AVIO: Detecting Atomicity Violations via Access Interleaving Invariants

Shan Lu, Joseph Tucek, Feng Qin and Yuanyuan Zhou

Department of Computer Science,
University of Illinois at Urbana Champaign, Urbana, IL 61801
{shanlu,tucek,fengqin,yyzhou}@uiuc.edu

Abstract

Concurrency bugs are among the most difficult to test and diagnose of all software bugs. The multicore technology trend worsens this problem. Most previous concurrency bug detection work focuses on one bug subclass, data races, and neglects many other important ones such as *atomicity violations*, which will soon become increasingly important due to the emerging trend of transactional memory models.

This paper proposes an innovative, comprehensive, invariant-based approach called AVIO to detect atomicity violations. Our idea is based on a novel observation called *access interleaving invariant*, which is a good indication of programmers' assumptions about the atomicity of certain code regions. By automatically extracting such invariants and detecting violations of these invariants at run time, AVIO can detect a variety of atomicity violations.

Based on this idea, we have designed and built **two** implementations of AVIO and evaluated the **trade-offs** between them. The first implementation, AVIO-S, is purely in software, while the second, AVIO-H, requires some simple extensions to the cache coherence hardware. AVIO-S is cheaper and more accurate but incurs much higher overhead and thus more run-time perturbation than AVIO-H. Therefore, AVIO-S is more suitable for in-house bug detection and postmortem bug diagnosis, while AVIO-H can be used for bug detection during production runs.

We evaluate both implementations of AVIO using large real-world server applications (Apache and MySQL) with six representative real atomicity violation bugs, and SPLASH-2 benchmarks. Our results show that AVIO detects more tested atomicity violations of various types and has around 15 times fewer false positives than previous solutions on average.

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

General Terms Languages, Reliability

Keywords concurrent program, atomicity violation, concurrency bug, bug detection, hardware support, program invariant

1. Introduction

1.1 Motivation

Concurrency bugs in multi-threaded and multi-process programs are among the most difficult to test and diagnose of all software bugs. As they are non-deterministic, requiring specific thread or process schedulings to expose, they are hard to trigger. This frustrates both in-house testing and reproduction for postmortem diagnosis. In the real world, most server and high-end critical softwares are multi-threaded or multi-process. 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 [35].

Recent hardware advances further worsen the concurrency bug problem. With SMT and CMP architectures becoming mainstream, more and more multi-threaded applications are being written in order to take advantage of the available processors. As a result, one can expect an increasing number of concurrency bugs in the near future. Therefore, good techniques to automatically detect these bugs are greatly desired.

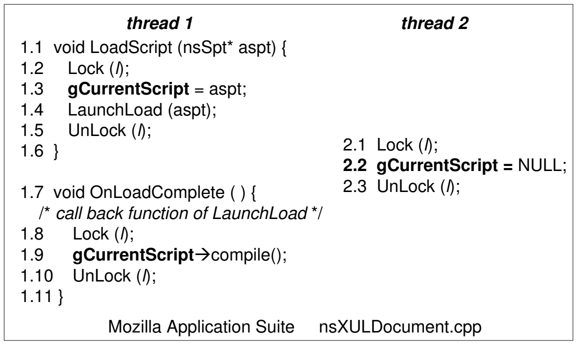
Most previous concurrency bug detection work focuses on one subclass: data race detection. A data race occurs when two accesses, at least one of which being a write, from different threads to the same memory location execute without proper synchronization. Many dynamic detection tools have been proposed to address this problem. They can be grouped into three categories: lockset bug detection tools [6, 34], happens-before bug detection tools [7, 26, 28], and various hybrid tools combining the two [27, 29, 43]. The lockset algorithm reports a data race bug when it finds that there is no common lock held during accesses to a shared memory location. Happens-before bug detection is based on Lamport's happens-before relation [19]. During concurrent execution, the happens-before partial order of all memory accesses is maintained based on synchronization activities. A data race bug is reported when two conflicting memory accesses do not have a strict happens-before relation. Different forms of happens-before algorithm have been implemented in hardware [23, 30, 31] to overcome the overhead problem (i.e. factor of 10 to 100 slowdowns) in software race detection. The third class of tools combine the above two techniques for better detection accuracy.

1.2 Limitations with Previous Approaches

While the above three approaches can effectively detect some concurrency bugs in multi-threaded programs, they are far from providing a complete solution for detecting concurrency bugs due to the following reasons:

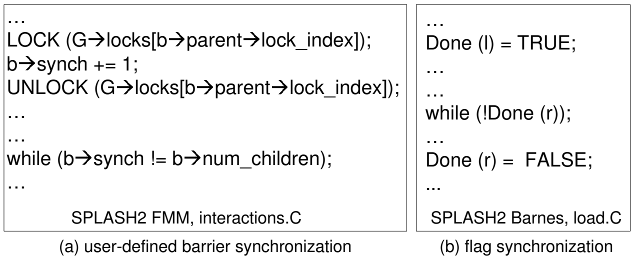
(1) In many cases, what programmers want is atomicity, not freedom from data races. Being data race-free does not necessarily indicate correct synchronization. Many concurrency bugs would

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.



```
            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 }
        Mozilla Application Suite     nsXULDocument.cpp
```

Figure 1. Being data-race free does not guarantee correct synchronization. This is a real bug in the Mozilla Application Suite. It is slightly simplified for illustration. When thread 2 violates the atomicity of thread 1's accesses to gCurrentScript, the program crashes.



```
…
LOCK (G→locks[b→parent→lock_index]);
b→synch += 1;
UNLOCK (G→locks[b→parent→lock_index]);
…
…
while (b→synch != b→num_children);
…
        SPLASH2 FMM, interactions.C
```

(a) user-defined barrier synchronization

```
…
Done (l) = TRUE;
…
…
while (!Done (r));
…
Done (r) =  FALSE;
...
  SPLASH2 Barnes, load.C
```

(b) flag synchronization

Figure 2. User-defined synchronization mechanisms in SPLASH-2 benchmarks Barnes and FMM

still occur even when all accesses are protected by correct locks and have strict happens-before relations between each other. Figure 1 is a real bug example from the Mozilla Application Suite. As we can see, gCurrentScript is a shared handler to the script being processed. Thread one first sets this handler and launches the script loading as an asynchronous event. After the loading finishes, thread one will continue the script processing based on the previously set handler. Meanwhile, thread two may nullify the handler.

This example does **not** have any data race because one common lock protects every access to gCurrentScript. However, it still contains a severe concurrency bug: an *atomicity violation*, which will lead to a crash once triggered.

Atomicity, also referred to as **serializability**, is a property for several concurrently executed actions, when their data manipulation effect is equivalent to that of a serial execution of them [1]. For the Figure 1 example, the two parts of script processing from thread 1 are expected to be atomic, i.e., never be unserializablly interleaved by any modification to the gCurrentScript. Otherwise thread 1 will consume the wrong script handler on line 1.9.

An important lesson learned from the above example is that, in many cases, what programmers want is the *atomicity* of code segments, not necessarily freedom from data-race [12, 37, 42]. When writing code, programmers tend to assume nearby operations in one thread to be atomic. If an atomicity expectation is not satisfied by the actual implementation, some executions will have unserializable interleavings. This violates the programmers' assumptions, and can manifest as concurrency bugs. Using locks is just *one way* to ensure atomicity (assuming that the locks are correctly placed), but, as demonstrated in Figure 1, being data-race free does not guarantee proper atomicity.

(2) Data race is not a problem for future transaction-based concurrent programs, but atomicity violation still is. Recently there is an emerging trend toward transactional memory programming model via hardware or software support [1, 13, 16, 18]. Transactions provide programmers with an interface to ensure atomicity. Programmers using this model needn't worry about data races because the underlying system and hardware automatically detects conflicting accesses from different transactions and will roll back one to resolve any conflict. In contrast, atomicity violations will still happen in transaction-based programs when programmers do not group operations that should be atomic into the same transaction. For example, if the programmer divides the code segment shown in Figure 1 into two separate transactions, the atomicity violation bug still remains.

(3) A data race is not always a bug. Many important synchronization mechanisms are actually implemented using data races. Examples include barriers, flag synchronization, producer-consumer queues, etc. Figure 2 gives two such examples from the SPLASH-2 parallel benchmark suite [40], in which user-defined barrier and flag synchronization are achieved via races on the shared variables $b{\rightarrow}synch$ and r, respectively. Furthermore, sometimes programmers intentionally choose to allow a data race for better performance. For example, in the Microsoft CLR library, *performance counter* is regarded as inconsequential, so accesses to it are not protected [43]. Unfortunately, previous data-race detection solutions cannot differentiate these benign races from true bugs.

(4) Most previous techniques rely on specific synchronization semantics. Both the happens-before and lockset algorithms need to be informed in advance about the synchronization primitives used in the target program. Ignorance of non-lock based synchronization primitives, such as *barrier*, thread *create/join*, or raising the interrupt level, has caused many false positives in previous work [6, 34, 43]. Worse yet is user-defined synchronization, as shown by the examples in Figure 2, which is both common and difficult to recognize without prior knowledge. As a result, shared data access protected by user-defined synchronization is an inevitable source of spurious bug reports in all happens-before and lockset based tools (also shown in our experimental results), reducing their effectiveness for programmers.

1.3 State of the Art

Although the problem of atomicity violations has been known for years, few good solutions exist to address this challenging problem. Most state-of-the-art techniques [11, 12] for detecting these bugs require programmers' annotations on atomic regions, which is too time-consuming for programmers. Additionally, in many cases programmers are not consciously aware of their atomicity assumptions, and therefore cannot provide accurate annotations.

Most recently, the SVD tool [42] pioneers the direction of atomicity violation detection without programmer annotation. It automatically infers *computation units*, i.e. atomic regions, based on data and control dependencies. SVD reports bugs when such regions are interleaved by unserializable writes.

However, while SVD makes an inspiring start, it only looks at a limited subset of atomicity violation bugs: unserializable interleavings to atomic regions that start with *reads* on shared variables and expand based on read-write or control dependencies. We will show later in Section 2 that among all eight possible pairwise interleavings, SVD is applicable for only two out of the four (50%) unserializable cases. The bug in Figure 1 cannot be automatically detected by SVD for exactly this reason. Finally, due to its com-

[1] In databases, atomicity includes two properties: serializability and indivisibility. But in general programming, atomicity is usually considered equivalent to serializability

plex dynamic dependency analysis, SVD is implemented purely in software and slows applications by up to 65 times [42].

1.4 Our Contributions

This paper makes three contributions in order to address the limitations of previous works:

Contribution 1: **Access Interleaving (AI) Invariant based detection**. We propose an innovative, comprehensive, invariant-based approach called AVIO to detect a class of hard bugs: general atomicity violations. Our idea is based on the following two novel observations:

(1) AI Invariants: There exists a unique type of invariants in code regions that are expected by programmers to be atomic (regardless whether the implementation actually guarantees atomicity or not). That is, during correct runs (runs that do not expose any concurrency bugs), two consecutive accesses from one thread to the same shared variable are not interleaved by an unserializable access from another thread. This invariant reflects the programmer's intention at these accesses: are conflicting accesses from other threads welcomed, forbidden, or do-not-care? If such an invariant, i.e. the programmers' assumption, is violated, a concurrency bug will happen (Section 2.1).

(2)Feasibility of AI Invariant Extraction: Large number of correct runs is what we need to extract invariants. Concurrency bugs happen only in rare cases even with bug-exposing inputs, because their manifestation requires specific access interleavings. In addition, running a concurrent program multiple times, even with the same input, generates many different access interleavings. Therefore, it is feasible and relatively easy to obtain a collection of many different and correct access interleavings. AI invariants can then be extracted from the correct runs (Section 2.3).

Based on the above two observations, the main idea of AVIO is to automatically discover from correct runs those important code segments that are assumed to be atomic by programmers, and then to use such invariants to perform on-line detection of atomicity violation bugs.

Contribution 2: **design and trade-off study of software and hardware implementations of AVIO**. We design and build **two** implementations of our idea: AVIO-H in hardware and AVIO-S in software. AVIO-H requires *simple* hardware extensions to the hardware cache coherence protocol and achieves negligible overhead and little perturbation to application execution. In contrast, AVIO-S is a pure software approach which is more accurate than AVIO-H but incurs larger but still moderate overhead. Therefore, AVIO-S is more suitable for in-house bug detection and postmortem diagnosis while AVIO-H can be used for bug detection during production runs. Specifically, AVIO can be used for:

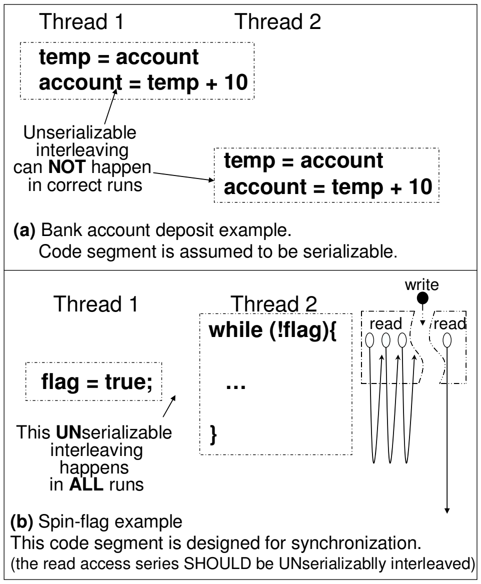
(1) Postmortem analysis: Programmers can use AVIO to diagnose the root cause of software failures. Given a failure to track down, a programmer can use AVIO to collect and compare AI invariants during correct runs and buggy runs to check for possible atomicity violation root causes.
(2) On-the-fly detection: Programmers can also use AVIO to automatically extract AI invariants during in-house testing. Then these AI invariants can be used during production runs to detect atomicity violations and pinpoint code regions that lack atomicity protection.

Contribution 3: **Evaluations with real-world server applications**. We have evaluated the **two** implementations of AVIO using six representative **real** atomicity violation bugs from two large real-world server applications, the Apache HTTP Server and the MySQL database server, and an extracted version of Mozilla, running on either real machines (for our software-only AVIO-S) or the whole-system simulator Simics [22] (for our hardware AVIO-H). Our experimental results show that, compared to previous approaches, AVIO has the following unique advantages:

- **Detects a variety of atomicity violation bugs**. Our comprehensive serializability analysis shows that, among eight possible access interleavings, AVIO can detect more cases of atomicity violations than previous algorithms. In addition, AVIO's detection also covers many bugs that are not addressed by data race detection. Our experiments show that AVIO detects more tested real atomicity violations of various types than previous algorithms (SVD, happens-before and lockset).
- **Oblivious to benign races**. Since benign races are good races desired by programmers (as shown in Figure 2), unserializable access interleavings are allowed and even welcomed at such places. During correct execution, these unserializable interleavings will happen often, and therefore no AI invariants will be observed at these accesses. As such, AVIO can easily differentiate true bugs from benign races.
- **Few false positives**. As AVIO does not rely on synchronization primitives and is oblivious to benign races, AVIO reports only a few (on average 3-5) static false positives for our evaluated applications. In contrast, previous methods have an average of 51 false positives, which significantly undermines their bug detection capability because programmers need to sift through 51 error reports in order to find one true bug.
- **Requires no annotations or specifications**. Unlike many previous approaches, our AVIO does not require programmers to provide any specifications about synchronization primitives or atomic region assumptions. Therefore, our idea applies for not only multi-threaded programs using standard lock-based synchronization, but also those using application-specific synchronizations as well as future applications that are written in transactional memory models.
- **Non-stringent requirements for training data generation**. Training is important in all invariant-based approaches [14, 44]. Fortunately, employing the nondeterministic characteristic of concurrent programs, i.e. different runs with the *same* input *automatically* have drastically different access interleavings, training data generation in AVIO has unique advantages over previous invariant-based approaches. Our results show that, running less than 5 times for SPLASH-2 benchmarks and less than 100 requests for server applications are well more than enough to generate a reasonably accurate set of AI invariants (see Section 6.3 for a detailed sensitivity analysis).
- **Imposes low overhead**. AVIO, in particular the hardware implementation AVIO-H, imposes very little (0.4–0.5%) overhead, orders of magnitudes smaller than software-based concurrency bug detection tools. Our software implementation AVIO-S incurs 15-42 times overhead, which is still lower than previous software-based tools such as SVD (65X) [42] and Valgrind-lockset (> 200X).

2. AVIO Idea

Terminology definitions: For simplicity, unless otherwise mentioned, all accesses are to the same shared memory location. We refer to the thread whose atomicity is interrupted as the *local thread* and its accesses as *local accesses* or *local reads/writes* (note that this does NOT mean a local variable). We refer to the thread with the interleaving access as the *remote thread* and its accesses as *remote accesses* or *remote reads/writes*. A *serializable* interleaving is an interleaving between local and remote accesses that is equivalent to a serial non-interleaving execution.



(a) Bank account deposit example.
Code segment is assumed to be serializable.

(b) Spin-flag example
This code segment is designed for synchronization.
(the read access series SHOULD be UNserializablly interleaved)

Figure 3. Example with an AI invariant (a) and WITHOUT an AI invariant (b)

2.1 Access-Interleaving Invariance

The *essence* of atomicity violation bugs is no different than other types of bugs: they are caused by a mismatch between the code implementation and the programmer intention. Specifically, programmers assume that a sequence of shared variable accesses is atomic, never interleaved by unserializable accesses, but the implemented code does not guarantee this property and thus bugs emerge.

Programmers' atomicity intention comes in different formats. The most common and fundamental one can be represented by a type of invariant that we refer to as an **Access-Interleaving invariant** (AI invariant). Such an invariant is held by an instruction if the access pair, composed of itself and its preceding local access to the same location, is never *unserializably* interleaved. We denote this instruction as **I-instruction** (invariant instruction) and the preceding access instruction as **P-instruction** (preceding instruction). Note that with an AI invariant, it is perfectly OK to have interleavings. Atomicity would be maintained as long as the interleavings are serializable. Section 2.2 will further discuss interleaving serializability.

Figure 3(a) gives a simple demonstration of an AI invariant using the classic banking account example. In this code, programmers assume that the read and modification of *account* are always together and never be unserializably interleaved by a conflicting remote access. Otherwise, an atomicity violation can result in program misbehavior.

An AI invariant indicates programmers' atomicity assumption. Such assumption is the essence of concurrent execution correctness. Atomicity assumptions, as in the banking account example, are made during design and implementation by programmers who are more comfortable with sequential thinking. Assumptions may be enforced through locks, barriers, flags or other synchronization mechanisms such as transactions. Poorly enforced atomicity assumptions cause synchronization errors during some executions.

We should note that programmers do *not* assume all code regions to be atomic, nor does AI invariant held for every shared variable access instruction. Contrarily, some instructions often *allow* unserializable interleavings. For example, in cases like flag-based synchronization implementation, programmers explicitly do not want an AI invariant. Automatically differentiating code that is or is not expected to have AI invariant would allow us to avoid many benign races, which are the major source of false positives in previous techniques. For example, Figure 3(b) illustrates synchronization implemented using a flag variable. During execution, no AI invariant will be observed at the `flag` read access of the while loop, because unserializable interleavings happen in every run. Such AI non-invariant matches the programmers' intention in this example at this position: an unserializable interleaving by a remote access is *required* to ensure liveness.

Of course, AI invariant is not the only format of programmers' atomicity assumption, but it is the most common and fundamental one. Other assumptions involving multiple shared variables are rarer and can be potentially extended from AI invariants. We will discuss them in Section 7.

In summary, atomicity violation bugs are code regions that are expected to be atomic but implemented as non-atomic. At run time, serial execution or serializable interleavings definitely maintain the atomicity; meanwhile, unserializable interleavings do NOT necessarily violate correctness. Which part of code needs to be atomic and serializable depends on programmers' intentions and is well indicated by AI invariants. Therefore, if we can automatically extract AI invariants, this knowledge can then be used to detect atomicity violation bugs by monitoring "unexpected" unserializable interleavings in code segments where AI invariants should hold.

2.2 Serializability Analysis

Not all interleavings are unserializable, and serializable interleavings do not lead to atomicity violation. In this section, we first analyze what interleavings are serializable and what are not. There are totally eight ways that two consecutive local accesses to the same shared variable can be interleaved by a remote access. Table 1 describes every cases, explaining why each case is serializable, with equivalent serial accesses, or unserializable, with a bug example.

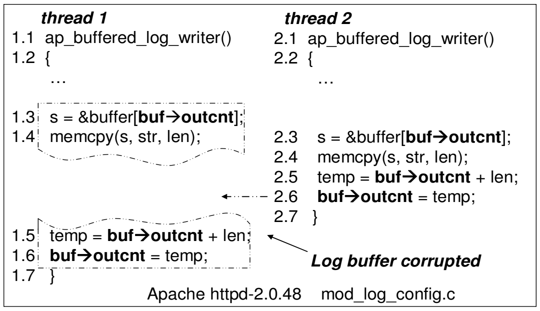
Among the eight cases, four (cases 0, 1, 4, 7) are serializable interleavings while the other four (case 2, 3, 5, 6) are not. We have an example bug for each unserializable case in which programmer's assumptions about atomicity is violated. For example, Figure 4 gives a real bug from the Apache httpd server whose root cause is a case 2 unserializable interleaving. Figure 5 shows a real bug example from the MySQL database server for case 5. Similarly, case 3 and case 6 are exemplified by the examples shown early in Figures 1 and Figure 3(a). But note that, as discussed in the previous section, unserializable interleavings are not necessarily bugs (Figure 3(b)) unless they violate programmers' assumptions.

Above we get the unserializable condition, composed of four cases, for *single* interleaving remote access. Extending it, we get following similar four-case unserializable conditions with *multiple* remote accesses to the same shared variable taken into account. *This condition will be used in the rest of the paper, guiding AVIO bug detection* (For illustration, interleaving remote accesses are put in parentheses; $*$ denotes zero or multiple interleaving read or write accesses; superscript i and p stand for one access and its preceding access from the same thread):

- Case2: $r^p[*w_r*]r^i$, two local reads are interleaved by *at least one* remote write, so they may have different views.
- Case3: $w^p[*w_r*]r^i$, a local read after write is interleaved by *at least one* remote write. Due to this remote write, the read would fail to get the local result that it expects.
- Case5: $w^p[r_r*]w^i$, a local write after write is interleaved by a remote access sequence that *starts with* read, making the local intermediate result visible to a remote thread.
- Case6: $r^p[*w_r*]w^i$, a local write after read is interleaved by *at least one* remote write. It makes the previous reading result stale.

Interleaving	Case #	Description	Serializability	Equivalent serial accesses	Problems (for unserializable cases)	Bug Example
read^p $read_r$ read^i	0	two reads interleaved by a read	serializable	read^p read^i $read_r$	N/A	N/A
write^p $read_r$ read^i	1	read after write interleaved by a read	serializable	write^p read^i $read_r$	N/A	N/A
read^p $write_r$ read^i	2	two reads interleaved by a write	*unserializable*	N/A	The interleaving write makes the two reads have different views of the same memory location	Apache Figure 4
write^p $write_r$ read^i	3	read after write interleaved by a write	*unserializable*	N/A	The local read does not get the local result it expects	Mozilla Figure 1
read^p $read_r$ write^i	4	write after read interleaved by a read	serializable	$read_r$ read^p write^i	N/A	N/A
write^p $read_r$ write^i	5	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	MySQL Figure 5
read^p $write_r$ write^i	6	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	Bank account Figure 3 (a)
write^p $write_r$ write^i	7	two writes interleaved by a write	serializable	$write_r$ write^p write^i	N/A	N/A

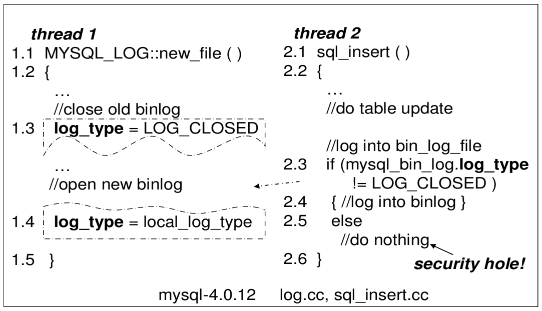
Table 1. Eight cases of access interleavings. All accesses are to the *same* shared variables. Besides read/write, subscript r denotes remote interleaving access; superscript i and p denotes one access and its preceding access from the same thread. Note: we will differentiate general and invariant-related instructions by lower-case i/p and upper-case I/P.



```
thread 1                               thread 2
1.1 ap_buffered_log_writer()           2.1 ap_buffered_log_writer()
1.2 {                                  2.2 {
     ...                                    ...

1.3  s = &buffer[buf→outcnt];
1.4  memcpy(s, str, len);              2.3  s = &buffer[buf→outcnt];
                                       2.4  memcpy(s, str, len);
                                       2.5  temp = buf→outcnt + len;
                                       2.6  buf→outcnt = temp;
                                       2.7  }
1.5  temp = buf→outcnt + len;
1.6  buf→outcnt = temp;                    Log buffer corrupted
1.7  }
             Apache httpd-2.0.48   mod_log_config.c
```

Figure 4. A real bug from Apache httpd server with case 2 unserializable interleaving: read-read interleaved by a write. Specifically, the read accesses at lines 1.3, 1.5 are interleaved by line 2.6 write of thread 2. buf→outcnt is read for index update after an append, however its value has already been modified since line 1.3 read. As a result, the server log is corrupted. (there is also a potential case 6 unserializable interleaving between 1.5 and 1.6)



```
thread 1                              thread 2
1.1 MYSQL_LOG::new_file ( )           2.1 sql_insert ( )
1.2 {                                 2.2 {
     ...                                   ...
     //close old binlog                    //do table update
1.3  log_type = LOG_CLOSED
                                           //log into bin_log_file
     ...                              2.3  if (mysql_bin_log.log_type
     //open new binlog                          != LOG_CLOSED )
                                      2.4   { //log into binlog }
1.4  log_type = local_log_type        2.5   else
                                              //do nothing
1.5  }                                2.6  }            security hole!
            mysql-4.0.12   log.cc, sql_insert.cc
```

Figure 5. A real bug from MySQL database server with a case 5 unserializable interleaving: write-write interleaved by a read. Specifically, the write accesses at lines 1.3, 1.4 are interleaved by line 2.3 read of thread 2. As a result, thread 2 at line 2.3 reads an intermediate value produced by thread 1 and thinks that the log file is already closed. As a result, thread 2's database action is not recorded in log, which creates a security back door.

2.3 Automatically Extract AI Invariants

A challenging question is how to obtain AI invariants, knowing which code regions do not welcome unserializable interleavings. In this section, we describe the high level idea used in AVIO. The detailed process will be given in section 3.2.

Obviously, we cannot expect programmers to provide such invariants because atomicity violations usually occur in code segments where programmers are not consciously aware of their assumptions. Similarly, we cannot use lockset analysis to extract AI invariants without suffering from the same limitations (discussed in Section 1) as previous lockset based algorithms.

To automatically learn a programmer's intention, the best way is to study the program's behavior in correct execution: if a code segment is always serializable in correct runs (runs where no bug manifests), it is probably assumed to be so always. In other words, we can *statistically* "learn" a program's AI invariants through training. Specifically, to collect and analyze access interleavings from a set of correct runs (**training runs**), we can see which shared accesses (such as the one in Figure 3(b)) allow unserializable interleavings, and which shared accesses *never* have unserializable interleavings.

The feasibility of the above idea depends on how well the training can be: (1) How to ensure training is dominated by *correct* runs (correctness issue)? (2) How to get *sufficient different* training samples (sufficiency issue)? These two issues are critical in all invariant-based techniques [9, 14, 44]. Fortunately, two unique and "notorious" properties of concurrency bugs make training in AVIO easier than general invariant training. In other words, we have turned the negative "troublesome" bug characteristics into positive characteristics in detecting these types of bugs.

First, the **correctness** issue is addressed by two facts: (1) Concurrency bugs manifest very infrequently, even with bug-exposing inputs. Their manifestation usually requires specific access interleaving, a notorious feature that makes concurrency bug very hard to reproduce for postmortem diagnosis. Practical experience with real bugs shows that, even with bug-triggering inputs, usually it still takes hundreds, thousands, or more of repeated executions to trigger a bug. As a result, we can easily get correct-dominated training. (2) Existing infrastructure and research in software testing can be leveraged to label training runs as correct or incorrect. In particular, according to a previous work [5], most concurrency bugs are fail-stop. Furthermore, software testers usually have various methods (beyond crashes or hangs) during in-house testing to determine the correctness of test runs. Additionally, assertions and automatically extracted predicates [20] can further help to filter out incorrect training runs. The AI extraction algorithm can also be designed to tolerate a small percentage of unfiltered incorrect training runs.

Second, the **sufficiency** issue is addressed by the fact that concurrent execution is **non-deterministic** due to the underlying thread interleaving. As we all know, both multi-processor execution and operating system thread scheduling have a lot of randomness. As such, even with just one input, we can easily get a large number of distinct access interleavings. For example, in our experiments, 100 runs of a SPLASH-2 benchmark with just one input always generate 100 different traces.

Benefiting from the *non-determinism*, training in AVIO's post-mortem analysis (usage model 1) is very easy. Just running the program with the bug triggering input many times, and we will get sufficient access-interleaving training results. This is a big advantage over traditional invariant-based tools. As for on-the-fly-detection (usage model 2), the capability of AVIO is related to its path coverage, which is a problem for *all* dynamic bug detection tools, not only for invariant-based techniques. In AVIO, if the training does not cover a particular code block, no AI invariant is available there and false negatives may occur. We need to rely on a reasonable branch coverage of in-house testing suite and AVIO can be extended to actively learn new invariants during detection. Value coverage is less of a concern, because AI invariants are associated with instructions and interleavings, not with data addresses or values. With a different input, an instruction may access data with a different value, but the programmer's assumption about the desired atomicity associated with this instruction remains the same.

The above analysis indicates it is feasible to extract AI invariant by training. Our experiments further validate this. All the real server bugs detected by AVIO are based on training with just one input and fewer than 100 training requests. The training input value is also flexible, as indicated in our experimental input sensitivity study (section 6.3).

3. AVIO Algorithms

AVIO automatically extracts AI invariants from off-line testing runs, and then detects potential violations to the extracted AI invariants during monitored runs. As the AI invariant extraction algorithm is based on the detection algorithm, we will first describe the detection algorithm and then present the extraction algorithm.

3.1 Detection Algorithm

Suppose that we already have a set of AI invariants, which gives 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. Based on our serializability analysis in section 2.2, to detect any such unserializable interleaving, the detection process can simply follow the binary decision diagram in Figure 6, which summarizes all the four unserializable interleaving cases.

The decision diagram in figure 6 clearly shows that AVIO needs four pieces of information to tell unserializable interleavings from serializable ones. These four pieces of information are: access type of the current instruction (i.e. I-instruction Type); access type of the preceding local instruction to the same memory location (P-instruction Type); interleaving remote write information and interleaving remote read information. With these four pieces of information, AVIO can easily detect violations to AI invariants. We will show later in Section 4 how these four pieces of information are collected in both our hardware and software implementations of AVIO.

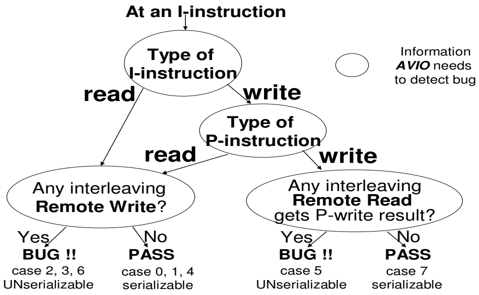


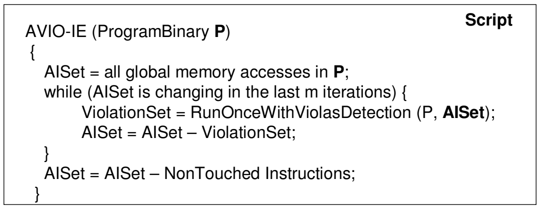
Figure 6. AVIO bug detection procedure (This diagram can be better understood when referring to table 1)

3.2 Extraction Algorithm

The goal of the AI invariant extraction, referred to as AVIO-IE, is to extract AI invariants from multiple correct runs.

Interestingly, AVIO-IE can be easily implemented by leveraging the AI invariant violation detection process. Specifically, the extraction process is a series of correct runs with the AVIO detection enabled. As shown in Figure 7, initially the set of AI invariants, $AISet$, includes all global memory accesses in the target program. Then it runs the program on top of AVIO multiple times. At the end of each run, AVIO reports "violations" to the current $AISet$ in this run. A violation at an instruction i indicates that an unserializable interleaving is encountered in the current *correct* run (labeled by the testing oracle). Therefore, there is no true AI invariant at i, i.e. i should be removed from $AISet$. This process will repeat many times until $AISet$ remains unchanged for the last m runs, where m is adjustable. In Section 6, we will show sensitivity results of the number of training runs. Finally we filter out never-executed instructions and return $AISet$.

To tolerate a small percentage of incorrectly labeled training runs (i.e. an incorrect run is labeled as correct), AVIO-IE can introduce an invariant filtering threshold T. Only when an invariant is violated in more than T training runs that pass the testing oracle, this invariant is removed from the $AISet$. This technique can avoid some actual invariants being filtered due to some incorrectly labeled training run, but at the cost of potentially more false positives in violation detection. So the best way is for programmers to adjust the threshold parameter based on the accuracy of their testing oracles as well as their false positive tolerance level.



```
                                                          Script
AVIO-IE (ProgramBinary P)
{
   AISet = all global memory accesses in P;
   while (AISet is changing in the last m iterations) {
         ViolationSet = RunOnceWithViolasDetection (P, AISet);
         AISet = AISet – ViolationSet;
   }
   AISet = AISet – NonTouched Instructions;
 }
```

Figure 7. AVIO's process of extracting AI invariants.

4. Two AVIO Implementations

To study the trade-offs between hardware and software, we implement our AVIO idea and algorithms in two different approaches: a software-only approach AVIO-S and a hardware-assisted approach AVIO-H. As the AI invariant extraction is done during in-house testing, it is less overhead critical. Therefore, extraction is implemented based on AVIO-S.

4.1 Hardware AVIO (AVIO-H)

4.1.1 AVIO-H Overview

The hardware implementation, AVIO-H, takes advantage of existing cache coherence protocol and achieves negligible overhead and little execution perturbation with simple hardware extensions as shown in Figure 8. AVIO-H currently assumes a CMP machine with a physical-address indexed private-L1 cache and a unified-L2 cache hierarchy, using an invalidation-based cache coherence protocol. Extending it to other multiprocessor architecture such as SMP and other cache coherence protocols is relatively straightforward, especially since our detection algorithms described in Section 3 are not implementation specific.

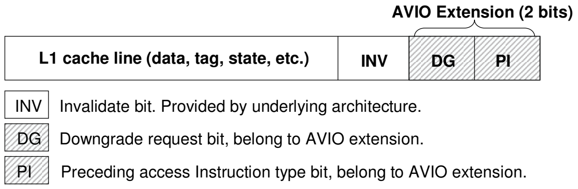

AVIO Extension (2 bits)
L1 cache line (data, tag, state, etc.)
INV
DG
PI
INV
Invalidate bit. Provided by underlying architecture.
DG
Downgrade request bit, belong to AVIO extension.
PI
Preceding access Instruction type bit, belong to AVIO extension.

Figure 8. AVIO-H's extension to each L1 cache line

First, AVIO-H appends each L1 cache line with two new access-information bits. These new bits, together with the existing invalidate (INV) bit used by the cache coherence protocol, provide enough information to perform the AVIO detection algorithm described in Section 3:

- **PI bit** (Preceding access Instruction bit): This bit provides the "Type of P-instruction" information. It is set to 1 at each local read to the corresponding cache line and is unset at each local write.
- **DG bit** (Downgrade bit): This bit provides information to find out whether the previous local write's result has been read by a remote thread. Interestingly, in existing invalidation-based cache coherence protocols, such an action is associated with a *Downgrade* request sending from the reader to the recent writer. Therefore, AVIO-H just needs to set the DG bit upon a *Downgrade* request and unset the bit after each local access.
- **INV bit**: This bit already exists in current cache coherence hardware. It provides information about any "interleaving remote write" after the previous local memory access. In existing invalidation-based cache coherence protocol, interleaving remote writes will invalidate all other L1 caches' copies. Therefore, AVIO-H just needs to check the INV bit to see whether a remote write has happened.

Second, the hardware cache coherence protocol is extended to support the above information bits and violation detection. Finally, we add special instruction encodings for I-instructions (reads and writes) and a special bit in the L1 cache access command to indicate when a memory instruction is an I-instruction. Using these extensions, we can easily implement the detection protocol in hardware as shown in figure 9.

Complexity and Overhead Both the state maintenance and bug detection in the AVIO-H have very simple logic, as shown in figure 9. Interestingly, further studying the detection protocol indicates that unserializable interleaving only happens when the original cache coherence protocol cannot use the local copy and needs to contact L2 to get the most-up-to-date copy and/or exclusive write permission. Therefore, AVIO-H's detection process is triggered only when an I-instruction cannot be satisfied by its local L1 cache.

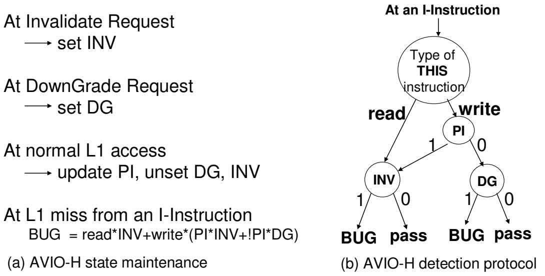


Figure 9. AVIO-H state maintenance and bug detection (This is a hardware version of figure 6)

The whole detection phase has small space overhead and negligible time overhead. The extra space is just two bits per L1 cache line, less than 0.4% overhead. Because the invariant check is conducted only when an I-instruction has to go to the shared L2 cache, the check is not in the critical path–the simple detection protocol can be hidden by the L2 cache access latency. Only when a bug is found, AVIO-H needs to impose overhead recording it.

4.1.2 Design Issues of AVIO-H

After describing AVIO-H's basic mechanism, this section discusses some design issues. Some of these issues have no effect on AVIO-H, while some others can, in rare cases, affect AVIO-H's accuracy in ways similar to previous hardware data race detectors. All these issues are specific to our hardware implementation, and do not affect our software implementation (described in Section 4.2).

Recording and reporting atomicity violations After detecting an AI invariant violation, AVIO-H marks the I-instruction in the reorder buffer and sends a signal when this instruction retires. Therefore, no bug is reported for speculative instructions. AVIO-H supports two bug reporting options: either break the execution with an exception, or only record the I-instruction's PC and accessed address to a memory location specified by the software.

Cache line displacement and context switch Recent access history of a cache line may be lost when it is displaced. This problem is also encountered in most previous hardware race detectors [30, 31] and was simply ignored because it only results in false negative in very rare cases. This is especially true for AVIO because AI invariants focus on two *consecutive* accesses to the same memory location from the same thread. Intuitively, these two accesses are nearby (which is why programmers forget to protect them in the first place) and therefore the probability for them to be interleaved by a displacement of an involved cache line is very small. In addition, we can always postpone displacing such a cache line by first evicting a private (e.g. a stack) cache line. Similarly, context switches can also create some false positives in AVIO-H as well as most previous hardware race detectors, and its probability is also very low for similar reasons. These issues can be addressed in the future by thread-id tagging or employing directory as victim buffer in directory-based cache coherence protocol.

Load-store queue and write-coalescing Some read access may be invisible to AVIO-H if they hit the load-store queue. Fortunately, if this access is an I-instruction, a hit in load-store queue definitely indicates *no remote interleaving* between this read and previous local access, so it is perfectly fine that AVIO-H is not checking this "invisible" access. For the same reason, write-coalescing also has no effect on bug detection. But if this access is a P-instruction right before an I-instruction, it can lead to a two-fold effect. On the positive side, AVIO-H may thus be enabled to detect atomicity violation to a larger code region, since it mistakes an access sequence $w^{P1}r^{P2}r_rw^I$ by $w^{P1}r_rw^I$, with r^{P2} hit the load-store queue and invisible to L1 cache. On the negative side, $w^{P1}r^{P2}w_rw^I$ may be mistaken as $w^{P1}w_rw^I$, and AVIO-H may miss the bug. In summary, in most cases, the load-store queue has no effect; in a very small percentage of cases, it may either help or harm AVIO-H in detecting some bugs. Note that similar issues are faced by previous hardware race detectors. Previous solution forces global memory accesses to go through the lower memory hierarchy [32]. Since this issue rarely has bad effect on AVIO-H, we did not choose this solution in our current prototype.

Strict/weak consistency model, out-of-order access and execution issues Different memory consistency models may cause different memory access orders for the same concurrent program. However, it does not affect the bug detection. No matter what the access order is, what AVIO-H sees is the *actual* order executed on hardware. Out-of-order execution similarly has no effect on AVIO-H. The only exception here is when prefetching results are finally discarded *and* the access interleaving matches case 5, which is a low probability event, there may be some false positives.

False sharing due to cache line granularity In our design AVIO-H uses a cache line as the unit for information keeping and bug detection. It may introduce some false sharing, an issue also faced by previous hardware race detectors [31]. It can be solved by simply using a smaller granularity (e.g. word) at the expense of increasing space overhead and bus traffic. This problem can also be alleviated by using profiling to find false sharing and then using a compiler to automatically add paddings. Such processes have already become a standard optimization to reduce unnecessary cache coherence traffic and cache misses for performance reasons.

Other sources of cache line invalidation In AVIO-H, we use each cache line's INV bit to record remote write access information. In addition to cache coherence invalidation, this bit can also be set by other sources such as DMAs. This does not interfere AVIO-H's bug detection capability. Atomicity violation would still be correctly reported even though it may be from a DMA operation.

Support for SMT Our current AVIO design is based on CMP/SMP. To support SMT, AVIO needs a simple extension: tag L1 cache lines with thread-ids.

Compatibility with certain processor and cache coherence protocol The cache management policy required by AVIO is general: an invalidation-based cache coherence protocol. However, there may be some real processor that is incompatible with AVIO's current prototype. In that case, AVIO needs simple extension to the cache coherence protocol to get some bug detection required information, such as downgrade information.

4.2 Software AVIO (AVIO-S)

To study the trade-offs between efficiency and accuracy, we also implemented the AVIO techniques purely in software.

Like AVIO-H, the key task of AVIO-S is to collect and maintain access information required by the AVIO detection protocol. Specifically, for all global memory, AVIO-S maintains the most recent local and remote access history information, and then uses it to check for possible violations at I-instructions. In software, various access information is collected by binary instrumentation at every global memory access and maintained in an access-table data structures. Each thread has an access-table, holding the type information of its latest access to each global memory location. There is also a global access-owner-table, holding the identifier of the thread that most lately wrote to each global memory location. At each memory access from I-instruction, the P-Instruction Type can be obtained from the local-access table; and the information about remote write and read can be inferred and bookmarked by comparing local thread-id with the owner-id.

Once an atomicity violation is detected, as in AVIO-H, AVIO-S will either stop the program and raise an exception, or log all the debugging information and continue the execution. Debugging information, such as the address of the three involving instructions, P-instruction, I-instruction and the remote interleaving access, can be recorded in the global and local access tables.

4.3 Trade-offs between AVIO-H and AVIO-S

AVIO-H and AVIO-S each have their own advantage and disadvantages. First, AVIO-S is cheaper because it does not require any hardware extensions. Second, AVIO-S is also more accurate because: (1) AVIO-S's detection granularity is very flexible, ranging from a byte to a cache line with a word as default. Therefore, AVIO-S suffers much less from the false sharing problem than AVIO-H. (2) Since AVIO-S monitoring and detection are done by instrumented code, it is not affected by the cache displacement, load-store queues, context switches, or other hardware-related issues.

However, as a trade-off, AVIO-S incurs much higher overhead and run-time perturbation, which comes from two sources. The first is monitoring overhead–each global access is instrumented to update the access information of the accessed data. The second is detection overhead. At each I-instruction, AVIO-S needs to detect possible violations to the corresponding AI invariant. To reduce overhead, hashing is used for quick locating the information tables. Since the global owner-table can be accessed by all threads, spin-locks are used for fast synchronization. All these optimizations are helpful in reducing overhead. However, as we will show in the experimental results (section 6), even though AVIO-S' performance is better than several other software concurrency bug detectors, the overhead is still much higher (4 orders of magnitude) than AVIO-H. Even if static analysis may further optimize AVIO-S, it is still too hard to reach the level to fit for production run as AVIO-H. In addition, the bug detection capability may also be affected by the larger execution perturbation from AVIO-S.

5. Evaluation Methodology

Our software implementation, AVIO-S, is implemented using the PIN binary instrumentation tool [21] and runs on a *real machine* with four Intel Pentium processors. Our hardware implementation, AVIO-H, is implemented on the Simics [22] whole system simulator, based on the SimFlex timing model [15] because it can run a real OS on the top, allowing us to run real-world server programs in a realistic simulation environment. Specifically, we use a full system, cycle-accurate, x86 simulator that models a 4-core CMP in-order x86 machine. Non-memory operations have a fixed one cycle latency and memory operations go through the cache and memory hierarchy. The parameters of the architecture are shown in Table 2. With AVIO-H, we assume a 0.4% extra slow down on whole chip frequency due to the 0.4% larger L1 cache and 500 cycle penalty at each bug report to stall pipeline and prepare debugging information.

CPU	2.0 GHz in-order; 1 issue width each core
L1 cache (private)	32K, 4 way, 64B/line, 2 cycle latency
L2 cache (shared)	1M, 8 way, 64B/line, 10 cycle latency
Memory	200 cycles latency
Cache coherence protocol	Derived from Piranha [2] CMP cache coherence protocol

Table 2. **Simulation configuration**

Application	BugNo.	Bug description
Apache HTTP server (253K LOC)	#1	Unprotected buffer length read and write corrupt log file(Figure 4)
	#2	Unprotected reference counter write-read causes null pointer reference
MySQL DB server (688K LOC)	#1	Unprotected database bin log close and open cause some actions not logged (Figure 5)
	#2	Unprotected query-id set and read crashes database server
	#3	Unprotected 'delete table' query and logging causes database log disorder
Mozilla -extract[2]	#1	Unprotected script handler set and read causes null pointer reference (Figure 1)

Table 3. Evaluated applications and atomicity bugs.

Two sets of applications are used in our experiments. The first set is used to evaluate AVIO's bug detection capability. Unlike many previous hardware race detection studies [30, 31] that evaluated with manually injected bugs, we use six *real* atomicity violation bugs which were unintentionally introduced by the original programmers in two large real-world server applications (Apache and MySQL) and Mozilla[2]. Table 3 shows the buggy applications and the description of the six real bugs. For these applications, we evaluate whether the bug can be detected and how many false positives are reported during the bug manifestation run.

In the second set, we use several well known SPLASH-2 benchmarks to evaluate AVIO's overhead and false positive. SPLASH-2 has also been used in many previous works [29, 31, 32] to evaluate false positives because they have few concurrency bugs.

Besides comparing our two implementations AVIO-S and AVIO-H, we directly compare the false positives, negatives and overhead with an enhanced lockset algorithm implemented in Valgrind [25], which we will refer to as Val(grind)-Lockset algorithm. In addition, we also compare indirectly with the happens-before algorithm and the SVD algorithm [42] by analytically evaluating whether each bug can be detected by them based on our understanding of these two algorithms. In terms of false positive and overhead, we refer to previous papers: happens-before has similar level of overhead with lock-set algorithm; SVD reports an up to a 65X server application overhead and 1-60 static false positives for the same server applications, MySQL and Apache.

To demonstrate the less stringent requirement of AVIO on training runs, *we do not use same inputs for detection and training in our experiments*. To extract AI invariants, we examine multiple access interleavings during 100 training runs (or 100 server requests) for each application. The invariant filtering threshold T is set to 0. In addition, we also conduct sensitivity studies on the number of training runs for both server applications and SPLASH-2 benchmarks. The result shows that no more than 100 server requests or 5 training runs are enough to obtain reasonably accurate AI invariants for all the tested applications.

6. Experimental Result

6.1 Functional Results

(1) Bug detection capability AVIO detects more tested real bugs than the three alternatives (Val-Lockset, happens-before and SVD). Specifically, as shown in Table 4, AVIO can detect five out of the six tested bugs, while the three alternatives can detect only one or three.

MySQL bug3 requires atomicity among accesses to multiple global variables. These variables have no data or control dependency with each other, but must be consistent for semantic reasons. As a result, it is not detected by any evaluated tool. Besides this bug, the Lockset algorithm cannot detect the Mozilla-extract bug, because it is data-race free, as explained in Figure 1. For the same reason, the happens-before algorithm also fails to detect it. SVD cannot detect MySQL Bug1, because it is an atomicity violation involving a write-after-write access pair, with no true data dependency or control dependency within it. The pair are therefore not be put into one computation region and consequently not checked by SVD. Similarly, Apache Bug2, MySQL Bug2 and Mozilla-extract are atomicity violations with write-then-read access pairs, which are also not checked automatically by SVD.

In contrast, AVIO's bug detection capability is more comprehensive because, unlike race detectors, it does not rely on synchronization primitives; unlike SVD, it can detect atomicity violations with write-read and write-write dependencies based on our serializability analysis.

(2) False positives Table 5 shows that AVIO introduces only 1–11 static and 1–17 dynamic false positives on server applications, much fewer than the Lockset algorithm, which has on average 51.5 static and 118.5 dynamic false positives. Similarly, for the bug-free SPLASH-2 benchmarks, AVIO-S has no false positive and AVIO-H has only an average of 1.25 static and dynamic false positives, while Lockset has 8.25 static and 26313 dynamic false positives on average.

The lockset algorithm's high false positive rate is because, as discussed in Section 1, it incorrectly reports all shared accesses that are correctly synchronized using non-lock based methods, such as barriers, flag-synchronization, etc. Such cases account for 64% of the static false positives in the four SPLASH-2 benchmarks. In addition, lockset cannot differentiate benign races from real bugs. For the SPLASH-2 benchmarks, 21% of the static false positives are contributed by benign races. Even though we do not evaluate false positives with the happens-before algorithm, we expect that the results will be similar because the happens-before algorithm also relies on synchronization primitives to order execution segments and thereby suffers from the same problems such as benign races, etc. The large number of false positives in the previous algorithms requires much effort from programmers to sift through manually.

In contrast, AVIO reports many fewer false positives because it does not rely on any synchronization primitives. Instead, it bases its detection on access interleavings, which are more essential and fundamental to atomicity violation bugs. Correctly synchronized accesses are not reported as bugs no matter what synchronization methods are used because they do not violate AI invariants. Moreover, AVIO can easily differentiate benign races from true bugs because benign races do not have any AI invariants, i.e., these code segments actually welcome unserializable interleavings. Therefore, AVIO does not report bugs at these code points.

Application	Bug Detected				
	AVIO-H (Hardware)	AVIO-S (Software)	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*

Table 4. Bug detection results for various techniques, against buggy real applications. * Since the four bugs are not in the SVD paper, we evaluate them based on our understanding of the SVD algorithm. This is easy because these bugs involve either write-write or write-read dependencies, or multiple unrelated variables, and therefore cannot be detected by SVD at run time. The other two bugs are evaluated in the SVD work and our results agree with theirs.

[2] Since our instrumentation and simulation tools do not support Mozilla's graphic user interface, we use an extracted version of the real bug in Mozilla based on its nsXULDocument.cpp file.

Benchmark	Dynamic False Positive			Static False Positive		
	AVIO -H	AVIO -S	Lock-set	AVIO -H	AVIO -S	Lock-set
Apache #1	6	5	6	3	2	6
Apache #2	1	1	23	1	1	20
MySQL#1	4	4	107	4	4	7[illegible]
MySQL#2	17	6	338	11	6	10[illegible]
Average	7	4	118.5	4.75	3.25	51[illegible]
fft	1	0	4098	1	0	6
fmm	4	0	389	4	0	12
lu	0	0	65026	0	0	5
radix	0	0	35740	0	0	10
Average	1.25	0	26313	1.25	0	8.25

Table 5. False positives rates for server applications and bug-free SPLASH-2 benchmarks. We determine 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 and static false positives are static code segments incorrectly reported as bugs. Since Mozilla-extract is extracted from Mozilla by us, its false positive number is not objective and is therefore not reported here.

Bench- mark	Bug Detection Execution Slow Down		
	AVIO (Hardware)	AVIO (Software)	Valgrind- Lockset
fft	0.5%	42X	1217X
fmm	0.4%	19X	660X
lu	0.4%	23X	661X
radix	0.4%	15X	236X
Average	0.4%	25X	694X

Table 6. Overhead comparison of synchronization bug detection on SPLASH-2 benchmarks

AVIO still has a few false positives. For software AVIO, the false positives are due to insufficient training. Since server applications are very complicated, some correct interleavings do not occur during our short training (only 100 client requests). For hardware AVIO, apart from insufficient training, the reason for most false positives is false sharing at the cache line granularity. Unlike the lockset algorithm, AVIO never has huge numbers of dynamic false positives even when the static false positive rate is comparable to the Lock-set algorithm's, because most of AVIO's false positives occur due to rare interleavings or code paths.

(3) Comparison of AVIO-S and AVIO-H Functionality As AVIO-S has a much smaller granularity than that of AVIO-H, AVIO-S is more accurate. For example, it incurs up to 5 fewer static false positives than AVIO-H. But as shown in the next section, this improved accuracy is achieved with much higher overhead.

6.2 Overhead Results

AVIO has low detection overhead due to the hardware support and the simple detection algorithm we use. As shown from our experiments on SPLASH-2 benchmarks (table 6), AVIO-H imposes only 0.4-0.5% overhead, clearly feasible for production run use. Without hardware support, the software implementation AVIO-S imposes an average of 25 times slow down. Although this is acceptable for in house testing or postmortem analysis with deterministic replay support [24, 41], it is too high for production runs. We expect its overhead would still be substantially higher than AVIO-H even after aggressive static analysis optimizations.

Though much worse than AVIO-H, our software implementation AVIO-S still outperforms these previous software approaches. As shown in Table 6, the Valgrind-lockset imposes an average of 694 times slow down[3]. As the original SVD paper [42] reports, SVD imposes a factor of 65 times slow down to server applications. Such performance advantage is mainly due to the simplicity of our bug detection algorithm. In the future, static analysis can be used to further improve the performance of AVIO-S.

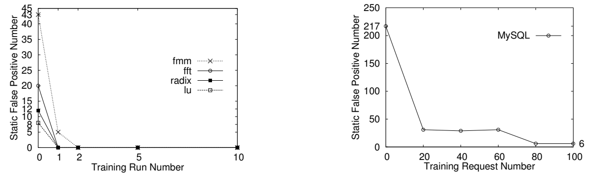
In summary, AVIO-S would be a good choice for off-line bug detection and diagnosis, while AVIO-H can be employed for production runs.

6.3 Training Sensitivity

Our sensitivity study results show that for most applications, a few runs are sufficient to get a set of reasonably accurate AI invariants, which are also robust to different inputs. Figure 10 shows the number of false positives reported when we use the invariants generated from a different number (1-10) of training runs for the SPLASH-2 benchmarks, or a different number (1-100) of requests for MySQL.

In all four SPLASH-2 benchmarks, the false positives drop to 0 when we use the invariants extracted from more than two training runs. Similarly, with MySQL, most false positives are eliminated after just 100 training requests. Such results indicate that AVIO's training requirements are not stringent.

As mentioned before, in all our experiments, the inputs used in detection runs are different from those in training runs. Therefore, our results also show that training with one input can be used to guide bug detection with other inputs. Of course, similar to other invariant-based approaches [14, 44] and general dynamic bug detectors, AVIO can only generate invariants from exercised code.



(a) SPLASH-2 Benchmarks **(b) MySQL 4.1.1-alpha**

Figure 10. Effect of training on static false positives for SPLASH-2 benchmarks and MySQL server. ((b) uses different x-axis scale–the number of requests)

7. Discussion: Limitations of AVIO

AVIO is definitely not a panacea. the current AVIO prototypes suffer from the following limitations and require more work in the future to enhance AVIO to address these problems.

(1) Bugs that are not exposed during the monitored run. Like many previous dynamic race and memory bug detectors, AVIO reports only those bugs that manifest in the monitored run and may miss potential bugs that do not happen during that run. It would be ideal if AVIO can *predict* non-exposed bugs like lockset algorithm. However, lockset algorithm focuses on data race and can not effectively detect atomicity violation. It also suffers from high false positive rates, less applicability to future transactional memory programs, etc. In the future, AVIO can use static and dynamic analysis to infer potential interleavings during detection, so that it can be less sensitive to scheduling. Advanced concurrency test generation techniques can also help AVIO to counter this problem in the future.

(2) Atomicity violations involving multiple variables. Like most previous tools, including Lockset, happens-before and SVD, AVIO focuses on single variable related bugs and cannot detect concurrency bugs that involve multiple shared variables, such as the MySQL bug3. Fortunately, real world concurrency bug characterization experience indicates that concurrency bugs involving single variables are more typical and can serve as the building blocks

[3] Part of the slow-down is from Valgrind's code emulation mechanism.

of multi-variable ones. In order to extend AVIO to detect multiple variable atomicity violation. In many cases, we can simply compose multi-address regions from several single-address atomicity regions. We just need to extend the detection protocol to look out when serialization of one variable's accesses conflicts with that of another variable. In more complicated and challenging cases when multi-addresses are correlated by high-level semantics, like the MySQL bug3, our AI-Invariance needs to be extended to consider multiple variable access interleaving invariance. How to systematically detect concurrency bugs involving multiple variables is still an open question left for our future work.

(3) Training Overhead. Training will also add some overhead to the whole AVIO bug detection process. Fortunately, the training results *can* be reused. Even when the code is changed, we can still save the training effort of unchanged part and only redo the training for those change-related or not well trained parts. For each training run, the overhead is similar to that of a detection run (reported in section 6). The training run number depends on the sufficiency requirements. As shown in section 6.3, as long as the related code region is covered, usually small number of training runs (less than 100 requests in our server applications) would provide sufficient interleaving samples.

8. Related Work

Due to space limit, we describe only closely related work:

Data race detection Previous work in data race detection can be grouped into the categories of dynamic and static. In dynamic race detection, happens-before [7, 26, 28] and lockset algorithms [6, 34, 38] have been thoroughly studied. Many extensions and combinations have been proposed to reduce false positives or overhead [27, 29, 43]. Static data race detection techniques include race type-safe systems [3, 10], static-versions of the lockset algorithm [8] and model checking techniques [17]. As discussed in Section 1, all data race detection techniques, while useful, still have several limitations, primarily because not all concurrency bugs are data races and not all data races are concurrency bugs. In contrast, the focus of our work is atomicity violation detection.

Atomicity Violation Detection Several static and dynamic techniques based on state reduction theory of right/left mover have been proposed [12, 33, 39] to detect atomicity violations. All of these methods require programmers to annotate all synchronization points, which is expensive; and also annotate code regions that need to be atomic, which is impractical–if programmers can properly do this, they can properly synchronize these areas. These problems are addressed to some extent by Atomizer [11], which gains synchronization knowledge through the lockset algorithm [34] and uses simple heuristics to identify atomic blocks. While an improvement, its synchronization knowledge is limited by the lockset algorithm. Also not requiring manual annotation, SVD [42] studies a subclass of atomicity problems based on computational region. In [4], *stale-value errors*, a subclass of atomicity violation, is studied.

Compared to previous atomicity violation work, our AVIO is more general: we indirectly infer atomic regions by observing AI invariants without knowledge of synchronization primitives. In addition, benefiting from our comprehensive serializability analysis, AVIO-S covers more atomicity violation cases. Moreover, since AVIO leverages the cache coherence protocol, our hardware implementation has negligible overhead, orders of magnitudes smaller than previous works on atomicity violation detection.

Bug Detection Our work is also related to invariant-based detection and hardware-support for bug detection. Invariant-based bug detection is a new and promising direction. Its feasibility and powerful bug detection capability is shown in previous work like DAIKON [9], DIDUCE [14], AccMon [44] and Liblit's work [20]. Our AVIO shares the strength of invariant-based bug detection with previous work. However, we are one of the first proposing invariants in access interleavings for concurrent programs. In addition, our work has less stringent requirement on training data generation since by simply running the same test case many times, we obtain many different access interleavings.

Recently several studies have exploited hardware support for bug detection, including memory related bug detectors [36, 44, 45], deterministic replay support [24, 41], etc. As mentioned in Section 1, previous works have also investigated hardware and coherence protocol supports for data race detection. Examples include [23, 28, 32], almost all of which implement the happens-before race detection algorithm by combining with cache coherence protocols in distributed shared memory systems. Most recently, ReEnact [31] employs advanced TLS architecture for race detection and CORD [30] uses scalar logical timestamps to improve the scalability. Since *all* these previous studies are based on happens-before, they all share the intrinsic and fundamental limitations of happens-before as described in Section 1.

Like previous techniques, AVIO also exploits hardware to reduce overhead to negligible level. Unlike them, AVIO uses a fundamentally different approach, AI invariants, to detect atomicity violation bugs instead of data races. It can therefore differentiate benign races, require no knowledge of synchronization mechanisms, and can apply to future transaction-based parallel programs.

New programming language and model for atomicity Recent research have conducted on new programming models such as transactional memory [1, 13, 16, 18] to replace the existing lock-based synchronization. These new programming language features allow programmers to explicitly specify code regions or data that need to be atomic [37]. All of these new approaches aims to eliminate error-prone explicit *locks* to make concurrent programming easier. However, although such approach can significantly reduce the number of concurrency bugs such as data races, they still cannot avoid atomicity violations. The reason is that programmers can make mistakes when dividing atomic regions. Our AVIO can help detect these bugs.

9. Conclusions

This paper has presented an innovative, invariant-based approach called AVIO to detect atomicity violations. By automatically extracting AI invariants and detecting violations of these invariants at run time, AVIO can detect a variety of atomicity violation bugs. We have implemented AVIO in two different ways, a software-only implementation (AVIO-S) and a hardware implementation (AVIO-H). AVIO is evaluated using two real-world server applications (Apache and MySQL) with five representative *real* bugs, one extracted-version of Mozilla with one real bug in it, and several SPLASH-2 benchmarks. Our results show that AVIO detects more bugs with much fewer false positives than previous algorithms. Comparing the two implementations of AVIO, AVIO-H incurs very little (0.4–0.5%) overhead while AVIO-S introduces fewer false positives. We are in the process of extending this work to address the limitations discussed in Section 7.

As the emerging transactional memory programming model becomes more attractive due to the multi-core technology trend, atomicity violation detection will become increasingly urgent because parallel programs written using such models will suffer more from atomicity violations instead of data races (explained in Section 1). To the best of our knowledge, our work provides one of the first practical, comprehensive, low-overhead approaches in detecting various atomicity violations, especially with a fundamentally different and novel idea and an evaluation using real bugs from real-world server programs. In addition, our hardware AVIO is also the *first* using hardware support to detect atomicity violations.

10. Acknowledgments

We thank the anonymous reviewers for useful feedback, the Opera groups for useful discussions and paper proofreading. We also thank Min Xu for sharing resources on server applications' concurrency bugs. This research is supported by NSF CNS-0347854 (career award), NSF CCR-0325603 grant, DOE DE-FG02-05ER25688, and Intel gift grant.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.

[2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA*, 2000.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.

[4] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. In *Compaq SRC Technical Note 2002-04*, 2002.

[5] S. Chandra and P. M. Chen. How fail-stop are faulty programs? In *FTCS*, 1998.

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

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP*, 1990.

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

[9] M. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[10] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI*, 2000.

[11] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[12] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[14] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[15] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4), 2004.

[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[17] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[22] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrom, and B. Werner. Simics/sun4m: A virtual workstation. In *Usenix Annual Technical Conference*, 1998.

[23] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *ASPLOS*, 1991.

[24] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[25] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *ENTCS*, 2003.

[26] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[27] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[28] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, 1996.

[29] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP*, 2003.

[30] M. Prvulovic. Cord:cost-effective (and nearly overhead-free) order-reordering and data race detection. In *HPCA*, 2006.

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

[32] B. Richards and J. R. Larus. Protocol-based data-race detection. In *SPDT*, 1998.

[33] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP*, 2005.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[35] SecurityFocus. Software bug contributed to blackout.

[36] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *MICRO*, 2004.

[37] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[38] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, 2001.

[39] L. Wang and S. D. Stoller. Static analysis for programs with non-blocking synchronization. In *PPoPP*, 2005.

[40] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

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

[42] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[43] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[44] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *MICRO*, 2004.

[45] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *ISCA*, 2004.