Detecting Concurrency Bugs from the Perspectives of Synchronization Intentions

Shan Lu, Soyeon Park, and Yuanyuan Zhou, *Member*, *IEEE*

Abstract—Concurrency bugs are among the most difficult to detect and diagnose of all software bugs. This paper combats concurrency bugs from the perspective of programmers' synchronization intentions. We first study the root causes of 74 real-world concurrency bugs to understand what types of synchronization assumptions are violated in real world. This study reveals two classes of synchronization intentions that are common, frequently violated, and understudied—single-variable atomicity intention and multivariable correlation intention. Following this study, two bug detection tools, AVIO and MUVI, are proposed to automatically infer these two types of synchronization intentions and detect related bugs. Specifically, AVIO automatically extracts *access interleaving invariants* and detects a variety of atomicity-violations during production runs. It can work both with and without special hardware support in our implementation. MUVI automatically infers multivariable correlations through static analysis and detects multivariable concurrency bugs. Our evaluation with real-world large multithreaded applications shows that AVIO can detect more atomicity-violation bugs with 15 times fewer false positives on average than previous solutions. Besides, AVIO-H incurs negligible (0.4-0.5 percent) overhead. MUVI successfully extracts 6,449 access correlations from Linux, Mozilla, MySQL, and PostgreSQL with high (83 percent) accuracy. Race detectors extended by MUVI can correctly identify the root causes of real-world multivariable concurrency bugs in our experiments. They also report four new multivariable concurrency bugs that have never been reported before.

Index Terms—Concurrency bugs, atomicity violation, bug detection.

1 Introduction

1.1 Motivations

Concurrency bugs are synchronization problems in multithreaded and multiprocess programs. They are extremely difficult to detect because they are nondeterministic. In the real world, most server and high-end critical software are multithreaded or multiprocessed. Concurrency bugs in this software have caused serious accidents, including a blackout that left tens of millions of people without electricity [36].

Recent hardware advances further worsen the concurrency bug problem. With multicore becoming mainstream, many more multithreaded applications are being developed to take advantage of the available processors and cores. As a result, the number of concurrency bugs will inevitably increase in the near future. Therefore, good techniques to automatically detect these bugs are greatly desired.

The state of the art in concurrency bug detection focuses on data races. A data race occurs when more than one thread access the same memory location without proper synchronization and at least one of them is a write. Three classes of tools have been proposed to detect data races. They are lockset race detection tools [6], [35], happens-before race detection tools [7], [29], [31], and hybrid tools combining the above two [30], [42]. The lockset algorithm reports a race when it finds no common lock protecting accesses to a shared memory location. The happens-before algorithm reports a race when two conflicting memory accesses do not have a strict happens-before relation. Pure-software implementation of the above detection algorithms often introduces slowdowns of 10X-100X. To address this performance problem, hardware extensions [26], [32], [33] are also proposed.

Although race detection has made great progress, it is far from sufficient to detect concurrency bugs because data races and concurrency bugs are fundamentally different.

Many data races are intentionally introduced by developers for better performance [35], [42], and should not be treated as software bugs introduced accidentally. More importantly, many concurrency bugs are not data races. Concurrency bugs occur when programmers' synchronization intentions are not enforced. Simply protecting all accesses with locks and forcing strict happens-before orders among them cannot guarantee the synchronization intentions, as we will see in many real-world examples in this paper.

Overall, data races are not the root cause of concurrency bugs. The root cause of concurrency bugs is not different from other types of software bugs—mismatches between the code implementation and the programmers' intentions. In order to improve the state of the art and effectively detect concurrency bugs, we need to understand what are the common root causes of real-world concurrency bugs. That is, what synchronization intentions are violated in typical real-world concurrency bugs.

Our study of real-world concurrency bugs (in Section 2) reveals two types of programmers' intentions that are most frequently violated.

Intention 1: **atomicity**. Atomicity, also called as serializability, is a property for a sequence of instructions S in multithreaded software. If the concurrent execution of S

- *S. Lu is with the Computer Science Department, University of Wisconsin - Madison, 1210 W. Dayton Street, Madison, WI 53706. E-mail: shanlu@cs.wisc.edu.*
- *S. Park and Y. Zhou are with the Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, M/C 0404, La Jolla, CA 92093. E-mail: {soyeon, yyzhou}@cs.ucsd.edu.*

Manuscript received 13 Apr. 2011; revised 10 Aug. 2011; accepted 28 Aug. 2011; published online 30 Sept. 2011.
Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-04-0225.
Digital Object Identifier no. 10.1109/TPDS.2011.254.

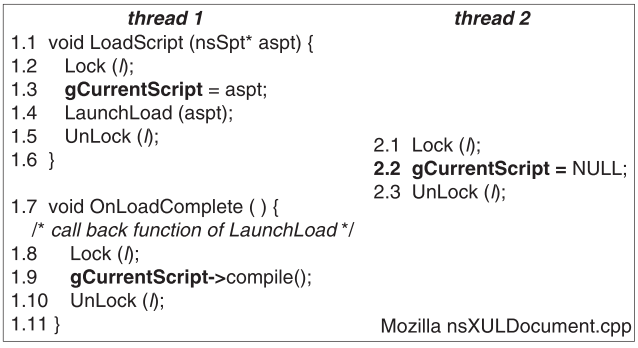


Fig. 1. Race-free does not guarantee correct synchronization. This is a real bug in the Mozilla Application Suite. Although there is no race, *thread 2* can still violate the atomicity of *thread 1*'s accesses to `gCurrentScript` and make the program crash.

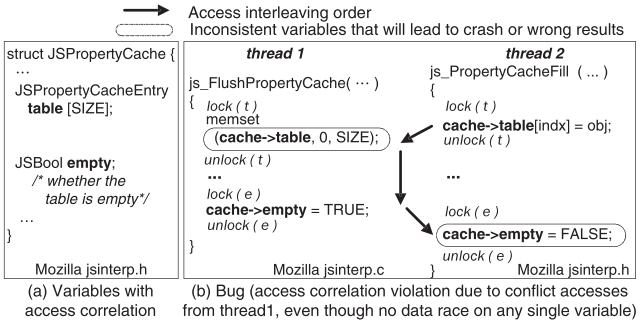


Fig. 2. A multivariable access correlation and the related concurrency bug in Mozilla. Data-race detectors or single-variable atomicity violation detectors cannot correctly detect this bug.

produces equivalent execution effects with a serial execution of S, the atomicity of S is preserved. Serial execution is that the execution of multiple threads does not overlap in time but completes consecutively, one after the other.

Atomicity violation bugs are introduced when programmers expect some code regions to always behave atomically without enforcing the atomicity in their implementation. Fig. 1 shows a real-world atomicity violation bug from the Mozilla Application Suite. The *thread 1* stores a pointer into the shared script-handler `gCurrentScript`, and later uses the pointer to compile the script. The two operations on `gCurrentScript` are expected to be atomic, but *thread 2* may nullify it in the middle and causes the program to crash.

The above atomicity violation bug is *race free* , because every access to `gCurrentScript` is protected by lock *l*. In general, race-free does not guarantee proper atomicity.

Intention 2: **variable correlation.** Some variables have semantic correlations among each other. For example, one variable may represent a constraint or the status of another variable, such as `empty` representing the status of `table` being empty in Fig. 2. We will refer to this type of relationship as *variable access correlation*, short as *variable correlation*. Correlated variables need to be accessed in a consistent manner, such as being updated (or read) together.

Multivariable concurrency bugs occur when concurrent accesses to correlated variables are not synchronized. Fig. 2 shows an example of this type of bugs. In Mozilla, `cache` → `table` is an array and cache → `empty` indicates whether the array is empty or not. In this example, *thread 1* uses function `js_FlushPropertyCache` to nullify the whole `table` array and set the `empty` flag to true. Unfortunately, these two actions can be interleaved by a function `js_PropertyCacheFill` from another thread. As a result, `empty` may be false when `table` has already been cleaned, causing invalid accesses to `table` and Mozilla to crash.

The above bug is clearly *race free*. Even if the two accesses to `cache` → `table` are synchronized and the two accesses to `cache` → `empty` are also synchronized, the bug still exists, as shown in Fig. 2. This type of multivariable concurrency bug cannot be correctly identified by previous concurrency bug detection tools that separately look at the accesses to individual variable [8], [35], [29], [6].

Violations to the above two types of intentions are very common among concurrency bugs. They contribute to about 70 and 30 percent of real-world nondeadlock bugs, respectively, in our empirical study (Section 2). Tools to detect these two types of bugs are desirable.

The above two intentions are orthogonal to each other. Violations to variable correlations can lead to multivariable atomicity violations and also many other types of bugs, as we will see in Section 8 and the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.254.

1.2 Challenges

A fundamental challenge in detecting concurrency bugs, such as atomicity-violation bugs and multivariable concurrency bugs, is to infer programmers' synchronization intentions, e.g., which code regions need to be atomic and which variables are correlated. Without knowing these intentions, bug detection can only blindly check the synchronization around all code regions and among all memory accesses, which is not only extremely slow but also unacceptably inaccurate.

Unfortunately, identifying synchronization intentions of a program is difficult.

State-of-the-art atomicity violation bug detectors [12] rely on programmers' *annotations* to specify atomic regions. SVD [39] uses data/control dependency to infer atomic regions. Although inspiring, SVD only covers a limited subset of atomicity violation bugs and incurs huge runtime overhead.

State-of-the-art multivariable concurrency bug detection is even poorer. The only piece of previous work (to the best of our knowledge) [3] on this topic used a lock-based heuristic: if two variables, `x` and `y`, are ever accessed within one lock critical-section, they should never be separately accessed in different critical sections throughout the program. This heuristic did not work well in practice: all the bugs reported in their experiments turned out to be false positives. The reason is that many variables might coincidentally appear in one critical section that is set up for other nearby accesses.

In summary, we need new techniques to infer programmers' synchronization intentions and detect concurrency bugs.

1.3 Contributions

This paper first studies what are common synchronization intentions behind real-world concurrency bugs. Based on this study, it proposes novel techniques to automatically infer common synchronization intentions and detect concurrency bugs. Specifically, it makes the following contributions.

1. Conducting a study of the characteristics of real-world concurrency bugs. This study reveals common patterns of concurrency bugs and typical synchronization intentions, providing guidance to concurrency bug detection.

Our study of 74 real-world concurrency bugs collected from four representative open-source applications shows that atomicity violation is the most common root cause of concurrency bugs. In addition, about one third of the studied concurrency bugs involve multiple variables.

2. Designing two simple program invariants to capture common synchronization intentions.

The first is called Access-Interleaving invariant, short for AI-invariant (Section 3). It captures the most common type of atomicity intentions: two consecutive accesses from one thread to the same shared variable are not interleaved by unserializable accesses from another thread.

The second is variable-access correlation (Section 6). It captures the access consistency relationship among variables.

3. Automatically extracting likely invariants to infer synchronization intentions. We designed AVIO that automatically infers likely AI-invariants based on information collected during training runs (Section 4). We also designed MUVI that infers variable correlation based on source code analysis and data mining (Section 7).

4. Implementing two new bug detection tools, AVIO and MUVI, that can effectively detect atomicity-violation bugs and multivariable concurrency bugs, respectively.

AVIO monitors memory accesses at runtime. It automatically detects violations to AI-invariants and reports atomicity violation bugs (Section 4). We built **two** implementations of AVIO: AVIO-H in hardware and AVIO-S in software.

MUVI extends two classic race detection methods with variable-access correlation to detect multivariable concurrency bugs (Section 8).

5. Evaluating AVIO and MUVI with real-world applications and detecting previously unknown real-world bugs.

We evaluated AVIO with six real-world atomicity violation bugs in big server applications. AVIO could detect these bugs more accurately and more efficiently than existing concurrency bug detection tools. *AVIO is under technology transfer to Intel*.

We evaluated MUVI with five real-world multivariable concurrency bugs. MUVI correctly identified the root causes of four bugs, none of which could be identified by existing techniques. MUVI also found four previously unknown multivariable concurrency bugs in Mozilla.

Since the bugs used to evaluate AVIO and MUVI come from different software or software versions, we discuss their evaluation results separately in Sections 5 and 9.

2 Synchronization Intentions

This section studies real-world concurrency bugs to answer which type of synchronization intention is violated and how many variables are involved in each concurrency bug.

2.1 Methodology of Bug Characteristics Study

Our study looks at four mature and representative open source applications: MySQL database server, Apache HTTPD web server, Mozilla browser suite, and OpenOffice editor suite.

We *randomly* collect 74 nondeadlock concurrency bugs from the bug databases of these applications: 13 from Apache, 41 from Mozilla, 14 from MySQL, and six from OpenOffice. We manually check the root causes of these 74 bugs based on developers' discussion, source code, and final patches.

TABLE 1
Root Causes of Nondeadlock Concurrency Bugs and the Number of Variables Involved in These Bugs

Application	Total	Atom.	Order	Other	>1 var	1 var
MySQL	14	12	1	1	6	8
Apache	13	7	6	0	4	9
Mozilla	41	29	15	0	15	26
OpenOffice	6	3	2	1	0	6
Overall	74	51*	24*	2	25	49

*: *there are three bugs that are both atomicity violation and order violation bugs.*

More details about our methodology and the threats-to-validity discussion can be found in the supplemental material, available online.

2.2 Characteristics and Implications

Finding 1. Atomicity violation is the most common (51 out of 74) root cause of the examined nondeadlock concurrency bugs, as shown in Table 1. This implies that concurrency bug detection should focus on atomicity violation bugs.

Our study finds that concurrency bugs mostly happen when programmers assume a code segment to be atomic but forget to or fail to correctly implement the atomicity. This finding is consistent with the fact that most programmers are used to sequential thinking and frequently assume atomicity.

We also find other types of concurrency bugs. For example, order violation bugs occur when the intended order between two operations (e.g., `A` is expected to execute before `B`) is flipped. They are less common than atomicity bugs.

Finding 2. A significant portion (34 percent) of nondeadlock concurrency bugs involve more than one variable (Table 1).

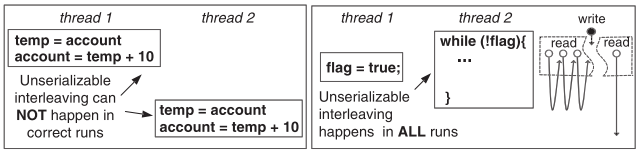
Our study defines the *number of variables involved in concurrency bugs* based on the set of memory accesses whose execution order can determine the manifestation of a concurrency bug (e.g., {1.3, 2.2, 1.9} in Fig. 1). Multiple variable concurrency bugs occur when unsynchronized accesses to correlated variables cause inconsistent program state.

Note that many previous concurrency bug detectors, including all race detectors and some atomicity violation detectors [39], assume that only one shared variable is involved in a bug. Although this simplification provides a good starting point for concurrency bug detection, the above finding confirms the importance of designing *new* concurrency bug detection tools to address multiple variable concurrency bugs.

In summary, atomicity and variable correlation are both important programmer intentions. Violations to them have caused many real-world concurrency bugs. In the following, we will discuss how to infer single-variable atomicity intentions and multivariable correlation intentions, as well as how to detect related concurrency bugs.

3 Programmers' Atomicity Intention

Terminology definitions. *Interleaving*, in general, means an execution order among accesses from multiple threads. Our discussion about interleavings in this paper assumes the



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

(b) While-flag synchronization example. The read access series SHOULD be unserializably interleaved.

Fig. 3. (a) An example with AI invariant; (b) An example without the AI invariant.

sequential consistency memory model. For simplicity, in our discussion of AVIO, unless otherwise mentioned, all accesses are to the same shared memory location. We refer to the thread whose atomicity is violated as the *local thread* and its accesses as *local accesses* (note that this does NOT mean a local variable). We refer to the atomicity violating thread as the *remote thread* and the violating accesses as *remote accesses*.

A *serializable* interleaving is an execution order among multithreaded accesses that produces equivalent execution effects with a serial execution of these accesses. Here, a serial execution is that the execution of different threads does not overlap in time and is completed one after the other. The execution effects include any changes on program states (e.g., registers and memory) due to these accesses.

3.1 Access-Interleaving Invariants

Programmers' atomicity intention comes in different formats. Based on our characteristics study, the most common and fundamental one can be represented by a type of invariants that we refer to as an **Access-Interleaving invariant** (AI invariant). This invariant is held by an instruction if the execution of this instruction and its immediately preceding local access to the same memory location are always atomic. We denote this instruction as the **I-instruction** (invariant instruction) and the preceding access instruction as the **P-instruction** (preceding instruction).

Fig. 3a shows an example of instructions holding AI invariants. In this code, programmers assume that the read and modification of *account* are never unserializably interleaved by conflicting remote accesses. The concurrent execution effect, including the values of *temp* and *account*, are supposed to be the same with one of the serial execution. An atomicity violation here will result in program misbehavior.

Fig. 3b shows an example of instructions not holding AI invariants. In this code, programmers assume that unserializable interleavings will occur to the while loop in *thread 2*. Otherwise, *thread 2* will hang.

Automatically differentiating code that is expected to have AI invariants from other code will allow us to avoid many false positives in concurrency bug detection.

Of course, the 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 extended from AI invariants. We will discuss them in Section 6.

3.2 Serializability Analysis

In order to differentiate unserializable interleavings from serializable interleavings, we studied all possible scenarios when two local accesses to the same shared variable are interleaved by a remote access. As demonstrated in Table 2, some interleavings are serializable because their execution effects are equivalent with some serial execution sequences. Some interleavings are unserializable and may lead to bugs, which is also explained in Table 2.

TABLE 2
Eight Cases of Access Interleavings to the Same Variable

Interleaving	Case No.	Serializability	Equivalent serial acc.	Problems (for unserializable cases)
read^p $read_r$ read^i	0	serializable	read^p read^i $read_r$	N/A
write^p $read_r$ read^i	1	serializable	write^p read^i $read_r$	N/A
read^p $write_r$ read^i	2	*unserializable*	N/A	The interleaving write gives the two reads different views of the same memory location (Fig. 4(a)).
write^p $write_r$ read^i	3	*unserializable*	N/A	The local read does not get the local result it expects (Fig. 1).
read^p $read_r$ write^i	4	serializable	$read_r$ read^p write^i	N/A
write^p $read_r$ write^i	5	*unserializable*	N/A	Intermediate result that is assumed to be invisible to other threads gets exposed (Fig. 4(b)).
read^p $write_r$ write^i	6	*unserializable*	N/A	The local write relies on a value that is returned by the preceding read and becomes stale due to the remote write (Fig. 3(a)).
write^p $write_r$ write^i	7	serializable	$write_r$ write^p write^i	N/A

Subscript r: remote interleaving access; i and p: one access and its preceding access from the same thread.

We further extend the table to consider *multiple* remote accesses and get the following four cases of unserializable interleavings (remote accesses are in parentheses; $*$ denotes zero or multiple interleaving accesses):

- 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 read result stale.

4 Inferring AI Invariants and Detecting Violations

4.1 Overview of Inferring AI Invariants

It is challenging to know which code regions can be adversely affected by unserializable interleavings and should hold AI invariants. Obviously, we cannot expect programmers to provide this information because bugs usually occur when programmers are not consciously aware of their assumptions.

The best way to automatically learn a programmer's intention is to study the program's behavior in correct execution. If we have never (or rarely) observed a code segment to be unserializably interleaved in correct runs, we can guess that developers expect this code region to always be atomic (e.g., the code of *thread 1* in Fig. 3a). If we have observed that a code segment is frequently unserializably interleaved in correct runs, we can guess that developers do not expect this code region to be atomic (e.g., the code of *thread 2* in Fig. 3b).

The feasibility of the above idea depends on the quality of training. Specifically, a good training process has to include a sufficient number of different training runs (sufficiency issue). In addition, a good training process has to be dominated by *correct* training runs (correctness issue).

Fortunately, two unique and "notorious" properties of concurrency bugs make training in AVIO easier than general invariant training [11], [16], [43].

The **correctness** issue can be addressed by several facts. First of all, Concurrency bugs manifest very infrequently, even with bug-exposing inputs. Their manifestation usually requires rare access interleavings, a notorious feature that makes concurrency bugs very hard to reproduce for postmortem diagnosis. As a result, we can easily get correct-dominated training. In addition, Existing software testing infrastructures can be leveraged to label training runs as correct or incorrect. Finally, the AI extraction algorithm can be designed to tolerate a small percentage of unfiltered incorrect training runs.

The **sufficiency** issue can be addressed by the natural **nondeterminism** of concurrent execution. Even with just one input, we can easily get a large number of distinct access interleavings by running a concurrent program on a multicore machine. For example, in our experiments, 100 runs of a SPLASH-2 benchmark with just one input always generate 100 distinct traces, each of which includes different access interleavings on shared variables.

Overall, the nondeterminism of concurrent programs provides AVIO with unique advantages: just running the program with the bug triggering input many times will provide good-quality training, which is much easier than traditional invariant-based tools.

4.2 AVIO Algorithms

AVIO automatically extracts AI invariants from offline training runs, and then detects potential violations of the extracted AI invariants during monitored runs.

Detection Algorithm. Detecting violations of AI invariants is straightforward based on our serializability analysis in Section 3.2. The process can follow the binary decision diagram in Fig. 5, which summarizes all the four unserializable interleaving cases.

Inference Algorithm. The goal of AI invariant extraction (AVIO-IE) is to extract AI invariants from multiple correct runs. Its results will be used by the above detection algorithm.

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 AVIO detection enabled. Initially the set of AI invariants, $AISet$, includes all global memory accesses in the target program. Then, the program runs 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 should be removed from $AISet$. This process will repeat many times until $AISet$ remains unchanged for the last m runs, where m is adjustable. Finally, we filter out never-executed instructions and return $AISet$.

4.3 Two AVIO Implementations

The key part of an AVIO implementation is to collect sufficient information to carry out the detection procedure depicted in Fig. 5. In the following, we discuss a hardware-assisted implementation (AVIO-H) and a software-only implementation (AVIO-S) of AVIO.

4.3.1 Hardware AVIO (AVIO-H)

The hardware implementation, AVIO-H, takes advantage of existing cache coherence protocol and achieves negligible overhead with simple hardware extensions. 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.

AVIO-H appends two bits onto each L1 cache line. These two bits, together with the existing invalidate (INV) bit used by the cache coherence protocol, provide sufficient information for the AVIO detection algorithm described in Section 4.2.

Preceding-access Instruction bit (*PI bit)* indicates the type of the preceding access. It is set to 1 at each local read to the corresponding cache line and unset at each local write.

DownGrade bit (*DG bit)* indicates whether a remote thread has read a value written by the preceding write access. It is unset at each local access and is set at every *Downgrade* request generated by invalidation-based cache-coherence protocols.

INValidate bit (INV *bit)* indicates whether an interleaving remote write has happened since the last local memory access. This bit exists in an invalidation-based cache coherence hardware. After a CPU core issues a memory write, the cache coherence protocol will set INV bits of other cores' corresponding L1 cache lines to invalidate old copies of this memory location. Therefore, AVIO-H simply checks the existing INV bit to see whether a remote write has happened.

Since an unserializable interleaving only happens when the original cache coherence protocol needs to contact the L2 cache to get the most up-to-date copy and/or exclusive write permission, AVIO-H's detection process is also triggered only at such moments. This helps AVIO-H achieve small space overhead and negligible time overhead.

More details of the AVIO-H design can be found in [22].

4.3.2 Software AVIO (AVIO-S)

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 them to check for possible

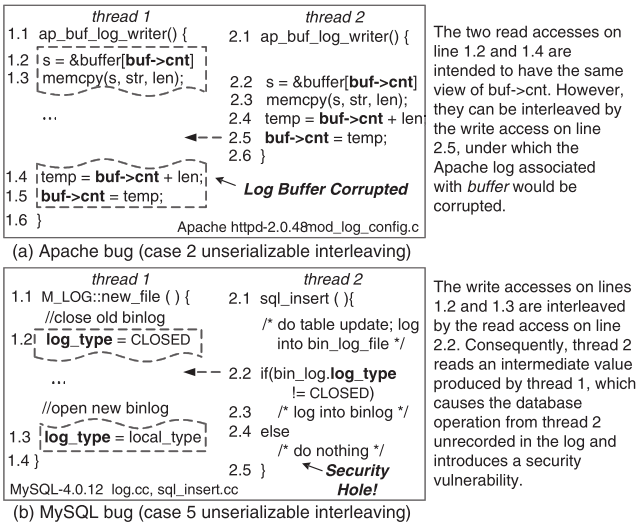


Fig. 4. Real bugs from Apache and MySQL caused by unserializable interleavings.

violations at I-instructions. In software, access information is collected by binary instrumentation at every global memory access and maintained in an access-table data structure. 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 recently 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 writes and reads can be inferred and bookmarked by comparing the local thread-id with the owner-id.

Comparing AVIO-S with AVIO-H. AVIO-S does not require any hardware extensions. AVIO-S is also more accurate because it does not suffer from the cache displacement, context switch, cache-line false sharing, and other problems that bother AVIO-H. However, AVIO-S inevitably incurs much higher overhead and runtime perturbation than AVIO-H. Its bug detection capability may also be affected by the large execution perturbation.

5 Experimental Results of AVIO

5.1 Methodology of AVIO Evaluation

Our software implementation, AVIO-S, is implemented using the PIN binary instrumentation tool [24] and runs on a real machine with four Intel Pentium processors. Our hardware implementation, AVIO-H, is implemented on the Simics whole system simulator, based on the SimFlex timing model [17].

We use six *real-world* atomicity violation bugs from two large open-source server applications (Apache and MySQL) and Mozilla[1] to evaluate AVIO's bug detection capability. These include the bugs described in Figs. 1 and 4. We also use SPLASH-2 benchmarks to evaluate AVIO's performance and false positives.

1. Since our hardware simulator does not support Mozilla's graphic user interface, we use an extracted version of the real bug in Mozilla.

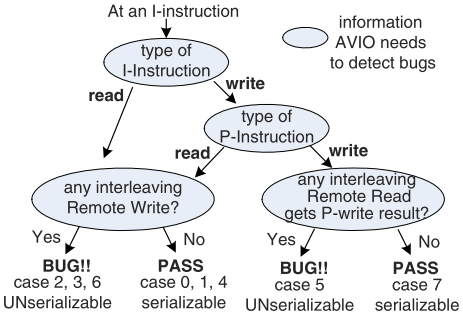


Fig. 5. AVIO bug detection. Cases 0-7 are explained in Table 2.

We compare AVIO with an enhanced lockset algorithm implemented in Valgrind [28] (referred to as Val-Lockset). We also analytically evaluated whether happens-before and the SVD algorithm [39] can detect each evaluated bug based on our understanding of the algorithms.

To demonstrate the less stringent requirement of AVIO on training runs, *we do not use same inputs for detection and training in our experiments*. We use 100 training runs (or 100 server requests) for each application.

5.2 Functional Results of AVIO

Bug detection capability As shown in Table 3, AVIO detects more bugs in our experiments than existing bug-detection tools (Val-Lockset, happens-before, and SVD).

MySQL#3 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 Fig. 1. For the same reason, the happens-before algorithm also fails to detect it. SVD misses MySQL#1, because it cannot infer the atomicity intention between a pair of write-after-write accesses with no data/control dependency. Similarly, it misses Apache#2, MySQL#2, and Mozilla-extract.

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.

False positives As shown in Table 4, AVIO introduces only from 1 to 11 static and from 1 to 17 dynamic false positives on server applications. In comparison, Val-Lockset introduces from 6 to 101 static and from 6 to 338 dynamic false positives. Similarly, for the bug-free SPLASH-2, AVIO-S has no false positive and AVIO-H has only 1.25 false positives on average, while Val-Lockset has many more.

Most false positives reported by Val-LS are memory accesses that are protected by nonlock synchronizations, such as barriers and spin-loops. The happens-before algorithm would have similar results since it uses similar knowledge of synchronization primitives to order execution segments.

In contrast, AVIO reports 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,

TABLE 3
AVIO Bug Detection Results

Application	Bug Detected				
	AVIO-H	AVIO-S	Val-LS	H.B.	SVD
Apache #1	Yes	Yes	Yes	Yes	Yes
Apache #2	Yes	Yes	No	No	No
MySQL#1	Yes	Yes	Yes	Yes	No
MySQL#2	Yes	Yes	Yes	Yes	No
MySQL#3	No	No	No	No	No
Mozilla-ext	Yes	Yes	No	No	No

H.B.: happens-before.

no matter what synchronization methods are used, because they do not violate AI invariants. Moreover, AVIO can easily differentiate benign atomicity violations from true bugs because benign violations do not have any AI invariants (i.e., these code segments actually welcome unserializable interleavings).

The false positives of AVIO-S are due to insufficient training. AVIO-H has more false positives than AVIO-S, because AVIO-H uses a cache-line, rather than a word, as its memory-access monitoring granularity. The coarser granularity leads to false sharing problems.

Performance AVIO-H has less than 1 percent overhead. The performance of AVIO-S is comparable with or slightly better than Val-Lockset and SVD. AVIO-S is suitable for offline bug detection and diagnosis, while AVIO-H can be employed for production runs. More detailed performance and training sensitivity results are presented in the supplemental material, available online.

6 Variable Correlation Intention

Although powerful, AVIO has a limitation. Like many previous bug detectors, AVIO focuses on synchronization among accesses to a single variable and cannot detect concurrency bugs that involve multiple shared variables (e.g., Fig. 2).

Multivariable concurrency bugs widely exist in real world, as discussed in Section 2. They are caused by violations to variable correlations intended by programmers. In the following sections, we will deepen our understanding of variable-correlation intentions and develop tools to detect multivariable concurrency bugs.

TABLE 4
False Positive Results

Benchmark	Dynamic False Positive			Static False Positive		
	AVIO-H	AVIO-S	Val-LS	AVIO-H	AVIO-S	Val-LS
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	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

It shows the number of dynamic instances of false positives and static code segments incorrectly reported as bugs. Since Mozilla-ext is extracted from Mozilla by us, its false positive number is not objective and not reported here.

TABLE 5
Real-World Examples with and without Access Correlation (the Numbers of Times that Variables Are Accessed Separately in Different Functions Are Shown in the Parentheses)

	ID	source	Variable definitions	# of functions they are together (not)

Variables **With** access correlation	a	Linux net-device.h	struct net_device_stats {
			u64 rx_bytes; /* #of received bytes */
			u64 rx_packets; /* #of received packets*/
			}
				49 (1)

	b	PgSQL time.h	struct tm { int tm_sec; /* second */ int tm_min; /* minute */ } /* time */	25 (0)
	c	Linux fb.h	struct fb_var_screeninfo { u32 red_msb; /* red */ u32 blue_msb; /* blue */ u32 green_msb; /*green*/ u32 transp_msb; /*transparency*/ } /* for color display */	11 (1)

	d	Linux libiscsi.h	struct iscsi_session {
			spinlock_t lock; /* lock */
			int state; /* critical data */
			}
				20 (0)

	e	Linux list.h	struct hlist_node { struct hlist_node *next; /* next */ struct hlist_node **pprev; /* pevious */ } /* linked list */	32 (0)
	f	MySQL mysql-test.c	struct st_test_file* cur_file; struct st_test_file* file_stack; /* cur_file points to the top of stack */	69 (0)

Variables **WithOUT** access correlation	g	Linux net-device.h	struct net_device_stats {
			u64 rx_bytes; /* #of received bytes */
			u64 tx_aborted_erros; /* #of transfer aborts*/
			}
				4 (68)

	h	MySQL sql_class.h	Class THD {
			NET net; /* client connection descriptor */
			uint db_length; /*length of database name*/
			}
				3 (87)

Correlation, originating from statistics, means a pair of items' departure from independence. Many items in real-world programs are not independent from each other, nor are program variables in software. Variable access correlation is inherent in program semantics. The following shows the typical semantic reasons of variable access correlations.

1. *Constraint specification.* A variable specifies a constraint, a property or a state of its correlated peer. For example, in Fig. 2a, `cache->empty` describes the state of the `cache->table`.
2. *Different representation.* Two variables represent the same information in different ways. The `rx_bytes` and `rx_packets` shown in the row (a) of Table 5 represent such an example. They are accessed together[2] in 49 functions except for one, which is a new bug detected by MUVI and confirmed by the Linux developers.
3. *Different aspects.* The correlated variables specify different aspects of complex data to emulate correlated real-world entities. The row (c) of Table 5 shows four correlated fields representing the red, blue, green, and transparency information in a Linux color display. They are accessed together in 11 functions with only one exception, which is also a confirmed as a new bug detected by MUVI. The `tm_min` and `tm_sec` in the row (b) of Table 5 are another example.

2. More formal discussion of *together*ness is in Section 7.1.

TABLE 6
Four Examples of Access Constraints in Correlations

Constraint	Definition	Example
read(x) ⇒ read(y)	Every read to x semantically requires a read to y	Fig. 2: read to `cache->table` has to be preceded by checking `cache->empty`
write(x) ⇒ write(y)	Every write to x semantically requires a write to y	Row (a) in Table 5: two statistic variables `rx_bytes` and `rx_packets` are always updated together.
write(x) ⇒ AnyAcc(y)	Every write to x semantically requires an access to y	Row (d) in Table 5: write to `state` has to check or grab the lock `lock`
AnyAcc(x) ⇒ AnyAcc(y)	Every access to x semantically requires an access to y	Row (c) in Table 5: accesses to `red, blue, green, transp` fields are always together.

`AnyAcc` means read or write. There are totally nine types of access constraints.

4. *Implementation demand.* The correlated variables cooperate with each other in order to implement a specific functionality. The row (d) of Table 5 shows that the field `lock` is used to protect the critical data `state` in structure `iscsi_session`. Therefore, accesses to `state` always occur together with accesses to `lock`. Similar examples are shown in row (e) and row (f) of Table 5.

   Obviously, not all of the variable pairs are access-correlated. In the rows (g-h) of Table 5, although the fields in each pair belong to the same structure, they do not have access correlation. They are accessed together in three or four functions and are accessed separately in 68 or 87 functions.

Since correlated variables are connected semantically, violating an access correlation poses the risk of breaking the semantic connections and consistency, and might threaten the program correctness, as demonstrated in Fig. 2.

Access constraints. Access constraints have many different formats. As shown in Table 6, correlated variables are sometimes always accessed (either read or write or both) together; sometimes reading a correlated variables is preceded by checking (reading) the other correlated variable; and in other cases, writing one variable requires checking (reading) *or* writing its correlated variables.

In addition, constraints could be either symmetric or asymmetric between two variables: modifying one may require modifying or checking the others accordingly, but the other way around is not necessarily true. For example, in the row (d) of Table 5, updating `state` requires accessing `lock`, but accessing `lock` does not require modifying `state`.

In general, there are nine different types of access constraints for two correlated variables (four of which are illustrated in Table 6). For simplicity of description, for two variables x and y, we use the following notation to formally represent an access correlation:

$$A1(x) \Rightarrow A2(y),$$

where $A1$ and $A2$ can be any of "read," "write," or "AnyAcc" (either read or write). For example, an access correlation `write`(x) ⇒ `read` (y) means that every time x is modified, the program needs to read the value of y.

7 Inferring Variable Correlations

7.1 Overview of Inferring Correlations

MUVI statistically infers access correlations by checking which variables are usually read or written together. For example, if variable y is read or written whenever variable x is updated, it is very likely that an underlying program semantic connects them (i.e., access correlation). The underlying assumption is that the target program is reasonably mature. Almost all open source and commercial software meet this requirement [10], [19], [20], [11], [16].

Definition of "Access Together." A nontrivial question that immediately emerges is how we claim that two accesses are "together." There could be many possible measures, and we considered the following issues: 1) dynamic execution distance is not a good measure because two correlated accesses can easily be separated by a loop or a function and get a large dynamic distance; 2) simply counting the number of source code lines between accesses is also bad. It will consider two accesses from two adjacent functions as "together," which is unreasonable. Based on these considerations, MUVI defines "access together" as: if two accesses (reads or writes) appear *in the same function*-with less than *MaxDistance* statements apart, these two accesses are considered*together*, where *MaxDistance* is an adjustable threshold.

Definition of "Access Correlation." x has access correlation with y (i.e., $\mathrm{A1}(x) \Rightarrow \mathrm{A2}(y)$), iff $\mathrm{A1}(x)$ and $\mathrm{A2}(y)$ appear together at least *MinSupport* times and whenever $\mathrm{A1}(x)$ appears, $\mathrm{A2}(y)$ appears together with at least *MinConfidence* probability. Here, *MinSupport* and *MinConfidence* are tunable parameters, and A1 and A2 can be read, write or AnyAcc.

Correlation inference. MUVI's correlation analysis is then to find out all $\mathrm{A1}(x) \Rightarrow \mathrm{A2}(y)$ from the target program. MUVI conducts this analysis in three steps. At the Step 1, MUVI collects access information of program variables. At the Step 2, MUVI uses a data mining technique to efficiently identify variables that are accessed together in many functions. At the Step 3, MUVI considers the access type information and uses statistical metrics to produce a ranked list of variable access correlations. Each MUVI step is discussed in the following three sections one by one.

7.2 Step 1: Access Information Collection

As the first step, MUVI conducts flow-insensitive and interprocedural static analysis to collect variable access information (`Acc_Set`) from every function. To conduct this step, MUVI needs to address several issues as follows:

1. *Which variables are we interested in?* Correlations involving structure/class fields and global variables are usually more common and more important than those short-lived correlations involving only scalar local variables. Therefore, MUVI considers two types of variables: global variables and structure fields (regardless of which object instance the field is associated with), represented by structure/class-`name::field-name` (e.g., `JSPropertyCache::empty`).

2. *What detailed access information do we need?* For each variable access, we need its access type (read or write) and its position in the source code (file name and line number).
3. *How to handle function calls?* A function can access a variable directly (referred to as a *direct access*) or via its callees (referred to as an *indirect access*). The `Acc_Set` of a function should include not only direct but also indirect accesses. Otherwise, some access correlations would be missed, especially in cases when a variable is read or written inside some utility functions, such as `get(` or `put()`. MUVI conducts bottom up analysis along a program's call graph and includes variables directly accessed by a function into the `Acc_Set` of its caller, but not the caller's caller. The rationale is that, if two accesses are several functions apart in the call chain, they are unlikely to be correlated (otherwise it is difficult for programmers to maintain the code). Future work can allow propagation across many levels of function calls with decreasing weights down the call chain.

7.3 Step 2: Access Pattern Analysis

The goal of this step is to identify variables that are accessed in the same function (i.e., appear in the same `Acc_Set`) for more than a threshold number of times. Each set of variables that satisfies this property is referred to as an *access pattern*. Note that a pattern is *not* an access correlation yet. Read and write accesses are *not* differentiated in this step.

In order to carry out this task in an efficient and scalable way, we leverage a well-studied data mining technique: frequent item set mining [14]. Frequent item set mining examines a database where each entry is an item set (i.e., a set of items) and tries to *efficiently* discover which subitem sets (subsets of an item set) are *frequent*, i.e., contained in more than a threshold (called *MinSupport*) number of database entries. For example, in an item set database $\mathbb{ID}$,

$$\mathbb{ID} = \{\{w, y, z\}, \{v, w, y, z\}, \{w, x, y\}\},$$

if $MinSupport = 3$, the mining result will show that item sets $\{w\}$, $\{y\}$, $\{w, y\}$ are frequent. If $MinSupport = 2$, item sets $\{w\}$, $\{y\}$, $\{z\}$, $\{w, y\}$, $\{w, z\}$, $\{y, z\}$, $\{w, y, z\}$ are frequent.

The specific frequent item set mining algorithm used in MUVI is called *FPclose* [14]. It is one of the most efficient frequent item set mining algorithms.

We apply FPClose to our `Acc_Set`database. In this database, each item set includes all variables accessed by a function in the target program, and each item is a program variable. The FPClose algorithm computes the frequent subitem sets. Each frequent subitem set is an access pattern, composed of variables that are accessed (regardless of their access types) in more than *MinSupport* number of functions. For example, at the threshold $MinSupport$=10, the noncorrelated variable examples shown in the rows (g-h) of Table 5 in the previous section will not be selected as candidates, since they are accessed in only a few functions together.

7.4 Step 3: Correlation Generation and Pruning

Each access pattern (x, y) reported by Step 2 implies 18 access correlation candidates in the form of $\mathrm{A1}(x) \Rightarrow \mathrm{A2}(y)$ or $\mathrm{A1}(y) \Rightarrow \mathrm{A2}(x)$, where A1 and A2 can be read, write, or AnyAcc.

At this final step, MUVI first calculates the *support* and *confidence* of each correlation candidate, and then prunes out candidates with low *support* or low *confidence*. At the end, MUVI generates a ranked list of access correlations.

Next, we use $\mathrm{read}(x) \Rightarrow \mathrm{read}(y)$ as an example to explain how to calculate *support* and *confidence*. Similar calculation is applied to other types of correlation candidates.

1. *Support.* The support of a correlation **C**: $\mathrm{read}(x) \Rightarrow \mathrm{read}(y)$, denoted as *support* (**C**), is the number of functions in which x and y are read together (based on the definition of togetherness in Section 7.1). Such a function is called a *supporter* of this correlation. If a correlation candidate has fewer than *MinSupport* number of supporters, it is pruned.
2. *Confidence.* The confidence of a correlation **C**: $\mathrm{read}(x) \Rightarrow \mathrm{read}(y)$ measures the conditional probability that, given $\mathrm{read}(x)$'s presence in a function, y is read nearby in the same function. It is calculated via *support* (**C**)/*support* ($\mathrm{read}(x)$), where *support* ($\mathrm{read}(x)$) is the number of functions that read x. MUVI uses a threshold *MinConfidence* to prune out correlation candidates with low confidence.

No static analysis is conducted at this step, because all the information is available in the `Acc_Set` database. Furthermore, since most irrelevant variable pairs are already pruned by Step 2, Step 3 is scalable for large real-world applications, as shown in our experimental results (Section 9).

Parameter setting. Setting above threshold parameters needs to consider the tradeoff between false positives and negatives. Our default setting provides a good initial point for most applications, as shown in Section 9. Users can further tune the parameters based on the properties of the targeting program and the usage scenarios. For example, small applications can use relatively small *MinSupport*. If correlations are considered as strict requirements and all violations are reported as bugs, it is better to be conservative and pick large parameters. If correlations are considered as hints to help bug detection, as we will see in multivariable concurrency-bug detection, it is fine to be aggressive and choose relatively small parameters.

8 Detecting Multivariable Bugs

The access correlations inferred by MUVI can help detect various types of bugs. This section presents how we extend race detectors to catch multivariable concurrency bugs. We will discuss other use of MUVI in the supplemental material, available online.

Although traditional race detectors only focus on the synchronization among accesses to the same shared variable, some researchers have recognized that unsynchronized accesses to related variables could also lead to software failures. They refer to this problem as *high-level data races* [3]. Unfortunately, previous work did not find an effective method to identify variable correlations, as discussed in Section 1. This challenge can be addressed by MUVI.

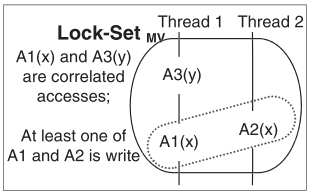


Fig. 6. Multivariable extension to the lockset algorithm.

Multivariable extensions to lockset. The lockset algorithm [8], [9], [35], [42] reports a data race bug when it finds that there is no common lock held during accesses to a shared memory location. To perform such a check, it maintains the set of locks currently held by each thread and the set of locks that have been used to protect each variable so far. A data race is detected when the latter set is empty.

The multivariable extension to the lockset algorithm, referred to as Lock-set$_{\mathrm{MV}}$, is as follows: For every pair of accesses, $\mathrm{A1}(x)$ and $\mathrm{A2}(x)$ (one is a write), from different threads to the same shared variable x, we check if they are protected by at least one common lock (the basic lockset algorithm), *and also* check if any correlated access A3(y), of $\mathrm{A1}(x)$ or $\mathrm{A2}(x)$, is also protected by a common lock with $\mathrm{A1}(x)$ *and* $\mathrm{A2}(x)$, as shown in Fig. 6. In our study, we extend an existing dynamic implementation of lockset algorithm in Valgrind [28].

Multivariable extensions to happens-before. The happens-before algorithm [7], [29], [31] detects data-race bugs by comparing the logic timestamps of accesses from different threads to the same shared variable. If the timestamps do not indicate a happens-before order among these accesses, a race is reported. Extending the happens-before algorithm to consider variable correlations is quite similar with our extension to the lockset algorithm: Happens-before$_{\mathrm{MV}}$ compares the logic timestamp between not only accesses to the same memory location but also correlated accesses.

9 Experimental Results of MUVI

9.1 Methodology of MUVI Evaluation

To evaluate MUVI's capability of analyzing access correlations, we use the latest versions of Linux (drivers), Mozilla-FireFox, MySQL, and PostgreSQL.

To evaluate MUVI's capability of detecting multivariable concurrency bugs, we use five real-world bugs from Mozilla and MySQL, including the one in Fig. 2. None of these bugs can be correctly detected by lock-set or happens-before detectors without MUVI-extensions. MUVI also found *four* **previously unknown** *multivariable concurrency bugs*.

More details of our experiment methodology and parameter setting can be available in the online supplemental material.

9.2 Variable-Access Correlation Inference

Table 7 shows the variable access correlation analysis results, including Step 1-3 presented in Section 7. As we can see, variable access correlations are very common in real applications: totally 6,449 `AnyAcc` $\Rightarrow$ `AnyAcc` correlations are inferred, with 5,954 variables and 1,467 structures involved. The analysis is efficient. For 3-4 million lines of code as Linux kernel has, MUVI takes only 3 hours to infer 3,353 access correlations.

TABLE 7
`AnyAcc` $\Rightarrow$ `AnyAcc` Correlations Inferred by MUVI

App.	LOC	#Access-Correlations	#Involved Variables	#Involved Structures	%False Positive	Analysis Time
Linux	3.6M	3353	3038	587	19%	175m2s
Mozilla	3.4M	1431	1380	394	16%	157m40s
MySQL	1.9M	726	703	209	13%	19m25s
PostgreSQL	832K	939	833	277	15%	98m23s
Total	-	**6449**	**5954**	**1467**	-	**450m30s**

To evaluate the accuracy of MUVI, we randomly sampled 100 correlations from each application and manually checked whether they are true. The results show that the false positive rate is reasonably low, around 17 percent on average.

The above results indicate that MUVI can efficiently and reasonably accurately infer access correlations, which can be stored in a database as a specification for reference by programmers or leveraged by other tools such as AutoLocker [25], DAIKON [11] and Colorama [5].

False positives and negatives. About 17 percent of correlations reported by MUVI are false positives. They are mainly caused by noninformative read-read coappearance or inflated appearance supports from macro or inlined functions. MUVI could miss correlations in two cases: 1) true correlation with low supports, a common problem for almost all statistics-based techniques [10], [20]; 2) *conditional correlation*, correlations that only exist under certain program contexts.

9.3 Bug Detection Results of MUVI

Overall. Table 8 shows the evaluation results on five real-world multivariable concurrency bugs. Both Lock-set$_{\mathrm{MV}}$ and Happens-before$_{\mathrm{MV}}$ can correctly identify the root causes of four tested multivariable concurrency bugs. None of these tested bugs' true root causes, i.e., multivariable concurrency bugs, can be identified by the original lock-set or happens-before algorithms without MUVI's extensions.

Furthermore, MUVI also detects four *new* multivariable concurrency bugs that have never been reported before.

False positives and negatives. The MUVI extension also introduces a small number (0-6) of false positives, caused by wrong correlations and benign multivariable races.

MUVI extension misses one tested bug. The reason is that MUVI cannot detect the *conditional correlation*, which only holds under certain program contexts, associated with

TABLE 8
Multivariable Concurrency Bug Detection Results

Bug	Lock-set$_{\mathrm{MV}}$		Happens-before$_{\mathrm{MV}}$	
	Detect	FP	Detect	FP
Moz-js1	Yes	1	Y	1
Moz-js2	Yes	2	Y	5
Moz-imap	Yes	0	Y	0
MySQL-log	Yes	3	Y	6
MySQL-blog	No	0	N	1
Note: In addition to the above existing bugs, we detected four *new* multi-variable concurrency bugs that were never reported before.				

None of the root causes can be correctly identified without MUVI extension. FP means the additional static false positives introduced by MUVI.

this bug. How to combine program contexts with variable correlations remains as future work. The detail of this bug can be available in the online supplemental material.

Additional results of MUVI parameter sensitivity and detecting other types of bugs with MUVI can be available in the online supplemental material.

10 Discussion

Comparing AVIO with MUVI. MUVI and AVIO share the same philosophy in combating concurrency bugs. They both consider concurrency bugs as violations to programmers' synchronization intentions. They both automatically infer underlying semantic information assuming that *whatever is abnormal is suspicious and unintended*. At the end, they both use the inferred information to detect concurrency bugs.

The specific inference approaches used by AVIO and MUVI are different. MUVI uses a data mining algorithm and works on source code. Instead, AVIO uses a simpler algorithm and works on execution traces. This difference comes from the different natures of the problems they target.

AI-invariants are almost inconceivable to infer through static analysis. Static analysis may tell whether two accesses are always atomic or which accesses can violate their atomicity. However, few "normal" or "abnormal" behavioral patterns can be collected from source code regarding atomicity. Different from AI-invariants, the variable-access correlations in MUVI could be collected through either static analysis or training-based trace-analysis. MUVI chooses to static analysis, because it can easily provide complete coverage of the whole program without being limited to the training input sets.

AVIO and MUVI also have different performance properties. Before detecting any bugs, MUVI has to spend time on static analysis to infer access correlations. Fortunately, it is required only once for all inputs and all detection runs. Similarly, before detecting any bugs, AVIO has to spend time on training to collect AI-invariants. When the input changes, AVIO may need to conduct further training, if the new input covers new code regions.

MUVI and AVIO complement each other well in concurrency bug detection, because they focus on two different types of synchronization intentions and consequently two different types of bugs. In addition, MUVI can extend AVIO to detect multivariable atomicity violation bugs. We discuss this topic which available in the online supplemental material.

Going beyond AVIO and MUVI. AVIO and MUVI can be extended in several ways. First, they provide a general and extensible framework for invariant-based concurrency-bug detection. It is conceivable to extend them by looking at more sophisticated synchronization intentions and interleaving patterns [23], [41]. Second, the synchronization intentions inferred by AVIO and MUVI can be used to annotate the program and help other tools. For example, MUVI can be used to automatically annotate programs for tools such as AutoLocker [25] and Colorama [5]. Future work can also extend AVIO and MUVI by collecting programmers' intentions from other places, such as program comments.

11 Related Work

There are many previous works related to AVIO and MUVI. Due to space limitations, this section only reviews the most related works that have not been discussed in earlier sections.

Atomicity violation detection. Several static and dynamic techniques based on state reduction theory of right/left mover have been proposed [13], [34], [37] to detect atomicity violations. These methods require programmers to annotate synchronization operations and code regions that need to be atomic, which is impractical—if programmers can properly do this, they can properly synchronize these areas. This problem is addressed to some extent by Atomizer [12], which gains synchronization knowledge through the lockset algorithm [35] 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 [39] 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 works, AVIO is fully automated: it infers atomic regions by observing AI invariants without knowledge of synchronization primitives. It also covers more atomicity violation cases than previous works inferring intended atomicity regions [39], benefiting from our comprehensive serializability analysis. Moreover, by leveraging the cache coherence protocol, AVIO-H has negligible overhead, orders of magnitudes smaller than previous works on atomicity violation detection. Finally, MUVI can further strengthen AVIO to automatically detect bugs that involve more than one variable.

Invariant-Based bug detection. AVIO follows the invariant-based bug detection approach, similar to DAIKON [11], DIDUCE [16], AccMon [43], and Liblit's work [21]. These work observe many successful runs of a software and infers likely program invariants. Different from them, AVIO focuses on invariants in access interleavings for concurrent programs.

MUVI shares a similar philosophy with previous works that automatically extract specifications from source code [1], [19], [20], [38], [40], [10]. Most of previous works focused on procedures and component interfaces instead of variable correlations. Instead, MUVI focuses on a different type of code patterns—multivariable access correlations.

Other related works. Much work has been done to ease the implementation of concurrent programs. Transactional memory (TM) [2], [15], [18], [27] is a widely recognized approach along this direction. TM can potentially raise the multithreaded programming level and remove deadlocks at the programmer level. Unfortunately, TM cannot remove atomicity-violation bugs or multivariable concurrency bugs. The reason is that programmers can make mistakes when dividing atomic regions. In addition, basic TMs only monitor conflicting accesses to each single variable and do not handle conflicting accesses to multiple correlated variables. AutoLocker [25] proposes using compilers to add locks automatically. It assumes that programmers need to *manually* annotate correlated variables to use the same lock. AVIO and MUVI well complement these works by automatically inferring intended atomicity regions and variable correlations.

12 Conclusions

This paper proposes a new concurrency-bug detection approach that focuses on programmers' synchronization intentions. Specifically, 74 real-world concurrency bugs are studied to understand the typical synchronization intentions that are violated in real-world bugs. Following this study, two concurrency-bug detection tools, AVIO and MUVI, are proposed to automatically infer synchronization intentions and detect two types of concurrency bugs that have not been well addressed before—atomicity-violation bugs and multivariable concurrency bugs.

Our evaluation with large real-world applications shows that both AVIO and MUVI are effective. AVIO can detect more atomicity-violation bugs with many fewer false positives than previous algorithms. MUVI successfully extracts 6,449 access correlations from Linux, Mozilla, etc., with high (83 percent) accuracy. Using the extracted correlations, MUVI extends existing race detectors to correctly identify the root causes of five tested real-world multivariable concurrency bugs, and also four new multivariable concurrency bugs that have never been reported before.

Our results indicate that it is promising to look at concurrency bugs from the perspective of synchronization intentions. It is also feasible to automatically infer these intentions. Our results also show that multivariable access correlation, which is largely ignored by previous research, is a common and important property in the real-world software.

Acknowledgments

The authors thank the anonymous reviewers whose comments and suggestions have greatly improved our paper. This work is supported by US National Science Foundation (NSF) CNS-0720743, NSF CCF-0325603, NSF CNS-0615372, NSF CNS-0347854 (career award), NSF CCF-1018180 grant, NSF CCF-1054616, NetApp Gift grant, and a Claire Boothe Luce faculty fellowship.

References

[1] R. Alur, P. Cerny, P. Madhusudan, and W. Nam, "Synthesis of Interface Specifications for Java Classes," *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*, 2005.

[2] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie, "Unbounded Transactional Memory," *Proc. 11th Int'l Symp. High-Performance Computer Architecture (HPCA)*, 2005.

[3] C. Artho, K. Havelund, and A. Bierre, "High-Level Data Races," *Proc. First Int'l Workshop Verification and Validation of Enterprise Information Systems*, 2003.

[4] M. Burrows and K.R.M. Leino, "Finding Stale-Value Errors in Concurrent Programs," Compaq SRC Technical Note 2002-04, 2002.

[5] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas, "Colorama: Architectural Support for Data-Centric Synchronization," *Proc. IEEE 13th Int'l Symp. High-Performance Computer Architecture (HPCA)*, 2007.

[6] J.-D. Choi et al., "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2002.

[7] A. Dinning and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *Proc. Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 1990.

[8] A. Dinning and E. Schonberg, "Detecting Access Anomalies in Programs with Critical Sections," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging (AOWPDD)*, 1991.

[9] D. Engler and K. Ashcraft, "RacerX: Effective Static Detection of Race Conditions and Deadlocks," *Proc. ACM 19th Symp. Operating Systems Principles (SOSP)*, 2003.

[10] D. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. ACM 18th Symp. Operating Systems Principles (SOSP)*, pp. 57-72, 2001.

[11] M. Ernst, A. Czeisler, W.G. Griswold, and D. Notkin, "Quickly Detecting Relevant Program Invariants," *Proc. 22nd Int'l Conf. Software Eng. (ICSE)*, 2000.

[12] C. Flanagan and S.N. Freund, "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs," *Proc. 18th Int'l Parallel and Distributed Processing (POPL)*, 2004.

[13] C. Flanagan and S. Qadeer, "A Type and Effect System for Atomicity," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2003.

[14] G. Grahne and J. Zhu, "Efficiently Using Prefix-Trees in Mining Frequent Itemsets," *Proc. IEEE First ICDM Workshop Frequent Item Set Mining Implementations (FIMI '03)*, Nov. 2003.

[15] 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," *Proc. 31st Ann. Int'l Symp. Computer Architecture (ISCA)*, 2004.

[16] S. Hangal and M.S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proc. 24th Int'l Conf. Software Eng. (ICSE)*, 2002.

[17] 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 Performance Evaluation Rev.*, vol. 31, no. 4, pp. 31-35, 2004.

[18] M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. 20th Ann. Int'l Symp. Computer Architecture (ISCA)*, 1993.

[19] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From Uncertainty to Belief: Inferring the Specification within," *Proc. Seventh Symp. Operating Systems Design and Implementation (OSDI)*, 2006.

[20] Z. Li and Y. Zhou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code," *Proc. 10th European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE)*, 2005.

[21] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, "Bug Isolation via Remote Program Sampling," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2003.

[22] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting Atomicity Violations via Access Interleaving Invariants," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[23] B. Lucia and L. Ceze, "Finding Concurrency Bugs with Context-aware Communication Graphs," *Proc. IEEE/ACM 42nd Ann. Int'l Symp. Microarchitecture (MICRO)*, 2009.

[24] 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," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2005.

[25] B. McCloskey, F. Zhou, D. Gay, and E. Brewer, "Autolocker: Synchronization Inference for Atomic Sections," *Proc. Ann. Symp. Principles of Programming Languages (POPL)*, 2006.

[26] S.L. Min and J.-D. Choi, "An Efficient Cache-Based Access Anomaly Detection Scheme," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[27] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood, "Logtm: Log-Based Transactional Memory," *Proc. 12th Int'l Symp. High-Performance Computer Architecture (HPCA)*, 2006.

[28] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2007.

[29] R.H.B. Netzer and B.P. Miller, "Improving the Accuracy of Data Race Detection," *Proc. Third ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 1991.

[30] R. O'Callahan and J.-D. Choi, "Hybrid Dynamic Data Race Detection," *Proc. Ninth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 2003.
[31] D. Perkovic and P.J. Keleher, "Online Data-Race Detection via Coherency Guarantees," *Proc. Second USENIX Symp. Operating Systems Design and Implementation (OSDI)*, 1996.
[32] M. Prvulovic, "Cord:cost-Effective (and Nearly Overhead-Free) Order-Reordering and Data Race Detection," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, 2006.
[33] 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)*, 2003.
[34] A. Sasturkar, R. Agarwal, L. Wang, and S.D. Stoller, "Automated Type-Based Analysis of Data Races and Atomicity," *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 2005.
[35] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems*, vol. 15, pp. 391-411, 1997.
[36] SecurityFocus, Software Bug Contributed to Blackout, 2003.
[37] L. Wang and S.D. Stoller, "Static Analysis for Programs with Non-Blocking Synchronization," *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 2005.
[38] J. Whaley, M.C. Martin, and M.S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA)*, 2002.
[39] 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)* 2005.
[40] J. Yang, D. Evans, D. Bhardwaj, T.B. t, and M. Das, "Perracotta: Mining Temporal API Rules from Imperfect Traces," *Proc. 28th Int'l Conf. Software Eng. (ICSE)*, 2006.
[41] J. Yu and S. Narayanasamy, "A Case for an Interleaving Constrained Shared-Memory Multi-Processor," *Proc. 36th Ann. Int'l Symp. Computer Architecture (ISCA)*, 2009.
[42] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," *Proc. ACM 20th Symp. Operating Systems Principles (SOSP)*, 2005.
[43] 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," *Proc. 37th Int'l Symp. Microarchitecture (MICRO)*, 2004.

Shan Lu received the PhD degree from the University of Illinois at Urbana-Champaign. She is an assistant professor in the Computer Science Department, the University of Wisconsin at Madison. Her main research interests are software reliability and concurrent software systems.

Soyeon Park received the PhD degree from Korea Advanced Institute of Science and Technology. She is a project scientist in the Department of Computer Science and Engineering at the University of California, San Diego. Her research interests include software reliability, operating system, and computer architecture.

Yuanyuan Zhou received the PhD degree in computer science from Princeton University. She is a Qualcomm chair professor in the Department of Computer Science and Engineering at the University of California, San Diego. Her research interests include software reliability, operating systems, and storage systems. She is a member of the IEEE.