iWatcher: Simple, General Architectural Support for Software Debugging

iWatcher lets programmers associate specified functions to "watched" memory locations or objects. Access to any such location automatically triggers the monitoring function in the hardware. Relative to other approaches, iWatcher detects many real bugs at a fraction of the execution-time overhead.

Pin Zhou
Feng Qin
Wei Liu
Yuanyuan Zhou
Josep Torrellas
University of Illinois at Urbana-Champaign

Recent impressive advances in microprocessor performance have failed to deliver significant gains in ease of software debugging. Given that software bugs account for as much as 40 percent of computer system failures[1] and cost the US $59.5 billion annually—0.6 percent of the gross national product[2]—this lapse represents a major shortcoming in state-of-the-art microprocessors.

Current debugging techniques consist largely of static tools and dynamic monitors, both of which have significant limitations. Static tools[3-6] analyze programs statically. Because they suffer from aliasing problems and other compile-time limitations, especially for C/C++ programs, many bugs often remain in the software even after aggressive static checking. Dynamic monitors—including Purify,[7] Valgrind (http://valgrind.kde.org), Intel's Thread Checker (http://developer.intel.com/software/products/threading/tcwin), DIDUCE,[8] Eraser,[9] and CCured,[10,11]—are more effective in that they base the analysis on actual execution paths and accurate values of variables and aliasing information. However, most dynamic checkers are often computationally expensive, some slowing down the program by a factor of 6 to 30.[8,9] With these slowdowns, some timing-sensitive bugs may never occur. Another drawback is that most dynamic checkers rely on compilers or preprocessing tools to insert instrumentation. These tools are limited by imperfect variable disambiguation, which may cause them to miss accesses to monitored locations. As a result, they often catch bugs much later than their actual occurrence, which makes it hard to find the bugs' root cause. The sidebar "A Limitation of Software-Only Dynamic Checkers" shows the consequences of imperfect variable disambiguation.

The state of the art in microarchitectural support for software debugging is limited largely to watchpoints, as in the IA-32 architecture and to event or branch trace buffers, as in the IA-32 architecture and Pentium 4.[12] Watchpoints, such as those that Intel's x86 and Sun's SPARC support, trigger an exception whenever the program accesses a programmer-specified memory location. Although a good beginning, watchpoint-based tools tend to suffer from high overhead, since they trigger the exception mechanism and disrupt application execution.

Consequently, they are unusable for production runs, where checks must be on all the time. Moreover, current architectures support only a handful of watchpoints. The Intel x86, for example, supports only four.

Branch or event-trace buffers are also potentially useful for certain types of debugging, such as providing more program state information in a crash. They do not provide, however, highly processed information that could truly boost a microprocessor's debugging capability.

Some recent research that proposes microarchitectural support for software debugging, such as ReEnact[13] or the Flight Data Recorder,[14] is promising, but these techniques are expensive and offer limited bug coverage.

To address the shortcomings in these existing approaches, we propose Intelligent Watcher (iWatcher), a combination of hardware and software support that can detect a large variety of software bugs with only modest hardware changes to current processor implementations. iWatcher lets programmers associate automated debugging tools monitoring functions to memory locations or objects. Access to any such watched location automatically triggers the monitoring function in hardware with very low overhead and without generating an exception.

As Table 1 shows, iWatcher compares favorably with existing debugging approaches.

- It monitors *all* accesses to the watched memory locations. Consequently, it catches hard-to-find bugs such as updates through stray pointers and stack-smashing attacks that viruses commonly exploit. It is very effective for bugs such as buffer overflow, memory leaks, uninitialized reads, or accesses to freed locations.
- It has low overhead because it monitors only *true* accesses to the watched memory locations, and because the hardware automatically triggers the monitoring functions with minimal overhead.
- It is flexible in that it supports any checks that the programmer codes. It is also language independent and cross-module and cross-developer.
- It can *optionally* leverage thread-level speculation (TLS) to hide monitoring overhead and support program rollback. With TLS, a monitoring function executes in parallel with the rest of the program, and can roll back the program if it finds a bug.

A limitation of software-only dynamic checkers

Software-only dynamic checkers are limited by imperfect variable disambiguation, as the following C code illustrates:

```
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];
...
```

Although **x** is corrupted in line A, the bug is not detected until the invariant check at line B. It is hard to perform perfect pointer disambiguation, so a dynamic checker is not likely to know that it must insert an invariant check right after line A, and the bug will remain undetected. In contrast, iWatcher detects the bug in line A.

Table 1. How iWatcher compares to popular debugging approaches.

Feature	Software-only dynamic monitoring*	Hardware watchpoints**	iWatcher
Checks all monitored locations and only those?	Very hard to ensure because of aliasing and incomplete information	Yes	Yes
Language independent, cross-module and cross-developer?	Typically no	Yes	Yes
Execution-time overhead?	Varies, but typically high	High; checks are interrupt driven	Low
Flexible?	Yes	No; only a few watchpoints	Yes

* Includes assertions and automatic checkers such as DIDUCE.

** Such as those supported by Intel's x86 and Sun's Sparc.

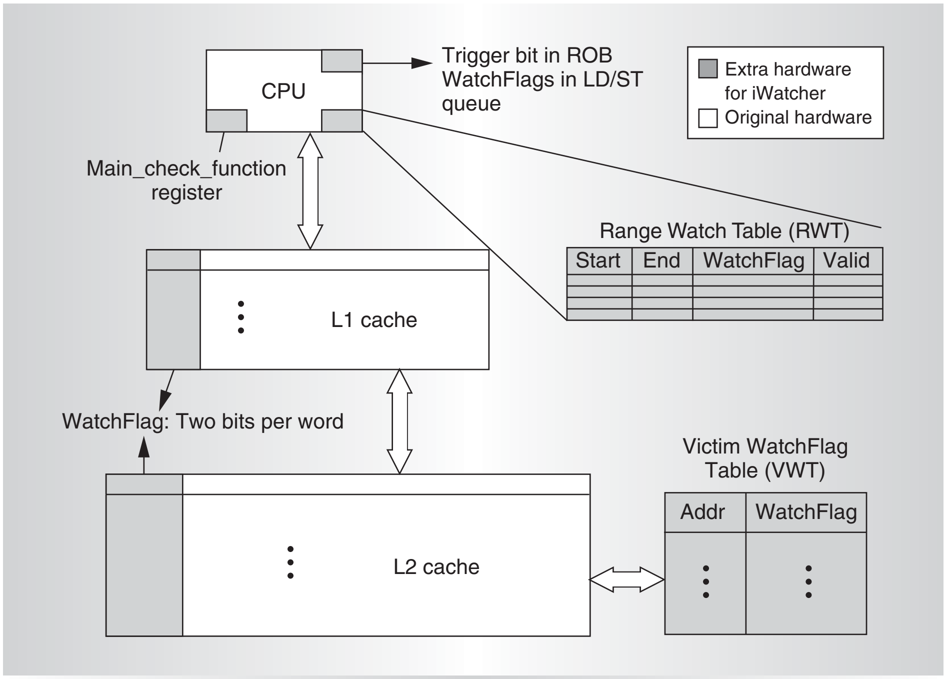
CPU
Trigger bit in ROB
WatchFlags in LD/ST queue
Extra hardware for iWatcher
Original hardware
Main_check_function register
Range Watch Table (RWT)
Start
End
WatchFlag
Valid
L1 cache
WatchFlag: Two bits per word
Victim WatchFlag Table (VWT)
Addr
WatchFlag
L2 cache

Figure 1. iWatcher hardware architecture. White areas indicate original hardware; gray areas, the hardware that iWatcher implementation requires.

iWatcher interface

Programs can turn the monitoring of a memory object on and off with **`iWatcherOn()`** and **`iWatcherOff()`**, system calls that the compiler, instrumentation tools, or programmer can insert into programs. The interfaces of **`iWatcherOn()`** and **`iWatcherOff()`** are

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

When called, **`iWatcherOn()`** associates monitoring function **`MonitorFunc()`** to the memory object that begins at **`MemAddr`** and has size **`Length`**. **`MonitorFunc()`** takes arguments **`Param1, Param2, ... ParamN`**. **`WatchFlag`** specifies what types of accesses to this memory object (read, write, or both) will trigger **`MonitorFunc()`**.

`ReactMode` determines the action the program will take at runtime if the monitoring function detects a bug. (We describe this in more detail later.)

After the program calls **`iWatcherOff()`**, accessing the object with **`WatchFlag`** no longer invokes **`MonitorFunc()`**.

In general, a program can associate multiple monitoring functions to the same object, and when that object is accessed, all the associated monitoring functions execute. Programmers can remove individual monitoring functions as needed.

iWatcher implementation

iWatcher uses a combination of hardware and software. Logically, it has four components. The first *detects accesses to monitored locations* (triggering accesses). For this, iWatcher uses two structures, as Figure 1 shows: **`WatchFlag`**s in both level 1 (L1) and level 2 (L2) cache lines to detect accesses to small monitored memory regions, and a small RangeWatch Table (RWT) to detect accesses to large monitored memory regions. The second of iWatcher component *keeps a common entry point for all monitoring functions*. For this, the processor provides the **`Main_check_function`** register. As the third component, the software *manages the associations between watched locations and monitoring functions*. Finally, programmers can opt to use TLS, *which hides monitoring overhead* by executing a monitoring function in parallel with the rest of the program, and *adds ease of use* by supporting program rollback if the monitoring function finds a bug.

As Figure 1 shows, each cache line has two **`WatchFlag`** bits per word: one for read monitoring and one for write monitoring. If the read (write)-monitoring bit is set for a word, all loads (stores) to this word trigger a monitoring function. 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 **`WatchFlag`**s for lines of small watched regions that have at some point been displaced from the L2 cache.

The RWT is a set of registers that detect accesses to large (multipage) monitored memory regions. Each RWT entry stores the region's start and end virtual addresses plus the **`WatchFlag`** bits. The RWT prevents lines in large monitored regions from overflowing the L2 cache and the VWT. These lines are not loaded into the caches in an **`iWatcherOn() call.`** The **`WatchFlag`**s of these lines do not need to be set in the caches.

A software table, *Check Table,* stores detailed monitoring information for each watched memory location, including **`MemAddr`**, **`Length`**, **`WatchFlag`**s, **`ReactMode`**, **`MonitorFunc`**, and **`Parameters`**. An **`iWatcherOn/Off()`** call adds/removes an entry to/from Check Table.

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

Finally, using iWatcher requires enhancing the processor core with a **`Trigger`** bit for each reorder buffer (ROB) entry, and two **`WatchFlag`** bits for each load-store queue entry.

Watching a range of addresses

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

Detecting triggering accesses

The hardware can identify a triggering access at two points: early in the pipeline when it checks the RWT in parallel with the translation look-aside buffer (TLB) lookup, or later in the pipeline when it accesses the memory system and checks the **`WatchFlag`**s in the cache.

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 the hardware has already read them from an RWT entry). If the **`WatchFlag`** bits indicate that the load is a triggering one, the hardware sets the **`Trigger`** bit in the load's ROB entry. When any instruction reaches the head of the ROB and its **`Trigger`** bit is set, the hardware triggers the corresponding monitoring function.

Typically, a processor does not send a store to the memory system until that store reaches the head of the ROB. At that point, the hardware immediately retires the store, but the store can still cause a cache miss. Consequently, the processor might have to wait a long time to know if the store is a triggering access. During that time, the processor cannot retire any subsequent instruction because the processor may have to trigger a monitoring function. To reduce this delay, iWatcher changes the microarchitecture so that as soon as the processor resolves a store address early in the pipeline, the hardware issues a prefetch to the memory system. The 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 after the store reaches the head of the ROB. We provide more details about this elsewhere.[15]

Executing monitoring functions

When the hardware retires a triggering load or store, it automatically saves the architectural registers and the program counter, and redirects execution to the address in the **`Main_check_function`** register. After the monitoring function completes, execution resumes from the saved program counter.

As an *optimization*, programmers can leverage TLS mechanisms. Specifically, when the hardware retires the triggering access, it could automatically spawn a new microthread to speculatively execute the rest of the program in parallel with the microthread that executes the monitoring function nonspeculatively. TLS would track data dependencies between the monitoring function and the rest of the program, and any violation would result in the rollback of the speculative microthread to right after the triggering access.

With or without TLS, iWatcher triggers monitoring functions in hardware. iWatcher can skip the operating system because monitoring functions do not depend on any resource management in the system; moreover, they are *not* executed in privileged mode and, in addition, are in the same address space as the monitored program. Therefore, a "bad" program cannot use iWatcher to mess up other programs.

Reaction modes

If the monitoring function detects a bug, different actions take place depending on **`ReactMode`**, one of the pieces of information in **`iWatcherOn()`**, as we described earlier. **`ReactMode`** can be **`ReportMode`**, **`BreakMode`**, and **`RollbackMode`**. In **`ReportMode`**, the monitoring function reports the check's outcome and lets the program continue. This mode, which is for profiling and error reporting, does not interfere with program execution.

In **`BreakMode`**, if the monitoring function detects an error, the program pauses at the state right after the triggering access, and control passes to an exception handler. Users can attach an interactive debugger 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 requires checkpointing and rollback support. This mode supports transaction-based programming or the replay of a code section to analyze a bug.

Key results

To analyze iWatcher's features, we simulated a system with a four-context simultaneous multithreading processor and iWatcher hardware. We used this setup to compare iWatcher's functionality and overhead to those of Valgrind, a well-known software-only dynamic checker. Valgrind is an open-source memory debugger for x86 programs. We used seven applications that contained various real or injected bugs, including buffer overflow, memory leaks, accesses to freed locations, stack smashing, and invariant violations. We added five new applications with real bugs relative to earlier work.[15]

Table 2 compares Valgrind and iWatcher in **`ReportMode`** *without TLS*. For each of the buggy applications considered, the table shows if the schemes detect the bug and, if so, the overhead they add to the program's execution time. iWatcher detects all the bugs considered, while Valgrind detects only a fraction. For bugs that both iWatcher and Valgrind detected, iWatcher adds only 4 to 179 percent overhead; the program slowdown is 17 to 165 times smaller than that with Valgrind. More details and experiments are presented elsewhere.[15]

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 multithreaded programs, which often exhibit hard-to-debug bugs such as data races and deadlocks. Finally, we plan to test more applications, especially large server programs, with real bugs, and are in the process of upgrading our simulation infrastructure accordingly. MICRO

References

1. E. Marcus and H. Stern, *Blueprints for High Availability*, John Wiley & Sons, 2000.
2. "Software Errors Cost U.S. Economy $59.5 Billion Annually," *NIST News Release 2002-10*, Nat'l Inst. of Standards and Technology (NIST), US Dept. of Commerce, June 2002.
3. J.-D. Choi et al., "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," *Proc. Conf. Pro-*

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

Application**	Valgrind		iWatcher without TLS	
	Bug detected?	Overhead (%)	Bug detected?	Overhead (%)
gzip-STACK	No	NA	Yes	80.0
gzip-FREE	Yes	1,466	Yes	8.9
gzip-BO1	Yes	1,514	Yes	10.4
gzip-ML	Yes	936	Yes	53.5
gzip-COMBO	Yes	1,650	Yes	61.5
gzip-BO2	No	NA	Yes	10.6
gzip-IV1	No	NA	Yes	10.5
gzip-IV2	No	NA	Yes	9.7
cachelib	No	NA	Yes	4.4
bc-1.06	Yes	7,367	Yes	178.8
ncompress-4.2.4	No	NA	Yes	3.1
gzip-1.2.4	No	NA	Yes	169.4
polymorph-0.4.0	No	NA	Yes	0.1
tar-1.13.25	Yes	132	Yes	3.6

* We analyze iWatcher's overhead *with* TLS in P. Zou et al., "iWatcher: Efficient Architectural Support for Software Debugging," *Proc. 31st Ann. Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 224-235.

** The applications listed in boldface are new relative to work reported in the paper listed above.

gramming Language Design and Implementation (PLDI 02), ACM Press, 2002, pp. 258-269.

4. D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," *Proc. Symp. Operating Systems Principles* (SOSP 03), ACM Press, 2003, pp. 237-252.
5. S. Hallem et al., "A System and Language for Building System-Specific, Static Analyses," *Proc. Conf. Programming Language Design and Implementation* (PLDI 02), ACM Press, 2002, pp. 69-82.
6. U. Stern and D.L. Dill, "Automatic Verification of the SCI Cache Coherence Protocol," *Proc. Conf. Correct Hardware Design and Verification Methods—IFIP WG10.5 Advanced Research Working Conf., Lecture Notes in Computer Science* 987, Springer-Verlag, 1995, pp. 21-34.
7. R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," *Proc. Usenix Winter Tech. Conf.*, Usenix Assoc., 1992, pp. 125-138.
8. S. Hangal and M.S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proc. Int'l Conf. Software Eng.* (ICSE 02), IEEE CS Press, 2002, pp. 291-301.
9. S. Savage et al., "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems*, vol. 15, no. 4, Nov. 1997, pp. 319-411.

10 J. Condit et al., "CCured in the Real World," *Proc. Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 232-244.

11. G.C. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," *Proc. Symp. Principles of Programming Languages* (POPL 02), ACM Press, 2002, pp. 128-139.
12. B. Sprunt, "Pentium 4 Performance-Monitoring Features," *IEEE Micro*, vol. 22, no. 4, July-Aug. 2002, pp. 72-82.
13. M. Prvulovic and J. Torrellas, "ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes," *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 110-121.
14. M. Xu, R. Bodik, and M.D. Hill, "A 'Flight Data Recorder' for Enabling Full System Multiprocessor Deterministic Replay," *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 122-133.
15. P. Zhou et al., "iWatcher: Efficient Architectural Support for Software Debugging," *Proc. 31st Ann. Int'l Symp. Computer Archi-*

tecture (ISCA 04), IEEE CS Press, 2004, pp. 224-235.

Pin Zhou is a PhD student in the Department of Computer Science, University of Illinois at Urbana-Champaign. Her research interests include architectural support for software debugging, energy management for storage system, and memory management. Zhou has an MS in computer science from Tsinghua University, China.

Feng Qin is a PhD student in the Department of Computer Science, University of Illinois at Urbana-Champaign. His research interests include software debugging with system support. Qin has an ME in software engineering from the Institute of Software, Chinese Academy of Sciences.

Wei Liu is a research scientist for computer science at the University of Illinois at Urbana-Champaign. His research interests include architecture and compiler support for thread-level speculation and software debugging. Liu has a PhD in computer science from Tsinghua University, China.

Yuanyuan Zhou is an assistant professor at University of Illinois at Urbana-Champaign. Her research interests include database storage, architecture and OS support for software debugging, power management, and memory management. She has an MA and a PhD, both in computer science, from Princeton University.

Josep Torrellas is a professor of computer science and Willett Faculty Scholar at the University of Illinois at Urbana-Champaign. He is also vice-chair of the IEEE Technical Committee on Computer Architecture (TCCA). His main research interests are in multiprocessor computer architecture, thread-level speculation, low-power design, reliability and debuggability. Torrellas has a PhD in electrical engineering from Stanford University. He is an IEEE Fellow and a member of the ACM.

Direct questions and comments about this article to Pin Zhou, Univ. of Illinois, 4219 Siebel Center for Computer Science, 201 N. Goodwin Ave., Urbana, IL 61801; pinzhou@cs.uiuc.edu.