A Lightweight System for Detecting and Tolerating Concurrency Bugs

Mingxing Zhang, Yongwei Wu, *Member, IEEE*, Shan Lu, Shanxiang Qi, Jinglei Ren, and Weimin Zheng, *Member, IEEE*

Abstract—Along with the prevalence of multi-threaded programs, concurrency bugs have become one of the most important sources of software bugs. Even worse, due to the non-deterministic nature of concurrency bugs, these bugs are both difficult to detect and fix even after the detection. As a result, it is highly desired to develop an all-around approach that is able to not only detect them during the testing phase but also tolerate undetected bugs during production runs. However, existing bug-detecting and bug-tolerating tools are usually either *1)* constrained in types of bugs they can handle or *2)* requiring specific hardware supports for achieving an acceptable overhead. In this paper, we present a novel program invariant, name Anticipating Invariant (A_I), that can detect most types of concurrency bugs. More importantly, A_I can be used to anticipate many concurrency bugs before any irreversible changes have been made. Thus it enables us to develop a software-only system that is able to forestall failures with a simple thread stalling technique, which does not rely on execution roll-back and hence has good performance. Experiments with 35 real-world concurrency bugs demonstrate that A_I is capable of detecting and tolerating many important types of concurrency bugs, including both atomicity and order violations. It has also exposed two new bugs (confirmed by developers) that were never reported before in the literature. Performance evaluation with 6 representative parallel programs shows that A_I incurs negligible overhead ($< 1\%$) for many nontrivial desktop and server applications.

Index Terms—Concurrency bugs, software reliability, bug tolerating

1 Introduction

1.1 Motivation

Nowadays, in order to better utilize multi-core systems concurrent programs are becoming more and more prevalent. But due to the inherent complexity of concurrency, these programs are remarkably error-prone [2]. Even worse, unlike sequential bugs, the manifestation of concurrency bugs depends not only on input data but also on many other timing-related events (e.g., thread interleavings). Thus it is both hard to detect concurrency bugs during the in-house testing phase, and difficult to fix the detected concurrency bugs. As a demonstration of the later case, recent investigations have shown that it frequently takes more than one month to fix a concurrency bug [3], [4], and, even after consuming so many development resources, nearly 70 percent of the patches are buggy in their first release [5], [6]. Consequently, an all-around approach that is able to not only detect bugs during in-house testing but also tolerate undetected bugs in production runs are highly desired.

Generally speaking, an ideal bug-tolerating tool should satisfy requirements from two aspects. First, it should have a high bug-tolerating *coverage*, which means that the tool can handle a wide variety of concurrency bugs that are hidden in the deployed applications. Specifically, the tool should be able to detect and tolerate both atomicity violations and order violations that involve only two threads and one variable, which, based on a previous empirical study [2], are the two most common types of concurrency bugs in the real world. Second, in order to ensure that the tool is useable in practice, it needs to only incur a low run-time *overhead* even on the commodity machines. Hence the tool must not rely on a custom hardware that currently does not exist.

According to our investigation, the existing techniques for tolerating concurrency bugs can be categorized into three types, depending on when bug toleration takes effect (Fig. 1). But none of them can satisfy the above two requirements simultaneously.

(1) *Always-on approach*. The first type of existing approaches is the *Always-on* approach, which constrains the program's execution *all the time* to prevent potential manifestations of concurrency bugs. This kind of approach constrains the threads' interleavings even in correct runs and hence relies on transactional memory or other custom hardware to achieve good performance. For example, both the Atomtracker [7] and the AtomAid [8] will automatically group instructions into chunks and require the hardware to execute each chunk atomically. In addition, this kind of approach often only probabilistically tolerates concurrency bugs with a specific root

- *M. Zhang, Y. Wu, J. Ren, and W. Zheng are with the Tsinghua National Laboratory for Information Science and Technology (TNLIST), the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, the Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China, and the Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, Zhejiang, China.*
  E-mail: {zhangmx12, wuyw, renjl10, zwm-dcs}@mails.tsinghua.edu.cn.
- *S. Lu is with the University of Chicago, Chicago, IL.*
  E-mail: shanlu@cs.uchicago.edu.
- *S. Qi is with the UBER growth team, San Francisco, CA.*
  E-mail: qi@uber.com.

Manuscript received 19 May 2015; revised 1 Feb. 2016; accepted 14 Feb. 2016. Date of publication 17 Feb. 2016; date of current version 21 Oct. 2016. Recommended for acceptance by C. Zhang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2531666

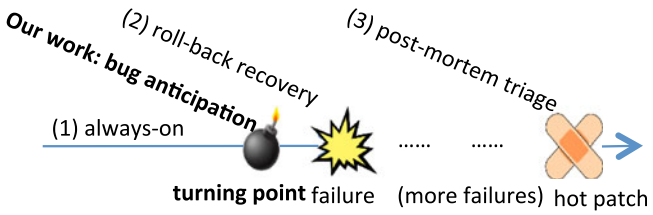


Fig. 1. Categorization of bug-tolerating tools. *The arrow in the figure represents the time-line during production runs.*

cause pattern (e.g., atomicity violations), and are unable to handle bugs with other types of root causes (e.g., order violations).

(2) *Failure-recovery approach*. Tools within this category, such as PSet [9] and Frost [10], will roll back the program's execution to a recent checkpoint *when failures or errors occur* and rely on the re-execution for automated recovery. However, the checkpoint and rollback mechanism is quite complex. Hence it will incur significant overhead on commodity machines, which makes it impractical to be used in practice. A recent work ConAir [11] has achieved the low overhead by only rolling back idempotent regions of one thread, which can be reexecuted for any number of times without changing the program semantics. However, as we will discuss in Section 5.4, ConAir can only achieve that overhead by significantly sacrificing the tool's bug-tolerating coverage.

(3) *Post-mortem approach*. Finally, the third kind of bug-tolerating tools is the *Post-mortem* approaches. This kind of approaches [12], [13] aims to prevent future manifestations of a concurrency bug, *after triaging* earlier manifestations of the bug. They can ease the pain of lengthy patch releasing period, but cannot prevent failures caused by unknown bugs (i.e., losing coverage).

1.2 Our New Approach

In this paper, we propose a new approach to tolerate concurrency bugs in production runs. Different from all the previous techniques, this new approach achieves both the coverage and the performance requirement by *anticipating* the manifestation of concurrency bugs at run time. This anticipating ability enables us to prevent bugs' manifestations through temporarily stalling the execution of one thread, which incurs much smaller overhead than the checkpointing and rollback mechanism used in the previous works.

The key observation behind our approach is that there exists a *turning point* t during the manifestation of a concurrency bug. Before t, the manifestation of the bug is non-deterministic and can be avoided by perturbing the threads' scheduling. In contrast, after t the manifestation of the bug becomes deterministic and cannot be avoided without a rollback. Thus if a concurrency bug can be anticipated before its turning point, its manifestation can be prevented by temporarily stalling a thread, which incurs little overhead.

As you can imagine, anticipating bugs right before the turning point is critical to bug tolerating. Anticipating too early will inevitably encounter many false positives, which causes unnecessary thread stalling and performance losses.

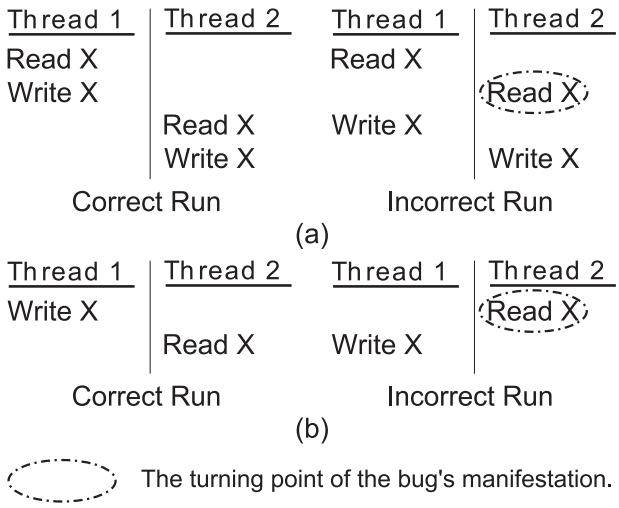


Fig. 2. Illustrations of bugs' turning points.

Anticipating too late will miss the chance of lightweight bug toleration—only the heavy weight checkpoint-rollback mechanism can restore the correct states after turning points.

However, anticipating bugs right before the turning point is also challenging. Previous concurrency-bug detection tools did not consider bug anticipation and would indeed detect many bugs *after* the turning points (we will discuss in more details in Section 5.2). Below, we simply demonstrate how two straw-man ideas do not work for bug anticipation.

Straw-man 1. Detecting a bug before the execution of buggy writes. Intuitively, one might think that it should be early enough to prevent a bug, if no buggy write has happened. Unfortunately, this is not true. Fig. 2a shows a typical atomicity violation pattern, where the expected atomicity of write-after-read is violated. Many real-world concurrency bugs follow this pattern [2]. Here, the turning point is actually right before the second read instruction, as circled in Fig. 2a. Once that read happens, although no bug-related write has executed, the atomicity violation is inevitable.

Straw-man 2. Detecting a bug before the execution of the second buggy thread. Suppose a bug involves two threads. Even if only one thread's buggy code region has executed, it could still be too late. Fig. 2b illustrates a typical order violation pattern, where a read in thread two unexpectedly executes before a write in thread 1. Many real-world concurrency bugs follow this pattern and lead to problems such as un-initialized reads [2]. The turning point in this example is right before the read in Thread 2, as circled in Fig. 2b. Once that read is executed, although the buggy code region in thread 1 has not executed yet, the order violation is inevitable.

1.3 Contributions

This paper makes the following contributions.

First, we proposed a new approach for tolerating production-run concurrency bugs. This new approach complements the existing bug tolerating approaches by anticipating concurrency bugs' manifestations right before their turning points. As a result, we can use a lightweight thread-stalling technique, instead of the heavyweight checkpoint-rollback mechanism, to get around the detected bugs.

Second, we designed a novel invariant, named Anticipating Invariant (AI), that is suitable for effective and efficient

concurrency-bug toleration. Roughly speaking, previous bug-detecting invariants [9], [14], [15], [16] usually focus on the ability to observe the concurrency bug ultimately, hence the detections are often too late for tolerating concurrency bugs without using roll-back. In contrast, AI will *immediately* be violated after bugs' turning points (e.g., after the instruction enclosed by dotted oval in Fig. 2 is executed), which makes the anticipation of bugs both possible and timely. Specifically, for an instruction I that accesses variable V, AI captures which instructions are allowed to access V right before I from a different thread. Consequently, we can check before the execution of each instruction I whether the immediate execution of I would violate its corresponding invariant. If that is the case, we need only to postpone I's execution by temporarily stalling I's thread to avoid the violation. What distinguishes AI from previously proposed interleaving invariants is that AI can help achieve both the coverage goal and the overhead goal of concurrency-bug toleration. In terms of coverage, it reflects programmers' intentions about the correct order of concurrent memory accesses, and its violation can be used to detect both atomicity and order violations. In terms of overhead, the violation of AI occurs at exactly the turning point of most concurrency bugs, not too early and not too late. More details are presented in Section 2.1.

Third, based on AI, we implemented a low-overhead software-only system[1] that is able to not only detect concurrency bugs during the testing phase but also tolerate them on-the-fly during the production runs. Our system includes several steps: it first automatically learns AI during the in-house testing phase; it then monitors violations to AI during the production runs; finally, it automatically and temporarily stalls a thread right before an AI violation to prevent the manifestation of concurrency bugs. To the best of our knowledge, this is the first attempt to efficiently tolerate previously unknown atomicity and order violations at run time without rollbacks. Our system also includes an optional bias instrumentation scheme and several APIs to allow easy performance tuning for memory-access intensive applications. More details are presented in Section 3.

Fourth, besides detecting and tolerating, we also explore the usage of AI at other phases of the whole bug-handling lifecycle. In Section 4, we demonstrate how to use AI to actively expose order violations, and discuss how to use AI as an emergency patch generator.

Finally, we conduct an evaluation that is based on 35 representative real-world concurrency bugs. The evaluation results show that our system can tolerate all of the 35 concurrency bugs, which is more than each of the existing techniques that we have evaluated. Our system also incurs low overhead—smaller than 1 percent overhead for many non-trivial desktop and server applications. Furthermore, we detected two previously unknown concurrency bugs from widely used open-source software, which are confirmed by the corresponding programmer and fixed in the nightly build.

1. We have made the source code of our tool publicly available at http://james0zan.github.io/AI.html. Related documentations and several demos are also presented there.

2 Anticipating Invariant

Program invariants are predicates that should always be true at certain points of the execution; they usually reflect programmers' intentions. Many recent works pay close attention to learn "likely invariants" from testing runs and use them to detect or tolerate software bugs [15], [16], [17]. These invariants are supposed to be held in all the correct runs, thus if one of them is violated at run time, a bug probably has manifested. However, although these invariants all differ vastly in their details, many of them are constrained in types of bugs they can handle. More importantly, they are designed for detecting bugs instead of anticipating bugs, which makes them unsuitable for lightweight bug-tolerating.

In order to resolve the above problems, we proposed a novel invariant, named Anticipating Invariant (AI), that will be violated immediately after the bugs' turning point. In this section, we first introduce the definition of AI. Then, we present some case studies to demonstrate AI's ability of anticipating concurrency bugs right before the turning points. Finally, we discuss why and how AI is different from prior works.

2.1 Definition

Through investigating many real-world bugs, we find that the manifestations of most concurrency bugs involve an instruction I_1 that is preceded by an unexpected instruction I_2 from a different thread, where I_1 and I_2 access the same variable. In addition, postponing the execution of I_2 can often prevent the bug's manifestation (i.e., the execution of I_2 is the turning point).

For example, most of the order violations occur when an instruction I_1 from Thread 1 unexpectedly executes after instruction I_2 from Thread 2, which causes I_2 to be preceded by a different instruction that accesses the same variable as I_2. And these bugs can be avoided by postponing I_2 until I_1 is executed. As for another example, most of the atomicity violations occur when instruction I_2 from Thread 2 unexpectedly interleaves the instruction I_1 and I_3 from Thread 1, so that I_2 is unexpectedly preceded by I_1 (instead of I_3). Similar to the order violations, postponing I_2 can effectively prevent this kind of atomicity violations.

Leveraging the above observation, we propose the Anticipating Invariant, which can satisfy both of the two requirements listed in Section 1.1. Specifically, in the rest of this paper we will use S_y to indicate a static instruction in the source code, which is a line of code that can be differentiated by its program counter. And we will use I_xS_y to represent that the dynamic instruction I_x observed at run time is derived from static instruction S_y. Here, the "dynamic instruction" means an execution instance of a static instruction, thus a static instruction in loops or recursions can have many dynamic instructions that are derived from it.

Following these definitions, we define a *remote predecessor*, expressed as $RPre(I_x)$, for every dynamic instruction I_x in the execution traces. $RPre(I_x)$ is a static instruction, which has at least one dynamic instruction derived from it that *1)* accesses the same memory address as I_x; *2)* comes from another thread (besides I_x's thread); and *3)* accesses the address *immediately before* I_x. By using the phrase "immediately before", we mean that there is no instruction

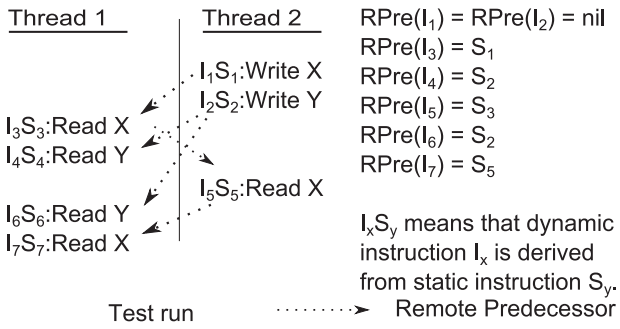


Fig. 3. Demonstration of remote predecessors.

that has accessed the same address is interleaved. In other words, we consider I_2 from Thread 2 to be immediately before I_1 from Thread 1 if and only if, except instructions from Thread 1, there is no instruction that accesses the same address of I_1 between the execution of I_2 and I_1. And $RPre(I_x) = nil$ if there is no such dynamic instruction. For example, Fig. 3 shows an interleaving and each instruction's corresponding remote predecessor. As we can see from the figure, although I_4 executes between I_2 and I_6, $RPre(I_6) = S_2$, because I_4 is executed by the same thread as I_6.

To be more explicit, the remote predecessor has the following characteristics: *1)* It is defined for every dynamic instruction. Thus if one static instruction is executed more than once in an execution, there will be multiple dynamic instructions and hence multiple remote predecessors that are needed to be calculated; *2)* the nil state is specially defined to describe the state that the instruction can be executed before any instructions from other threads that access the same address. As we will discuss later in Section 2.2, nil is very useful for anticipating bugs in practice; *3)* Different from many previous works [9], whether the instruction is a read or a write operation does not matter in calculating the remote predecessors.

Fig. 4 shows another interleaving, in which all the memory operations access the same address. Note that I_1's remote predecessor is nil, because no instruction has accessed the variable X before it. And, as there are two dynamic instructions derived from the same static instruction S_2 in this test run, their remote predecessors are calculated separately, and the results are S_1 and S_3 respectively.

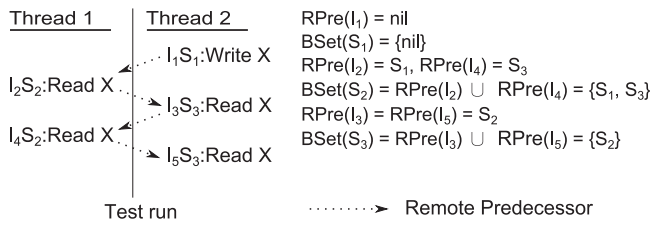


Fig. 4. Demonstration of belonging sets.

After investigating many concurrency bugs, we observe that: In all the correct runs, the remote predecessor of the same static instruction's dynamic instructions has fixed candidates. And once a dynamic instruction's remote predecessor does not belong to this set, it implies the occurrence of a concurrency bug. Hence we calculate a *Belonging Set*, expressed as $BSet(S_y)$, for every static instruction, which is the union of all the remote predecessors of its dynamic instructions that have been seen in verified interleavings. And we define the *Anticipating Invariant* to be:

$$RPre(I_xS_y) \in BSet(S_y), \;\; I_xS_y \; is \; derived \; from \; S_y$$

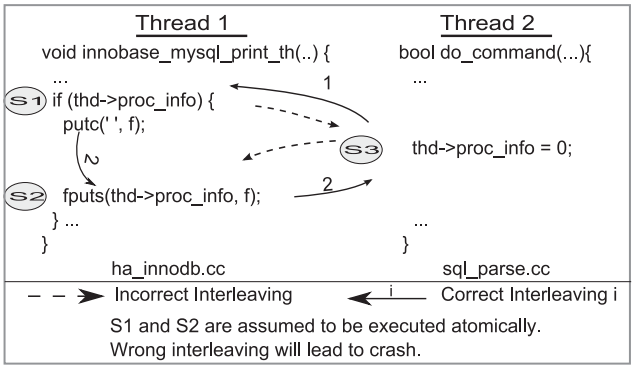
As an illustration, $BSet$ of S_2 in Fig. 4 is calculated as $BSet(S_2) = RPre(I_2) \cup RPre(I_4) = \{S_1, S_3\}$. And the belonging set of S_3 is $\{S_2\}$, since the two dynamic instructions derived from it have the same remote predecessor.

2.2 Case Studies

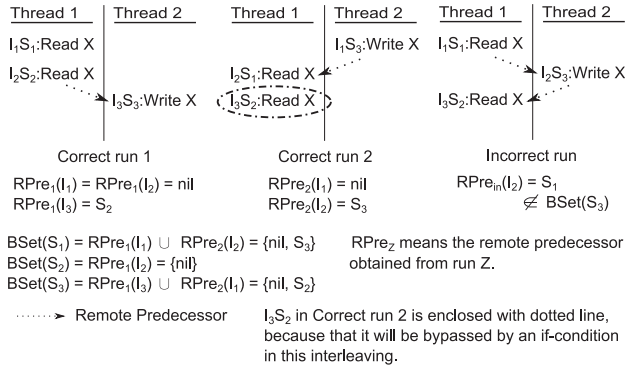
2.2.1 Atomicity Violation

Fig. 5a shows a real-world atomicity violation from the MySQL database server, while Fig. 5b is the corresponding simplified code of this bug. As we can see from the figure, a total of two possible interleavings can be found in correct test runs, and only one possible interleaving can be found in incorrect runs.

After observing the correct test runs, we can calculate that $BSet(S_3)$ is $\{nil, S_2\}$. Then in the incorrect case, when I_2S_3 in Thread 2 wants to be executed before I_3S_2 in Thread 1 after I_1S_1 has already been executed, its remote predecessor will be S_1. Since $S_1 \notin BSet(S_3)$, a violation is reported. Note that this bug can be anticipated before S_3's execution, at which point, the run-time environment still can prevent the bug from happening by temporarily



(a) The lines of code that are related to the bug.



(b) The simplified version of the bug.

Fig. 5. A real-world atomicity violation in MySQL. I_3S_2 in correct run 2 is enclosed in dotted line, because it is bypassed by an if-condition and hence not executed in that case. As a result, the belonging set of S_2 is $\{nil\}$ rather than $\{nil, S_3\}$.

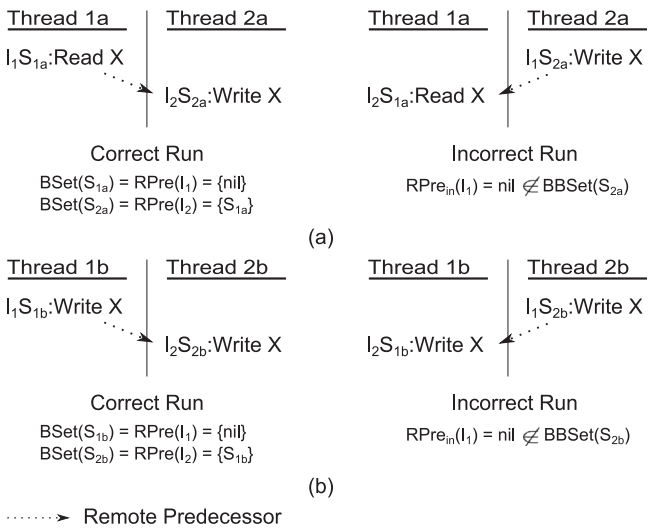


Fig. 6. Interleavings of two typical order violations.

stalling the execution of Thread 2. There is no need to roll back any executed instruction here.

Another example of the atomicity violations is shown in Fig. 2a. In the incorrect run, AI can anticipate the bug before $Read_{Thread\,2}$, because $Read_{Thread\,1}$ does not belong to its belonging set. Hence the bug can also be avoided by temporarily stalling the execution of Thread 2.

In contrast, although previous works have proposed many types of invariants to detect atomicity violations eventually [9], [14], [15], [16], they cannot predict many kinds of atomicity violations before their turning points. We will discuss the distinctions in more details in Section 5.2.

2.2.2 Order Violation

Fig. 6 shows two representative interleavings obtained from a R-W order violation and a W-W order violation respectively. The remote predecessor of I_1S_{2a} in subfigure (a) and I_1S_{2b} in subfigure (b) are both nil in the incorrect run. In this case, a violation will be reported because neither $BSet(S_{2a})$ nor $BSet(S_{2b})$ contains nil. Similar to atomicity violations, the bug is detected right before its turning point and hence can be avoided without using roll-back.

Unlike our Anticipating Invariant, previous works' [9], [15], [16] ability of anticipating order violations will be influenced by other conditions like whether there is another leading instruction accessing the same address. A formal discussion will be given later in Section 5.2.

2.3 Rationales

Although most of the invariant-based techniques share the same formation of learning some invariants from testing runs and then checking/guarding them later. AI differs from the others in its ability of avoiding roll-back, which requires the invariant designer's perspective to be shifted from "how to detect the bug eventually" to "how to anticipate the bug right before its turning point". Owing to this unique perspective, many special decisions are made during the design of AI:

(1) Many previous works [7], [14] record some states in a former instruction and check them at later instructions. Then they learn invariants about how these states will be preserved or altered. But when the invariant is violated at the later instruction, one can do nothing but roll-back to tolerate the bug. Instead, AI does not wait for the last instruction in a buggy region to detect the invariant violation. It constrains the instant state of each instruction not a continued state of an instruction region.

(2) AI does not differentiate read and write instructions, so that it can avoid the deficiency of straw-man idea 1 described in Section 1.1.

(3) A special state nil is explicitly defined in AI to represent the initial state. It helps us to address the limitation of straw-man idea 2 that is also listed in Section 1.1.

(4) AI tries to make minimal assumptions about where a bug may hide. For example, instructions involved in an Anticipating Invariant do not need to constitute the unserializable interleaving like AVIO [14], the read-write dependence described in DUI [15], or the memory-dependent used in PSet [9], etc. Thus, as shown in Section 5.2, AI can detect more bugs than each of the previous works can do.

3 Implementation

In order to utilize the Anticipating Invariant for detecting and tolerating concurrency bugs, we implement a software-only system by using the LLVM compiler framework [18]. In this section, we first give an overview of our system. Then, we describe how to automatically extract AI and how to use it for detecting/tolerating concurrency bugs. Finally, we discuss the usages of custom instrumentation strategies and the provided APIs.

3.1 Overview

In our implementation, we built a system mainly consists of three LLVM passes, namely AIPrepare, AITrace, and AITolerate. Each of them will perform a corresponding transformation to the input source code.

Specifically, the input of AIPrepare pass is the original source code. It will assign an universally unique access ID to each load/store instruction in the LLVM IR (by adding a metadata node). The marked code is stored in bitcode format for further usage. If the user also designs a custom instrumentation strategy with our API, a corresponding white list file will also be generated (elaborated in Section 3.4.2).

Then, the AITrace pass reads the marked code and adds a logging function before each memory access, which will output a triplet of access ID, thread ID, and the accessed memory address to the trace file. This instrumented code is used in the in-house testing phase to gather enough traces for computing BSets.

Finally, the AITolerate pass uses all the data generated before (white list and traces) to transform the marked code to an AI -guarded version of code. The generated code is compiled to executable objects and used in production runs.

3.2 Training

In order to infer AI automatically without any programmers' annotation, we rely on correct runs observed

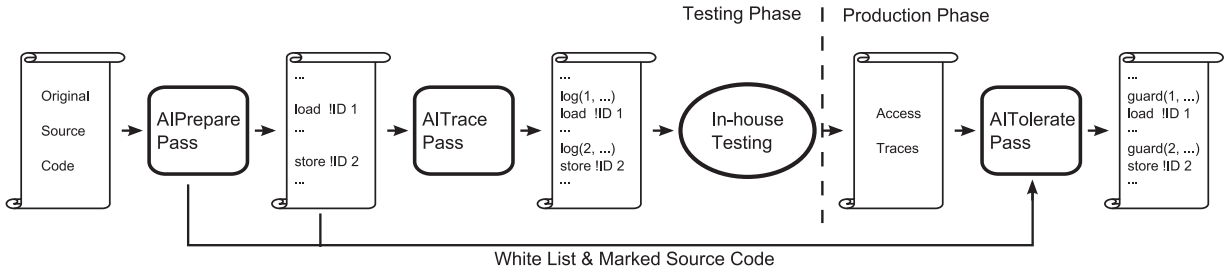


Fig. 7. Overview of how to use AI for tolerating concurrency bugs.

during the in-house testing phase. As pointed out by prior work [9], programmers can assert whether a test run is correct or not by verifying the outputs. Generally, the programmers should both run the application under different inputs to cover all the feasible paths, and run multiple times with every input to explore different interleavings. A systematic concurrency testing framework such as CHESS [19] or CTrigger [20] can also be used to systematically explore different interleavings for each input.

Then, after gathering enough trace files, we can now extracting AI from them. Specifically, as described in the above section, after running the instrumented program under various inputs for many times, the added logging functions will generate corresponding trace files that consist of triplets of (Access ID y, Thread ID tid, Accessed Memory Address addr). Each of these triplets represents a dynamic instruction defined in Section 2.1. By scanning the trace files chronologically, we can calculate the remote predecessor of each dynamic instruction according to the definition given before. In the meantime, the Anticipating Invariant for each static instruction is calculated by updating its belonging set, which can be formally described by:

$$BSet(y) = BSet(y) \cup RPre(Triplet_x)\ if\ Triplet_x = (y, .., ..).$$

Since disk I/Os are expensive, to shorten the time spent in training, the logging functions in our implementation first buffer logs in memory and then use direct memory access (DMA) APIs to flush logs to the disk.

3.3 Detecting & Tolerating

After obtaining the invariants, the AITolerate pass will encode the synthesized BSets (the union of the results of each trace file) into the application by adding an initialization function to the program's $llvm.global\_ctors$ array, which is the list of constructor functions and hence will be executed before the execution of other functions. The AITolerate pass will also add a guarding function before every shared-memory accesses to perform bug tolerating. These guarding functions ensure that all the anticipating invariants will not be violated. Since a shared variable will be accessed by at least one static instruction S that satisfies the property: $BSet(S) - \{nil\} \neq \emptyset$, we can identify shared-memory accesses during the training while inferring AI.

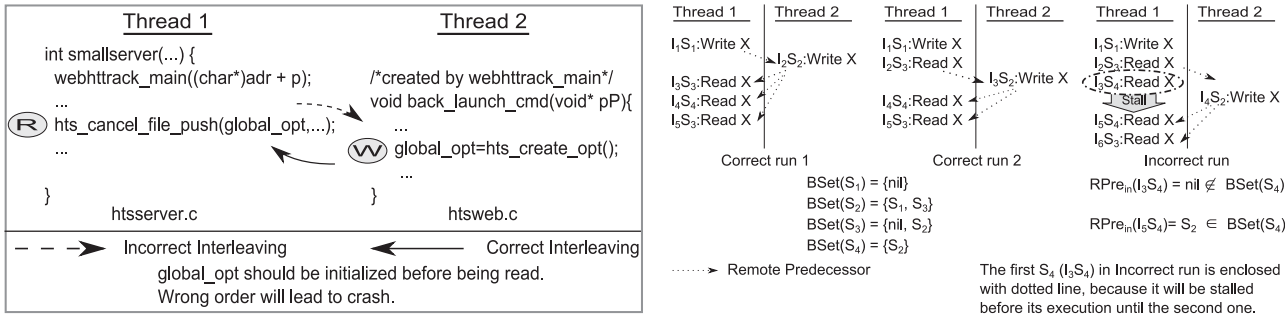
Specifically, the guarding functions will maintain a data structure $Recorder[M]$ to record the last two instructions that access memory M and are from different threads. This is enough for calculating the remote predecessors, because $RPre$ of the current operation is the last access (before updating) if the access is from a different thread, or it must be the second last one. After obtaining the $RPre$, the guarding function will check whether the corresponding Anticipating Invariant of the current instruction is held, and an *Anticipating Invariant Violation* is reported if it does not (i.e., $RPre(I_x) \notin BSet(S_y)$ although I_x is derived from S_y). Corresponding violation reports are generated to indicate the programmers that a potential site of concurrency bug is detected.

Moreover, as analyzed in Section 2.2, thanks to AI's capability of anticipating bugs before their occurrence, we can tolerate these violations by stalling the violating thread until the violation gets resolved. As for the implementation, the violated AI will be checked again and again, in order to determine whether the accesses from other threads have resolved it. If the check passes, the stalled thread will resume its execution.

Although it is rare, not all the Anticipating Invariant Violations can be tolerated by just perturbing the thread schedule. It is possible that the only correct interleaving for an input is untested. If such "fake" violation (i.e., false positive) is not properly treated, it may cause an indefinite stall. Thus, in order to ensure forward progress, we set the maximum stall time to a threshold (10 ms is used in our experiments). Once the threshold is reached, the system will log the violation and resume the stalled thread's execution. The log is sent back to developers to determine whether this violation is a bug. If it is not, we can update the relevant AI to green-light this new interleaving in future runs. Our algorithm ensures that stalling does not occur during the tested interleavings.

Fig. 8b shows the simplified version of Fig. 8a, which is a real-world W-R order violation in HTTrack. The order assumed by developers is that S_4 should always be executed after S_2. As pointed out by Shi et al. [15], since S_2 may inject between S_1 and S_3's execution in some cases (like correct run 1), PSet cannot detect this bug. But the remote predecessor of S_4 is always S_2 in both two correct test runs, and hence $BSet(S_4) = \{S_2\}$. This invariant constrains that S_4 is impossible to be executed before S_2 in production runs. More details about this violation in the incorrect run can be found in Fig. 8b.

To be more clear, Algorithm 1 gives the pseudo code of our guarding function, in which the *recorder* data structure is used for collecting runtime information and the learnt invariants are stored in $BSet$. Specifically, the elements of *recorder* are initialized with two *nil* flags, and for each tracked shared memory access the corresponding element is updated by right shifting the data array. In contrast, the $BSet$ data structure is initialized at the beginning of the program's execution and keeps unchanged for the whole execution. As for the stalling



(a) The lines of code that are related to the bug.

(b) The simplified version of the bug.

Fig. 8. A real-world W-R order violation in HTTrack, which cannot be detected by PSet.

mechanism, we currently use a simple method that just calls an "usleep()" functions to stall the thread for one millisecond, which is enough because the program will not encounter with unnecessary stalling after achieving sufficient training. Moreover, several tricks are used in our implementation to further lower the run-time overhead of our AI system: *1)* a fine grain locking mechanism is used for the implementation of the hash map that reserves the learnt belonging sets and the *recorder*, which allows synchronized and efficient hash-map accesses and hence the horizontal scalability of the original program will likely not to be constrained by AI; *2)* a tuned hash function that uses bitwise operators only are used for speeding up the hash-table queries; and *3)* thread local variables are used for caching the thread-depended information that is different for each thread and does not need to be synchronized with each other (e.g., the thread ID), which can save the time caused by repeated system calls.

3.4 Custom Instrumentation Strategy

Since only shared-variable accesses have to be instrumented and there is no need to roll back, AI incurs low overhead for many nontrivial desktop and server programs and is promising for production deployment. But for some applications, such as the high-performance computing (HPC) programs that have intensive heap accesses, the default instrumentation scheme may still incur very high overhead. To alleviate this problem, AI also provides users the ability to design custom instrumentation strategies that can decrease the overhead with little damage to AI's ability of detecting and tolerating bugs. For example, Lu et. al. [21] shows that a quarter of concurrency bugs in filesystem arise on failure paths. A custom strategy that preferentially covers these regions will definitely be very useful.

3.4.1 The Bias Instrumentation Scheme

As an illustration of the custom instrumentation strategy, we have proposed an optional bias instrumentation scheme that selectively instruments the "cold-region" of a program. We expect that this scheme can decrease the overhead while only missing few harmful bugs in practice. According to our evaluation, this scheme is effective for the aforementioned HPC programs (the results will be elaborated in Section 5.5).

Algorithm 1. Pseudo code of the Guarding Function

Global Variable:
recorder[M]={$< id, tid >$, $< id, tid >$}:
The map that maps a memory address M to the last two instructions that operate address M and are from different threads.
The elements of *recorder* are initialized with two *nil* flags.
$BSet[I]$: The map that maps an instruction I to its Belonging Set.

function $UpdateRecorder$ (Memory Address m, Access ID id, Thread ID tid)
  if $recorder[m][0] = nil$ **then**
    $recorder[m][0] := \ < id, tid >$
    return nil
  end if
  if $recorder[m][0].tid \neq tid$ **then**
    $recorder[m][1] := recorder[m][0]$
  end if
  $recorder[m][0] := < id, tid >$
  if $recorder[m][1] = nil$ **then**
    return nil
  else
    return $recorder[m][1].id$
  end if
end function
function $Guard$(Dynamic Instruction ins, Memory Address m)
  $iter := 0$; $id := GetAccessID(ins)$
  $tid := GetCurrentThreadID()$
  while $iter < threshold$ **do**
    $RPre := GetRPre(ins)$
    if $RPre \notin BSet[id]$ **then**
      Report an violation
      Stall this thread for a while.
    else
      Break
    end if
    $iter := iter + 1$
    if $iter = threshold$ **then**
      Report an unresolved violation
    end if
  end while
  $UpdateRecorder(m, id, tid)$
end function

Specifically, our bias instrumentation scheme is based on the so-called "cold-region" hypothesis [22], which is "in a

well-tested program, bugs usually occur in less-executed (i.e., cold) regions". The intuition of this hypothesis is simple: the more frequently (i.e., "hot") a code region is executed, the more it is examined in the standard stress testing period, and hence the less possible a bug is hidden in it. Many previous sampling-based tools [22], [23] leverage this hypothesis by sampling executions of code segments at a rate inversely proportional to their execution frequency. Thus, with this method, rarely executed code segments are effectively traced whereas frequently executed code segments are sampled at a very low rate. This approach trades the ability to collect more samples from frequently executed code segments for more comprehensive code coverage, while maintaining similar runtime overhead. According to their evaluation, this mechanism can significantly reduce the overhead and, at the same time, do not hurt the coverage, which achieves a good result in Google's practice [24].

Moreover, according to our investigation, the "cold-region" hypothesis can be further strengthened for the HPC programs, because the distribution of code execution frequency in HPC programs is extremely skewed. In HPC programs, there will usually be a few instructions that are responsible for calculating mathematical kernels. These instructions are hot since they account for a large portion of all the executed shared memory accesses, but they are also bug-sensitive, since if a bug is hidden in them it is not possible for the program to output correct results. As a result, our bias instrumenting scheme simply chooses to not instrument this kind of instructions, which will improve the performance significantly.

As for the implementation, we first group all the static instructions into maximal number of groups that: if instruction S_a and S_b belong to different groups, they will never access the same memory location. In order to achieve this goal, we leverage the traces obtained in the training phase and a data structure named disjoint-set. Specifically, a disjoint-set is a data structure that keeps track of a set of elements partitioned into a number of disjoint subsets [25]. It supports two useful operations: *1) Find*: determine which subset a particular element is in; and *2) Union*: join two subsets into a single subset. With this data structure, we can first let all the instructions and all the memory addresses belong to their own distinct groups. And then, for each memory access record in the trace "$IxSy : Read/Write\ Addr$", the group of instruction S_y is merged with the group of memory address $Addr$ by executing $Union(Find(S_y),\ Find(Addr))$. According to the complexity analysis, after adopting the common optimization technique of disjoint-set named "path compression" and "link-by-rank", the amortized cost of each *Find* and *Union* operation become $O(log(n))$ and $O(1)$ respectively.

After the grouping, we compute an ins-proportion (IP) for each group, which is the proportion of dynamic instructions generated by members of this group to all the shared memory accesses. This information can also be gathered from the traces and calculated in tandem with the grouping. As for the implementation, we maintain an execution count $ECount$ for each group, and for each *Union* operation the corresponding $ECount$ is accumulated by using the following formula:

$$ECount[Union(Find(S_y), Find(Addr_a))] := ECount[Find(S_y)] + ECount[Find(Addr_a)].$$

The procedure of this calculating is formalized as Algorithm 2.

Algorithm 2. Pseudo code for Calculating Instruction Groups and the Corresponding Execution Count

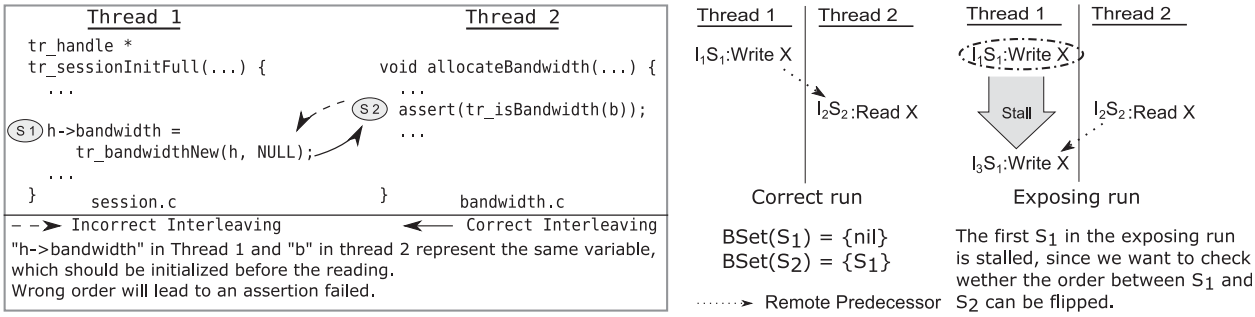
```
Global Variable:
    DisjointSet: A disjoint-set that keeps track of the groups of
        each static instructions and memory addresses.
    ECount[G]: The map that maps a group G to its
        execution count.
for all record ∈ trace do
    if record is a memory access then
        S_y := GetInstruction(record)
        G_a := DisjointSet.Find(S_y)
        Addr := GetMemoryAddress(record)
        G_b := DisjointSet.Find(Addr)
        G_c := DisjointSet.Union(G_a, G_b)
        ECount[G_c] := ECount[G_a] + ECount[G_b]
    end if
end for
```

Finally, if a specific flag is set when applying the AITolerate pass, those groups whose IP is larger than a threshold will not be instrumented. As we will discuss more in detail later in Section 5.5, in the HPC programs that we tested, there are usually several groups that only contains a few (< 20) instructions and in that same time, accounts to a large portion ($> 30\%$) of shared memory accesses. Through not instrumenting the instructions of these groups, we can significantly reduce the overhead and, at the same time, not miss the bugs we evaluated.

Theoretically, other metrics, such as the ins-proportion of each single instruction, can also be used to identify the "hot" instructions. But, in our case on HPC programs, the IP for each group is enough.

3.4.2 APIs for Designing Custom Strategies

Generally speaking, the users can implement their own instrumentation strategies very conveniently by directly modifying the generated BSets. However, we also provide several APIs to further facilitate this procedure. By default the AITolerate pass will instrument all the shared variable accesses, but if a specific flag is set, it will only instrument the instructions declared by the following annotations: *1)* the $AI\_INS\_THIS\_FUNC$ and $AI\_INS\_THIS\_BB$ macros are used to tell AI that all the shared memory accesses belong to this function (or basic block) should be instrumented; and *2)* the $AI\_INS\_THIS\_ADDR(void * addr)$ function is given to state that AI should instrument all the accesses to $addr$, which is implemented by using a dynamic analysis technique. Specifically, when applying the AITrace pass, this function will be replaced by a function that outputs the actual value of $addr$ to the trace file. Then, combining this information with the former mentioned triplets, all the related instructions (that have been observed during the testing phase) can be identified. The users only need to posit these annotations in the proper positions of the code and set



(a) The lines of code that are related to the bug.

(b) The simplified version of the bug.

Fig. 9. Another real-world W-R order violation in transmission.

the corresponding flags. Then the whole procedure, such as the recomputation of BSets (because the omitted instructions should not affect *Recorder*) and selective instrumentation, will all be automatically handled.

4 Other Usages of AI

Essentially, AI learns the correct order of concurrent memory accesses that are intended by programmers. And these learnt invariants can be used for many other purposes apart from passive bug detecting and tolerating. In this section, we will demonstrate the potentials of using AI to actively expose concurrency bugs, especially the order violations. Moreover, we will also show how to use AI as an emergency patch generator.

4.1 Exposing Order Violations

As we have discussed in Section 1.1, concurrency bugs rarely manifest during in-house testing as they depend on rare thread interleaving. Hence many active testing tools [20], [26], [27] have been proposed to predict and actively exercise the potentially buggy interleavings. However, most of these bug-exposing tools only target specific bug types such as data races or atomicity violations, and therefore can miss many concurrency bugs, especially the order violations [28].

Fortunately, by learning the anticipating invariants of a program, we have also inferred the "critical order" of shared-memory accesses that is assumed by programmers. Thus we can build an active testing tool to purposely promote memory-access orders that violate the learnt anticipating invariants, i.e., validating whether those critical orders are indeed enforced by the program. For example, if the belonging set of a static instruction S_2 contains only one instruction S_1, we know that S_1 should probably always be executed before S_2 at the run time. Consequently, we can intentionally stall S_1's thread before its execution to test whether S_1 is enforced to execute before S_2 by the program and check the impact of executing S_1 after S_2, if possible.

As an illustration, Fig. 9a shows a real-world order violation bug that is reported in the famous Transmission bittorrent client. Fig. 9b, which is a simplified version of the bug, illustrates the key memory accesses in this bug. Without active timing perturbation, the variable initialization at S_1 in Thread 1 almost always executes before the variable usage at S_2 in Thread 2, leading to correct execution. Thus this bug will very likely escape in-house testing and slip into production runs. Fortunately, AI will learn that $BSet(S_1) = S_2$ from these passive testing runs. Then if we apply the AI-based active testing described above to Transmission, the Thread 1 will be stalled before S_1's execution and, in the meantime, the Thread 2 is still allowed to make progress, which will inverse the common order between variable initialization (S_1) and variable usage (S_2) and hence expose the bug.

Implementing the above active testing scheme is straightforward. The only tricky issue is how to coordinate thread stalling and lock acquisition, which is illustrated in Fig. 10. In such a case, AI observes S_2 to always execute before S_5 during passive testing runs and tries to reverse this order during active testing. Unfortunately, simply stalling Thread 1 right before the execution of S_2 cannot make S_5 execute before S_2, because Thread 1 has already acquired $lock\_X$ at the moment of stalling, which makes it impossible for S_5 to execute.

In order to resolve this problem, the exposing tool needs to move the stalling up to where the lock is acquired. In the example shown in Fig. 10, Thread 1 should be stalled right before S_1, instead of S_2. In short, the AI-based active testing tool will obtain all the lock acquisition/release information through trace analysis and decide where to stall a thread accordingly.[2]

To be more clear, programmers can add an exposing phase between the training and production phases mentioned in Fig. 7. During this exposing phase, the workflow of using our algorithm is: *1)* calculating the belonging sets with the gathered traces; *2)* scanning the traces chronologically again for identifying the stalling points; *3)* adding stalling functions to the program by instrumenting; and finally *4)* executing the instrumented program for more traces and verify those traces with the invariants learnt before. Specifically, Algorithm 3 presents the algorithm of identifying stalling points, which will scan the traces chronologically and maintain the lock information of each thread correspondingly. After obtaining the stalling points, the program is instrumented once again for adding stalling function calls. In other words, the program is instrumented with not only the log functions for generating traces but also several *usleep()* calls for purposely perturbing the thread scheduling. This program will also generate a trace for each execution while its interleaving may be different from the uninstrumented ones. Thus we can scan the new traces for verifying whether the original leant invariants are still held

2. The thread is stalled at the earliest lock-acquiring site if it acquires more than one lock.

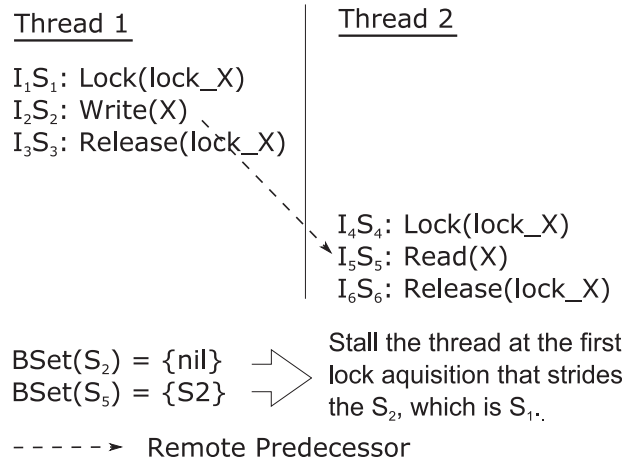


Fig. 10. Simplified code of a test interleaving involving locks.

or not. If there is an invariant violation, it indicates that some order that is preserved in normal execution is actually not enforced, which may be caused by a bug.

Algorithm 3. Pseudo code for Identifying the Stalling Points

```
Global Variable:
    recorder[M], BSet[I]: The definitions are the same
        as Algorithm 1.
    Locks[tid]: The map that maps a thread tid to the set of
        locks it acquires, initialized with ∅.
for all dynamic instance of instructions I_x S_y do
    tid := GetCurrentThreadID()
    if I_x S_y is acquiring a lock then
        Locks[tid] := Locks[tid] ∪ I_x S_y
    end if
    if BSet[S_y].size() = 1 then
        precedent := BSet[S_y][0]
        if S_y ∉ BSet[precedent] then
            first_lock = GetFirstLock(precedent)
            Record first_lock as a stalling point.
        end if
    end if
    if I_x S_y is releasing a lock then
        Locks[tid] := Locks[tid] - ins
    end if
end for
```

Moreover, we expect that the exposing phase is used as part of in-house testing, where test inputs and test oracles are provided. Thus, whether a run is a failure or success run can be determined by the test oracles. If the invariant violation happens during failure runs, our exposing phase generates bug reports to programmers. Otherwise, if the invariant violation happens during success runs, our exposing phase successfully prunes out wrong invariants before the production phase (through the relaxing tool described in Section 5.6).

4.2 Generating Emergency Patches

Additionally, AI can also be used as an emergency patch generator, Specifically, once a bug is detected or reported by the users, the programmers can apply the AITolerate pass to only the bug related instructions. The instrumented program can prevent all the future failures of this bug with only trivial overhead. Benefiting from the custom instrumentation mechanisms described in Section 3.4, the users can implement the above emergency patch very easily, and hence ease the pain caused by lengthy patch releasing period.

5 Experimental Evaluation

5.1 Test Platform, Applications, and Bugs

We analyzed AI's capability of detecting and tolerating concurrency bugs by using 35 representative real-world bugs from 11 multi-threaded applications. These applications include three widely used servers (Apache Httpd, Cherokee and MySQL), seven desktop/client applications (Mozilla, Axel, Pigz, HTTrack, Transmission, PBZip2, ZSNES) and the SPLASH-2 benchmarks [31]. As shown in Table 1, we group the found bugs into eight patterns: *1)* for atomicity violations, symbols on each side of the vertical line represent the assumed atomicity region in that thread. Thus a *R-R | W Atomicity Violation* is a bug in which two consecutive read operations in one thread are assumed to be executed atomically, but in fact they can be interleaved by a write operation from another thread. And a *R-R-W | R-R-W Atomicity Violation* occurs when two threads concurrently execute an atomic region of two read operations followed by a write operation without acquiring a lock; *2)* for order violations, the symbols represent the assumed order. For example, in a *W-R Order Violation*, the programmer intends that a write operation should always be executed before another read operation, but this intention is not guaranteed.

According to their particular conditions, we identify these bugs by a bug report ID in the software's bug database, a forum post ID, a paper/web page that describes them, or a commit ID that fixes them. Moreover, two of these bugs were never reported before but detected by our system.

We also evaluated the overhead of our software implementation with several real-world applications and the kernel programs from SPLASH-2. All these experiments were conducted on a 12-core Intel Xeon machine (2.67 GHz, 24 GB of memory) running Ubuntu-12.04.3-amd64 and using the LLVM 3.3 compiler.[3]

5.2 Detecting and Anticipating Capability

In order to evaluate whether each kind of bug can be detected or tolerated (without using roll-back) by AI and several other existing invariants (AVIO [14], DUI [15], CCI [16], and PSet [9]), we execute the corresponding buggy programs[4] under the bug-triggering input for 1,000 times and check whether the bug is detected or tolerated (random sleeps are added to increase the bug manifestation probability, following the methodology used by previous works [9], [16], [32]). Table 2 gives our results, in which ✓ represents that all the manifestations from this kind of bug are detected/tolerated in our experiments; ✓\ means that the

3. Since the compilation of MySQL requires the -fno-implicit-templates flag, which is not supported in clang++. We use llvm-g++ from LLVM 2.9 in that case.

4. Among all the 35 bugs we used, 13 of them (mainly from Mozilla) are bug kernels, which contain all bug-related code snippets extracted from the original buggy programs. We use the original programs for the experiments of the remaining 22 bugs.

TABLE 1
Evaluated Real-World Bugs

Category	Pattern	Number of bugs	Bugs		
Atomicity Violation	R-R | W	5	MySQL♯644;	MySQL♯3596;	MySQL♯12228;
	W-R | W	2	Mozilla♯341323; pigz♯9ef658babd†;	Mozilla♯224911 MySQL♯19938	
	W-W | R	5	MySQL♯791; Mozilla♯73761;	MySQL♯12848; Mozilla♯622691	Mozilla♯52111;
	R-W | R-W	7	MySQL♯56324; Apache♯21287;	MySQL♯59464; Mozilla♯342577;	Apache♯48735; Mozilla♯270689;
	R-R-W | R-R-W	3	Mozilla♯225525; Apache♯25520;	Apache♯46215;	Cherokee♯326
Order Violation	W-R	10	Axel♯313564†; Transmission♯1827;	Httrack♯20247; ZSNES♯10918;	Transmission♯1660; Mozilla♯61369;
	R-W	1	MySQL bug from paper Bugaboo [29]; FFT, LU, Barnes bug from paper LOOM [13] Pbzip2 bug from Yu's Homepage [30]		
	W-W	2	MySQL♯48930; Mozilla bug from paper Lu2008 [2]		

TABLE 2
Evaluation Results on Different Invariants' Bug Detecting and Tolerating (without Using Roll-Back) Capability

Category	Pattern	AI		AVIO		DUI		CCI		PSet	
		Detect	Tolerate	Detect	Tolerate	Detect	Tolerate	Detect	Tolerate	Detect	Tolerate
Atomicity Violation	R-R | W	✓	✓	✓		✓		✓		✓	✓
	W-R | W	✓	✓	✓		✓		✓		✓	✓
	W-W | R	✓	✓	✓		✓	✓	✓		✓	✓
	R-W | R-W	✓	✓	✓				✓		✓	
	R-R-W | R-R-W	✓	✓	✓		✓		✓		✓	
Order Violation	W-R	✓	✓			✓	✓	✓	✓	✓	✓
	R-W	✓	✓			✓		✓		✓	✓
	W-W	✓	✓					✓	✓	✓	✓

Due to space constraints, we aggregate the evaluation results of the 35 bugs into eight patterns. All results of AI in this table are obtained through experiments. And the results for other detectors are obtained based on our understanding of their algorithms.

bugs can only be detected/tolerated on particular interleavings; and the rest blank cells represent the corresponding invariant is not violated in all executions even when the bug is triggered.

Overall, AI can detect all the patterns of bugs we have found, which is more than each prior invariant. Moreover, it has a superior anticipating ability and thus can tolerate more bugs without using roll-back. In the rest of this section, we will compare AI with the prior invariants one by one.

5.2.1 AVIO

AVIO [14] invariant consists of two static instructions from one thread that should not be interleaved by an unserializable memory operation from a different thread. In AVIO, an interleaving is unserializable if the remote operation cannot be reordered out of the atomicity region without changing the result (two read operations or operations that access different locations can exchange their place without changing the result, but if they access the same variable and at least one of them is write, they cannot do this). It is an effective invariant for detecting atomicity violations.

However, *1)* as a representative of those tools that focus on detecting atomicity violations, AVIO cannot handle order violations at all; *2)* AVIO can only detect a subset of atomicity violations, because it only checks whether two consecutive memory accesses are unserializably interleaved. As an illustration, Fig. 11 shows a W-W | R Atomicity Violation given by Yu and Narayanasamy [9] that will be ignored by AVIO. In this bug, R_2's interleaving between $W_1 - R_1$ or $R_1 - W_2$ are both serializable; *3)* since AVIO invariant is checked when the second instruction is about to

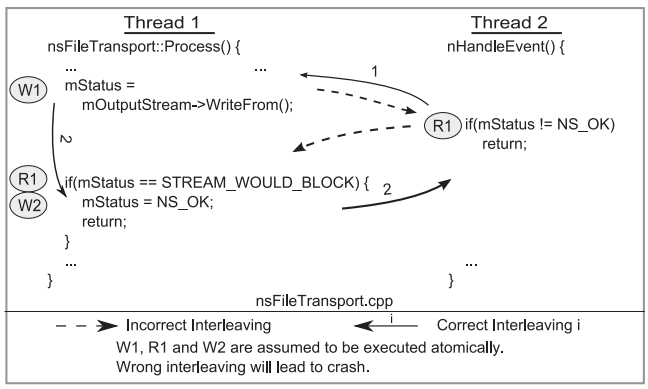


Fig. 11. An atomicity violation bug in Mozilla, which will not raise an AVIO invariant violation.

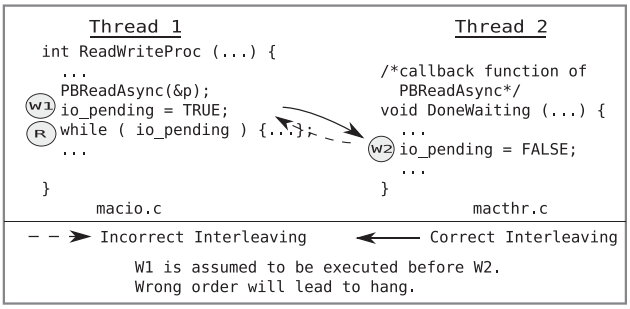


Fig. 12. A real-world W-W order violation in Mozilla nspr that cannot be detected by DUI.

be executed, it can hardly anticipate the bugs, hence is not able to be used for preventing them without using roll-back.

5.2.2 DUI

As stated by Shi et al. [15], DUI is a set of Definition-Use Invariants, which can be used to detect a large variety of software bugs including concurrency bugs (both order and atomicity violations) and sequential bugs. Specifically, *1)* DUI-LR describes the property that a local read should always read a value defined by a local or remote writer; *2)* DUI-Follower checks whether two consecutive reads from one thread must read the same value; and *3)* DUI-DSet defines a "definition set" for every read instruction, which encloses all the write instructions that the read instruction can read from.

However, since DUI concentrates on definition-use data flows which can only be undermined by unexpected write operations before reads, it cannot detect R-W | R-W Atomicity Violations (e.g., Fig. 2a). In that kind of bugs, the write operation is following the read operation, not antedating it. And it also cannot handle W-W Order Violations as shown in Fig. 12, because it is the order within write operations not the order between read and write that matters. As for R-W | R-W Atomicity Violations, although DUI can detect the most probable incorrect run shown in Fig. 13b by its DUI-Follower, it will miss the other probable incorrect interleaving (Fig. 13c).

Moreover, like AVIO, DUI-LR and DUI-Follower are checked at the last instruction of the buggy regions, thus they cannot be used to anticipate bugs. And DUI-DSet can only predict W-W | R Atomicity Violations and a subset of W-R Order Violation if there exists another leading write operation that is executed before the buggy region.

5.2.3 CCI

CCI [16], which tracks properties like "whether the last access was from the same thread" and "whether a variable has changed between two consecutive accesses from one thread", can detect both atomicity and order violations.

But limitation still remains. Since CCI does not record the exact program counters, it is relatively simpler than our Anticipating Invariant. As a double-edged sword, this simplification both promotes its efficiency and restricts its capability. For example, Fig. 14 gives a complex version of the W-R Order Violation shown in Fig. 2b. In this example, the programmer's accurate intention is that the read operation in Thread 2 should be executed after the second write operation in Thread 1. But CCI cannot tell different write operations

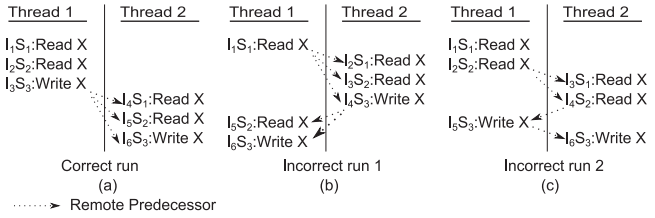


Fig. 13. The simplified code of a R-R-W | R-R-W Atomicity violation.

apart and will ignore this kind of bugs. CCI may also miss the bug shown in Fig. 2a, because it cannot differentiate it from a common benign race, where a single read operation interleaves the R-W atomicity region. In contrast, AI can correctly distinguish them by checking program counters.

Then, once again, CCI is similar to AVIO, DUI-LR and DUI-Follower in the respect that it is about whether some properties will be preserved until a later instruction. This kind of invariants is unsuitable for anticipating bugs.

5.2.4 PSet

Different from the above three invariants, PSet [9] was proposed to prevent undetected concurrency bugs from happening at the production phase, which is the same as our Anticipating Invariant. During the production runs, PSet will ensure that a memory operation M can only "immediately depend on" an instruction P that belongs to a specific set named PSet, which is established during the testing phase. Here, "immediately depend on" means that *1)* P and M should access the same memory location and at least one of them is a write operation; *2)* there is no instruction from either remote or local thread that accesses the same memory location between P and M.

Although similar to our belonging set in the format, PSet is still an invariant about data dependencies like DUI-DSet, thus it constrains that at least one of P and M should be write. As a result, it can only detect the bug after its turning point in many cases, which makes the heavy-weight roll-back mechanism indispensable. Take the R-W atomicity bug shown in Fig. 2a as an example, since PSet assumes that two consecutive read instructions do not construct any "depend on" relationship, it can only detect the bug at $Write_{Thread\,1}$, which is too late to prevent the bug without roll-back. According to their experiments [9], only 6 out of 15 bugs they have tested can be resolved by PSet without using roll-back, which are consistent with our evaluation results.

In contrast, AI does not differentiate read and write instructions. And it explicitly defines the state nil to represent the initial state, which is critical in anticipating W-R Order Violations. Therefore, as shown in Table 2, AI can prevent all the bugs we have found by merely temporarily stalling the thread.

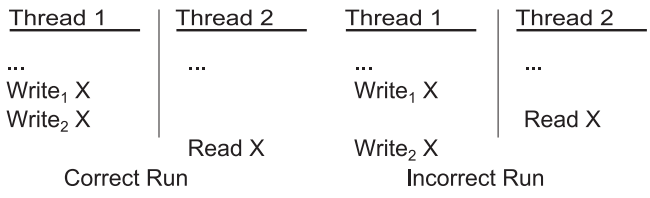


Fig. 14. A W-R order violation that CCI will ignore.

Additionally, Shi et al. [15] pointed out that PSet cannot detect the bug if it is similar to the W-R Order Violation Httrack‡20247 (Fig. 8a), because the influence of a remote operation will be blocked by local operations in PSet (caused by the second condition of PSet). This is not the case in AI.

5.2.5 Discussion

Our experiment shows that AI is able to detect and tolerate all the bugs we examined, but this does not mean that AI is the silver bullet for concurrency bugs. Currently, AI primarily aims to handle non-deadlock concurrency bugs where no more than *1)* two threads; *2)* one variable; and *3)* three memory accesses are involved. Thus AI cannot handle deadlocks, multi-variable bugs, and some high-level concurrency bugs such as linearizability violations. Park et al. [33] has concluded that the bugs within the aforementioned scope can be grouped into eight patterns. The 35 bugs we examined have covered 7 out of 8 them, which demonstrates a good representativeness. Even the neglected one, $W_1 - W_2 - W_1$, is semantically akin to our W-W | R Atomicity Violation since we can predict the bug before W_2's execution in AI (right before its turning point).

The reason that we choose to cover only these types of concurrency bugs is that we intend to keep our method lightweight and, at the same time, achieve a good coverage. According to a recent investigation [2], *1)* the manifestation of most (96 percent) concurrency bugs involves no more than two threads; *2)* only about 30 percent non-deadlock concurrency bugs access multiple variables; and *3)* almost all the single-variable concurrency bugs contain no more than three memory accesses. These investigation results conform with the theoretical deductions: the more complex a concurrency bug is; the more complex its exposing condition is; and hence the rarer it will manifest in production runs.

Moreover, AI's capability of detecting and tolerating bugs can be enhanced by integrating with other techniques. For example, AI can be easily extended to handle multi-variable bugs by leveraging the coloring technique proposed by ColorSafe [34]. Since ColorSafe is able to assign the same color to related variables, the only modification needed in AI is replacing the memory address with the color assigned to it. The bugs can be detected and tolerated without using roll-back, if the corresponding related variables are correctly colored. Since the cooperation with ColorSafe does not add any more instructions to our guard function, it should not incur additional overhead.

The other main drawback of AI is that it cannot tell different dynamic instructions apart if they are derived from the same static instruction, because it does not record any context information. As an illustration, the *allA-B* order violations described by Jin et al. [35] is a kind of bugs that: instruction B is expected to be executed after all instances of instruction A. Since AI only records the program counter of an instruction, it will allow B to be executed immediately after the first instance of A (AI can still detect this bug because the following A will find that it is un-expectedly preceded by B, but AI cannot tolerate this bug without roll-back). Theoretically, AI can be extended to handle this kind of bugs by adding context information, such as recent memory-access history [29] or call stack, to RPre. But, it is obvious that this integration will increase AI's overhead in both the runtime overhead and the time needed for achieving sufficient training, which may become non-trivial if one tries to record too much execution context. According to our experiments, recording five nearest previous access each shared memory access may increase the overhead for ten times and even higher overhead is imposed if the calling stack is recorded. Even for desktop and server applications, this may become not acceptable in practice.

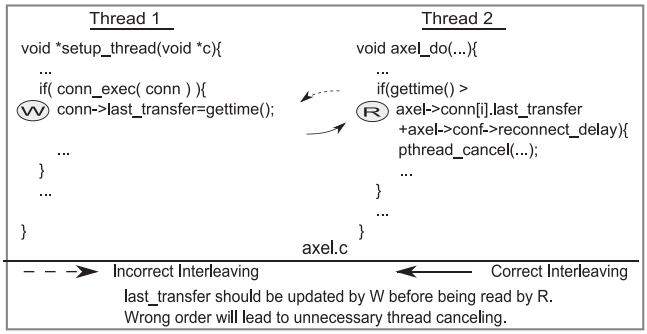


Fig. 15. An exposed order violation in Axel.

Since we use a dynamic analyzing technique to identify the shared memory accesses, our method may also miss a bug if we miss some instructions that access the same memory. However, as we will discuss later in Section 5.6, the time for achieving sufficient training is acceptable.

5.3 Exposing Capability

We have also evaluated the exposing capability of AI by using the order violations given in Table 1. According to the results, all these known order violations can be exposed by the exposing algorithm described in Section 4.1. Here, by using the word "exposed", we mean that through following the workflow described in Section 4.1 the possibility of the bug been observed (reported as a potential bug) is increased to almost 100 percent after the application is attached with our exposing tool.

Moreover, although Axel Download Accelerator and Pigz compressing tool († in Table 1) are two widely used applications, we have exposed two new bugs in them that have never been reported before. Since these two bugs are both dangerous ones that may lead to *infinite loop* and *assertion failed* respectively, both of them were confirmed by the developers and fixed in the nightly build.

Fig. 15 shows the detected order violation in Axel, in which the *last_transfer* should be updated before it is read in Thread 2. If this order is flipped, Thread 1 will be unnecessarily canceled, although it has already downloaded the current chunk. Moreover, if this order is always flipped, there will be an infinite loop. This bug has been confirmed by the developer, and fixed in the developing version by using unblocked asynchronous I/O model instead of the previous block one.

We have also detected an atomicity violation in Pigz. It is a data race in *pigz.c*, where an instruction reads a shared variable *pool→made* in *free_pool()* after releasing the corresponding mutex lock *pool→have*. This bug has also been confirmed by the developer and fixed in the developing version 2.2.5.

TABLE 3
Run-Time Overheads

Applications		Overhead	
		Default	Bias
Desktop Application	PBZip2	0.38%	-
	Pigz	0.20%	-
Server Application	Apache	0.34%	-
	MySQL	0.57%	-
SPLASH-2 Benchmarks	FFT	1,345%	115%
	LU	1,613%	127%

In this table, the "Bias" column and the "Default" column give the overhead with and without bias instrumentation respectively.

5.4 Performance

Table 3 gives the evaluation result of the performance for our current AI implementation.[5] Since we want our benchmark suit to cover different types of multi-threaded applications, we select two representatives from each category (desktop applications, server applications, and the scientific-computing kernels). These applications are chosen because they are also the choices of many previous bug detection papers [16], [27]. We compute the overhead by counting the "Total time" or "Wall Clock" field of output for the kernel programs and desktop applications, and the throughput output by testing benchmarks (httperf, supersmack) for server applications.

The applications shown in Table 3 can be roughly split into three categories. The first category includes desktop applications like PBZip2 and Pigz, which do not have many instructions that access shared variables. Hence only a little run-time overhead is imposed. The second category includes those server applications. Although they may have relatively more heap accesses, the overheads are still low, it is usually other factors, such as the I/O latencies, that obstruct these applications' performance. Applications from the SPLASH-2 benchmark suites belong to the third category, they have extremely intensive heap accesses and loops. In this case, a proper custom instrumentation scheme is critical for low overhead. As shown in the table, our general bias instrumentation scheme (with threshold 30 percent) can reduce the overhead to about 100 percent. And it will not ignore the bug listed in Table 1.

Overall, since only shared-variable accesses have to be instrumented and there is not need to roll back, AI is much faster than those existing software-only concurrency bug tolerating tools, which usually impose an impractical overhead. For example, even in terms of I/O intensive applications, the software implementation of PSet incurs more than $100\times$ overhead, which is caused by its heavyweight software roll-back implementation [9]. And, thanks to the use of static instrumentation, AI is also much more lightweight than those dynamic instrumentation based bug detection tools, such as AVIO's [14] software implementation and DUI [15], which incur $15\times - 40\times$ and $5\times - 20\times$ overhead respectively.

5. Since the overhead for desktop and server applications are low enough even when instrumenting all the shared-memory accesses, their overheads after applying bias instrumentation are omitted.

Contrary to our method, ConAir [11] takes a different approach to achieve low run-time overhead. It only aims to tolerate bugs that can be recovered by rolling back an idempotent region in one thread, which can be reexecuted for any number of times without changing the program's semantics. This policy allows ConAir to eschew the time-consuming memory-state checkpoint in general roll-back. Nevertheless, it also restricts ConAir's ability. First, ConAir cannot handle concurrency bugs that have I/O operations. A study [36] shows that about 15 percent of concurrency bugs belong to this kind. Second, an idempotent region should not contain any shared variable write. Thus ConAir is not able to tolerate W-R | W Atomicity Violations and some of R(-R)-W | R(-R)-W Atomicity Violations. Third, even some local variable writes are not idempotent, which constrains an idempotent region's length. But in ConAir, the idempotent region should both cover the whole error-propagation region to tolerate a bug, since ConAir can only affirm a bug after it has incurred some kinds of program failures.

Frost [10] is a novel technique to tolerate races. It is efficient in terms of overhead (12 percent). But it gains this efficiency on the cost of high CPU utilization ($3\times$), because it needs to run three independent instances of the program simultaneously. And it can only process data races.

5.5 Bias Instrumentation

As mentioned above, the bias instrumentation strategy that we used can decrease the overhead dramatically while still detecting and tolerating the bugs we evaluated. However, the readers may wonder whether these are just coincidences, whether there are other bugs that will be omitted by this custom instrumentation strategy. In this section, we will give a more detailed discussion about the code that we chose to not instrument.

As we can see from Table 4, a distinguishable property of HPC programs is that the distribution of the instruction groups' ins-proportion is extremely skewed. There are usually a dozen of instructions that dominate almost all the shared memory accesses. As an illustration, in FFT, two groups of instruction are omitted by the bias instrumentation. Each of them contains only 19 instructions but has an *IP* of 37.8 percent. These instructions are related to the exact fast Fourier transformation and will be executed for millions of times in an execution. As a result, they are carefully synchronized by the programmers for getting the correct result. It is hard to imagine that the program can pass in-house testing with a bug hiding in them.

The case of LU is similar, only one group of instructions is removed, which contains only 18 instructions while accounting for more than 80 percent of all the shared memory accesses. The instructions within this group are used for updating the result matrix and hence must have been checked carefully by the programmers.

However, our bias instrumentation scheme is only suitable for those programs whose distribution of memory-accesses frequency is skewed. It may not be able to filter out any distinct group for programs that use shared memory accesses only for synchronizations. Thus, we also expect that the programmers can use the APIs provided by us to design even more effective custom instrumenting strategies

TABLE 4
The Instruction Groups That Are Removed by Bias Instrumentation

Application	List of Instruction Groups (ranked by IP in descending order). For each group, we present $< IP,$ # of instructions $>$.
FFT	$<$ **37.8%**, **19** $>$, $<$ **37.8%**, **19** $>$, $<$ 5.67%, 3 $>$, $<$ 5.67%, 3 $>$, $<$ 4.13%, 179 $>$, $<$ 0.012%, 29 $>$, ...
LU	$<$ **81.9%**, **18** $>$, $<$ 11.6%, 242 $>$, $<$ 1.91%, 12 $>$, $<$ 0.014%, 21 $>$, ...

For each application, we list all the instruction groups of it that has an ins-proportion (IP) larger than 0.1 *percent. The groups whose IP are larger than* 30 *percent are removed, which are labeled in bold type.*

that further lower the overhead for that kind of CPU-intensive programs, because they are more familiar with the programs. In fact, they can manually check part of the code that is important and hot and let our tool to guard the other part of the program for them. Essentially, the custom instrumenting mechanism gives the programmers a tradeoff between overhead and coverage.

5.6 Sufficient Training

Similar to all the other invariant-based techniques, AI needs sufficient execution traces to achieve a good coverage of the possible interleavings. Although the invariants are only a compressed expression of interleaving, whose number will not grow exponentially as the number of possible interleavings, it still requires a sufficient exercise. Otherwise, there will be *1)* false positives: some correct interleavings are not observed during the training phase and hence will be classified as potential bugs in production runs, which will cause unnecessary stallings; and *2)* false negatives: certain shared-memory accesses are not identified during the training thus the concurrency bugs related to them will be omitted by AI.

In order to avoid these problems we need to gather sufficient training data so that all the possible positive/negative examples are observed and learnt by AI. In general, this requires that the programmers should both run the application under different inputs and configurations to cover all the feasible paths, and run multiple times with every input to explore different interleavings. The programmers can also use a systematic concurrency testing framework such as CTrigger [20] to systematically explore different interleavings for different inputs.

In our evaluation, we use another metrics to evaluate the complexity of training AI, which is the number of independent executions that AI needed to become converge. Here, by using the word "converge", we mean that the learnt invariants will not change even if new traces are provided. In our experiments, all the evaluated bugs are detected/tolerated and no more false positive or unnecessary stalling arises during the testing, if such a set of converged invariants is obtained. Fig. 16 shows the results of our evaluation. As we can see from the figure, small applications like PBZip2, Pigz, FFT, LU will converge in about 200 runs (i.e., 200 times of execution of each kind of inputs). As for large server applications like MySQL and Apache, less than 5,000 executions are sufficient. In other words, even for Apache, the training can be completed within several days. Comparing to the release cycle of large software (usually several months) and the fixing period of every bug (more than a month on average [37]), we think that this cost is acceptable.

Moreover, since the logging function added by the AITrace pass and the guarding function added by the AITolerate pass impose a different overhead, there may exist some interleavings that are less likely to happen in the testing phase. Thus we also provide a tool to automatically relax the BSets (i.e., reduce the false positives). It simply run the AI-guarded application and verifies the output. If the outcome is correct, the tool will relax the BSets with the generated violation report. Specifically, if a violation of "$RPre \notin BSet[S_y]$" is reported, the programmers are required to manually check the results. If it is not a bug, the corresponding BSet is relaxed by $BSet[S_y] := BSet[S_y] \cup RPre$.

After the aforementioned two steps, the applications reach to a stage we named as sufficient training. Here, by sufficient training, we mean that no false positive and false negative are observed in our experiments. Although the possibility of false positives cannot be eradicated, it will only incur a stalling timeout in our work. And the corresponding invariants can be updated immediately (used the relax mechanism described above), in order to green-light all the future runs. These stallings will never affect the program's correctness. Since the threads are randomly scheduled, a correctly synchronized program will not depend on time delays.

6 Related Work

6.1 Concurrency-Bug Detecting

Data race detection has been the subject of many works (e.g., [38], [39], [40], [41], [42], [43], [44]). In general, these works can be classified into two classes namely the lock-set approach, as in Eraser [45], and the happened-before ones, as in SigRace [46]. However, it's claimed in [14] that detecting data race is not sufficient. Because, *1)* many data races in real-world applications are benign thus race detectors may have too many false alarms; *2)* Data race free atomicity violation is prevalent and *3)* order violations usually cannot be fixed by simply using critical sections.

Due to the limitations of detecting data races only, detecting atomicity violations become a hot topic recently. A lot of

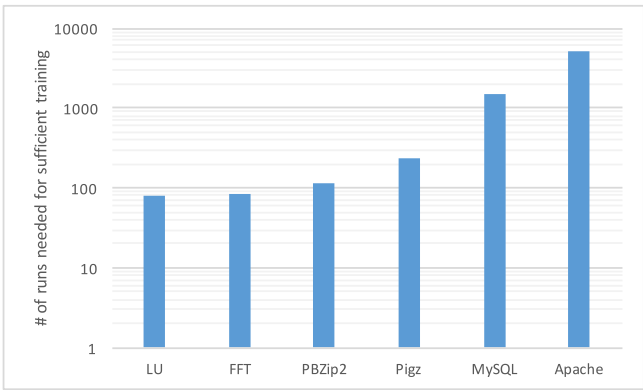


Fig. 16. The number of runs needed for achieving sufficient training.

research has been conducted on it [7], [14], [32], [47], [48], [49], [50], [51], [52], [53], [54]. Most of them attempt to detect Atomic Regions (AR) by either annotation from programmers or invariants obtained from test runs. However, the order violations has so far received much less attentions. Only a few works [15], [27], [29] deal with this kind of bugs and with limited types of bugs supported, mostly due to the limitation of the type of code constructs in which these works are embedded.

In contrast, AI can collect invariants automatically without any annotations and is capable for dealing with various types of concurrency bugs.

6.2 Concurrency-Bug Tolerating

A complementary approach to detecting and removing bugs through in-house testing is to tolerate the remaining ones during production runs. This approach is attractive for that *1)* even after extensive in-house testing, bugs have been shown to remain in the code after deployment; *2)* the misunderstanding of a reported bug may introduce new bugs; and *3)* it takes a long time between the detection of the bug and the release of a fix by the manufacturer. Therefore, bug prevention techniques have gained interests recently.

Among this category, the most closely related work of AI is PSet [9], which also proposes an invariant-based technique to tolerate both atomicity and order violations at run time. However, as shown in Section 5.2.4, PSet's ability of tolerating bugs relies heavily on roll-back, which makes it hard to be applied in production environments. Although one can substitute roll-back with the idempotent reexecution technique introduced by ConAir [11] to reduce the overhead, there will be a consequent decrease in comprehensiveness as a side-effect. As we have discussed in Section 5.4, ConAir is incapable for tolerating many kinds of bugs.

Moreover, the other concurrency bug tolerating methods like Frost [10], LifeTx [32] and AtomAid [8] are all constrained in type of bugs that they can handle, such as data races or atomicity violations. In contrast, AI can tolerate both atomicity and order violations without roll-back and incurs moderate overhead.

On the other hand, EnforceMOP [55] and Zhang and Wang [56] both propose a method that enables the programmers to specify and enforce complex properties in multithreaded programs. They can also be used tolerate concurrency bugs that are caused by unexpected interleavings. However, both of them require the programmers to manually specify the correct order either by a type-state automaton or annotations in the program. In other words, they can be used as the implementation method of AI but cannot replace AI.

6.3 Concurrency-Bug Exposing

Apart from passive bug detecting and tolerating, many techniques have been proposed to actively expose concurrency bugs [53], [57]. These interleaving testing papers use different "heuristics" to insert delay and enhance the chance of bugs' exposedness. For example, RaceFuzzer [58] and CTrigger [20] try to exercise a suspicious buggy interleaving in a real execution to verify whether it is really a bug or merely a false positive. Similar to the above two methods, Maple [59] is a new coverage-driven approach to test multithreaded programs, which achieves high efficiency by avoiding testing the same thread interleavings across different test inputs. There also exist approaches, such as ConTest [60] and PCT [61], that randomly insert delays or assign priority of threads to improve stress-testing.

Different from them, AI mainly targets on bug detection and avoidance, which is not part of these works. These works can also be used to complement AI by producing new thread interleavings for training.

In addition, as we have mentioned in Section 4.1, AI can also be extended to expose the order violations. The algorithm we proposed is similar to HaPSet [62], which uses an enhanced version of PSet to speed up systematic concurrency testing. Specifically, HaPSet leverages the property that invariants are essentially a compressed expression of interleavings and hence one of the two interleavings can be omitted if they produce the same invariant set. Thus, in HaPSet, the invariant is used as a metric of testing coverage and the target of HaPSet is covering all the possible kinds of interleavings. In contrast, our method is based on AI rather than PSet, which observes more types of ordering (e.g., the order between two read instructions). Moreover, our exposing method only purposely examines whether a certain type of order intension is guaranteed by the program. As a result, our method will consume less time (in cost of missing more bugs).

6.4 Concurrency-Bug Fixing

Finally, many solutions have been proposed to help fixing concurrency bugs. There are both dynamic methods that integrate dynamic bug detection and fixing and static methods that generate the patches offline. More specifically, Gadara [63] is a tool that can automatically fixing the deadlocks. CFix [35], on the other hand, automates the fixing procedure of both atomicity violations and order violations. The researchers have also tried using Petri-Net mechanisms to automatically fix concurrency bugs [64], [65]. And, the usages of satisfiability (SAT) solver and bounded model checker in fixing concurrency bugs have been explored by ConcBugAssist [66]. These tools can be used for repairing the program after the bug is detected by AI.

Liu et al. [67] have proposed a method for fixing linearizability violations in concurrent data structures. However, these kind of high-level violations is not targeted by AI and hence the cooperating with it is remained as future works.

7 Conclusion

This paper presents Anticipating Invariant, whose violations can anticipate bugs right before their turning points. Based on it, we implement a software-only tool that can tolerate both atomicity and order violations with a lightweight stalling strategy, instead of roll-back mechanism or chunk based execution used in prior works. Our experiment results with 35 real-world bugs of different types have shown that AI is capable of detecting and tolerating all the eight patterns of bugs we have found. In addition, AI only incurs negligible overhead ($< 1\%$) for many nontrivial desktop and server applications. And its slowdown on computation-intensive programs can be reduced to about $2\times$

after using the bias instrumentation. In contrast, prior bug tolerating tools are usually constrained in types of bugs or incurring very high (100× for PSet [9]) run-time overhead in their software-only implementation. Additionally, we also explore the usage of AI at other phases of the whole bug-handling lifecycle, such as bug exposing and patch generation. Two previously unknown concurrency bugs are exposed by our method, and they are confirmed by the corresponding developers and fixed in the nightly build.

ACKNOWLEDGMENTS

The authors from Tsinghua University are sponsored by the Natural Science Foundation of China (61433008, 61373145, 61170210, U1435216), the National Basic Research (973) Program of China (2014CB340402), National High-Tech R&D (863) Program of China (2013AA01A213), Chinese Special Project of Science and Technology (2013zx01039002-002). Shan Lu's research is partly supported by US National Science Foundation (NSF) grant CCF-1217582 and CCF-1439091. An earlier version of this work [1] appeared in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14).

REFERENCES

[1] M. Zhang, Y. Wu, S. Lu, S. Qi, J. Ren, and W. Zheng, "AI: A lightweight system for tolerating concurrency bugs," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 330–340.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2008, pp. 329–339.

[3] (2005, Feb.). Microsoft. revamping the microsoft security bulletin release process. [Online]. Available: http://www.microsoft.com/technet/security/bulletin/revsbwp.mspx.

[4] MySQL. bug report time to close stats. (2016). [Online]. Available: http://bugs.mysql.com/bugstats.php.

[5] C. Cowan, H. Hinton, C. Pu, and J. Walpole, "The cracker patch choice: An analysis of post hoc security techniques," in *Proc. Nat. Inform. Syst. Security Conf.*, 2000, pp. 16–19.

[6] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 55–64.

[7] A. Muzahid, N. Otsuki, and J. Torrellas, "Atom tracker: A comprehensive approach to atomic region inference and violation detection," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 287–297.

[8] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-aid: Detecting and surviving atomicity violations," in *Proc. 35th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 277–288.

[9] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *Proc. 36th Annu. Int. Symp. Compu. Archit.*, 2009, pp. 325–336.

[10] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and surviving data races using complementary schedules," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 369–384.

[11] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam, "ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 113–126.

[12] B. Lucia and L. Ceze, "Cooperative empirical failure avoidance for multithreaded programs," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 39–50.

[13] J. Wu, H. Cui, and J. Yang, "Bypassing races in live applications with execution filters," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 1–13.

[14] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2006, pp. 37–48.

[15] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng, "Do I use the wrong definition?: DeFuse: Definition-use invariants for detecting concurrency and sequential bugs," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2010, pp. 160–174.

[16] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for cooperative concurrency bug isolation," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2010, pp. 241–255.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.: Feedback-Directed Runtime Optim.*, 2004, p. 75.

[19] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 446–455.

[20] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2009, pp. 25–36.

[21] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 31–44.

[22] M. Hauswirth and T. M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," in *Proc. 11th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2004, pp. 156–164.

[23] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective sampling for lightweight data-race detection," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, vol. 44, no. 6, 2009, pp. 134–143.

[24] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with LLVM compiler," in *Proc. 2nd Int. Conf. Runtime Verification*, 2012, pp. 110–114.

[25] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen, *Introduction to Algorithms*. Cambridge, MA, USA: MIT press, 2001.

[26] K. Sen, "Race directed random testing of concurrent programs," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2008, pp. 11–21.

[27] W. Zhang, C. Sun, and S. Lu, "ConMem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proc. 15th Ed. ASPLOS Archit. Support Program. Lang. Operating Syst.*, 2010, pp. 179–192.

[28] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2012, pp. 485–502.

[29] B. Lucia and L. Ceze, "Finding concurrency bugs with context-aware communication graphs," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 553–563.

[30] (2016). [Online]. Available: http://web.eecs.umich.edu/~jieyu/bugs.html

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit*, 1995, pp. 24–36.

[32] J. Yu and S. Narayanasamy, "Tolerating concurrency bugs using transactions as lifeguards," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 263–274.

[33] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *Proc. 32nd ACM/IEEE Int. Conf. Softw Eng.* - vol. 1, 2010, pp. 245–254.

[34] B. Lucia, L. Ceze, and K. Strauss, "ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 222–233.

[35] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 221–236.

[36] H. Volos, A. J. Tack, M. M. Swift, and S. Lu, "Applying transactional memory to concurrency bugs," in *Proc. 17th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2012, pp. 211–222.

[37] Mysql bugs: Statistics. (2016). [Online]. Available: http://bugs.mysql.com/bugstats.php.

[38] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2009, pp. 121–133.

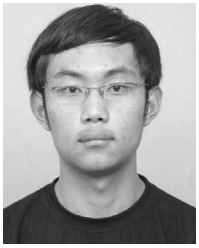
[39] S. L. Min and J.-D. Choi, "An efficient cache-based access anomaly detection scheme," in *Proc. 4th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 1991, pp. 235–244.
[40] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 22–31.
[41] R. H. B. Netzer and B. P. Miller, "Improving the accuracy of data race detection," in *Proc. 3rd ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 1991, pp. 133–144.
[42] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient detection of data race conditions via adaptive tracking," in *Proc. 20th ACM Symp. Operating Syst. Principles*, 2005, pp. 221–234.
[43] D. Li, W. Srisa-an, and M. B. Dwyer, "SOS: Saving time in dynamic race detection with stationary analysis," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Programm. Syst. Lang. Appl.*, 2011, pp. 35–50.
[44] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient data race detection for distributed memory parallel programs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 51:1–51:12.
[45] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 27–37, Nov. 1997.
[46] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas, "SigRace: Signature-based data race detection," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 337–348.
[47] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Proc. 38th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2004, pp. 256–267.
[48] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2003, pp. 338–349.
[49] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, "Automated type-based analysis of data races and atomicity," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2005, pp. 83–94.
[50] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatically detecting memory-related bugs via program counter-based invariants," in *Proc. 37th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2004, pp. 269–280.
[51] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 1–14.
[52] J. Burnim, G. Necula, and K. Sen, "Specifying and checking semantic atomicity for multithreaded programs," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2011, pp. 79–90.
[53] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 135–145.
[54] J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 144–154.
[55] Q. Luo and G. Roşu, "Enforcemop: A runtime property enforcement system for multithreaded programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 156–166.
[56] L. Zhang and C. Wang, "Runtime prevention of concurrency related type-state violations in multithreaded applications," in *Proc Int. Symp. Softw. Testing Anal.*, 2014, pp. 1–12.
[57] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 267–280.
[58] K. Sen, "Race directed random testing of concurrent programs," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2008, pp. 11–21.
[59] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proc. ACM Int. Conf. Object Oriented Programming Syst. Lang. Appl.*, 2012, pp. 485–502.
[60] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation," *IBM Syst. J.*, vol. 41, no. 1, pp. 111–125, 2002.
[61] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proc. 15th Ed. ASPLOS Archit. Support Program. Lang. Operating Syst.*, 2010, pp. 167–178.
[62] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 221–230.
[63] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke, "Gadara: Dynamic deadlock avoidance for multithreaded programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 281–294.
[64] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 299–309.
[65] P. Liu, O. Tripp, and C. Zhang, "Grail: Context-aware fixing of concurrency bugs," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 318–329.
[66] S. Khoshnood, M. Kusano, and C. Wang, "Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 165–176.
[67] P. Liu, O. Tripp, and X. Zhang, "Flint: Fixing linearizability violations," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. & Appl.*, 2014, pp. 543–560.

Mingxing Zhang received the BE degree from the Beijing University of Posts and Telecommunications, China, in 2012. He is currently working toward the PhD degree in the Department of Computer Science and Technology, Tsinghua University, China. His research interests include parallel and distributed systems. He can be reached at: zhangmx12@mails.tsinghua.edu.cn.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences in 2002. He is currently a professor in computer science and technology at the Tsinghua University of China. His research interests include parallel and distributed processing, and cloud storage. He has published more than 80 research publications and has received two Best Paper Awards. He is currently on the editorial board of the *International Journal of Networked and Distributed Computing* and *Communication of China Computer Federation*. He is a member of the IEEE. He can be reached at: wuyw@tsinghua.edu.cn.

Shan Lu received the PhD degree from the University of Illinois at Urbana Champaign. She is an associate professor in the Computer Science Department of the University of Chicago. Her research focuses on building tools to help improve software reliability and efficiency.

Shanxiang Qi received the BS degree from Tsinghua University and the PhD degree from the University of Illinois at Urbana-Champaign, both in computer science. He is a software engineer at Uber working on China growth. Prior to Uber, Shanxiang was a software engineer in Google working on knowledge graph.



Jinglei Ren received the BE degree from Northeast Normal University, China, in 2010 and is currently working toward the PhD degree in the Department of Computer Science and Technology, Tsinghua University, China. His research interests include distributed systems, storage techniques, and operating systems. He has developed a storage system for virtual machines with enhanced manageability, and is currently working on the flash-aware and energy-efficient smartphone filesystem.

Weimin Zheng received the BS and MS degrees in 1970 and 1982, respectively, from Tsinghua University, China, where he is currently a professor of computer science and technology. He is the research director of the Institute of High Performance Computing at Tsinghua University, and the managing director of the Chinese Computer Society. His research interests include computer architecture, operating system, storage networks, and distributed computing. He is a member of the IEEE.