iWatcher: Simple and General Architectural Support for Software Debugging

Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou and Josep Torrellas
PROBE Group, Department of Computer Science, University of Illinois at Urbana-Champaign
http://carmen.cs.uiuc.edu/probe

1. Motivation

Recent impressive advances in microprocessor performance have failed to deliver significant gains in ease of software debugging. This is a major shortcoming of the state of the art, given that software bugs have major implications on computer system reliability and programmer time. Specifically, software bugs account for as much as 40% of computer system failures [10] and cost the U.S. economy $59.5 billion annually, or 0.6% of the GDP [11]!

Code debugging is currently largely done using software techniques. One approach is to perform checks statically [2, 4, 5, 18]. 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, especially for C/C++ programs. As a result, many bugs often remain in the software even after aggressive static checking.

A second approach is to monitor execution dynamically. Examples of dynamic monitors include Purify [7], Valgrind [15], the Intel thread checker [9], DIDUCE [6], Eraser [14], CCured [3, 12], and other tools [1]. The strength of this approach is that the analysis is based on actual execution paths and accurate values of variables and aliasing information. However, most dynamic checkers suffer from two limitations. First, they are often computationally expensive. Some dynamic checkers slow down a program by 6-30 times [6, 14], 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: unintended 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 right after line A.

While there is interest in providing micro-architectural support for software debugging, the current state of the art is very primitive. Such state of the art is largely limited to watchpoints [8, 16] and event or branch trace buffers [8, 17]. Watchpoints, such as those supported by Intel's x86 [8] and Sun's SPARC [16], trigger an exception every time that a programmer-specified memory location is accessed. While they are a good starting point, they have several limitations. First, they do not provide *low-overhead* checks that can be on *all the time* in a production run. This is because they trigger the exception mechanism, which has very high overhead and disrupts the execution of the application. Second, current architectures only support a handful of watchpoints (four in Intel x86). Besides watchpoints, branch or event trace buffers [8, 17] can also potentially be used for debugging purposes, such as providing more program state information in a crash. However, they do not provide highly-processed information that could truly boost debuggability.

Recently, there have been some research proposals for micro-architectural support for software debugging [13, 19]. These proposals are promising. However, they provide relatively limited bug coverage and are also relatively expensive. Specifically, ReEnact uses the state buffering, rollback and re-execution features of Thread Level Speculation (TLS) to debug data races [13]. The Flight Data Recorder extends the cache coherence protocol to record the ordering of shared-memory accesses for postmortem analysis [19].

In contrast, our paper presents new architectural support that is able to *detect a large variety* of software bugs with only *modest hardware changes* to current microprocessor implementations.

2. Our Contribution

We propose the *Intelligent Watcher (iWatcher)* — architectural support to automatically monitor dynamic execution for a *wide variety* of software bugs with minimal overhead and *modest hardware requirements*. With iWatcher, programmer-specified monitoring functions are associated to monitored memory locations or objects. When any such "watched" location is accessed, the monitoring function is automatically triggered in hardware with very low overhead, without generating an exception. iWatcher is very flexible because monitoring functions are not hard-wired into the architecture, but are provided by programmers or external automated software tools.

The main characteristics that distinguish 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 is very effective for bugs such as buffer overflow, memory leaks, uninitialized reads, or accesses to freed locations.

Feature	Software-Only Dynamic Checkers	Hardware Watchpoints	iWatcher
Checking all monitored locations and only those?	Very hard to ensure due to aliasing & incomplete info	Yes	Yes
Language independent, cross-module, and cross-developer?	Typically no	Yes	Yes
Overheads	Variable Typically high	High: checks are interrupt driven	Low
Flexibility	High	Currently low: only a few watchpoints	High

Table 1. Comparing different approaches. Examples of software-only dynamic checkers include assertions and automated checkers such as DIDUCE [6].

— It has low overhead because it only monitors *true* accesses to the watched memory locations and uses minimal-overhead hardware-supported triggering of monitoring functions.

— It is flexible in that it supports any checks coded by the programmer. It is also 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.

Table 1 compares iWatcher to other approaches.

3. iWatcher Overview

3.1. iWatcher Interface

Programs can turn on and off the monitoring of a memory object with the *iWatcherOn* and *iWatcherOff* system calls, respectively. These calls can be inserted into programs by the compiler, instrumentation tools, or the programmer. The interfaces of iWatcherOn and iWatcherOff are:

```
iWatcherOn(MemAddr, Length, WatchFlag, ReactMode,
           MonitorFunc, Param1, Param2, ... ParamN)
iWatcherOff(MemAddr, Length, WatchFlag, MonitorFunc)
```

When iWatcherOn is called, it associates monitoring function *MonitorFunc()* to the memory object that begins at *MemAddr* and has size *Length*. MonitorFunc() takes arguments *Param1*, *Param2*, ... *ParamN*. The *WatchFlag* specifies what types of accesses (read, write, or both) to this memory object will trigger MonitorFunc(). *ReactMode* determines the action that will be taken at run time if the monitoring function detects a bug (Section 3.2.4).

After iWatcherOff is called, MonitorFunc() is no longer invoked when the object is accessed with WatchFlag.

In general, a program can associate multiple monitoring functions to the same object. In this case, upon an access to the watched object, all its monitoring functions are executed. Monitoring functions can be individually removed.

3.2. iWatcher Implementation

iWatcher is implemented using a combination of hardware and software. Logically, it has four parts. (1) To detect accesses to monitored locations ("triggering" accesses), we use two structures: WatchFlags in the tags of both L1 and L2 cache lines to detect accesses to small monitored memory regions, and a small *Range Watch Table (RWT)* to detect accesses to large monitored memory regions; (2) The processor provides a special *Main_check_function* register to store the common entry point for all monitoring functions; (3) Software manages the associations between watched locations and monitoring functions; and (4) TLS can be *optionally* used to hide monitoring overhead by executing a monitoring function in parallel with the rest of the program, and to add ease of use by supporting program rollback if a bug is found.

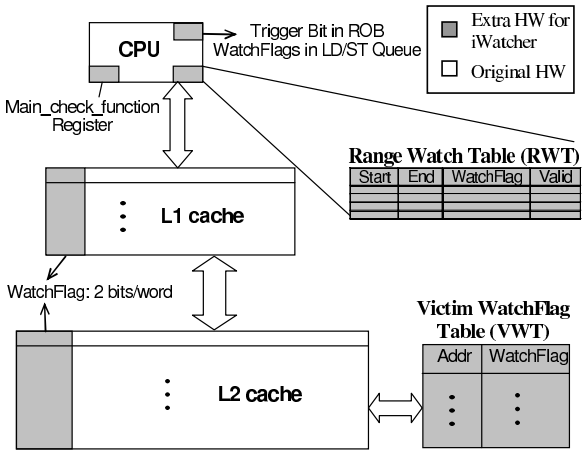

Figure 1. iWatcher hardware architecture.

Figure 1 overviews the iWatcher hardware. 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 are triggering. The Main_check_function register holds the address of the *Main_check_function()*, which is the common entry point to all program-specified monitoring functions. iWatcher also has a *Victim WatchFlag Table (VWT)*, which stores the WatchFlags for lines of small watched regions that have at some point been displaced from L2.

The RWT is a set of registers to detect accesses to large (multi-page) monitored memory regions. Each RWT entry stores the virtual start and end addresses of the region plus WatchFlag bits. The RWT is used to prevent lines from large monitored regions from overflowing the L2 cache and the VWT. The WatchFlags of these lines do not need to be set in the caches.

A software table called *Check Table* stores detailed monitoring information for each watched memory location. The information stored includes MemAddr, Length, WatchFlags, ReactMode, MonitorFunc, and Parameters. An iWatcherOn/Off() call adds/removes an entry to/from the Check Table.

When a triggering access occurs, the hardware saves the architectural registers and the program counter, then sets the program counter to the address in the Main_check_function register. The Main_check_function() searches the Check Table and calls the monitoring function(s) associated with the accessed location.

Finally, the processor core is enhanced with a *Trigger* bit for each reorder buffer (ROB) entry, and 2 WatchFlag bits for each load-store queue entry (Section 3.2.2).

3.2.1. Watching a Range of Addresses

When a program calls iWatcherOn() for a memory region as large as or larger than *LargeRegion*, iWatcher allocates a RWT entry. If the RWT already has an entry for this region, iWatcherOn() updates the entry's WatchFlags. If, instead, the region is smaller than *LargeRegion*, iWatcher loads the watched memory lines into L2 (but not into L1 to avoid polluting it). 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 of the L2 line to be the logical OR of the old and new values. In all cases, iWatcherOn() also adds the monitoring function to the Check Table.

3.2.2. Detecting Triggering Accesses

A triggering access can be identified at two points: early in the pipeline when the RWT is checked in parallel with the TLB lookup, or later in the pipeline when the memory system is accessed and the WatchFlags in the cache are checked.

A load can access the memory system before reaching the head of the ROB. As a load reads the data from the cache into the load queue, it also reads the WatchFlag bits into the load queue entry (unless they have already been read from a RWT entry). If the WatchFlag bits indicate that the load is a triggering one, the Trigger bit in the load's ROB entry is set. When any instruction reaches the head of the ROB and its Trigger bit is set, the hardware triggers the corresponding monitoring function.

Typically, 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 iWatcher, this means that the processor may have to wait a long time to know whether the store 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, we change the micro-architecture so that, as soon as a store address is resolved early in the pipeline, a prefetch is issued to the memory system. Such prefetch reads the data into the cache, brings the WatchFlag bits into the store queue entry, and may set the Trigger bit in the ROB entry. With this support, the processor is much less likely to have to wait when the store reaches the head of the ROB. More details are in [20].

3.2.3. 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 TLS mechanisms. Specifically, when the triggering access is retired, a new microthread could be automatically spawned to speculatively execute the rest of the program in parallel with the microthread that executes the monitoring function non-speculatively. Data dependencies between monitoring function and rest of the program would be tracked by TLS, and any violation would result in the rollback of the speculative microthread.

With or without TLS, a monitoring function is triggered in hardware. iWatcher can skip the OS because monitoring functions are not related to any resource management in the system and, in addition, are *not* executed in privileged mode. Moreover, they are in the same address space as the monitored program. Therefore, a "bad" program cannot use iWatcher to mess up other programs.

3.2.4. Different Reaction Modes

If the monitoring function detects a bug, different actions take place depending on *ReactMode* (Section 3.1). *ReactMode* can be *ReportMode*, *BreakMode*, and *RollbackMode* (the last one requires checkpointing and rollback support).

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 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 the most recent 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.

4. Key Results

We simulate a desktop with a 4-context SMT and iWatcher hardware. We compare its functionality and overhead to Valgrind [15], a well-known software-only dynamic checker. Valgrind is an open-source memory debugger for x86 programs. We use 7 applications that contain various real or induced bugs, including buffer overflow, memory leak, accessing freed locations, stack smashing, and invariant violations. Relative to [20], we add 5 new applications with real bugs.

Table 2 compares Valgrind and iWatcher in ReportMode *without TLS*. 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.

iWatcher detects all the bugs considered, whereas Valgrind can detect only a fraction of the bugs. For bugs that can be detected by both iWatcher and Valgrind, iWatcher adds only 4-179% overhead, a factor of 17-165 times smaller than Valgrind. More details and many other experiments are in [20].

5. Conclusions

iWatcher is a significant advance in the state of the art of micro-architecture for software debugging. First, it targets a wide variety of memory-related bugs, such as memory leaks, accesses to freed locations, buffer overflow, stack smashing, and memory corruption. These bugs are hard to debug because their effect is typically observed much later than when they occur; with iWatcher, they are detected immediately. Secondly, iWatcher offers high flexibility

Application	Valgrind		iWatcher Without TLS	
	Bug Detected?	Overhead (%)	Bug Detected?	Overhead (%)
gzip-STACK	No	-	Yes	80.0
gzip-FREE	Yes	1466	Yes	8.9
gzip-BO1	Yes	1514	Yes	10.4
gzip-ML	Yes	936	Yes	53.5
gzip-COMBO	Yes	1650	Yes	61.5
gzip-BO2	No	-	Yes	10.6
gzip-IV1	No	-	Yes	10.5
gzip-IV2	No	-	Yes	9.7
cachelib	No	-	Yes	4.4
bc-1.06	Yes	7367	Yes	178.8
ncompress-4.2.4	No	-	Yes	3.1
gzip-1.2.4	No	-	Yes	169.4
polymorph-0.4.0	No	-	Yes	0.1
tar-1.13.25	Yes	132	Yes	3.6

Table 2. Comparing Valgrind and iWatcher. The iWatcher overheads with TLS are presented in our ISCA-2004 paper [20]. The applications in bold are new relative to our ISCA-2004 paper [20].

to the programmer, who can write very sophisticated monitoring functions. Thirdly, iWatcher is highly usable, as it is language independent, cross-module and cross-developer. Fourthly, it is very effective — it is able to detect many real bugs, and with program slowdowns that are a factor of 17-165 times smaller than a popular dynamic monitoring tool. Finally, iWatcher provides a framework for general-purpose debugging, including performance debugging through value and address profiling.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, June 1994.

[2] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, June 2002.

[3] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *PLDI*, June 2003.

[4] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, October 2003.

[5] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, June 2002.

[6] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, May 2002.

[7] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference*, January 1992.

[8] Intel Corporation. The IA-32 Intel architecture software developer's manual, volume 3: System programming guild, 2004.

[9] KAI-Intel Corporation. Intel thread checker. URL: http://developer.intel.com/software/products/threading/tcwin.

[10] E. Marcus and H. Stern. Blueprints for high availability. John Willey and Sons, 2000.

[11] National Institute of Standards and Technlogy (NIST), Department of Commerce. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002-10, June 2002.

[12] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, January 2002.

[13] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, June 2003.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, November 1997.

[15] J. Seward. Valgrind. URL: http://valgrind.kde.org/.

[16] SPARC International. *The SPARC architecture manual: Version 8*. Prentice-Hall, 1992.

[17] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, July-August 2002.

[18] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Conference on Correct Hardware Design and Verification Methods*, October 1995.

[19] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, June 2003.

[20] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.