AVIO: DETECTING ATOMICITY VIOLATIONS VIA ACCESS-INTERLEAVING INVARIANTS

THIS ARTICLE PROPOSES AN INNOVATIVE CONCURRENT-PROGRAM INVARIANT THAT CAPTURES PROGRAMMERS' ATOMICITY ASSUMPTIONS. IT DESCRIBES A TOOL WITH TWO IMPLEMENTATIONS, ONE IN SOFTWARE AND THE OTHER USING HARDWARE SUPPORT, THAT CAN AUTOMATICALLY EXTRACT SUCH INVARIANTS AND DETECT ATOMICITY VIOLATION BUGS.

Shan Lu
Joseph Tucek
Feng Qin
Yuanyuan Zhou
University of Illinois at Urbana-Champaign

•••••• Of all software bugs, concurrency bugs in multithreaded and multiprocess programs are among the most difficult to test for and diagnose. Their notorious nondeterminism frustrates both in-house testing and postmortem diagnosis. In the real world, most server and high-end critical software is multithreaded or multiprocess. Concurrency bugs in such applications have caused some of the most serious computer-related accidents in history, including a blackout leaving tens of millions of people without electricity (http://www.securityfocus.com/news/8016).

The recent multicore hardware trend has worsened this problem. Relying on more concurrent programs to take advantage of hardware resources, multicore techniques inevitably create more concurrency bugs. Addressing this issue is one of computer architecture's grand research challenges.

Most previous concurrency-bug detection work focused on one subclass of concurrency bugs—data races—which occur when two conflicting accesses, from different threads to the same memory location, execute without proper synchronization. Existing solutions for data races include lockset bug detection tools,[1] happens-before bug detection tools,[2] and hybrids of these two tools.[3] Some researchers have suggested using hardware support[4] to alleviate these solutions' bug detection overhead. Although these approaches can effectively detect some race bugs in concurrent programs, they have several limitations.

First, what programmers often want is atomicity, not freedom from data races. The absence of data race does not necessarily indicate correct synchronization. Figure 1 is a real bug example from the Mozilla application suite. Thread 1 sets the shared script handler *gCurrentScript* and wants to read the value to continue processing the script at a later step. However, thread 2 may nullify *gCurrentScript* midway. This example has no data race because one common lock protects every access to *gCurrentScript*. Nevertheless, it contains a severe concurrency bug: an atomicity violation that, once triggered, will lead to a crash.

Many programmers are accustomed to thinking sequentially and therefore usually assume the atomicity of code segments. In Figure 1, they might expect the two parts of script processing from thread 1 to be atomic and free from interference by other accesses to *gCurrentScript*. Formally, atomicity—also called serializability—is a property of several concurrently executed actions, when their data manipulation effect is equivalent to that of their serial execution. If the implementation fails to satisfy an atomicity expectation, some executions will have unserializable interleavings that violate programmers' assumptions and manifest as concurrency bugs. Using locks or transactions is one way to ensure atomicity, but, as we demonstrated, the absence of data races doesn't guarantee proper atomicity.

```
                Thread 1                                   Thread 2
1.1  void LoadScript (nsSpt* aspt) {
1.2    Lock (l);
1.3    gCurrentScript = aspt;
1.4    LaunchLoad (aspt);
1.5    UnLock (l);
1.6  }                                          2.1  Lock (l);
                                                2.2  gCurrentScript = NULL;
1.7  void OnLoadComplete ( ) {                  2.3  UnLock (l);
     /* call back function of LaunchLoad */
1.8     Lock (l);
1.9     gCurrentScript→compile();
1.10    UnLock (l);
1.11 }
```

Figure 1. A real bug in the Mozilla application suite, file nsXULDocument.cpp, slightly simplified for illustration. When thread 2 violates the atomicity of thread 1's accesses to *gCurrentScript*, the program crashes.

The second limitation is that although data races are not a problem for future transaction-based concurrent programs, atomicity violations still are. There is a recent, emerging trend toward a transactional memory programming model using hardware or software support.[5,6] Programmers using this model needn't worry about data races because the underlying transaction system can automatically detect and resolve memory access conflicts. But atomicity violations will still occur in transaction-based programs when programmers do not group operations that should be atomic into the same transaction. For example, if a programmer divides the code segment shown in Figure 1 into two separate transactions, the atomicity violation bug remains.

Third, a data race is not always a bug. Many important synchronization mechanisms actually use data races in their implementation. Examples include barriers, flag synchronization, and producer-consumer queues. Furthermore, programmers might choose to allow data races on unimportant variables to achieve better performance. The inability to differentiate these benign races has caused many false positives in previous tools.

Finally, most previous techniques rely on specific synchronization semantics. The happens-before and lockset algorithms both need to know what synchronization primitives the target program uses. Ignorance of non-lock-based synchronization, such as barrier, thread create/join, and many user-defined synchronizations (for example, flag-based synchronization), has caused many false positives in previous work.

Although researchers have known about this issue for years, the problem of atomicity violations has few good solutions. Most state-of-the-art techniques[7] rely on programmers' annotations to recognize atomic regions, requiring extensive human effort and risking errors based on programmers' unconscious atomicity assumptions. Recently, the serializability violation detector (SVD) approach has tried to use data/control dependency to automatically infer atomic regions.[8] However, SVD's dependency heuristics cover only a limited subset of atomic regions—those that start with the read of a shared variable and expand on the basis of write-after-read or control dependencies. For example, SVD doesn't cover the bug-related atomic region in Figure 1 and therefore cannot automatically detect that bug. In addition, these atomicity violation bug detection tools are all implemented entirely in software and greatly reduce program execution speed (up to 65× with SVD).[8]

Proposed solutions

This article offers three proposals to address the limitations of previous work. The first concerns the access interleaving (AI) invariant, a unique invariant for concurrent programs. It exists in a code region that programmers expect to be atomic (regardless of whether the implementation actually guarantees atomicity). This invariant accurately reflects the programmer's intention concerning shared-variable accesses: Are conflicting accesses from other threads welcomed, irrelevant (don't care), or forbidden? Violation of such an invariant, or the programmer's assumption, results in a concurrency bug. AI invariants provide a new way to approach concurrent program correctness.

The second proposal is an innovative, comprehensive, invariant-based approach called AVIO, a scheme to detect general atomicity violations. On the basis of AI invariants, AVIO automatically identifies from correct runs (training runs) the important code segments that programmers assume to be atomic and then uses those invariants to perform online detection of atomicity violation bugs. AVIO requires no programmer annotation or knowledge of synchronization. During its operation, AVIO also uses two originally notorious properties of concurrent programs to make the invariant collection even easier than traditional sequential program invariant collection.

The third proposal implements the AVIO idea in both hardware (AVIO-H) and software (AVIO-S), and evaluates them using real-world bugs in server applications. Experiments show that AVIO can detect more cases of atomicity violations, including those not addressed by data-race detection, than previous approaches. Because AVIO can differentiate benign races and doesn't rely on prior knowledge about synchronization primitives used in the programs, it introduces far fewer false positives (an average of four) than previous techniques (an average of 51). AVIO-H imposes negligible overhead (0.4 percent to 0.5 percent) with the help of a simple hardware extension on a cache coherence protocol and is fit for production-run monitoring. AVIO-S is slower (25× on average) but more accurate and more suitable for in-house bug detection and diagnosis.

In summary, AVIO can provide effective help in two scenarios:

- *Postmortem analysis.* Programmers can use AVIO to diagnose the root cause of software failures. Given a failure to track down, a programmer can use AVIO to collect and compare AI invariants during correct runs and bug-infested runs, and thereby identify possible atomicity violation root causes.
- *On-the-fly detection.* Programmers can use AVIO to automatically extract AI invariants during in-house testing. These AI invariants serve during production runs to detect atomicity violation bugs.

The AVIO concept

In our discussion, we refer to the thread whose atomicity is interrupted as the local thread and its accesses as local accesses. (This does *not,* however, mean a local variable.) We refer to the thread with the interleaving access as the remote thread and its accesses as remote accesses. A serializable interleaving is an interleaving between local and remote accesses that is equivalent to a serial noninterleaving execution.

Access-interleaving invariance

Atomicity violation bugs are no different in essence than other types of bugs: They result from a mismatch between programmer intention and implementation. Specifically, programmers who are more comfortable with sequential thinking assume that a sequence of shared-variable accesses is atomic (serializable), free from interference by unserializable accesses. If the implementation doesn't enforce such atomicity assumptions correctly, bugs emerge.

Programmers' atomicity intentions take different forms. The most common and fundamental can be represented by a type of invariant that we call an AI invariant. Such an invariant is held by an instruction if—for the execution to be correct—the access pair, composed of itself and its preceding local access to the same location, should never be unserializably interleaved. We denote these as I-instruction (invariant instruction) and P-instruction (preceding access instruction).

Programmers don't assume all code regions to be atomic, nor does an AI invariant hold for every shared-variable access instruction. For example, a flag-based synchronization, shown in Figure 2a, has no AI invariant because unserialized interleavings are actually welcomed in this case. In contrast, in the classic bank-account example, shown in Figure 2b, programmers assume the read and modification of an account to be atomic—that is, the AI invariant should be held for the execution to be correct. Whether or not an AI invariant holds indicates programmers' different assumptions about correct synchronization.

Synchronization primitives, such as locks, barriers, flags, or transactions, are means for enforcing AI invariants. If programmers don't correctly use synchronization primitives, resulting in a failure to enforce AI invariants, atomicity assumptions could be violated and the program would misbehave.

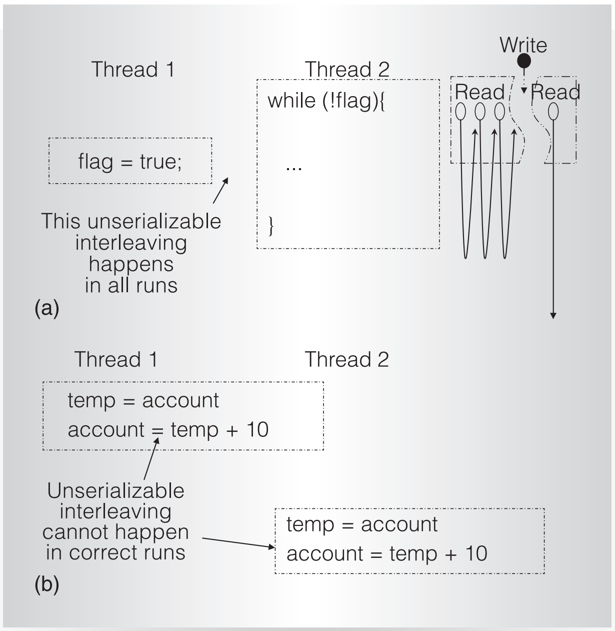

Figure 2. Spin-flag example, without an AI invariant (a). This code segment is designed for synchronization. (The read access series should be unserializably interleaved.) Bank-account deposit example, with an AI invariant (b). The code segment is assumed to be serializable.

Serializability analysis

Not all interleavings are unserializable, and serializable interleavings do not lead to atomicity violation. In this section, we analyze what interleavings are serializable and what are not.

There are eight ways in which one remote access can interleave two consecutive local accesses to the same shared variable. Table 1 describes each case and shows that four cases (cases 2, 3, 5, and 6) are unserializable. In a longer version of this article, we listed bug examples for each unserializable case.[9]

Extending this analysis to consider multiple remote accesses, we encounter more-general unserializable conditions:

- Case 2: Two local reads are interleaved by at least one remote write, so they might have different views.
- Case 3: A local read after a local write is interleaved by at least one remote write. Because of this remote write, the read would fail to get the local result it expects.
- Case 5: A local write after a local write is interleaved by a remote access sequence that starts with a read. This makes the local intermediate result visible to a remote thread.
- Case 6: A local write after a local read is interleaved by at least one remote write, making the previous reading result stale.

We use these four case conditions in the rest of the article as a guide to AVIO bug detection.

Automatic extraction of AI invariants

Because AI invariants reflect programmers' atomicity intentions, we can view the detection of atomicity violation bugs as detecting violations to AI invariants. So, knowing which code regions are intended to be atomic, how do we obtain AI invariants? Obviously, we cannot expect programmers to provide such invariants, because many atomicity violations occur in code segments where programmers are not aware of their assumptions.

The best way to automatically discover a programmer's intention is to study the program's behavior in correct execution: If a code segment is always serializable in correct runs (bug-free runs), then programmers probably intend this code segment to be serializable. In other words, we can statistically "learn" a program's AI invariants through training. We just need to

Table 1. Eight cases of access interleavings. All accesses are to the same shared variables. In the read/write entries, subscript r denotes remote interleaving access; superscripts i and p denote one access and its preceding access from the same thread.

Case no.	Interleaving	Description	Serializability	Equivalent serial accesses	Problems (for unserializable cases)
0	Read^p Read$_r$ Read^i	Two reads interleaved by a read	Serializable	Read^p Read^i Read$_r$	N/A
1	Write^p Read$_r$ Read^i	Read after write interleaved by a read	Serializable	Write^p Read^i Read$_r$	N/A
2	Read^p Write$_r$ Read^i	Two reads interleaved by a write	Unserializable	N/A	The interleaving write makes the two reads have different views of the same memory location.
3	Write^p Write$_r$ Read^i	Read after write interleaved by a write	Unserializable	N/A	The local read does not get the local result it expects.
4	Read^p Read$_r$ Write$_i$	Write after read interleaved by a read	Serializable	Read$_r$ Read^p Write^i	N/A
5	Write^p Read$_r$ Write^i	Two writes interleaved by a read	Unserializable	N/A	Intermediate result that is assumed to be invisible to other threads is read by a remote access.
6	Read^p Write$_r$ Write^i	Write after read interleaved by a write	Unserializable	N/A	The local write relies on a value from the preceding local read that is then overwritten by the remote write.
7	Write^p Write$_r$ Write^i	Two writes interleaved by a write	Serializable	Write$_r$ Write^p Write^i	N/A

observe which shared accesses (such as the one in Figure 2a) allow unserializable interleavings and which never have unserializable interleavings during a set of correct (training) runs.

Like all previous invariant-based techniques,[10–12] AVIO can leverage the software testing infrastructure: Those test suites will help collect sufficient training samples, and the testing oracles (including various methods beyond crashes or hangs) can serve to differentiate correct from incorrect runs.

Training in AVIO can also take advantage of two unique and notorious properties of concurrency bugs. First, concurrency bugs are hard to trigger because their manifestation needs not only bug-exposing inputs but also special interleaving. This helps AVIO to easily acquire predominantly correct training runs. Second, concurrent execution's nondeterministic nature makes AVIO training—especially training in postmortem analysis—very easy. We simply run the program with one input (the bug-triggering input during postmortem analysis) many times to achieve sufficient and different AI training results. This is a big advantage over traditional invariant-based tools.

AVIO detection and extraction algorithms

Now let's look at the detailed AVIO algorithms for detecting AI-invariant violations and extracting AI invariants.

Detection algorithm. Suppose that we already have a set of AI invariants, that is, a list of I-instructions. Then an AI-invariant violation is an unserializable interleaving between an I-instruction and its preceding local access instruction (P-instruction) to the same shared variable. On the basis of our serializability analysis for detecting any such unserializable interleavings, the detection process can simply follow the binary decision diagram in Figure 3a, which summarizes all four unserializable interleaving cases.

Extraction algorithm. Leveraging the AI-invariant violation detection process can easily implement AI-invariant extraction. Specifically, the extraction process is a series of correct runs with AVIO detection enabled. If the process encounters an unserializable interleaving at an instruction i, AVIO detects this violation and removes i from the I-instruction candidate set (where i is a general instruction and I-instruction is an invariant-related instruction). This process repeats many times until we get a stable I-instruction set.

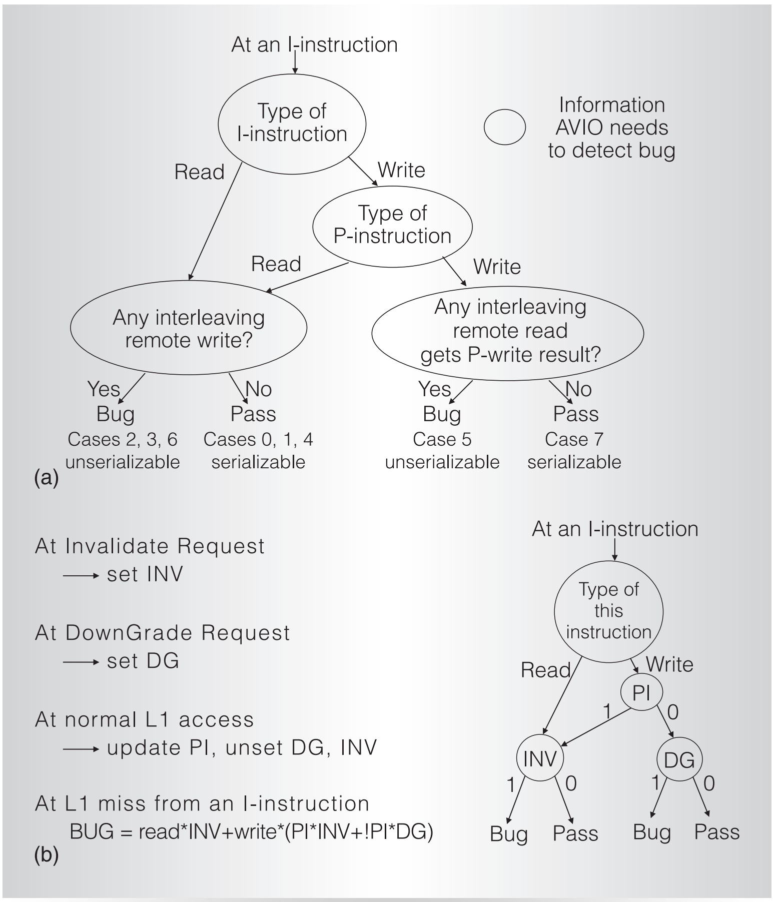

Figure 3. AVIO bug detection procedure. The general protocol (a) can be better understood in reference to Table 1. AVIO-H uses the hardware version protocol (b), showing AVIO-H state maintenance (left) and the AVIO-H detection protocol (right).

AVIO implementation

To study the trade-offs between hardware and software approaches, we implemented our AVIO idea and algorithms in a software-only approach, AVIO-S, and a hardware-assisted approach, AVIO-H.

Hardware (AVIO-H)

The hardware implementation uses very simple hardware extensions to the existing cache coherence protocol and achieves negligible overhead. (We assume the use of a chip multiprocessor, or CMP, machine with an invalidation-based cache coherence protocol.) The right-hand side of Figure 3b shows the AVIO-H protocol.

AVIO-H appends each L1 cache line with two new access information bits. These new bits, together with the existing invalidate (INV) bit, provide enough information for the AVIO detection algorithm to execute. The preceding access instruction (PI) bit provides the type of P-instruction information. The AVIO-H architecture sets it to 1 at each local read to a cache line and unsets it at each local write. The downgrade (DG) bit provides information to determine whether the previous local write's result has been read by a remote thread. In existing invalidation-based cache coherence protocols, such an action is associated with a downgrade request from the reader to the recent writer. Therefore, AVIO-H simply needs to set the DG bit upon a downgrade request and unset the bit after each local access. The INV bit already exists in current cache coherence hardware. AVIO-H can leverage it to determine the existence of an interleaving remote write after the previous local memory access, because in the existing invalidation-based cache coherence protocol, interleaving remote writes will invalidate all other L1 caches' copies.

Apart from the extra cache line bits, AVIO-H adds special instruction encodings for I-instructions (reads and writes) to indicate when an I-instruction is accessing memory.

AVIO-H requires very little extra hardware, yet it greatly alleviates the bug detection overhead problem. (The L2 cache access latency can easily hide the invariant violation checking.) Our longer version of this article provides more details on the design issues.[9]

Software (AVIO-S)

We also implemented the AVIO techniques just in software. AVIO-S uses binary instrumentation to collect access information at every global memory access and maintain the information in access-table data structures. On the basis of this information, atomicity violation detection operates easily following the general detection procedure (Figure 3a). Upon detecting an atomicity violation, AVIO-S, like AVIO-H, either stops the program and raises an exception or logs all the debugging information and continues executing.

Trade-offs between AVIO-H and AVIO-S

Each scheme has advantages and disadvantages. AVIO-S is cheaper because it doesn't require any hardware extensions. Moreover, AVIO-S is more accurate because it uses very fine granularity (a byte or a word) in detection and therefore suffers much less from the false-sharing problem. (AVIO-H uses cache-line granularity in detection and therefore would confuse accesses to different variables placed in the same cache line.) AVIO-S is not affected by cache displacement, context switches, or other hardware-related issues.

On the other hand, AVIO-S incurs much higher overhead and runtime perturbation because of the high demands of instrumentation-based information collection and violation detection. Such a performance disadvantage will prevent AVIO-S from production-run usage and might affect its bug detection capability due to the larger execution perturbation.

Experimental results

We implemented AVIO-S on the basis of the Pin binary instrumentation tool (http://rogue.colorado.edu/pin/) and evaluated it on a real machine. We implemented and evaluated AVIO-H in a full-system, cycle-accurate simulator that models a four-core CMP, in-order x86 machine (http://www.ece.cmu.edu/~simflex/).

Functional results

During our functionality experiments, we used different inputs for training and bug detection. All bug detection results were obtained by training with fewer than 100 client requests (for server programs) or fewer than 100 program runs.

Bug detection capability. We evaluated the bug detection capability using six real atomicity violation bugs from two large real-world server applications (Apache and MySQL) and Mozilla. As Table 2 shows, AVIO detects more tested real bugs than the three state-of-the-art alternatives: the lockset algorithm (we use the Valgrind implementation [www.valgrind.org], which we refer to as Val-Lockset); the happens-before algorithm; and SVD. (We conducted an indirect comparison with happens-before and SVD on the basis of our understanding of these two algorithms.)

MySQL bug 3 violates atomicity among accesses to multiple different global variables and therefore cannot be detected by any of the tools we evaluated. Along with this bug, the lockset algorithm and the happens-before algorithm miss bugs that are data-race free but still violating atomicity, as explained in Figure 1. SVD fails to automatically detect several bugs that are caused by a violation of write-after-write or read-after-write access atomicity; it does not consider these two types of atomic regions. In contrast, AVIO's bug detection capability is more comprehensive because, unlike race detectors, it doesn't rely on synchronization primitives. Unlike SVD, AVIO can detect atomicity violations with write-read

Table 2. Bug detection results for various techniques, against bug-infested real applications.

	Bug detected				
Application	**AVIO-H**	**AVIO-S**	**Val-Lockset**	**Happens-before**	**SVD**
Apache 1	Yes	Yes	Yes	Yes	Yes
Apache 2	Yes	Yes	Yes	Yes	No*
MySQL 1	Yes	Yes	Yes	Yes	No*
MySQL 2	Yes	Yes	No	No	No
MySQL 3	No	No	No	No	No*
Mozilla extract**	Yes	Yes	No	No	No*

* The SVD paper does not discuss these four bugs. We based this evaluation on our understanding of the SVD algorithm. The SVD work evaluated the other two bugs, and our results agree with those of the authors.
** Since our instrumentation and simulation tools do not support Mozilla's graphic user interface, we used an extracted version of the real bug in Mozilla, which was written on the basis of the original nsXULDocument.cpp file.

and write-write dependencies, on the basis of our serializability analysis.

False positives. Experiments with server applications and Splash-2 applications (http://www-flash.stanford.edu/apps/SPLASH/) show that AVIO introduces far fewer static and dynamic false positives than the lockset algorithm, as shown in Table 3. This is because the lockset algorithm incorrectly reports as bugs those shared accesses that are correctly synchronized using non-lock-based methods. Furthermore, the lockset algorithm cannot differentiate benign races from real bugs. Previous happens-before algorithms share similar limitations, but AVIO addresses them on the basis of AI invariants. AVIO's few false positives are caused by insufficient training (AVIO-H has some extra false positives caused by the false-sharing problem). The numerous false positives in the previous algorithms require much programmer effort to sift through manually.

Overhead results

Our experiments with four Splash-2 applications (fft, fmm, lu, and radix) show that AVIO has low detection overhead. With hardware support, AVIO-H imposes only 0.4 percent to 0.5 percent overhead and is therefore suitable for production-run monitoring. Our software implementation, AVIO-S, imposes an average slowdown of 25×, though it still outperforms many previous software approaches. Valgrind lockset imposes an average slowdown of 694× for the same set of applications (due partly to Valgrind's code emulation mechanism). SVD can impose a 65× slowdown on server applications.[8] AVIO-S would be a good choice for offline bug detection.

Training sensitivity

To demonstrate AVIO's nonstringent requirement on training runs, we used different inputs for detection and training in our experiments. We also conducted sensitivity studies on the number of training runs for both server applications and Splash-2 benchmarks. The results show that we need no more than 100 server requests or five training runs to obtain reasonably accurate AI invariants, resulting in just six static false positives for AVIO-S on MySQL 2 and no false positives for the Splash-2 benchmarks. Results are similar for MySQL 1 (four false positives). Of course, like other invariant-based approaches[11,12] and general dynamic bug detectors, AVIO can generate invariants only from exercised code.

AVIO's innovative AI-invariant-based approach provides one of the first practical, comprehensive, low-overhead solutions—and the first hardware support—for atomicity violation detection. The multicore hardware trend and the emerging transactional memory programming model will make the atomicity violation bug increasingly troublesome and critical. To help AVIO better address the atomicity violation problem, we are working first to

Table 3. False-positive rates for server applications and bug-free Splash-2 benchmarks.

	Dynamic false positives			Static false positives		
Benchmark	AVIO-H	AVIO-S	Val-Lockset	AVIO-H	AVIO-S	Val-Lockset
Apache 1	6	5	6	3	2	6
Apache 2	1	1	23	1	1	20
MySQL 1	4	4	107	4	4	79
MySQL 2	17	6	338	11	6	101
Average	7	4	118.5	4.75	3.25	51.5
fft	1	0	4,098	1	0	6
fmm	4	0	389	4	0	12
lu	0	0	65,026	0	0	5
radix	0	0	35,740	0	0	10
Average	1.25	0	26,313	1.25	0	8.25

We determined each server application's false positives by manually examining the application's Bugzilla database. Dynamic false positives are dynamic instances of false positives reported during execution; static false positives are static code segments incorrectly reported as bugs. Because we obtained the Mozilla extract ourselves, its false-positive number is not objective, and we do not report it here.

solve the challenging open question of detecting multiple-variable atomicity violation bugs, and second to provide good training runs to improve AVIO's bug detection capability and accuracy. MICRO

Acknowledgments

This research is supported by the National Science Foundation career award CNS-0347854, NSF grant number CCR-0325603, Department of Energy grant number DE-FG02-05ER25688, and an Intel gift grant.

References

1. S. Savage et al., ''Eraser: A Dynamic Data Race Detector for Multithreaded Programs,'' *ACM Trans. Computer Systems* (TOCS 97), vol. 15, no. 4, Nov. 1997, pp. 391-411.
2. R.H.B. Netzer and B.P. Miller, ''Improving the Accuracy of Data Race Detection,'' *Proc. 3rd ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPOPP 91), ACM Press, 1991, pp. 133-144.
3. R. O'Callahan and J.-D. Choi, ''Hybrid Dynamic Data Race Detection,'' *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPOPP 03), ACM Press, 2003, pp. 167-178.
4. 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.
5. T. Harris and K. Fraser, ''Language Support for Lightweight Transactions,'' *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 03), ACM Press, 2003, pp. 388-402.
6. M. Herlihy, J. Eliot, and B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. 20th Ann. Int'l Symp. Computer Architecture* (ISCA 93), IEEE CS Press, 1993, pp. 289-300.
7. C. Flanagan and S.N. Freund, ''Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs,'' *Proc. Principles of Programming Languages Symp* (POPL 04), ACM Press, 2004, pp. 256-267.
8. M. Xu, R. Bodík, and M.D. Hill, ''A Serializability Violation Detector for Shared-Memory Server Programs,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 05), ACM Press, 2005, pp. 1-14.
9. S. Lu et al., ''AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 37-48.

10. M. Ernst et al., ''Quickly Detecting Relevant Program Invariants,'' *Proc. 22nd Int'l Conf. Software Engineering* (ICSE 2000), ACM Press, 2000, pp. 449-458.
11. S. Hangal and M.S. Lam, ''Tracking Down Software Bugs Using Automatic Anomaly Detection,'' *Proc. 24th Int'l Conf. Software Engineering* (ICSE 02), ACM Press, 2002, pp. 291-301.
12. P. Zhou et al., ''AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants,'' *Proc. 37th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 04), IEEE CS Press, 2004, pp. 269-280.

Shan Lu is a PhD candidate in the Department of Computer Science of the University of Illinois at Urbana-Champaign. Her research interests include system and architecture support for software reliability. Lu has a BS in computer science from the University of Science and Technology of China. She is a student member of the ACM.

Joseph Tucek is a PhD candidate in the Department of Computer Science of the University of Illinois at Urbana-Champaign. His research interests include systems and architecture support for reliability and debugging. Tucek has BS degrees in computer science and computer engineering from Washington University in St. Louis. He is a student member of both the ACM and Usenix.

Feng Qin is an assistant professor at Ohio State University. His research interests include operating systems, software dependability, security, and computer architecture. Qin has a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the ACM.

Yuanyuan Zhou is an associate professor in the Department of Computer Science of the University of Illinois at Urbana-Champaign. Her research interests include software reliability, operating systems, computer architecture, and storage systems. Zhou obtained her PhD and MA in computer science from Princeton University. She is a member of the ACM and the IEEE Computer Society.

Direct questions and comments about this article to Shan Lu, 201 N. Goodwin Ave., Thomas M. Siebel Center for Computer Science, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2302; shanlu@uiuc.edu.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.