is done by wrapping the memory allocation and deallocation functions to insert iWatcherOn() and iWatcherOff() calls. For monitoring static arrays, the paddings and iWatcherOn/Off() calls are added manually now, although they can be done automatically with compiler support. For STACK_check, we use the compiler support to identify the stack location that stores the return address and add the iWatcherOn/Off() calls. The iWatcherOn/Off() calls for IV_check are inserted manually now, but can be automated using tools such as DAIKON and DIDUCE.

7. EVALUATION METHODOLOGY

7.1 Simulated Architecture

To evaluate iWatcher, we have built an execution-driven simulator that models a workstation with a 4-context SMT processor augmented with TLS support and iWatcher functionality. The experiments are conducted on this default architecture unless specifically mentioned. The parameters of the architecture are shown in Table II. We model the overhead of spawning a monitoring-function microthread as 5 cycles of processor stall visible to the main-program thread. The reaction mode used in all experiments is ReportMode, so that all programs can run to completion.

To isolate the benefits of TLS, we evaluate the same architecture without TLS support. In this case, on a triggering access, the processor first executes the monitoring function, and then proceeds to execute the rest of the program. For the evaluation without TLS support (with or without iWatcher support), the single microthread running is given a 64-entry load-store queue.

To study TLS scalability, we vary the number of contexts from 2 to 8 in the SMT machine, using the same number of shared resources as listed in Table II. The number of contexts limits the maximum number of concurrently running microthreads.

CPU frequency	2.4GHz	ROB size	360
Fetch width	16	I-window size	160
Issue width	8	Int FUs	6
Retire width	12	Mem FUs	4
Ld/st queue entries	32/thread	FP FUs	4
Spawn overhead	5 cycles	Reaction mode	ReportMode
L1 cache	32KB, 4-way, 32B/line, 3 cycles latency		
L2 cache	1MB, 8-way, 32B/line, 10 cycles latency		
VWT	1024 entries, 8-way, 2B/entry		
LargeRegion	64Kbytes		
RWT	4 entries, 32bits for the start and end addresses		
Memory	200 cycles latency		

Table II. Parameters of the simulated architecture. Latencies are given as unloaded round-trips from the processor.

7.2 Valgrind

In our evaluation, we compare the functionality and overhead of iWatcher to Valgrind [Nethercote and Seward 2003], an open-source memory debugger for x86 programs. We choose Valgrind because it does not require modifying the tested applications and is publicly available. Valgrind is a binary-code dynamic checker to detect general memory-related bugs such as memory leaks, memory corruption and buffer overflow. It simulates every single instruction of a program. Because of this, it finds errors not only in the user code but also in all supporting dynamically-linked libraries. Valgrind takes control of a program before it starts. The program is then run on a synthetic x86 CPU, and every memory access is checked. All detected errors are reported.

Valgrind provides an option to enable or disable memory leak detection. We also enhance Valgrind to enable or disable variable uninitialization checks and invalid memory access checks (checks for buffer overflow and invalid accesses to freed memory locations).

In our experiments, we run Valgrind on a real machine with a 2.6 GHz Pentium 4 processor, 32-Kbyte L1 cache, 2-Mbyte L2 cache, and 1-Gbyte main memory. Since iWatcher runs on a simulator, we cannot compare the absolute execution time of iWatcher with that of Valgrind. Instead, we compare their relative execution overheads over runs without monitoring.

7.3 Tested Applications

We have conducted two sets of experiments. The first one uses seven applications with both *injected* and *real* bugs to evaluate the functionality and overheads of iWatcher for software debugging. The second one systematically evaluates the overheads of iWatcher and the effect of architecture resources when monitoring applications without bugs.

The applications used in our first set of experiments contain various bugs, including memory leaks, accesses to freed locations, buffer overflow, stack-smashing attacks, and value invariant violations. These applications are: *bc-1.06* (an arbitrary precision calculator language), *gzip-1.2.4* (GNU zip, a popular compression utility provided by the GNU project), *polymorph-0.4.0* (a tool to convert Windows style file names to something more portable for UNIX systems), *ncompress-4.2.4* (a compression and decompression utility that is compatible with the original UNIX compress utility), *tar-1.13.25* (a tool to create and manipulate tar archives), *cachelib* (a cache management library developed at the University of Illinois), and *gzip* (a SPECINT 2000 application running the Test input data set).

Of these programs, bc-1.06, gzip-1.2.4, polymorph-0.4.0, ncompress-4.2.4, tar-1.13.25 and cachelib already had real bugs (the bugs come with the code and were introduced by the original programmers), while we injected some common bugs into gzip.

Table III shows the details of the bugs and monitoring functions, as described in Section 6. We evaluate the case of single type of bugs: stack-smashing, accessing freed location, buffer overflow (dynamic buffer overflow and static array overflow), memory leak, and value-invariant violations. We also evaluate the case of a combination of bugs (memory leak, accessing a freed location, and dynamic buffer overflow). Table III shows the bug information and monitoring functions.

Application	Type of Monitoring	Bug Class, Location, and Origin	Monitoring Function
gzip-STACK	general	stack smashing: "huft_free()". Injected	STACK_check
gzip-FREE	general	accessing freed location: "huft_free()". Injected	FREE_check
gzip-BO1	general	dynamic buffer overflow: "huft_build()". Injected	BO_check
gzip-ML	general	memory leak: "huft_free()". Injected	ML_check
gzip-COMBO	general	combination of the bugs in gzip-ML, gzip-FREE, and gzip-BO1. Injected	COMBO_check
gzip-BO2	general	static array overflow: "huft_build()". Injected	BO_check
gzip-IV1	program specific	value invariant violation: "huft_build()". Injected	IV_check
gzip-IV2	program specific	value invariant violation: "inflate()". Injected	IV_check
cachelib	program specific	value invariant violation at option.c:line 90. Real	IV_check
bc-1.06	general	dynamic buffer overflow at storage.c:line 176 and util.c:line 557. Real	BO_check
ncompress-4.2.4	general	stack smashing at compress42.c:line 886. Real	STACK_check
gzip-1.2.4	general	static array overflow at gzip.c:line 1009. Real	BO_check
polymorph-0.4.0	general	stack smashing at polymorph.c:line 193 and 200. Real	STACK_check
tar-1.13.25	general	dynamic buffer overflow at pregargs.c:line 92. Real	BO_check

Table III. Bugs and monitoring functions. The applications with name in bold are new relative to [Zhou et al. 2004].

For fair comparison between Valgrind and iWatcher, in Valgrind we enable only the type of checks that are necessary to detect the bug(s) in the corresponding application. For example, for gzip-ML, we enable only the memory leak checks. Similarly, for gzip-FREE, gzip-BO1, bc-1.06 and tar-1.13.25, we enable only the invalid memory access checks. In all our experiments, variable uninitialization checks are always disabled.

Finally, our second set of experiments evaluates iWatcher overheads and the effect of architecture resources (microthread contexts) by monitoring memory accesses in two unmodified SPECINT 2000 applications running the Test input data set. These applications are gzip and parser. We first measure the overhead for iWatcher with a default 4-context SMT processor and without TLS, as we vary the percentage of *dynamic* loads monitored by iWatcher and the length of the monitoring function. Then, we measure the iWatcher overheads with different numbers (namely 2, 4, 6 and 8) of microthread contexts in the SMT processor.

8. EXPERIMENTAL RESULTS

8.1 Overall Results

Table IV compares the effectiveness and the overhead of Valgrind and iWatcher. For each of the buggy applications considered, the table shows whether the schemes detect the bug and, if so, the overhead they add to the program's execution time. Recall from Section 7 that Valgrind's times are measured on a real machine, while iWatcher's are simulated.

Application	Valgrind		iWatcher	
	Bug Detected?	Overhead (%)	Bug Detected?	Overhead (%)
gzip-STACK	No	-	Yes	80.0
gzip-FREE	Yes	1466	Yes	8.7
gzip-BO1	Yes	1514	Yes	10.4
gzip-ML	Yes	936	Yes	37.1
gzip-COMBO	Yes	1650	Yes	42.7
gzip-BO2	No	-	Yes	10.5
gzip-IV1	No	-	Yes	10.5
gzip-IV2	No	-	Yes	9.6
cachelib	No	-	Yes	3.8
bc-1.06	Yes	7367	Yes	178.9
ncompress-4.2.4	No	-	Yes	2.4
gzip-1.2.4	No	-	Yes	168.3
polymorph-0.4.0	No	-	Yes	0.1
tar-1.13.25	Yes	132	Yes	3.8

Table IV. Comparing the effectiveness and overhead of Valgrind and iWatcher.

Consider effectiveness first. Valgrind can detect accessing freed locations, dynamic buffer overflow, memory leak bugs, and the combination of them. iWatcher, instead, detects all the bugs considered. iWatcher's effectiveness is largely due to its flexibility to specialize the monitoring function, and its low-overhead that enables more sophisticated monitoring functionality.

The table also shows that iWatcher has a much lower overhead than Valgrind. For bugs that can be detected by both schemes, iWatcher only adds 4-179% overhead, a factor of 25-169 smaller than Valgrind. For example, in gzip-COMBO, where both iWatcher and Valgrind monitor every access to dynamically-allocated memory, iWatcher only adds 43% overhead, which is 39 times less than Valgrind. iWatcher's low overhead is the result of triggering monitoring functions only when the watched locations are actually accessed, and of using TLS to hide monitoring overheads. The difference in overhead between Valgrind and iWatcher is larger in gzip-FREE, where we are looking for a pointer that de-references a freed-up location. In this case, iWatcher only monitors freed memory buffers, and any triggering access uncovers the bug. As a result, iWatcher's overhead is 169 times smaller than Valgrind's. Similarly, for bc-1.06 and tar-1.13.25, the iWatcher's overheads are 41 and 35 times smaller than Valgrind's, respectively. Finally, our results with Valgrind are consistent with the numbers (12-48 times slowdown) reported in a previous study [Nethercote and Seward 2003].

If we consider all the applications, we see that iWatcher's overhead ranges from 0.1% to 179%. This overhead comes from three effects. The first one is the contention of

monitoring-function microthreads and the main program for processor resources (such as functional units or fetch bandwidth) and cache space. Such contention has a high impact when there are more microthreads that want to execute concurrently than hardware contexts in the SMT processor. In this case, the main-program microthread cannot run all the time. Instead, monitoring-function and main-program microthreads share the hardware contexts on a time-sharing basis.

Application	% Time with Microthreads		# Triggering Accesses per 1M Instr.	# iWatcher-On/Off() per 1M Instr.	Size of iWatcherOn/-Off() (Cycles)	Size of Monitoring Func. (Cycles)	Max Monitored Memory Size at a Time (Bytes)	Total Monitored Memory Size (Bytes)
	> 1	> 4						
gzip-STACK	0.1	0.0	0.2	8988.1	20.6	22.4	40	19558568
gzip-FREE	0.1	0.0	0.4	0.4	1291.3	24.4	246880	246880
gzip-BO1	0.1	0.0	0.4	0.9	210.4	177.0	80	1944
gzip-ML	23.1	16.9	13008.9	0.4	582.6	47.4	6613600	6847616
gzip-COMBO	26.2	15.2	13009.6	0.4	1082.3	45.2	6847616	6847616
gzip-BO2	0.1	0.0	0.2	1.6	59.0	24.8	32	3520
gzip-IV1	0.1	0.0	0.7	0.2	40.5	21.7	4	528
gzip-IV2	0.1	0.0	1.1	0.1	83.0	23.0	4	4
cachelib	0.4	0.0	91.6	0.2	128.9	16.5	40	40
bc-1.06	0.1	0.0	4.8	2594.0	412.7	412.0	3272	4336
ncompress-4.2.4	1.1	1.0	321.7	160.8	162.5	151.5	4	8
gzip-1.2.4	0.9	0.9	371.4	4827.8	280.7	429.0	208	208
polymorph-0.4.0	0.1	0.0	0.7	0.3	204.0	127.6	8	20
tar-1.13.25	0.1	0.0	0.6	15.4	363.4	174.0	96	96

Table V. Characterizing iWatcher execution.

Columns 2 and 3 of Table V show the fraction of time that there is more than one microthread running or more than four microthreads ready to run, respectively. These figures include the main-program microthread. These figures are closely related to the product of the number of triggering accesses per 1 million instructions (Column 4) times the average size of the monitoring function (Column 7). The larger the product, the bigger these figures. Note that having more than four microthreads running does not mean that the main-program microthread starves: the scheduler will attempt to share all the contexts among all microthreads fairly. From the table, we see that three applications use more than 1 microthread for more than 1% of the time. Of those, there are two that have more than 4 microthreads ready to run for a significant fraction of the time. Specifically, this fraction is 15.2% for gzip-COMBO and 16.9% for gzip-ML. Note that these applications have relatively high iWatcher overhead in Table IV.

A second source of overhead is the iWatcherOn/Off() calls. These calls consume processor cycles and, in addition, bring memory lines into L2, possibly polluting the cache. The overhead caused by iWatcherOn/Off() cannot be hidden by TLS. In practice, their effect is small due to the small number of calls, except in gzip-STACK, bc-1.06 and gzip-1.2.4. Indeed, Columns 5 and 6 of Table V show the number of iWatcherOn/Off() calls per 1 million instructions and the average size of an individual call. Except for gzip-STACK, bc-1.06 and gzip-1.2.4, the product of number of calls per 1M instructions times the size per call is tiny compared to the execution cycles taken by 1 million instructions. For these cases, it can be shown that, even if every line brought into L2 by iWatcherOn/Off() calls causes one additional miss, the overall effect on program execution time is very small.

For gzip-STACK, bc-1.06 and gzip-1.2.4, the number of iWatcherOn/Off() calls per 1M instructions is huge (8988, 2594 and 4828, respectively). These calls introduce a large

overhead that cannot be hidden by TLS. Moreover, iWatcherOn/Off() calls partially cripple some conventional compiler optimizations such as register allocation. The result is worse code and additional overhead. Overall, while for most applications the iWatcherOn/Off() calls introduce negligible overhead, for gzip-STACK, bc-1.06 and gzip-1.2.4, they are responsible for most of the 80%, 179% and 168% overheads of iWatcher, respectively.

For the applications with STACK_check (gzip-STACK, ncompress-4.2.4, and polymorph-0.4.0), the dominant overhead is the iWatcherOn/Off() calls. Since iWatcherOn() is called before entering any functions and iWatcherOff() is called before returning from any functions, the frequency of iWatcherOn/Off() calls is correlated to the function call frequency. Therefore, so is the iWatcher overhead for STACK_check. For example, since gzip-STACK has much more frequent iWatcherOn/Off() calls than ncompress-4.2.4 and polymorph-0.4.0, it has much higher overhead.

Finally, there is a third, less important source of overhead in iWatcher, namely the spawning of monitoring-function microthreads. As indicated in Section 7, each spawn takes 5 cycles. Column 4 of Table V shows the number of triggering accesses per million instructions. Each of these accesses spawns a microthread. From the table, we see that this parameter varies a lot across applications. For most of these applications, the triggering frequency is very small. Moreover, for all applications, even if we had a higher spawn overhead, such as 10 or 20 cycles, the total overhead is still insignificant.

Overall, we conclude that the overhead of iWatcher can be high (37-179%) if the application needs to execute more concurrent microthreads than contexts provided by the SMT processor (gzip-ML and gzip-COMBO), or the application calls iWatcherOn/Off() very frequently (gzip-STACK, bc-1.06, and gzip-1.2.4). For the other applications analyzed, the overhead is small, ranging from 0.1% to 10.5%.

Finally, the last three columns of Table V show other parameters of iWatcher execution: average monitoring function size, maximum monitored memory size at a time, and total monitored memory size, respectively. We can see that 6 monitoring functions take less than 25 cycles, and there are 8 applications where monitoring functions take 45-429 cycles. In some cases such as gzip-ML and gzip-COMBO, these relatively expensive monitoring functions occur in applications with frequent triggering accesses. When this happens, the fraction of time with more than 4 microthreads is high, which results in high iWatcher overhead (Table IV).

The last two columns show that in some applications such as gzip-ML and gzip-COMBO, iWatcher needs to monitor many addresses. In this case, the Check Table will typically contain many entries. Note, however, that even in this case, the size of the monitoring function, which includes the Check Table lookup, is still modest. This is because our Check Table lookup algorithm is efficient for most applications evaluated in our experiments.

8.2 Benefits of TLS

As indicated in Section 7, our experiments are performed using ReportMode. In this reaction mode, TLS speeds-up execution by running monitoring-function microthreads in parallel with each other and with the main program. To evaluate the effect of not having TLS, we now repeat the experiments executing both monitoring-function and main-program code sequentially, instead of spawning microthreads to execute them in parallel.

Figure 4 compares the execution overheads of iWatcher and iWatcher without TLS for all the applications. The amount of monitoring overhead that can be hidden by TLS in a program is the product of Columns 4 and 7 in Table V. For programs with substantial

monitoring, TLS reduces the overheads. For example, in gzip-COMBO, the overhead of iWatcher without TLS is 61.4%, while it is only 42.7% with TLS. This is a 30% reduction. As monitoring functions perform more sophisticated tasks such as DIDUCE's invariant inference [Hangal and Lam 2002], the benefits of TLS will become more pronounced.

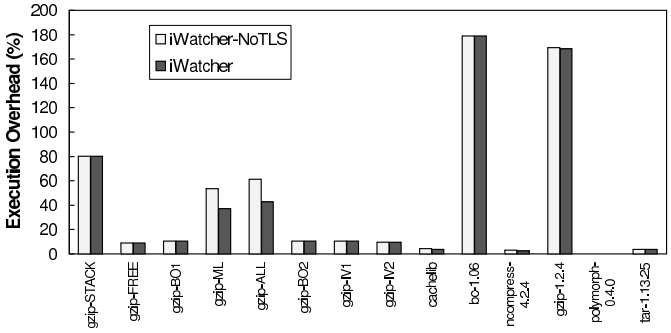


Fig. 4. Comparing iWatcher and iWatcher without TLS.

For programs with little monitoring, the product of Columns 4 and 7 in Table V is small. For these applications, TLS does not provide benefit, because there is not much overhead that can be hidden by TLS.

Overall, we recommend supporting TLS, as it reduces the overhead of iWatcher in some applications. We also note that TLS can be instrumental in efficiently supporting Rollback-Mode (Section 4.5).

8.3 Sensitivity Study

To measure the sensitivity of iWatcher's overhead, we artificially vary the fraction of triggering accesses and the size of the monitoring functions. We perform the experiments on the bug-free gzip and parser applications.

In a first experiment, we trigger a monitoring function every Nth *dynamic* load in the program[1], where N varies from 2 to 10. The function walks an array, reading each value and comparing it to a constant for a total of 40 instructions. The resulting execution overheads for iWatcher (with the default 4-context SMT processor), and iWatcher without TLS are shown in Figure 5 (bar iWatcher-TLS4 and iWatcher-NoTLS, respectively). The figure shows that the overhead of iWatcher with TLS with frequent triggering accesses is tolerable. Specifically, the gzip overhead is 72% for 1 trigger out of 5 dynamic loads, and 194% for 1 trigger out of 2 loads. The parser overheads are a bit higher, namely 182% for 1 trigger out of 5 loads, and 409% for 1 trigger out of 2 loads. If iWatcher does not support TLS, however, the overheads go up: 273% for gzip and 593% for parser for 1 trigger out of 2 loads.

In a second experiment, we vary the size of the monitoring function. We use the same function as before, except that we vary the number of instructions executed from 4 to 800. The function is triggered in 1 out of 10 dynamic loads. The resulting execution overheads

[1]For parser, we skip the program's initialization phase, which lasts about 280 million instructions, because its behavior is not representative of steady state.

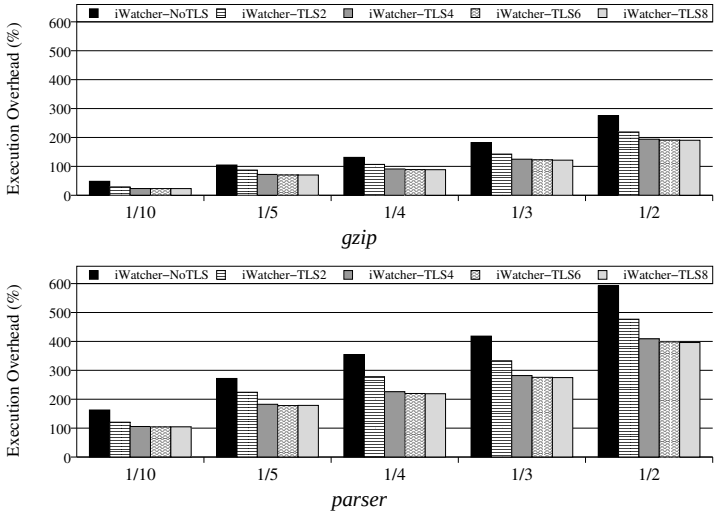


Fig. 5. Varying the fraction of triggering loads.

are shown in Figure 6 (iWatcher-TLS4 and iWatcher-NoTLS). The figure again shows that iWatcher overheads with TLS are modest. For 200-instruction monitoring functions, the overhead is 65% for gzip and 169% for parser. In iWatcher without TLS, the overhead is 173% for gzip and 356% for parser. As we increase the monitoring function size, the absolute benefits of TLS increase, as TLS can hide more monitoring overhead.

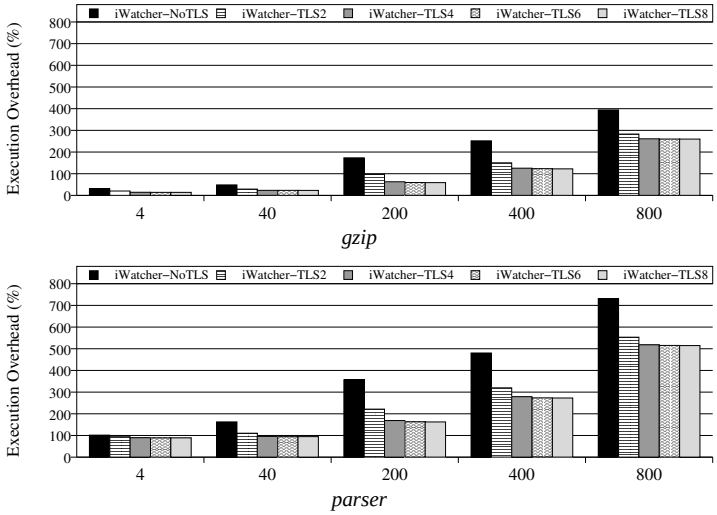


Fig. 6. Varying the size of the monitoring function.

8.4 Scalability Analysis

To evaluate the effect of architectural resources on iWatcher's overhead, we use different numbers (2, 4, 6 and 8) of microthread contexts for the two experiments performed in the sensitivity study (Section 8.3). Note that, in these experiments, the SMT processors with different numbers of contexts have the same number of shared resources.

The first experiment varies the fraction of triggering accesses, as we did in the first experiment of Section 8.3. The second to fifth bars in Figure 5 show that the execution overheads for iWatcher with TLS on a 2/4/6/8-context SMT processor, respectively. The results show that using a 4-context SMT reduces iWatcher's overhead more than using a 2-context SMT. However, using a 6 or 8-context SMT shows little improvement over using a 4-context SMT. More specifically, when using a 4-context SMT instead of a 2-context SMT, the gzip overhead decreases by 17.3% for 1 trigger out of 5 dynamic loads, and by 11.4% for 1 trigger out of 2 loads. For parser, the overhead reduction using a 4-context SMT rather than a 2-context SMT is 18.6% for 1 trigger out of 5 loads, and 14.2% for 1 trigger out of 2 loads. However, the overheads with a 6 or 8-context SMT are almost the same as the overheads with a 4-context SMT in all triggering fractions for both gzip and parser.

The second experiment varies the size of the monitoring function, as the second experiment in Section 8.3. The resulting execution overheads are shown in Figure 6, from the second to the fifth bars. The results again show that a 4-context SMT is enough to reduce the overheads to the minimum for almost all cases. There is no need to use more than 4 contexts for this experiment. For example, in the 200-instruction monitoring function case, the overhead reduction as we get from two to four contexts is 35.5% for gzip and 23.8% for parser. However, the overheads are pretty much the same from 4 contexts to 6 or 8 contexts.

9. DISCUSSION

Even though in this article, we have only demonstrated the use of iWatcher in detecting a few types of bugs, iWatcher can also have many other uses. For example, it can detect many other types of bugs, such as uninitialized reads and data races, and it can be used for performance debugging and interactive debugging.

To detect uninitialized reads, the monitoring function is responsible for checking if the first access to the monitored variable is a read instead of a write. If it is, it indicates an uninitialized read bug.

Detecting data race bugs is more complicated. For example, to detect a shared variable access without grabbing the corresponding lock, we need to monitor each shared variable. The monitoring function checks whether the thread currently holds the lock associated with the accessed variable.

iWatcher can also be used to provide more flexibility in interactive debuggers such as gdb. In particular, the BreakMode and RollbackMode can be exploited by interactive debuggers to support more watchpoints efficiently with low overhead, and to rollback to previous execution points in case of errors. Moreover, programmers can also *interactively* associate different monitoring functions with different memory locations.

iWatcher also provides a general framework for performance debugging. For example, iWatcher can be used for value or address profiling with *very low overhead* and *without instrumenting* every memory access. Value profiling detects the invariant of variables and

records the most likely values of a variable. This information can then be used to help value prediction, and guide automatically code generation for dynamic compilation, code specialization and other compiler optimizations. Address profiling profiles the use of variables to gather their temporal relationship, which can then guide memory optimization (e.g., data placement, and/or data layout) to improve data locality and reduce cache misses.

10. RELATED WORK

Our work builds upon many previous proposals to improve software robustness. Due to space limitations, we briefly describe only related work that has not been described in previous sections.

Many tools have been proposed for dynamic execution monitoring. Well-known examples include Purify [Hastings and Joyce 1992], Intel thread checker [KAI-Intel Corporation], Eraser [Savage et al. 1997], StackGuard [Cowan et al. 1998], DIDUCE [Hangal and Lam 2002], Valgrind [Nethercote and Seward 2003], CCured [Condit et al. 2003; Necula et al. 2002], and many others [Austin et al. 1994; Loginov et al. 2001; Patil and Fischer 1997; 1995]. Most of these tools rely on instrumentation to perform dynamic checks. Consequently, to check all possible accesses to a given location, they typically need to instrument more than necessary. Moreover, most dynamic checkers impose significant run-time overhead. iWatcher innovates with efficient and flexible location-controlled monitoring capability.

Schnarr and Larus have proposed using unused processor cycles to reduce overhead for *code-controlled* monitoring [Schnarr and Larus 1996]. Our work differs from theirs in that iWatcher provides convenient, flexible architectural support to perform *location-controlled* monitoring, and uses TLS to hide monitoring overheads.

Austin *et al.* [Austin et al. 1994] have proposed detecting errors in pointer or array accesses by validating dereferences against the pointer's attributes, such as bounds. This method can prevent a pointer pointing to an array from incorrectly moving out of bounds, but cannot detect errors for general pointers such as links between structures. Moreover, it performs checks sequentially and therefore can add large execution overheads (130-540%). iWatcher can be used to reduce these overheads. Moreover, iWatcher extends dynamic monitoring from out-of-bound checks for array accesses to location-controlled checks for any general memory accesses.

To improve software debugging, several hardware supports have been proposed beyond hardware-assisted watchpoints [Intel Corporation 2001; Johnson 1982; Kane and Heinrich 1992; SPARC International 1992], including the proposal of Oplinger and Lam [Oplinger and Lam 2002], ReEnact [Prvulovic and Torrellas 2003], and the Flight Data Recorder [Xu et al. 2003]. Oplinger and Lam use TLS to execute invariant checks to detect bugs in parallel with the main program. ReEnact uses the state buffering, rollback, and re-execute features of TLS to debug data races. The Flight Data Recorder logs coherence operations into a file that can potentially be used to support replay for debugging. Our work is different, since iWatcher's contribution is to provide efficient and flexible architectural support to monitor memory locations.

Our work is related to previous work on fine-grain access control [Schoinas et al. 1994; Witchel et al. 2002]. For example, Mondrian Memory Protection (MMP) [Witchel et al. 2002] provides access control at word granularity using a "protection look-aside buffer" (PLB) to record protection information. MMP can potentially be used to implement location-

controlled monitoring. However, like hardware-assisted watchpoints, it needs to raise an exception and, therefore can add significant overhead.

Our work is also related to some of the classic work on capability-based architectures [Fabry 1974; Levy 1984], protection-enhanced architectures [Koldinger et al. 1992], hardware support for security [Frantzen and Shuey 2001; Lie et al. 2000; Suh et al. 2003; Xu et al. 2002], TLS [Cintra et al. 2000; Sohi et al. 1995; Steffan et al. 2000; Tsai et al. 1999], and hardware support for instruction-level profiling [Dean et al. 1997].

11. CONCLUSIONS AND FUTURE WORK

This article has presented *iWatcher*, novel architectural scheme for minimal-overhead location-controlled monitoring. iWatcher detects all accesses to a watched memory location, including those by aliased pointer dereferences. To reduce overhead, iWatcher optionally leverages Thread-Level Speculation (TLS). We have evaluated iWatcher on applications with various bugs. iWatcher detects all bugs evaluated in our experiments with only a 0.1-179% execution overhead. In contrast, a well-known open-source bug detector called Valgrind induces orders of magnitude more overhead, and can only detect a subset of the bugs. Moreover, even with 20% of the dynamic loads monitored in a program, iWatcher only adds 72-182% overhead. Finally, TLS is effective at reducing overheads for programs with substantial monitoring, and a 4-context SMT is enough to achieve the best performance in our experiments.

We are in the process of extending this work in several ways. First, we plan to compare iWatcher to other dynamic checkers beyond Valgrind. Moreover, we will evaluate iWatcher for multi-threaded programs, which often exhibit hard-to-debug bugs such as data races and deadlocks. In addition, we plan to test more applications, especially large server programs with real bugs. In order to do this, we are in the process of upgrading our simulation infrastructure. We are also seeking help from the operating system to handle cases of VWT overflow. Finally, in this study, we have inserted iWatcherOn/Off() calls manually for some applications. It is more convenient to use a compiler or an instrumentation tool to insert them. Moreover, it is interesting to combine iWatcher with an invariant-inference tool such as DIDUCE [Hangal and Lam 2002], which can specify watched locations and their associated monitoring functions. We are addressing these issues.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback, and the University of Illinois PROBE, I-ACOMA, and Opera groups for useful discussions.

REFERENCES

AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*. 290–301.

BUCK, B. AND HOLLINGSWORTH, J. K. 2000. An API for runtime code patching. *The International Journal of High Performance Computing Applications 14*, 4 (Winter), 317–329.

CHOI, J.-D., LEE, K., LOGINOV, A., O'CALLAHAN, R., SARKAR, V., AND SRIDHARAN, M. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*. 258–269.

CINTRA, M., MARTINEZ, J., AND TORRELLAS, J. 2000. Architectural support for scalable speculative parallelization in shared-memory systems. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. 13–24.

CONDIT, J., HARREN, M., MCPEAK, S., NECULA, G. C., AND WEIMER, W. 2003. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*. 232–244.

COWAN, C. ET AL. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*. 63–78.

DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. Z. 1997. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 292–302.

ENGLER, D. AND ASHCRAFT, K. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. 237–252.

ERNST, M. D., CZEISLER, A., GRISWOLD, W. G., AND NOTKIN, D. 2000. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. 449–458.

FABRY, R. S. 1974. Capability-based addressing. *Commun. ACM 17*, 7 (July), 403–412.

FRANTZEN, M. AND SHUEY, M. 2001. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*. 55–66.

HALLEM, S., CHELF, B., XIE, Y., AND ENGLER, D. 2002. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*. 69–82.

HANGAL, S. AND LAM, M. S. 2002. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. 291–301.

HASTINGS, R. AND JOYCE, B. 1992. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1999 USENIX Winter Technical Conference*. 125–136.

INTEL CORPORATION. 2001. The IA-32 Intel architecture software developer's manual, volume 2: Instruction set reference.

JOHNSON, M. S. 1982. Some requirements for architectural support of software debugging. In *Proceedings of the 1st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 140–148.

KAI-INTEL CORPORATION. Intel thread checker. URL: http://developer.intel.com/software/products/threading/tcwin.

KANE, G. AND HEINRICH, J. 1992. *MIPS RISC architecture*. Prentice-Hall.

KOLDINGER, E. J., CHASE, J. S., AND EGGERS, S. J. 1992. Architectural support for single address space operating systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 175–186.

LARUS, J. AND SCHNARR, E. 1995. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*. 291–300.

LEVY, H. M. 1984. *Capability-based computer systems*. Digital Press, Bedford, MA.

LIE, D. ET AL. 2000. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 168–177.

LOGINOV, A., YONG, S. H., HORWITZ, S., AND REPS, T. W. 2001. Debugging via run-time type checking. In *the 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*. 217–232.

MARCUS, E. AND STERN, H. 2000. *Blueprints for high availability*. John Willey and Sons, New York.

MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D. R., AND DILL, D. L. 2002. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (SOSP)*. 75–88.

NATIONAL INSTITUTE OF STANDARDS AND TECHNLOGY (NIST), DEPARTMENT OF COMMERCE. 2002. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002-10.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. CCured: Type-safe retrofitting of legacy code. In *The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 128–139.

NETHERCOTE, N. AND SEWARD, J. 2003. Valgrind: A program supervision framework. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV)*.

ONE, A. 1996. Smashing the stack for fun and profit. Phrack Magazine.

OPLINGER, J. AND LAM, M. S. 2002. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 184–196.

PATIL, H. AND FISCHER, C. 1997. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software - Practice and Experience 27,* 1 (Jan.), 87–110.

PATIL, H. AND FISCHER, C. N. 1995. Efficient run-time monitoring using shadow processing. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*. 119–132.

PRVULOVIC, M. AND TORRELLAS, J. 2003. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*. 110–121.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15,* 4 (Nov.), 391–411.

SCHNARR, E. AND LARUS, J. R. 1996. Instruction scheduling and executable editing. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 288–297.

SCHOINAS, I. ET AL. 1994. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 297–306.

SOHI, G., BREACH, S., AND VIJAYAKUMAR, T. 1995. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*. 414–425.

SPARC INTERNATIONAL. 1992. *The SPARC architecture manual: Version 8*. Prentice-Hall.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*. 196–205.

STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. 1–12.

STERN, U. AND DILL, D. L. 1995. Automatic verification of the SCI cache coherence protocol. In *Conference on Correct Hardware Design and Verification Methods (CHARME)*. 21–34.

SUH, G., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. 2003. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*. 160–171.

TSAI, J.-Y., HUANG, J., AMLO, C., LILJA, D. J., AND YEW, P.-C. 1999. The superthreaded processor architecture. *IEEE Trans. Comput. 48,* 9 (Sept.), 881–902.

WAHBE, R., LUCCO, S., AND GRAHAM, S. L. 1993. Practical data breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*. 1–12.

WITCHEL, E., CATES, J., AND ASANOVIĆ, K. 2002. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 304–316.

XU, J., KALBARCZYK, Z., PATEL, S., AND IYER, R. K. 2002. Architecture support for defending against buffer overflow attacks. 2nd Workshop on Evaluating and Architecting System Dependability (EASY).

XU, M., BODÍK, R., AND HILL, M. D. 2003. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30st Annual International Symposium on Computer Architecture (ISCA)*. 122–133.

ZHOU, P., QIN, F., LIU, W., ZHOU, Y., AND TORRELLAS, J. 2004. iWatcher: Efficient Architecture Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. 224–237.