SCIENCE CHINA
Information Sciences

• REVIEW •
Special Focus on High-Confidence Software Technologies

May 2015, Vol. 58 052105:1–052105:18
doi: 10.1007/s11432-015-5315-9

Fixing, preventing, and recovering from concurrency bugs

DENG DongDong[1], JIN GuoLiang[1], de KRUIJF Marc[2], LI Ang[1], LIBLIT Ben[1], LU Shan[3*], QI ShanXiang[2], REN JingLei[4], SANKARALINGAM Karthikeyan[1], SONG LinHai[1], WU YongWei[4], ZHANG MingXing[4], ZHANG Wei[5] & ZHENG WeiMin[4]

[1]*Computer Science Department, University of Wisconsin, Madison WI* 53706*, USA;*
[2]*Google, USA;*
[3]*Computer Science Department, University of Chicago, Chicago IL* 60637*, USA;*
[4]*Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing* 100084*, China;*
[5]*IBM Research, USA*

Received February 10, 2015; accepted March 11, 2015; published online April 8, 2015

Abstract Concurrency bugs are becoming widespread with the emerging ubiquity of multicore processors and multithreaded software. They manifest during production runs and lead to severe losses. Many effective concurrency-bug detection tools have been built. However, the dependability of multi-threaded software does not improve until these bugs are handled statically or dynamically. This article discusses our recent progresses on fixing, preventing, and recovering from concurrency bugs.

Keywords concurrency bugs, bug detection, production run, performance counters

Citation Deng D D, Jin G L, de Kruijf M, et al. Fixing, preventing, and recovering from concurrency bugs. Sci China Inf Sci, 2015, 58: 052105(18), doi: 10.1007/s11432-015-5315-9

1 Introduction

Concurrency bugs are caused by unsynchronized or incorrectly synchronized memory accesses in multi-threaded programs. They are triggered when multiple threads access shared variables in some failure-inducing orders (i.e., under buggy interleavings). They exist widely in production-run software, causing severe failures in the field with huge financial losses[1)2)] [5]. When they finally get noticed by developers, fixing them takes substantial manual effort [6,7] and yet about 40% of released patches are incorrect [8]. Therefore, it is critical for *end-users* and *developers* to automatically handle production-run failures caused by concurrency bugs.

* Corresponding author (email: shanlu@uchicago.edu)
Authors are listed in alphabetical order. This article is based on the authors' previous papers [1–4], which were done when Marc de Kruijf, Shan Lu, and Wei Zhang were at University of Wisconsin-Madison, and Shanxiang Qi was at University of Illinois at Urbana-Champaign.

1) PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions. http://www.pcworld.com/businesscenter/article/255911/ nasdaqs_facebook_glitch_came_from_race_conditions.html.

2) SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

In the past, much research has focused on detecting concurrency bugs, including data races [9–13], atomicity violations [14–19], order violations [20–23], deadlocks [24–26], and others [27–30]. Although these tools are helpful in discovering concurrency bugs, software dependability does not improve until these bugs are handled. That is, we have to deal with these bugs so that they will not lead to production-run failures that affect end-users.

We have tackled this problem along two directions in the past few years.

1. For detected concurrency bugs, our tool, called CFix [1,2], statically patches the software to completely eliminate these bugs.

2. For undetected concurrency bugs, our tool dynamically adjusts the software execution to prevent these bugs from causing externally visible failures. Along this direction, we have explored two approaches: (1) a proactive approach, called AI [3], that prevents the failure-triggering interleavings from happening; (2) a reactive approach, called ConAir [4], that transparently recovers software right after a failure is triggered by concurrency bugs.

The above work is among the first to automatically fix concurrency bugs statically, and among the first to automatically prevent and recover from a wide variety of concurrency bugs with low overhead on commodity systems. Our evaluations using representative open-source multithreaded software and real-world concurrency bugs have shown that our tools can effectively help fix, prevent, and recover from concurrency bugs, and hence improve the dependability of multithreaded software. We will discuss each of these three approaches below.

2 Statically fixing concurrency bugs

2.1 High-level ideas

Goals & Challenges. As mentioned in Section 1, finding bugs is just a start. Software dependability does not improve until bugs are actually fixed. Unfortunately, fixing concurrency bugs is not trivial. Without end-to-end tool support from bug detection to bug fixing, developers are left to themselves to face the enormous pressure of fixing ever-so-many concurrency bugs, and concurrency-bug fixing remains time-consuming [6,7] and error-prone [8].

Our goal here is to statically fix concurrency bugs that are found by various bug detectors without programmer intervention. Specifically, we want to automatically generate patches that can achieve correctness, performance, and simplicity goals simultaneously: the patches should fix the bugs without introducing new functionality problems, degrading performance excessively, or being needlessly complex. Achieving this goal is challenging for several reasons.

• The bug fixing tool needs to automatically and accurately understand the root cause of a bug without programmers' help. We could get root-cause hints from automated bug-detection tools. However, existing bug detectors are not designed to work with bug fixing tools. As a result, they may merely report bugs' side effects, instead of their root causes. This can easily lead to incorrect or incomplete patches. See Subsection 2.3 for examples.

• The bug fixing tool needs to deal with a variety of concurrency bugs and different types of synchronization demands. Many different types of concurrency bugs exist in real world, such as data races, atomicity violations, order violations, communication pattern errors, and others. A good bug fixing tool should be able to handle different types of bugs using different types of synchronization primitives, such as locks and condition variables.

• The bug fixing tool needs to avoid introducing new bugs while adding new synchronization into the program. Patches to concurrency bugs often involve synchronization operations that have non-local impact. As a result, these patches can easily introduce additional bugs, such as deadlocks, lock-without-unlock, and unlock-without-lock, or cause unexpected and unnecessary performance degradation. See Subsection 2.3 for examples.

• The bug fixing tool needs to consider not only correctness, but also performance and code simplicity.

Figure 1 CFix bug fixing process.

CFix ideas. CFix is a system that automates the repair of concurrency bugs. The key observation we leverage is that concurrency bugs may be more amenable to automated repair than sequential bugs. Most concurrency bugs only cause software to fail rarely and nondeterministically. The correct behavior is already present as some safe subset of all possible execution interleavings. Thus, CFix fixes concurrency bugs by systematically adding synchronization into software and to disable failure-inducing interleavings.

We first decide to handle the wide variety of real-world concurrency bugs by focusing on enforcing two types of synchronization relationship: mutual-exclusion and pair-wise ordering. The rationale is that most synchronization primitives either enforce mutual-exclusion, such as locks and transactional memories [31–33], or enforce strict order between two operations, such as condition-variable signals and waits. Furthermore, Lu et al. [6] have shown that atomicity violations and order violations contribute to the root causes of 97% of real-world non-deadlock concurrency bugs.

We then design and implement two static analyses and code transformation tools: AFix [1] for mutual-exclusion enforcement and OFix [2] for order relationship enforcement, with a best effort to avoid deadlocks and excessive performance losses. AFix enforces the mutual-exclusion relationships among three operations, which prevents one operation from executing in between the other two operations. OFix enforces two common types of order relationships between two operations: (1) allA–B where an operation B cannot execute until *all* instances of operation A have executed; (2) firstA–B where an operation B cannot execute until at least *one* instance of operation A has executed, if operation A executes in this run at all.

Of course, these two synchronization-relationship enforcement tools (AFix and OFix) are only two building blocks of CFix. The overall CFix fixing process starts by taking inputs from some existing concurrency-bug detectors and automates the whole fixing process usually taken by developers, as shown in Figure 1.

2.2 CFix bug fixing process

Following the high-level fix strategy of fixing concurrency bugs by disabling bad interleavings, CFix automates a developer's typical bug fixing process in five steps, as shown in Figure 1.

The first step is *bug understanding*. CFix works with a wide variety of concurrency-bug detectors, such as atomicity-violation detectors, order-violation detectors, data race detectors, and abnormal inter-thread data-dependence detectors. These detectors report failure-inducing interleavings that bootstrap the fixing process.

The second step is *fix-strategy design*. We design a set of fix strategies for each type of bug report. Each fix strategy includes mutual-exclusion/order relationships that, once enforced, can disable the failure-inducing interleaving. By decomposing every bug report into mutual-exclusion and order problems, CFix addresses the diversity challenge of concurrency bugs and bug detectors. To extend CFix for a new bug detector, one only needs to design new fix strategies and can simply reuse other CFix components.

The third step is *synchronization enforcement*. Based on the fix strategies provided above, CFix uses static analysis to decide where and how to synchronize program actions using locks and condition variables, and then generates patches using static code transformation. Specifically, CFix uses AFix to enforce mutual-exclusion and OFix to enforce order relationships.

The fourth step is *patch testing and selection*. CFix tests patches generated using different fix strategies, and selects the best one considering correctness, performance, and patch simplicity. In this step, CFix addresses the challenge of multithreaded software testing by leveraging the testing framework of bug detectors and taking advantage of multiple patch candidates, as the testing result of one patch can sometimes imply problems of another. This step also addresses the challenge of bug detectors that report

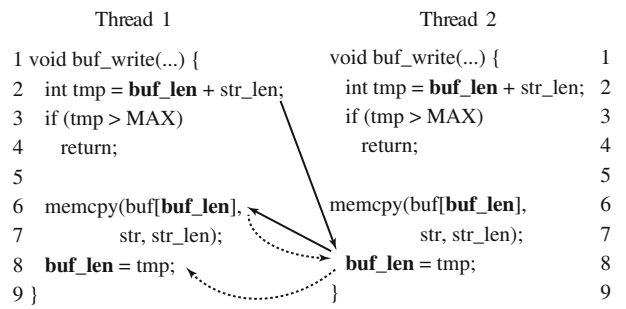

Figure 2 An atomicity violation simplified from Apache. Interleaving "⟶" could cause crashes. Interleaving "⋯▸" could corrupt the Apache log.

inaccurate root causes: patches that fix the real root cause are recognizable during testing as having the best correctness and performance.

The fifth step is *patch merging*. CFix analyzes and merges related patches. We use AFix to merge mutual-exclusion synchronizations (i.e., locks) and OFix to merge order synchronization operations (i.e., condition-variable signal/wait). This step reduces the number of synchronization variables and operations, significantly improving patch simplicity.

Finally, the CFix run-time monitors program execution with negligible overhead and reports deadlocks caused by the patches, if they exist, to guide further patch refinement.

More details about these algorithms and implementations can be found in our conference papers [1,2].

2.3 How CFix works for different types of concurrency bugs

Below we use two concurrency bugs reported by an automated concurrency-bug detector, CTrigger [34], to illustrate some challenges faced by CFix and the solutions provided by CFix.

CTrigger was designed to detect atomicity violations. An atomicity violation occurs when a code region in one thread is unserializably interleaved by accesses from another thread. Each CTrigger bug report is a triple of instructions (p, c, r) such that software fails almost deterministically when r is executed between p and c. Note that CTrigger may incorrectly identify a concurrency bug's root cause as atomicity violation, which we will discuss below.

Case 1 (Atomicity violation root cause). Figure 2 shows an example of a real-world concurrency bug. CTrigger can accurately report two problematic atomicity violations: (1) when line 2 and line 6 are interleaved (⟶) by line 8, Apache could crash; and (2) when line 6 and line 8 are interleaved (⋯▸) by line 8, Apache could corrupt its log. A natural strategy for fixing these two atomicity violations is enforcing mutual-exclusion. Unfortunately, even with accurate root cause understanding, bug fixing is still nontrivial:

• To fix the first report, if we simply lock before line 2 and unlock after line 6, the program could deadlock after buf_write exits at line 4. Our mutual-exclusion enforcement tool AFix will also add an unlock before line 4 to avoid the potential new deadlock.

• To fix the second report, we should not simply put line 6 to 8 and line 8 each into one critical region. As line 8 is part of the critical region for line 6 to 8, this would lead to deadlock again. AFix will recognize this and create just one critical region.

• Two patches that separately fix the above two atomicity violations could deadlock with each other: one thread acquires the first patch's lock at line 2 and waits for the second patch's lock before line 6; a different thread acquires the second patch's lock at line 6 and waits for the first patch's lock before line 8. AFix will heuristically merge these two patches and pick the merged patch after testing.

Case 2 (Order violation root cause). However, not all bugs reported by an atomicity-violation detector can be fixed by enforcing mutual-exclusion. CTrigger may have reported a side effect of a concurrency bug that dose not reflect its root cause. Figure 3 shows an order violation example, where the global variable Gend should be initialized before the two printf statements, but such an order is

```
// Thread 1                                  // Thread 2
printf("End at %f", Gend); //p               // Gend is uninitialized
...                                          // until here
printf("Take %f", Gend – init); //c          Gend = time(); //r
```

Figure 3 An order violation simplified from FFT, a SPLASH2 benchmark [35]. Making Thread 1 mutually exclusive with Thread 2 cannot fix the bug, because r can still execute after p and c.

not enforced. CTrigger reports p, c, and r as shown in the figure, and it is indeed a failure-inducing interleaving when r executes between p and c. However, the root cause here is missed order-relationship enforcement.

Instead of relying on CTrigger to point out the root cause, which is challenging, CFix explores multiple possible ways to disable the failure-inducing interleaving: (1) enforce an order relationship, making r always execute before p; (2) enforce an order relationship, making r always execute after c; or (3) enforce mutual-exclusion between p-c and r. Later on, the first patch will be picked after testing.

2.4 Results and discussion

We evaluate CFix using 10 software projects, including 13 different versions of buggy software. Four different concurrency-bug detectors have reported 90 concurrency bugs in total. CFix correctly fixes 88 of these, without introducing new bugs. This corresponds to correctly patching either 12 or all 13 of the buggy software versions, depending on the bug detectors used. CFix patches have excellent performance: software patched by CFix is at most 1% slower than the original buggy software. Additionally, manual inspection shows that CFix patches are fairly simple, with only a few new synchronization operations added in just the right places.

Overall, our CFix work makes two major contributions. First, we design and implement tools for mutual-exclusion and order relationship enforcement which our evaluation suggests that they are effective for real-world non-deadlock concurrency-bug fixing. Second, we assemble a set of bug detecting, synchronization enforcing, and testing techniques to automate the process of concurrency-bug fixing.

The main drawback of CFix is that it relies on automated concurrency-bug detectors to report failure-inducing buggy interleavings.

3 Proactively preventing concurrency-bug failures

3.1 High-level ideas

Goals. Although our CFix [2] work can ease the pain of lengthy patch releasing period, it heavily relies on automated bug-detection tools and cannot handle failures caused by bugs that have not been detected yet (i.e., unknown bugs). Thus, we also explore techniques that can proactively prevent the manifestation of unknown bugs during production runs.

An ideal production-run failure-prevention tool should satisfy requirements from two aspects.

• Generality. The tool should be able to handle a wide variety of concurrency bugs that are hidden in deployed applications, never been detected, including both atomicity violations and order violations: the two most common types of concurrency bugs based on a previous empirical study [6]);

• Performance. The tool should only incur small overhead on commodity machines.

To the best of our knowledge, none of the existing concurrency-bug failure prevention techniques can satisfy these requirements simultaneously.

AI ideas. AI achieves both the above requirements by anticipating the manifestation of concurrency bugs at run time and preventing the manifestation through temporarily stalling the execution of one thread, which incurs much smaller overhead than checkpointing and rollback.

The key observation behind AI 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

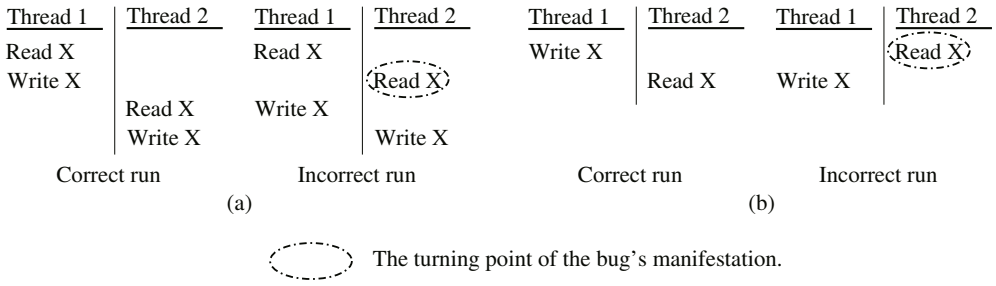

Figure 4 Illustrations of bugs' turning points.

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.

Challenges. Anticipating bugs right before the turning point is critical to failure prevention. Anticipating too early will lead to 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. Below, we simply demonstrate how two straw-man ideas do not work for bug anticipation.

Straw man 1. Anticipating a bug right before the execution of buggy writes. Intuitively, one might think that it should be early enough to prevent the manifestation of a bug, if no buggy write has happened. Unfortunately, this is not true. Figure 4(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 [6]. Here, the turning point is actually right before the second read instruction, as circled in Figure 4(a). Once that read happens, although no bug-related write has been executed, the atomicity violation is inevitable.

Straw man 2. Anticipating a bug right before the execution of the second buggy thread. Suppose a bug involves two threads. Even if only one thread's buggy code region has been executed, it could still be too late to prevent the bug manifestation. Figure 4(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 uninitialized reads [6]. The turning point in this example is right before the read in Thread 2, as circled in Figure 4(b). Once that read is executed, although the buggy code region in Thread 1 has not been executed yet, the order violation is inevitable.

3.2 How AI works for different types of concurrency bugs

Anticipating invariant. 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. Also, postponing the execution of I_2 can often prevent the bug (i.e., the execution of I_2 is the turning point). Following this observation, we propose the Anticipating Invariant (AI) as the central concept of our failure prevention system.

To ease the discussion, in the rest of this section 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.

The concept of Anticipating Invariant is based on our observation that, in all the correct runs, only a few static instructions ever appear as the remote predecessor of a given static instruction's dynamic instances. Once a dynamic instruction's remote predecessor does not belong to this trusted set, the manifestation of a concurrency bug can be *anticipated* and *prevented* by stalling a thread at this moment.

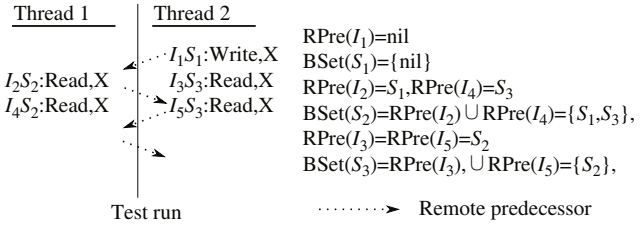

Figure 5 Demonstration of belonging sets.

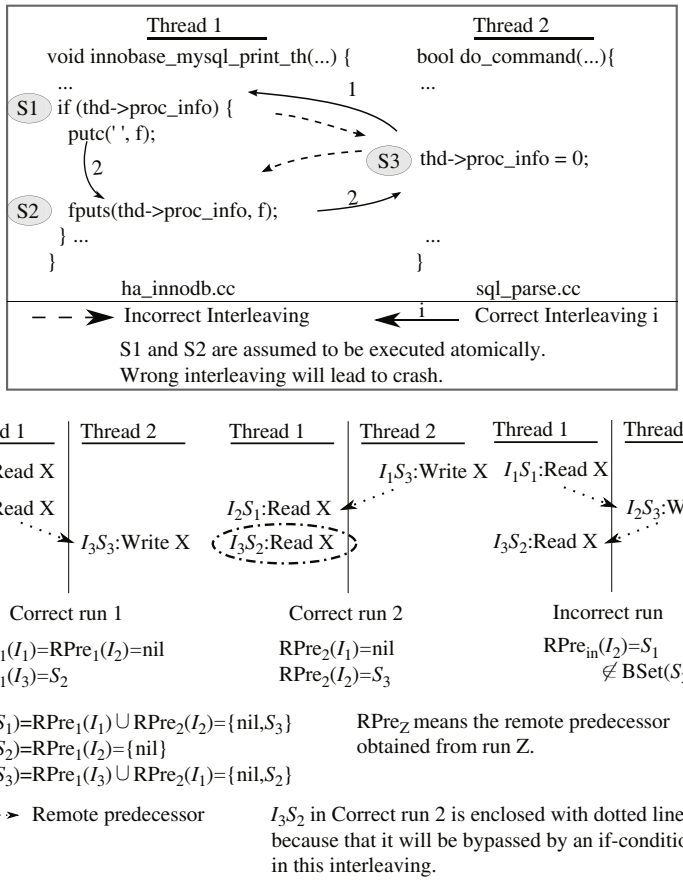

Figure 6 A real-world atomicity violation in MySQL and the corresponding simplified code.

The *remote predecessor* mentioned above will be expressed as $\mathrm{RPre}(I_x)$ for every dynamic instruction I_x in the execution traces. $\mathrm{RPre}(I_x)$ 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. If there is no such dynamic instruction, then $\mathrm{RPre}(I_x) = \mathrm{nil}$.

To strictly define the AI, we calculate a *Belonging Set*, expressed as $\mathrm{BSet}(S_y)$, 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 AI to be

$$\mathrm{RPre}(I_xS_y) \in \mathrm{BSet}(S_y), \quad I_xS_y \text{ is derived from } S_y.$$

As an illustration, *BSet* of S_2 in Figure 5 is calculated as $\mathrm{BSet}(S_2) = \mathrm{RPre}(I_2) \cup \mathrm{RPre}(I_4) = \{S_1, S_3\}$.

How it works for atomicity violations. Figure 6 shows a real-world atomicity violation from the

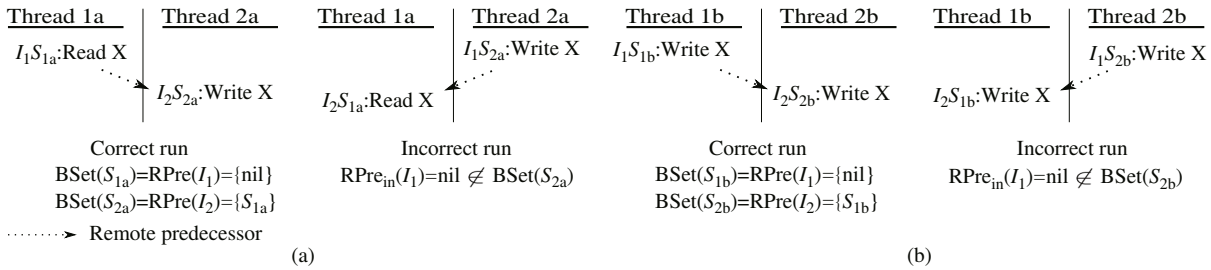

Figure 7 Interleavings of two typical order violations.

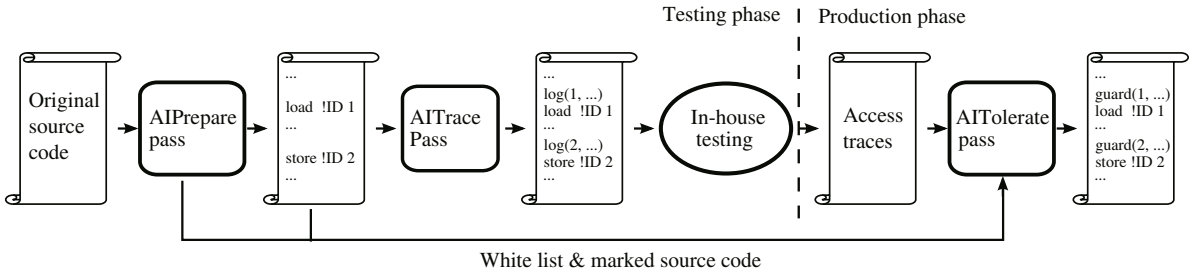

Figure 8 Overview of the whole system.

MySQL database server and the corresponding simplified code of this bug. The simplified part of the figure shows two interleavings that can be found in correct test runs and one found in incorrect runs.

After observing the correct test runs, we can calculate that $\mathrm{BSet}(S_3)$ is $\{\mathrm{nil}, S_2\}$. 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 $S_1 \notin \mathrm{BSet}(S_3)$, 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.

How it works for order violations. Figure 7 shows two representative interleavings obtained from an R–W order violation and a W–W order violation respectively. The remote predecessor of $I_1S_{2\mathrm{a}}$ in subfigure (a) and the remote predecessor of $I_1S_{2\mathrm{b}}$ in subfigure (b) are both nil in the incorrect run. In this case, a violation will be reported because neither $\mathrm{BSet}(S_{2\mathrm{a}})$ nor $\mathrm{BSet}(S_{2\mathrm{b}})$ contains nil. Similar to atomicity violations, the bug's manifestation is anticipated right before its turning point and hence can be prevented without using rollback.

3.3 Design and implementation details

Implementation. To utilize the Anticipating Invariant for tolerating concurrency bugs, we implement a software-only system using the LLVM compiler framework [36], as shown in Figure 8.

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 a 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 later usages.

Next the AITrace pass reads the marked code and adds a call to a logging function before each memory access. The logging function will output a triplet of (Access ID y, Thread ID tid, Accessed Memory Address addr) to the trace file. After gathering enough correct runs we can calculate the remote predecessor of each dynamic instruction by scanning the trace files chronologically, In the meantime, the belonging sets are updated as below:

$$\mathrm{BSet}(y) = \mathrm{BSet}(y) \cup \mathrm{RPre}(\mathrm{Triplet}_x) \text{ if } \mathrm{Triplet}_x = (y, \dots, \dots).$$

Table 1 Evaluation results on different invariants' bug detecting and tolerating (without using rollback) 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	Prevent	Detect	Prevent	Detect	Prevent	Detect	Prevent	Detect	Prevent
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	✓	✓					✓	✓	✓	✓

Finally, the AITolerate pass will add a call to a guarding function before every shared-memory accesses to perform bug prevention. The guarding function will maintain a data structure Recorder[M] to record the last two instructions that access memory M 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-to-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, that is, $\text{RPre}(I_x) \notin \text{BSet}(S_y)$, although I_x is derived from S_y. As analyzed in Subsection 3.2, due 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 until a maximum stalling time, to determine whether the accesses from other threads have resolved it. If the check passes or the threshold is reached, the stalled thread will resume its execution.

Custom instrumentation strategy. We also propose an optional bias instrumentation scheme. 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). As shown in Subsection 3.4, this scheme is effective in lowering the run-time overhead of AI, especially for applications with intensive memory accesses. Similar approaches have been used in several sampling-based race detecting methods [37], and achieve a good result in Google's practice [38].

3.4 Results and discussion

We analyzed AI's capability of anticipating (detecting) and preventing concurrency bugs by using 35 representative real-world bugs from 11 multithreaded applications. The results are shown in Table 1, in which ✓ represents that all the manifestations from this kind of bug are detected/prevented in our experiments; ✓́ means that the bugs can only be detected/prevented on particular interleavings; and blank cells represent that the corresponding invariant is not violated in all executions even when the bug is triggered. Table 1 groups 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 (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.

Table 2 Run-time overheads. The "Bias" and "Default" columns give the overhead with and without bias instrumentation respectively.

Applications		Overhead	
		Default (%)	Bias (%)
Desktop application	PBZip2	0.38	
	Pigz	0.20	
Server application	Apache	0.34	
	MySQL	0.57	
SPLASH–2 benchmarks	FFT	1345	115
	LU	1613	127

As shown in Table 1, AI can detect all the patterns of bugs we have found, which is more than each prior invariant. Also, it has a superior anticipating ability and thus can tolerate more bugs without using rollback.

Table 2 gives the evaluation result of the performance for our current AI implementation[3)]. As we can see, AI only incurs negligible overhead ($< 1\%$) for many nontrivial desktop and server applications. Furthermore, its slowdown on computation-intensive programs can be reduced to about $2\times$ after using the bias instrumentation (with threshold 30%).

Overall, AI provides a general solution for proactively preventing a wide variety of hidden concurrency bugs through temporary thread stalling. The main drawbacks of AI are that (1) it requires training; and (2) its overhead for some applications, such as scientific computing applications, is not negligible.

4 Reactively recovering from concurrency-bug failures

4.1 High-level ideas

Goals & Challenges. Our CFix work can statically fix detected concurrency bugs (Section 2) and our AI work can dynamically prevent some undetected concurrency bugs from manifesting (Section 3). However, neither technique is perfect, and some concurrency bugs will inevitably manifest and trigger failures at run time. Our goal here is to transparently recover a just-triggered concurrency-bug failure, so that end-users will not be affected by the failure.

An ideal failure recovery technique should have several key properties:

• Generality: helping bugs with a wide variety of root-cause interleaving patterns without reliance on accurate bug detection;

• Performance: small run-time overhead and fast failure recovery on commodity systems without any OS/hardware modification; and

• Correctness: not generating results infeasible for original software.

The state-of-the-art for concurrency-bug failure recovery is the *checkpoint-and-rollback* approach [39–41]. At run time, this approach periodically takes system checkpoints. When a failure is triggered, it rolls back the software to some recent checkpoints and reexecutes from the selected checkpoint. Given the non-determinism of multithreaded software, there is a good chance that the concurrency bug will not manifest during the reexecution, and hence the failure is transparently recovered.

This checkpoint-and-rollback approach achieves good *correctness* and *generality*, as it can help recover almost all types of concurrency bugs without changing the program semantics. Unfortunately, existing techniques based on this approach require periodic *whole-program* checkpoint at run time and *whole-program* rollback for failure recovery. As a result, they require OS/hardware modifications to achieve good performance, compromising the goal of achieving good *performance* on commodity systems.

3) 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.

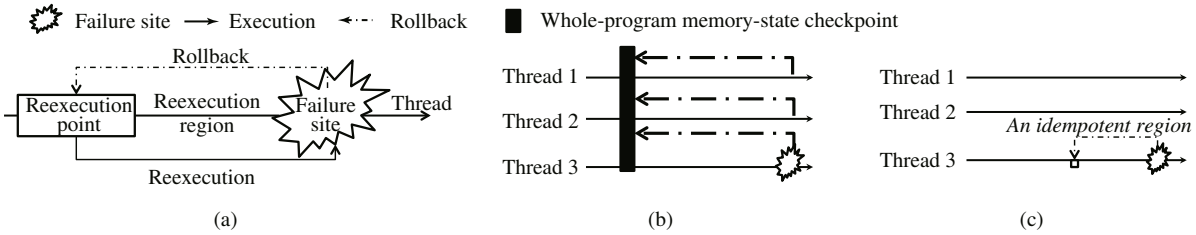

Figure 9 An overview of ConAir. (a) Key design points in rollback recovery; (b) traditional rollback recovery; (c) ConAir.

ConAir ideas. ConAir is a static analysis and code-transformation tool that automatically inserts rollback-recovery code into multithreaded software and allows software to recover from a wide variety of known and hidden concurrency bugs with little run-time overhead on existing OS/hardware platforms.

As shown in Figure 9, the design of a rollback-recovery system for multithreaded software includes two key components: (1) how many threads participate in the rollback recovery; and (2) what is the reexecution point in each participant thread. ConAirtakes a novel approach to both components:

1. **Single-threaded rollback**. ConAir only rolls back a single thread where the failure occurs, different from previous work that rolls back multiple (all) threads.

2. **Idempotent reexecution with no checkpoints**. ConAir always reexecutes an idempotent region[4)] surrounding the failure site. Consequently, no memory-state checkpoints are needed. This is different from previous work that periodically takes memory-state checkpoints and reexecutes from the latest checkpoint.

The above two ideas work together to make sure ConAir can achieve good run-time *performance*, with no modifications to OS/hardware. Its idempotent reexecution also guarantees that it never changes program semantics (*correctness*).

Finally, ConAir can still recover from a wide variety of concurrency bugs (*generality*), because of a key observation that many concurrency-bug failures occur quickly after incorrect interleaving and hence can be effectively recovered by rolling back and reexecuting an idempotent region in a single failing thread, while other threads make forward progress. We will discuss this in more detail below.

4.2 How ConAir works for different types of concurrency bugs

In the following, we will explain how ConAir works for all major types of concurrency bugs [6,42]: atomicity violations, order violations, and deadlocks. Particularly, we will focus on how the two key ideas work for each type of bug: (1) no multithreaded rollback; and (2) no memory checkpoints with idempotent reexecution.

How it works for atomicity violations. Atomicity violations contribute to about 70% of real-world non-deadlock bugs [6]. They occur when two code regions R_1 and R_2 from two threads interleave unserializably, which violates the expected atomicity of one or both regions. Clearly, if we can rollback and reexecute any one involved thread, the execution of R_1 and R_2 will be serialized and the failure will be recovered.

To understand whether ConAir recovery works, we need to answer two questions: (1) whether the failing thread is involved in the unserializable interleaving and (2) whether an idempotent region surrounding the failure site is large enough to cover the buggy code region.

To answer the first question, we checked 51 real-world atomicity-violation bugs collected by a previous work [6]. About 92% of them cause failures in a thread that is involved in the unserializable interleaving and hence can potentially be recovered by single-threaded recovery. This observation can be better understood through bug examples shown in Figure 10. This figure depicts the most common types of real-world atomicity violations [6,18,43]. As we can see, an atomicity violation usually causes an involved thread to read an unexpected value from a shared variable, such as log in Figure 10(a), ptr in Figure 10

4) A code region is idempotent if it can be reexecuted any number of times without changing the program semantics.

```
/*Thread 1*/
log=CLOSE;
log=OPEN;

/*Thread 2*/
if(log!=OPEN)
  {//output failure}
```
(a)

```
/*Thread 1*/
ptr=aptr;
tmp=*ptr;

/*Thread 2*/
ptr=NULL;
```
(b)

```
/*Thread 1*/
if(ptr)
  fputs(ptr);

/*Thread 2*/
ptr=NULL;
```
(c)

```
/*Thread 1*/
cnt+=deposit1;
printf("Balance=%d",cnt);

/*Thread 2*/
cnt+=deposit2;
```
(d)

Figure 10 Most failures caused by atomicity violations can be recovered by rolling back one thread: the failing thread. Different checkpoint/sandbox techniques may be needed to guarantee correctness. (a) Violating WAW atomicity: roll back Thread 2 to recover; (b) violating RAW atomicity: roll back Thread 1 to recover; (c) violating RAR atomicity: roll back Thread 1 to recover; (d) violating WAR atomicity: roll back Thread 1 to recover.

(b) and (c), and cnt in Figure 10(d). This incorrect value quickly leads to a failure in that thread. Clearly, the failure can be recovered by rolling back and reexecuting that thread.

It is not easy to answer the second question quantitatively. Qualitatively, we observe that failures caused by many concurrency bugs can be recovered by reexecuting an *idempotent* region surrounding the failure site, such as the one shown in Figure 10 (a) and (c). In fact, several previous empirical studies [23,44] indicate that many concurrency bugs have short error propagation and rarely contain I/O operations. This indicates that constraining reexecution regions to be free of idempotency-destroying operations does not eliminate the chance of recovering from many concurrency-bug failures.

How it works for order violations. Order violations contribute to nearly 30% of real-world non-deadlock concurrency bugs [6]. They occur when an operation A is expected to execute before an operation B, but instead executes after B due to lack of synchronization. Clearly, if we can rollback and reexecute the thread of B, the occurrence of B will be effectively delayed and the failure will be recovered.

To understand whether ConAir recovery works, we need to know (1) whether the thread of B is the failing thread; and (2) whether an idempotent region surrounding the failure site is large enough to cover B.

To answer the first question, we checked all 21 real-world order-violation bugs collected by a previous work [6]. We found that about 50% of order-violation bugs lead to failures in the thread of B, and hence can be recovered by the single-threaded recovery. Failures of the other bugs manifest in the thread of A and occasionally some other threads.

To better understand this observation, one can consider a common type of order-violation bugs; thread t_{B} reads a shared-variable V before V is initialized by thread t_{A}. In this case, the uninitialized value in V usually leads to a failure in t_{B}. By rolling back t_{B}, we can postpone the read of V until V is initialized.

The answer to the second question is similar to that for recovering atomicity violations. That is, since many concurrency bugs have short propagation distances, an idempotent reexecution region is long enough in many cases.

How it works for deadlock bugs. Deadlock contributes to about 40% of all concurrency bugs [42]. When a deadlock occurs, every thread involved is holding some resource that is blocking another thread. In a typical deadlock, making any thread release a resource will break the circular resource dependence. Clearly, ConAir can help recover from a deadlock, as long as any single failing thread t contains an idempotent region that covers both the location where t is blocked waiting for resources as well as the location where t acquires a resource another failing thread is waiting for.

Overall, ConAir does not aim to handle all possible software failures. Instead, it aims to handle a significant portion of concurrency-bug failures with a wide variety of symptoms and root causes with negligible overhead and no modification to the OS or hardware. ConAir also provides guarantee to never deviate from the original software semantics.

4.3 Design and implementation details

The basic ConAir framework includes three static analyses and code-transformation components.

A static analyses component identifying potential failure sites. Without any knowledge of hidden concurrency bugs, ConAir statically identifies program locations where four most common types

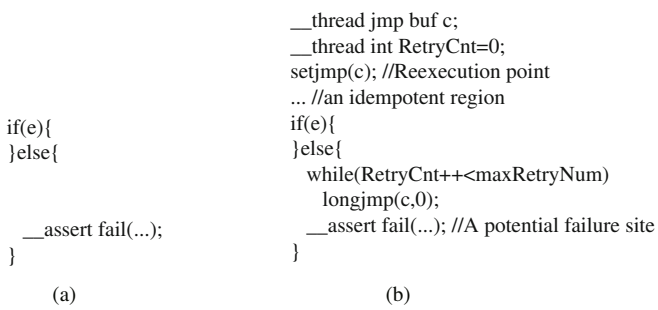

Figure 11 ConAir code transformation for assert. (a) Original code; (b) transformed code.

of failures [23] could occur: assertion failures, wrong output failures, segmentation-fault failures, and deadlocks. For example, ConAir would identify the invocation of __assert_fail in Figure 11 as a potential failure site.

A static analyses component identifying reexecution points. In ConAir, every reexecution region is an idempotent region that ends at a potential failure site. This component conducts a backward depth-first search from every site f along the control-flow graph. It identifies every location right after an idempotency-destroying operation as a reexecution point.

A static code-transformation component. As shown in Figure 11, this component inserts setjmp at the reexecution point and longjmp at the potential failure site, which work together to maintain correct reexecution during failure recovery.

ConAir further extends the above three basic components to handle interprocedural recovery and contains some library functions, such as malloc and pthread_mutex_lock, in reexecution regions. Furthermore, ConAir optimizes away useless reexecution points that can never help the recovery of concurrency-bug failures. The details of these algorithms and implementations can be found in our conference paper.

4.4 Results and discussion

Our evaluation uses 10 real-world concurrency bugs from 8 different open-source multithreaded software. These 10 bugs represent different types of root causes (atomicity violations, order violations, and deadlocks) and different types of failure symptoms. The evaluation shows that ConAir helps widely used open-source software recover from a wide variety of concurrency-bug failures on unmodified OS/hardware platforms, where ConAir incurs negligible overhead ($\leqslant 0.2\%$) and takes just a few microseconds for failure recovery.

Overall, our ConAir work makes two key contributions: (1) a novel observation that rolling back a single thread and reexecuting an idempotent region is sufficient to recover from many concurrency-bug failures; and (2) a static analysis and code-transformation algorithm that automatically harden multithreaded software against hidden concurrency bugs, and an implementation in LLVM.

The main drawback of ConAir is that it cannot help recover from failures that are preceded by a long and non-idempotent error propagation.

5 Related work

5.1 Software bug fixing

For general software bugs, automated bug fixing is very challenging. The recent GenProg [45] project smartly uses stochastic search methods like genetic programming with lightweight program analyses to find patches for real bugs. Although inspiring, it does not provide guarantee that the program is indeed fixed without new bugs introduced. Recent work by Logozzo and Ball [46] focuses on a small number of program repair patterns, such as fixing off-by-one errors. Software verification is used to make sure that the repairs do not make the programs more buggy regarding specifications provided by programmers.

For concurrency bugs, automated deadlock fixing has been studied before [26]. After our CFix work [1,2], researchers also tried using Petri-Net mechanisms to automatically fix concurrency bugs [47,48].

Hot-patching tools focus on how to apply program repairing while it is running. ClearView [49] patches security vulnerabilities by modifying variable values at run time. Its design is not suitable for concurrency bugs. The LOOM system [50] provides a language for developers to specify synchronizations they want to add to a running program and deploys these synchronization changes safely. Similar to CFix, LOOM also uses CFG reachability analysis for safety, and has a run-time component to recover from deadlocks. Since LOOM has different design goals from CFix, it does not need to consider issues like working with bug detectors, fix-strategy design, locating synchronization operations, handling statically unknown numbers of signals, simplicity concerns, patch merging and testing. Tools like CFix can potentially complement LOOM by automatically generating patches for LOOM to deploy.

5.2 Concurrency-bug failure prevention and recovery

Concurrency-bug prevention. AI is not the first tool to help prevent the manifestation of concurrency bugs. However, previous tools either rely on checkpoint and replay to some extent [51–53], which can lead to huge run-time overhead, or rely on the knowledge of previous manifestations of the same bug [24,50,54], which cannot handle failures caused by previously unknown bugs. Furthermore, some previously proposed tools [51,53] are constrained in their generality: they only handle certain types of concurrency bugs like data races or atomicity violations. In comparison, AI can handle a wide variety of previously unknown concurrency bugs without any rollback.

Concurrency-bug failure recovery. As discussed in Section 4, several rollback-recovery systems have been built before, such as Rx [39], ASSURE [40], and Frost [41]. They all change operating systems to support whole program checkpoint and rollback. Rx changes the program environment during reexecution to handle deterministic bugs. ASSURE rolls back a failed execution to an existing error-handling path. It is designed to mitigate the impact of deterministic bugs, and cannot help software generate correct results after the manifestation of a non-deterministic concurrency bug. Frost [41] proposes a novel solution to survive data races. With OS support, it executes multiple replicas of the program with complementary thread schedules at the same time. Periodically, it compares the states of different replicas and tries to survive state divergence caused by data races. In general, these systems all require checkpointing the whole program states and rolling back all threads during a failure. Consequently, they all require sophisticated changes to operating systems.

Microreboot [55] is a recovery technique that reboots only application components, instead of the whole program, when failures occur. To benefit from microreboot, programmers have to manually separate their systems into components (groups of objects) that can be individually restarted, such as Enterprise Java Beans components in J2EE applications. The ConAir technique discussed in this article shares a common high-level philosophy with microreboot of not rolling back the whole program. However, the similarity ends there. ConAir focuses on concurrency-bug failure recovery. It works on any C/C++ multithreaded software without manual changes. It automatically identifies reexecution points and conducts automated code transformation.

5.3 Others

Concurrency-bug detection. Many techniques have been proposed to detect data races [9,12,13,56], atomicity violations [15,18,57,58], order violations [20–22,27,52], and others concurrency problems. Bug-detection tools help developers discover and understand the defects in software. However, they cannot directly decrease software downtime or improve the dependability of software. The work presented in this article has a different goal from bug-detection tools. It aims to alleviate or eliminate the impact of concurrency-bug failures with or without the help of bug-detection tools.

Deterministic execution. Deterministic systems [59–63] force a multithreaded program to execute a deterministic interleaving under a given input. This promising approach still faces challenges, such as overhead, integration with system non-determinism, language design, etc. In general, these tools

address different problems from what are discussed in this article. Even inside a deterministic run time, concurrency bugs can still occur and require failure handling techniques.

Automated synchronization-primitive insertion. Techniques have been proposed to insert lock operations into software based on annotations [43,64], atomic regions inferred from profiling [65], and whole-program serialization analysis [66]. QuickStep [67] automatically selects functions to put into critical sections based on race-detection results during loop parallelization. Recent work by Navabi et al. [68] parallelizes sequential software based on future-style annotations. It automatically inserts barriers to preserve sequential semantics during parallelization.

Compared with the above techniques, our CFix is unique in fixing concurrency bugs reported by a wide variety of bug detectors and in synchronizing using both locks and condition variables. CFix addresses unique challenges such as fix-strategy design, simplicity optimization, patch merging, and patch testing. The static analysis conducted by CFix differs from that of Navabi et al. [68] by considering additional issues such as simplicity and performance.

6 Conclusion

This article reviews our recent work on fixing concurrency bugs (CFix), preventing concurrency bug manifestations (AI), and recovering from concurrency bug failures (ConAir). The three techniques, CFix, AI, and ConAir, aim the same goal—improving the dependability of multithreaded software—from different angles. They can well complement each other: CFix is most suitable when some concurrency bugs are already detected and not fixed yet; AI and ConAir are useful without any bug-detection information. Among AI and ConAir, ConAir provides better performance for a wide range of software applications and has no requirements on training. However, ConAir only works for concurrency bugs that have short error propagation distances, while AI does not have this limitation. Of course, the work presented in this article is just a starting point to concurrency bug fixing, prevention, and recovery. Future work can help further improve the simplicity and performance of automatically generated concurrency-bug patches and help further improve the generality and performance of concurrency-bug prevention and recovery.

Acknowledgements

This work was supported by Natural Science Foundation (Grant Nos. CCF-0701957, CCF-0953478, CCF-0845751, CCF-1018180, CCF-1054616, CCF-1217582, CCF-1439091); DOE contract DE-SC0002153; LLNL contract B580360; National Natural Science Foundation of China (Grant Nos. 61433008, 61373145, 61170210, U1435216), National High-tech R&D Program of China (863 Program) (Grant No. 2012AA012600), Chinese Special Project of Science and Technology (Grant No. 2013zx01039-002-002); a Google U.S./Canada PhD Fellowship, a Claire Boothe Luce Faculty Fellowship, and an Alfred P. Sloan Research Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

1 Jin G L, Song L H, Zhang W, et al. Automated atomicity-violation fixing. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, San Jose, 2011. 389–400

2 Jin G L, Zhang W, Deng D D, et al. Automated concurrency-bug fixing. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, Hollywood, 2012. 221–236

3 Zhang M X, Wu Y W, Lu S, et al. AI: a lightweight system for tolerating concurrency bugs. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, 2014. 330–340

4 Zhang W, de Kruijf M, Li A, et al. Conair: featherweight concurrency bug recovery via single-threaded idempotent execution. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, Houston, 2013. 113–126

5 Leveson N G, Turner C S. An investigation of the therac-25 accidents. Computer, 1993, 26: 18–41

6 Lu S, Park S, Seo E, et al. Learning from mistakes—a comprehensive study of real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, 2008. 329–339

7 Godefroid P, Nagappani N. Concurrency at Microsoft—an Exploratory Survey. Microsoft Research Technical Report MSR-TR-2008-75, 2008

8 Yin Z N, Yuan D, Zhou Y Y, et al. How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011. 26–36

9 Flanagan C, Freund S N. Fasttrack: efficient and precise dynamic race detection. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, 2009. 121–133

10 Kasikci B, Zamfir C, Candea G. Data races vs. data race bugs: telling the difference with portend. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, London, 2012. 185–198

11 Netzer R H B, Miller B P. Improving the accuracy of data race detection. In: Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg, 1991. 133–144

12 Savage S, Burrows M, Nelson G, et al. Eraser: a dynamic data race detector for multithreaded programs. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint Malo, 1997. 27–37

13 Yu Y, Rodeheffer T, Chen W. RaceTrack: efficient detection of data race conditions via adaptive tracking. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles, Brighton, 2005. 221–234

14 Chen F, Serbanuta T F, Rosu G. jpredictor: a predictive runtime analysis tool for java. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, 2008. 221–230

15 Flanagan C, Freund S N. Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, 2004. 256–267

16 Flanagan C, Qadeer S. A type and effect system for atomicity. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, 2003. 338–349

17 Flanagan C, Freund S N, Yi J. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, 2008. 293–303

18 Lu S, Tucek J, Qin F, et al. AVIO: detecting atomicity violations via access interleaving invariants. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, 2006. 37–48

19 Xu M, Bodík R, Hill M D. A serializability violation detector for shared-memory server programs. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, 2005. 1–14

20 Gao Q, Zhang W B, Chen Z Z, et al. 2ndStrike: toward manifesting hidden concurrency typestate bugs. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, 2011. 239–250

21 Shi Y, Park S, Yin Z N, et al. Do I use the wrong definition?: DefUse: definition-use invariants for detecting concurrency and sequential bugs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, Reno/Tahoe, 2010. 160–174

22 Zhang W, Sun C, Lu S. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In: Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, Pittsburgh, 2010. 179–192

23 Zhang W, Lim J, Olichandran R, et al. ConSeq: detecting concurrency bugs through sequential errors. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, 2011. 251–264

24 Jula H, Tralamazza D, Zamfir C, et al. Deadlock immunity: enabling systems to defend against deadlocks. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, San Diego, 2008. 295–308

25 Li T, Ellis C, Lebeck A, et al. Pulse: a dynamic deadlock detection mechanism using speculative execution. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, Anaheim, 2005. 3

26 Wang Y, Kelly T, Kudlur M, et al. Gadara: dynamic deadlock avoidance for multithreaded programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, Berkeley, 2008. 281–294

27 Lucia B, Ceze L. Finding concurrency bugs with context-aware communication graphs. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, New York, 2009. 553–563

28 Musuvathi M, Qadeer S, Ball T, et al. Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, San Diego, 2008. 267–280

29 Qi S X, Muzahid A, Ahn W, et al. Dynamically detecting and tolerating if-condition data races. In: Proceedings of 20th IEEE International Symposium on High Performance Computer Architecture, Orlando, 2014. 120–131

30 Yu J, Narayanasamy S, Pereira C, et al. Maple: a coverage-driven testing tool for multithreaded programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, Tucson, 2012. 485–502

31 Harris T, Fraser K. Language support for lightweight transactions. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, Anaheim, 2003. 388–402

32 Herlihy M, Moss J E B. Transactional memory: architectural support for lock-free data structures. In: Proceedings of

the 20th Annual International Symposium on Computer Architecture, San Diego, 1993. 289–300
33 Rajwar R, Goodman J R. Speculative lock elision: enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, Austin, 2001. 294–305
34 Park S, Lu S, Zhou Y Y. Ctrigger: exposing atomicity violation bugs from their finding places. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, 2009. 25–36
35 Woo S C, Ohara M, Torrie E, et al. The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, Margherita Ligure, 1995. 24–36
36 Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, Palo Alto, 2004. 75–86
37 Marino D, Musuvathi M, Narayanasamy S. Literace: effective sampling for lightweight data-race detection. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, 2009. 134–143
38 Serebryany K, Bruening D, Potapenko A, et al. Addresssanitizer: a fast address sanity checker. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, Boston, 2012. 28–28
39 Qin F, Tucek J, Sundaresan J, et al. Rx: treating bugs as allergies c a safe method to survive software failures. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles, Brighton, 2005. 235–248
40 Sidiroglou S, Laadan O, Perez C, et al. Assure: automatic software self-healing using rescue points. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, 2009. 37–48
41 Veeraraghavan K, Chen P M, Flinn J, et al. Detecting and surviving data races using complementary schedules. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, Cascais, 2011. 369–384
42 Li Z M, Tan L, Wang X H, et al. An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, San Jose, 2006. 25–33
43 Vaziri M, Tip F, Dolby J. Associating synchronization constraints with data in an object-oriented language. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, 2006. 334–345
44 Volos H, Tack A J, Swift M M, et al. Applying transactional memory to concurrency bugs. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, London, 2012. 211–222
45 Le Goues C, Dewey-Vogt M, Forrest S, et al. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 3–13
46 Logozzo F, Ball T. Modular and verified automatic program repair. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, Tucson, 2012. 133–146
47 Liu P, Zhang C. Axis: automatically fixing atomicity violations through solving control constraints. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 299–309
48 Liu P, Tripp O, Zhang C. Grail: context-aware fixing of concurrency bugs. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, 2014. 318–329
49 Perkins J H, Kim S, Larsen S, et al. Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, 2009. 87–102
50 Wu J Y, Cui H M, Yang J F. Bypassing races in live applications with execution filters. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Vancouver, 2010. 1–13
51 Lucia B, Devietti J, Strauss K, et al. Atom-aid: detecting and surviving atomicity violations. In: Proceedings of the 35th Annual International Symposium on Computer Architecture, Washington, 2008. 277–288
52 Yu J, Narayanasamy S. A case for an interleaving constrained shared-memory multi-processor. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, Austin, 2009. 325–336
53 Yu J, Narayanasamy S. Tolerating concurrency bugs using transactions as lifeguards. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Washington, 2010. 263–274
54 Lucia B, Ceze L. Cooperative empirical failure avoidance for multithreaded programs. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, Houston, 2013. 39–50
55 Candea G, Kawamoto S, Fujiki Y, et al. Microreboot—a technique for cheap recovery. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, San Francisco, 2004. 3
56 Erickson J, Musuvathi M, Burckhardt S, et al. Effective data-race detection for the kernel. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Vancouver, 2010. 1–16
57 Chew L, Lie D. Kivati: fast detection and prevention of atomicity violations. In: Proceedings of the 5th European Conference on Computer Systems, Paris, 2010. 307–320

58 Lu S, Park S, Hu C F, et al. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles, Stevenson, 2007. 103–116
59 Bergan T, Hunt N, Ceze L, et al. Deterministic process groups in dOS. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Vancouver, 2010. 1–16
60 Cui H M, Wu J Y, Gallagher J, et al. Efficient deterministic multithreading through schedule relaxation. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, Cascais, 2011. 337–351
61 Liu T P, Curtsinger C, Berger E D. Dthreads: efficient deterministic multithreading. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, Cascais, 2011. 327–336
62 Olszewski M, Ansel J, Amarasinghe S. Kendo: efficient deterministic multithreading in software. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, 2009. 97–108
63 Aviram A, Weng S-C, Hu S, et al. Efficient system-enforced deterministic parallelism. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. Berkeley: USENIX Association Berkeley, 2010. 1–16
64 McCloskey B, Zhou F, Gay D, et al. Autolocker: synchronization inference for atomic sections. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, 2006. 346–358
65 Weeratunge D, Zhang X Y, Jagannathan S. Accentuating the positive: atomicity inference and enforcement using correct executions. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, Portland, 2011. 19–34
66 Upadhyaya G, Midkiff S P, Pai V S. Automatic atomic region identification in shared memory SPMD programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, Reno/Tahoe, 2010. 652–670
67 Misailovic S, Kim D, Rinard M. Parallelizing Sequential Programs with Statistical Accuracy Tests. MIT Technical Report, MIT-CSAIL-TR-2010-038. 2010
68 Navabi A, Zhang X Y, Jagannathan S. Quasi-static scheduling for safe futures. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, 2008. 23–32