AI: A Lightweight System for Tolerating Concurrency Bugs

Mingxing Zhang[1] Yongwei Wu[1] Shan Lu[2,*] Shanxiang Qi[3,†]
Jinglei Ren[1] Weimin Zheng[1]

[1]Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China
[2]University of Wisconsin-Madison, USA [3]University of Illinois at Urbana-Champaign, USA

ABSTRACT

Concurrency bugs are notoriously difficult to eradicate during software testing because of their non-deterministic nature. Moreover, fixing concurrency bugs is time-consuming and error-prone. Thus, tolerating concurrency bugs during production runs is an attractive complementary approach to bug detection and testing. Unfortunately, existing bug-tolerating tools are usually either *1)* constrained in types of bugs they can handle or *2)* requiring roll-back mechanism, which can hitherto not be fully achieved efficiently without hardware supports.

This paper presents a novel program invariant, called Anticipating Invariant (AI), which can help anticipate bugs before any irreversible changes are made. Benefiting from this ability of anticipating bugs beforehand, our software-only system is able to forestall the 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 AI is capable of detecting and tolerating most types of concurrency bugs, including both atomicity and order violations. Two new bugs have been detected and confirmed by the corresponding developers. Performance evaluation with 6 representative parallel programs shows that AI incurs negligible overhead ($< 1\%$) for many nontrivial desktop and server applications.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*

General Terms

Reliability

Keywords

Concurrency Bugs, Software Reliability, Bug Tolerating

*Shan is now with University of Chicago.
†Shanxiang is now with Google Inc., Mountain View.

FSE '14, November 16–22, 2014, Hong Kong, China

1. INTRODUCTION

1.1 Motivation

Unlike sequential bugs, the manifestation of concurrency bugs depends not only on inputs, but also on thread interleavings and other timing-related events. Thus it is hard to expose all the concurrency bugs during in-house testing, and it is also difficult to fix concurrency bugs quickly [2] and correctly [6]. Consequently, tools that can not only detect concurrency bugs during in-house testing but also tolerate them during production runs are highly desired.

An ideal production-run bug-tolerating tool should satisfy requirements from two aspects: *1)* bug-tolerating **coverage**. The tool should be able to handle a wide variety of concurrency bugs that are hidden in deployed applications, including both atomicity violations and order violations[1]; and *2)* run-time **performance**. The tool should only incur small overhead on commodity machines.

Existing techniques that tolerate concurrency bugs can be categorized into three types, depending on when bug-tolerating takes effect (Figure 1). But none of them can satisfy the above two requirements simultaneously.

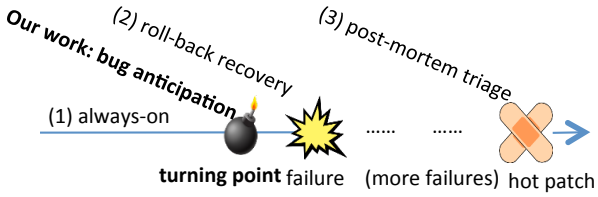

Figure 1: Categorization of bug-tolerating tools. ***The arrow in the figure represents the time-line during production runs.***

1) Always-on approach. This approach constrains program execution *all the time* to prevent potential manifestations of some concurrency bugs. For example, Atomtracker [21] and AtomAid [17] group instructions into chunks and execute every chunk atomically. This approach constrains interleavings even during correct runs and relies on transactional memory or other custom hardware to achieve good performance. In addition, it often only probabilistically tolerates concurrency bugs with a specific root cause pattern (e.g., atomicity violations), and unable to handle bugs with other types of root causes (e.g., order violations).

2) Failure-recovery approach. This approach [35, 30] rolls back program execution to a recent checkpoint *when failures or errors*

[1]These are the two most common types of concurrency bugs in real world based on a previous empirical study [12].

occur, and relies on re-execution for automated recovery. Unfortunately, low-overhead checkpoint and roll-back cannot be achieved on existing machines without significantly sacrificing the bug-tolerating coverage [37].

3) Post-mortem approach. This approach [14, 34] aims to prevent future manifestations of a concurrency bug, *after triaging* earlier manifestations of the bug. It 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

This paper proposes a new approach to tolerating 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 and preventing bug manifestation through temporarily stalling the execution of one thread, which incurs much smaller overhead than checkpointing and rollback.

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 is non-deterministic; after t, the manifestation becomes deterministic. 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.

Anticipating bugs right before the turning point is critical to bug tolerating. Anticipating too early will inevitably encounter many false positives, causing unnecessary thread stalling and performance losses. Anticipating too late will miss the chance of lightweight bug toleration — only heavy weight checkpoint-rollback can restore correct states after the turning points.

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, which we will discuss in more details in Section 4.2. Below, we simply demonstrate how two straw-man ideas do not work for bug anticipation.

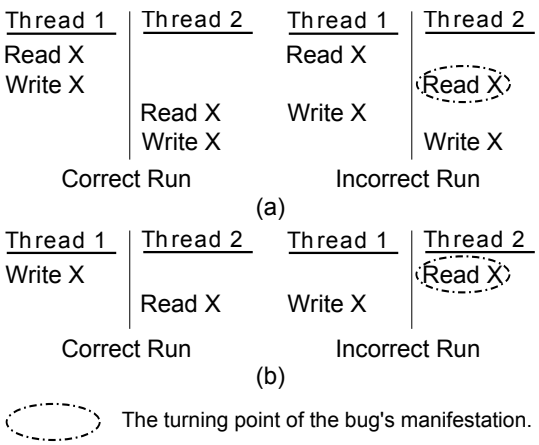

Figure 2: Illustrations of bugs' turning points.

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. Figure 2 (a) shows a typical atomicity violation pattern, where the expected atomicity of write-after-read is violated. Many real-world concurrency bugs follow this pattern [12]. Here, the turning point is actually right before the second read instruction, as circled in Figure 2 (a). 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. Figure 2 (b) illustrates a typical order violation pattern, where a read in thread 2 unexpectedly executes before a write in thread 1. Many real-world concurrency bugs follow this pattern and lead to problems like un-initialized reads [12]. The turning point in this example is right before the read in Thread 2, as circled in Figure 2 (b). 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.

i) A new approach to tolerating production-run concurrency bugs. This new approach complements existing bug tolerating approaches by anticipating concurrency bugs right before their turning points and leveraging lightweight thread-stalling, instead of heavyweight checkpoint-rollback, to get around the bugs' manifestation.

ii) A novel invariant, named Anticipating Invariant (AI), that is suitable for effective and efficient concurrency-bug toleration. Roughly speaking, for an instruction I that accesses variable V, AI captures which instructions are allowed to access V right before I from a different thread. What distinguishes AI from previously proposed interleaving invariants [13, 28, 7, 35] is that AI can help achieve both the coverage goal and the performance 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 performance, 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.

iii) A low-overhead software-only bug-tolerating system[2], designed and implemented based on AI . Our system includes several steps: it first automatically learns AI during in-house testing; it then monitors violations to AI during 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 optional bias instrumentation scheme and APIs to allow easy performance tuning for memory-access intensive applications. More details are presented in Section 3.

iv) An evaluation based on 35 representatives real-world concurrency bugs. The evaluation shows 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% overhead for many non-trivial desktop and server applications. Furthermore, our system detected two previously unknown concurrency bugs from widely used open-source software.

2. ANTICIPATING INVARIANT

Program invariants are predicates that should always be true at certain points of the execution; they 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 [5, 28, 7]. These invariants are supposed to be held in all the

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

correct runs. If one of them is violated at run time, a bug probably has manifested. 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 this section, we first introduce the Anticipating Invariant (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 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 (i.e., the execution of I_2 is the turning point).

For example, most order violations occur when an instruction I_1 from Thread 1 unexpectedly executes after instruction I_2 from Thread 2, causing I_2 to be preceded by a different instruction that accesses the same variable as I_2. Postponing I_2 can effectively make I_1 execute before I_2. As another example, most atomicity violations occur when instruction I_2 from Thread 2 unexpectedly interleaves instruction I_1 and I_3 from Thread 1, causing I_2 to be unexpectedly preceded by I_1. Similar to the order violations, postponing I_2 can effectively prevent this kind of atomicity violations.

Following this 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 (a line of code that can be differentiated by its program counter), and 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. We also 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 same address *immediately before* I_x. 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, Figure 3 shows an interleaving and each instruction's corresponding remote predecessor. 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.

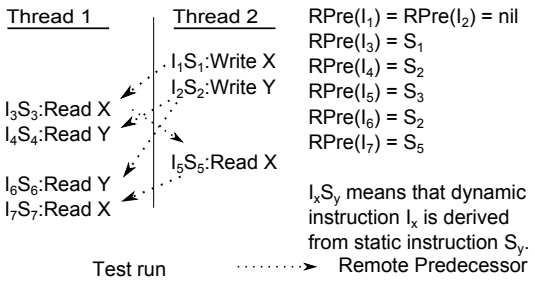

Figure 3: Demonstration of remote predecessors.

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 different remote predecessors that are needed to be calculated;

2) nil 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) Whether the instruction is a read or a write operation does not matter in calculating remote predecessors.

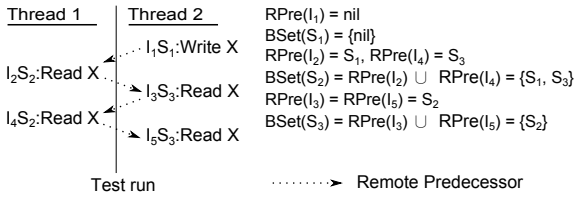

Figure 4: Demonstration of belonging sets.

Figure 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 S_1 and S_3 respectively.

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 Figure 4 is calculated as $BSet(S_2) = RPre(I_2) \cup RPre(I_4) = \{S_1, S_3\}$.

2.2 Case Studies

2.2.1 Atomicity Violation

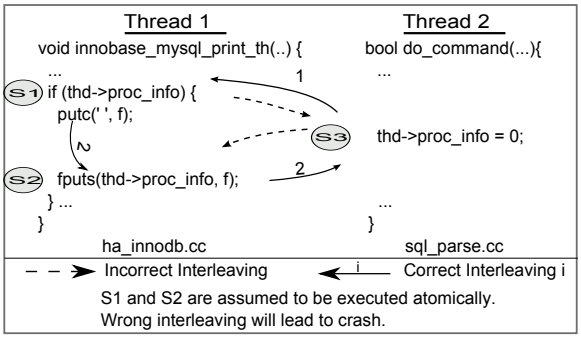

Figure 5: A real-world atomicity violation in MySQL.

Figure 5 shows a real-world atomicity violation from the MySQL database server, while Figure 6 is the corresponding simplified code

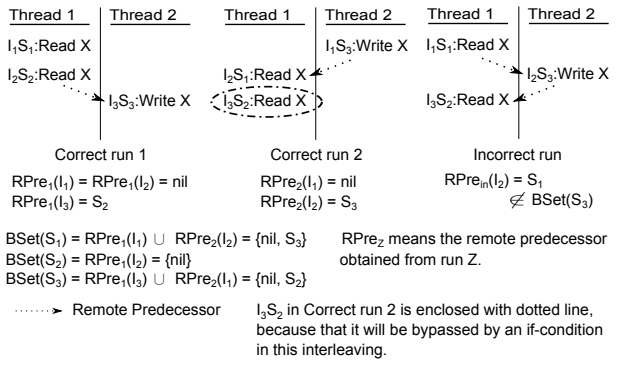

Figure 6: The simplified code of Figure 5.

of this bug. Two interleavings that can be found in correct test runs and one found in incorrect runs are given in Figure 6.

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 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 Figure 2 (a). In the incorrect run, AI can anticipate the bug before $Read_{Thread\ 2}$, because $Read_{Thread\ 1}$ does not belong to its $BSet$.

Previous works have proposed different types of invariants to detect atomicity violations ultimately [13, 28, 7, 35]. However, they cannot predict many atomicity violations before their turning points. We will discuss this in more details in Section 4.2.

2.2.2 *Order Violation*

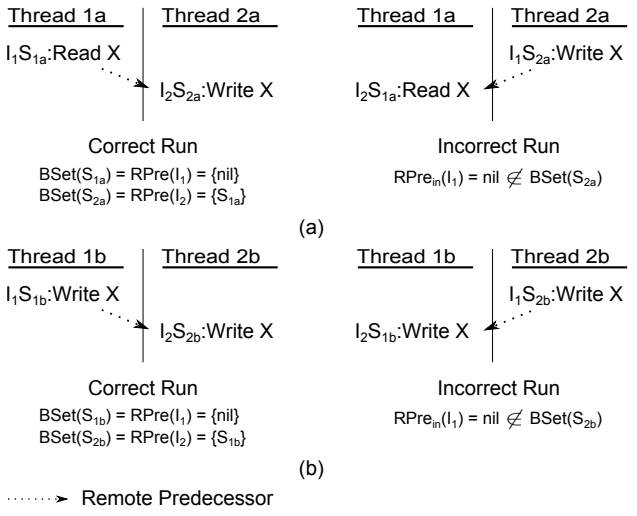

Figure 7: Interleavings of two typical order violations.

Figure 7 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 the Anticipating Invariant, previous works' [28, 7, 35] 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 4.2.

2.3 Rationales

It's true that most of the invariant-based techniques share the same formation of learning some invariants from testing runs and checking/guarding them later. But AI can be distinguished from the others by its ability of avoiding roll-back, which shifts the invariant designer's perspective from "how to detect the bug ultimately" to "how to anticipate the bug right before its turning point". Owing to this unique perspective, many special decisions are made:

1) Many previous works 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, and it explicitly defines nil to represent the initial state. This helps to avoid the first and the second misconceptions listed in Section 1.1 respectively. *3)* 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 [13], the read-write dependence like DUI [28], or the memory-dependent like PSet [35], etc. Thus, as shown in Section 4.2, AI can detect more bugs than each of the previous works can do.

3. IMPLEMENTATION

In order to utilize the Anticipating Invariant for tolerating concurrency bugs, we implement a software-only system by using the LLVM compiler framework [10]. In this section, we first give an overview of our system. Then, we describe how to automatically extract AI and use it to tolerate 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 consisting 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 usages. If the user also designs a custom instrumentation strategy with our API, a corresponding white list file will also be generated (elaborated in Section 3.3).

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

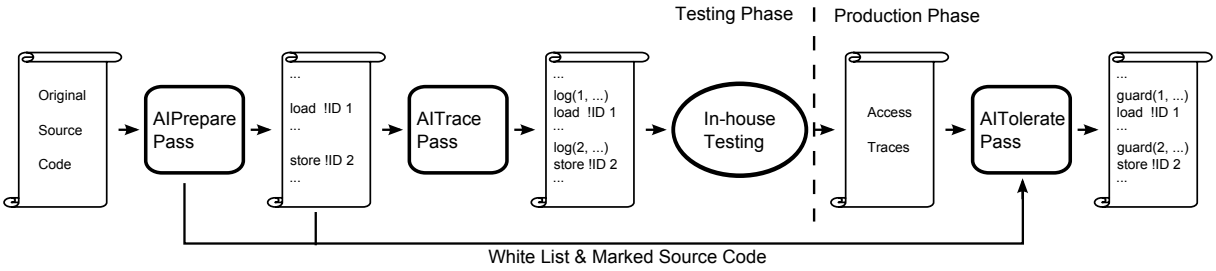

Figure 8: Overview of the whole system.

version of code. The generated code is compiled to executable objects and used in production runs.

3.2 Training & Tolerating

In order to infer AI automatically without any programmers' annotation, we rely on correct runs observed during the in-house testing phase. As pointed out by prior work [35], 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 [24] can also be used to systematically explore different interleavings for each input.

As described above, the trace files 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 as stated before. In the meantime, the belonging sets are updated as below:

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

After scanning all the traces 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. It will also add a guarding function before every shared-memory accesses[3] to perform bug tolerating.

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. Then after obtaining $RPre$, the guarding function will check whether the corresponding Anticipating Invariant is held. 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. As analyzed in Section 2.2, thanks to AI's capability of anticipating bugs before their occurrence, we can tolerate this violation by stalling the violating thread until the violation gets resolved. 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 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. 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

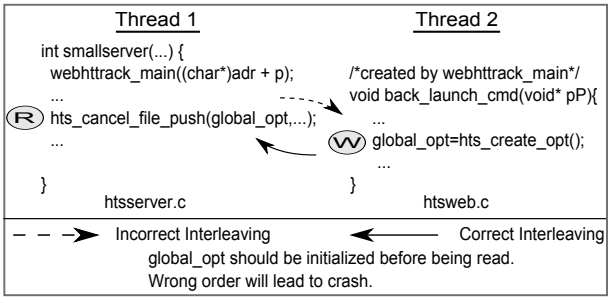

Figure 9: A real-world W-R order violation in HTTrack, which cannot be detected by PSet.

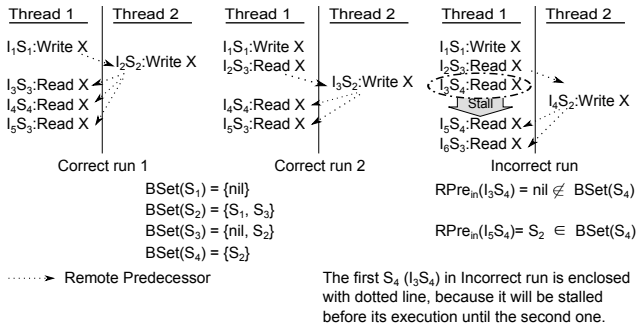

Figure 10: The simplified code of Figure 9.

Figure 10 shows the simplified version of Figure 9, 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. [28], since S_2 may inject between S_1 and S_3's execution in some cases (like correct run 2), PSet cannot detect this bug. But the remote predecessor of S_4 is always S_2 in both two correct test runs. Thus $BSet(S_4) = \{S_2\}$, which makes S_4 impossible to be executed before S_2 in production runs. More details about this violation in the incorrect run can be found in Figure 10.

[3]A shared variable will be accessed by at least one static instruction S that satisfies the property: $BSet(S) - \{nil\} \neq \emptyset$. Thus we can identify shared-memory accesses during the training while inferring AI .

Algorithm 1 Pseudocode of the Guarding Function.

```
Global Variable:
    recorder[M]: The map that maps a memory address M to
                 the last two instructions that operate address M
                 and are from different threads
    BSet[I]: The map that maps an instruction I to its
             Belonging Set

function Guard(Dynamic Instruction ins, Memory Address m)
    iter := 0; id := GetAccessID(ins)
    tid := GetCurrentThreadID()
    while iter < threshold do
        RPre := GetRPre(ins)
        if RPre ∉ BSet[id] then
            Report an violation
            Stall this Thread for a while
        else
            Break
        iter := iter + 1
        if iter = threshold then
            Report an unresolved violation
    UpdateRecorder(recorder[m], id, tid)
```

In summary, the pseudocode of guarding function is given in Algorithm 1. It should be noted that, since we implement AI with per-variable synchronization, the horizontal scalability of the original program will likely not to be constrained by AI .

3.3 Custom Instrumentation Strategy

Since only shared-variable accesses have to be instrumented and there is not 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. [9] 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.

Moreover, we propose an optional bias instrumentation scheme, which is effective for the aforementioned HPC programs. The scheme is based on a key observation that, in a well-tested program, bugs usually occur in cold (less-executed) regions. Thus if an access is deemed to be very "hot", we chose to not instrument it. We expect this scheme to miss few harmful bugs in practice, because, if a bug is lurking in these instructions, it will probably be found by the in-house testing (or is benign). Similar approaches have been used in several sampling-based race detecting methods [18], and achieve a good result in Google's practice [27].

As for 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 address. (This can be achieved by using a greedy algorithm that merges two instructions that have accessed a same address into the same group with a disjoint-set [11].) Then 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 (by counting the trace files). Finally, if a specific flag is set when applying the AITolerate pass, those groups whose IP is larger than a threshold (30% is used in our experiments) will not be instrumented. Other metrics, such as IP of each instruction, can also be used to identify the "hot" instructions.

Additionally, There is another usage of custom instrumentation strategies. Once a bug is detected or reported by others, the users can choose to only instrument the bug related instructions to prevent all the future failures. This can help to ease the pain caused by lengthy patch releasing period, just like Aviso [14] and Loom [34].

Generally speaking, the users can implement their own instrumentation strategies very conveniently by directly modifying the generated BSets. But 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 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. EXPERIMENTAL EVALUATION

4.1 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 [33]. 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 (2.67GHz, 24GB of memory) running Ubuntu-12.04.3-amd64 and using the LLVM 3.3 compiler[4].

[4]Since the compilation of MySQL requires the -fno-implicit-

Table 1: Evaluated real-world bugs.

Category	Pattern	Number of bugs	Bugs
Atomicity Violation	R-R | W	5	MySQL♯644; MySQL♯3596; MySQL♯12228; Mozilla♯341323; Mozilla♯224911
	W-R | W	2	pigz♯9ef658babd†; MySQL♯19938
	W-W | R	5	MySQL♯791; MySQL♯12848; Mozilla♯52111; Mozilla♯73761; Mozilla♯622691
	R-W | R-W	7	MySQL♯56324; MySQL♯59464; Apache♯48735; Apache♯21287; Mozilla♯342577; Mozilla♯270689; Mozilla♯225525;
	R-R-W | R-R-W	3	Apache♯25520; Apache♯46215; Cherokee♯326
Order Violation	W-R	10	Axel♯313564†; Httrack♯20247; Transmission♯1660; Transmission♯1827; ZSNES♯10918; Mozilla♯61369; MySQL bug from paper Bugaboo [15]; FFT, LU, Barnes bug from paper LOOM [34]
	R-W	1	Pbzip2 bug from Yu's Homepage [1]
	W-W	2	MySQL♯48930; Mozilla bug from paper Lu2008 [12]

Table 2: Evaluation results on different invariants' bug detecting and tolerating (without using roll-back) capability. ***Due to space constraints, we aggregate the evaluation results of the 35 bugs into 8 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.***

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	✓	✓					✓	✓	✓	✓

4.2 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 [13], DUI [28], CCI [7], and PSet [35]), we execute the corresponding buggy programs[5] 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 [7, 35, 36]). 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 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.

templates flag, which is not supported in clang++. We use llvm-g++ from LLVM 2.9 in that case.

[5]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.

4.2.1 AVIO

AVIO [13] invariant is consisting of two static instructions from one thread that should not be interleaved by an unserializable memory operation from a different thread. In which, 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, Figure 11 shows a W-W | R Atomicity Violation given by Yu et al. [35] 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 be executed, it can hardly anticipate the bugs, hence is not able to be used for preventing them without using roll-back.

4.2.2 DUI

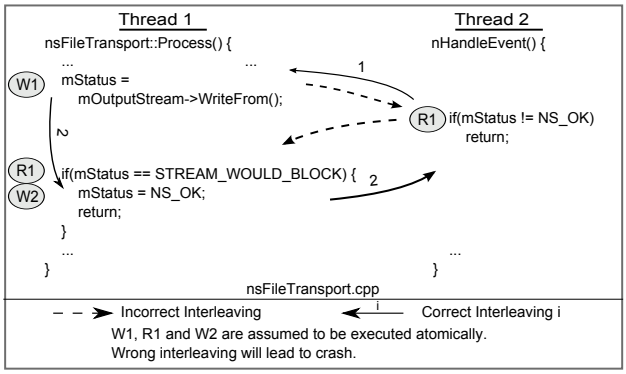

Figure 11: An atomicity violation bug in Mozilla, which will not raise an AVIO invariant violation.

As stated by Shi et al. [28], 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.

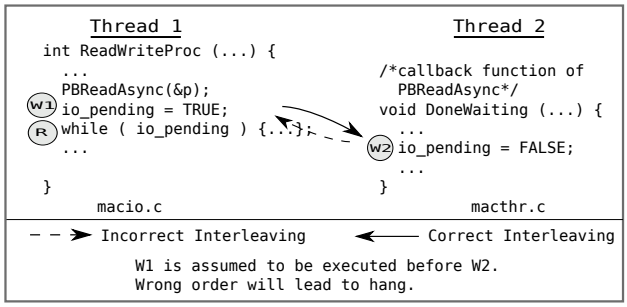

Figure 12: A real-world W-W order violation in Mozilla nspr that cannot be detected by DUI.

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. Figure 2 a). 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 Figure 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 Figure 13 (b) by its DUI-Follower, it will miss the other probable incorrect interleaving (Figure 13 c).

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.

4.2.3 CCI

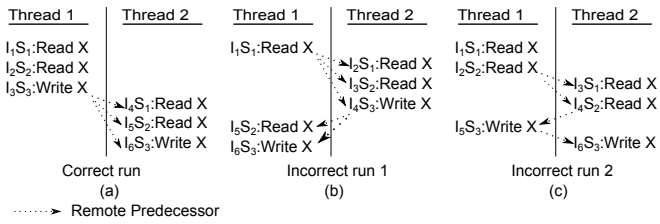

Figure 13: The simplified code of a R-R-W | R-R-W Atomicity Violation

CCI [7], 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.

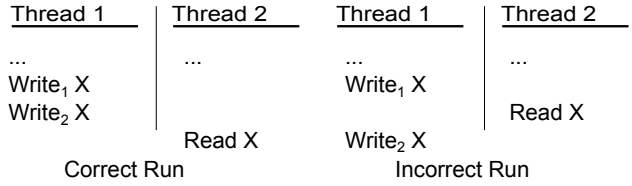

Figure 14: A W-R Order Violation that CCI will ignore.

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, Figure 14 gives a complex version of the W-R Order Violation shown in Figure 2 (b). In which 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 apart and will ignore this kind of bugs. CCI may also miss the bug shown in Figure 2 (a), 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.

4.2.4 PSet

Different from the above 3 invariants, PSet [35] 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 Figure 2 (a) 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 [35], 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.

Additionally, Shi et al. [28] pointed out that PSet cannot detect the bug if it is similar to the W-R Order Violation Httrack♯20247 (Figure 9), 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 .

4.2.5 Discussion

The 35 bugs we used have covered 7 out of 8 order and atomicity violation patterns concluded by Park et al. [25], which demonstrates a good representativeness. And 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). But AI's ability of detecting and tolerating bugs can still 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 [16]. 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.

And, AI cannot detect a synthetic bug described in Bugaboo [15]: in which two threads' repeated accesses to a shared variable must be interleaved. But no code constraint enforces the interleaving. This is caused by that fact that, without recording any context information, AI cannot tell different dynamic instructions apart if they are derived from the same static instruction. However, 1) Programmers usually pay more attentions on these kinds of complicate synchronizations (as a matter of fact, we do not find any real-world example belongs to this type); and 2) AI can integrate with Bugaboo easily by adding context information to RPre.

4.3 New Bugs

Although Axel Download Accelerator and Pigz compressing tool († in Table 1) are both widely used, we detected two new bugs in them. Since they are both dangerous bugs that may lead to *infinite loop* and *assertion failed* respectively, both of them were confirmed by the developers and fixed in the nightly build.

Figure 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.

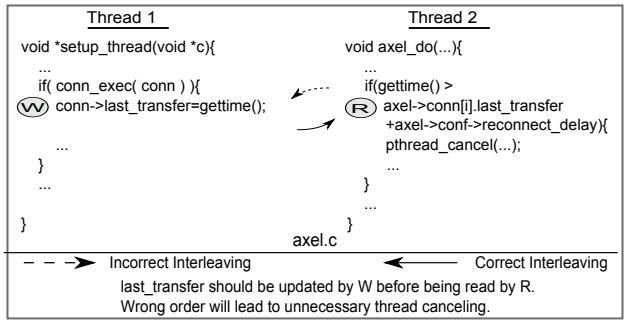

Figure 15: A detected Order Violation in Axel.

4.4 Performance

Table 3 gives the evaluation result of the performance for our current AI implementation[6]. Since we want our benchmark suit to cover different types of multi-threaded applications, we select two representatives from each categoy (desktop applications, server applications, and the scientific-computing kernels). These applications are chosed because they are also the choices of many previous bug detection papers [7, 38]. 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, super-smack) for server applications.

Table 3: Run-time overheads. In this table, the "Bias" column and the "Default" column give the overhead with and without bias instrumentation respectively.

Applications		Overhead	
		Default	Bias
Desktop	PBZip2	0.38%	-
Application	Pigz	0.20%	-
Server	Apache	0.34%	-
Application	MySQL	0.57%	-
SPLASH-2	FFT	1345%	115%
Benchmarks	LU	1613%	127%

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, because it is usually other factors, such as the I/O latencies, that obstructs 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%) can reduce the overhead to about 100%. And it will not ignore the bug listed in Table 1. 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%. Since they are the updating operations to the result array, it is hard

[6]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.

to imagine that the program can pass a comprehensive stress testing with a bug hiding in them. We also expect that the programmers can use the APIs provided by us to design even more effective custom instrumenting strategies that further lower the overhead for these CPU-intensive programs.

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 [35]. 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 [13] software implementation and DUI [28], which incur $15 \times -40\times$ and $5 \times -20\times$ overhead respectively.

Contrary to our method, ConAir [37] 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 [31] shows that about 15% 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 [30] is a novel technique to tolerate races. It is efficient in terms of overhead (12%). 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.

4.5 Sufficient Training

Similar to all the other invariant-based techniques, AI needs sufficient tested execution traces to achieve a good coverage. There may be false negatives (i.e., fail to predict some bugs) if some shared-memory accesses are not identified during training; and there may be false positives (i.e., unnecessary stallings) if some important correct interleavings are not covered during training. 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. And 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. The programmers can also use a systematic concurrency testing framework such as CTrigger [24] to systematically explore different interleavings for different inputs.

In our evaluation, the numbers of execution needed for sufficient training (i.e. all the bugs are detected/tolerated and no more false positive or unnecessary stalling arises) are about 200 for PBZip2, Pigz, FFT, LU; about $1,000$ for MySQL; and about $5,000$ for Apache. Even for Apache, the training can be completed within half a day. Comparing to the release cycle of large software (usually several months) and the fixing period of every bug (more than a month on average [3]), we think the cost is acceptable.

Moreover, 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, 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.

5. RELATED WORK

The most closely related work of AI is PSet [35] that also proposes an invariant-based technique to tolerate both atomicity and order violations at run time. However, as shown in Section 4.2.4, 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 [37] to reduce the overhead, there will be a consequent decrease in comprehensiveness as a side-effect. As we have discussed in Section 4.4, ConAir is incapable of tolerating many kinds of bugs. And the other concurrency bug tolerating methods like Frost [30], LifeTx [36] and AtomAid [17] 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.

As a complementary approach to concurrency bug detecting and tolerating, several methods have recently been proposed to expose concurrency bugs during software testing [20, 23, 22, 29, 8, 32]. These interleaving testing papers use different "heuristics" to insert delays and enhance the chance of bugs' exposedness. For example, RaceFuzzer [26] and CTrigger [24] try to exercise a suspicious buggy interleaving in a real execution to verify whether it is really a bug or merely a false positive. There also exist approaches, such as PCT [4], that randomly insert delays or assign priority of threads to improve stress testing. Apart from them, AI targets on bug avoidance, which is not part of these works. And these works can be used to complement AI by producing new thread interleavings for training.

6. 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.

7. ACKNOWLEDGMENTS

The authors from Tsinghua University are sponsored by the National Basic Research (973) Program of China (2011CB302505), Natural Science Foundation of China (61373145, 61170210), National High-Tech R&D (863) Program of China (2012AA012600), Chinese Special Project of Science and Technology (2013zx01039-002-002). Shan Lu's research is partly supported by NSF grant CCF-1217582.

8. REFERENCES

[1] http://web.eecs.umich.edu/∼jieyu/bugs.html.

[2] MySQL. Bug report time to close stats. http://bugs.mysql.com/bugstats.php.

[3] Mysql bugs: Statistics. http://bugs.mysql.com/bugstats.php.

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. ASPLOS '10.

[5] 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.*, 69(1-3), Dec. 2007.

[6] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? ICSE '10.

[7] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. OOPSLA '10.

[8] S. Kundu, M. K. Ganai, and C. Wang. Contessa: Concurrency testing augmented with symbolic analysis. CAV'10.

[9] R. H. A.-D. S. L. Lanyue Lu, Andrea C. Arpaci-Dusseau. A study of linux file system evolution. FAST '04.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. CGO '04.

[11] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[12] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. ASPLOS '08.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. ASPLOS '06, 2006.

[14] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. ASPLOS '13.

[15] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. MICRO 42, 2009.

[16] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. ISCA '10, 2010.

[17] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. ISCA '08, 2008.

[18] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. PLDI '09.

[19] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. PLDI '07, 2007.

[20] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. OSDI '08.

[21] A. Muzahid, N. Otsuki, and J. Torrellas. AtomTracker: A comprehensive approach to atomic region inference and violation detection. MICRO 43, 2010.

[22] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. PLDI '12, 2012.

[23] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. SIGSOFT '08/FSE-16, 2008.

[24] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. ASPLOS '09, 2009.

[25] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. ICSE '10.

[26] K. Sen. Race directed random testing of concurrent programs. PLDI '08.

[27] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with llvm compiler. In *Runtime Verification*, pages 110–114. Springer, 2012.

[28] 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. OOPSLA '10, 2010.

[29] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. POPL '12.

[30] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. SOSP '11.

[31] H. Volos, A. Tack, M. Swift, and S. Lu. Applying transactional memory to concurrency bugs. ASPLOS '12.

[32] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. ICSE '11.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. ISCA '95, 1995.

[34] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. OSDI '10, pages 1–13, 2010.

[35] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. ISCA '09, 2009.

[36] J. Yu and S. Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. MICRO 43, 2010.

[37] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. ASPLOS '13.

[38] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. ASPLOS '10, 2010.