Adaptively Generating High Quality Fixes for Atomicity Violations

Yan Cai †
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
ycai.mail@gmail.com

Lingwei Cao
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences, Beijing, China
lingweicao@gmail.com

Jing Zhao
School of Computer Science and Technology, Harbin Engineering University, Harbin, China
jingzhao.duke@gmail.com

ABSTRACT

It is difficult to fix atomicity violations correctly. Existing gate lock algorithm (*GLA*) simply inserts gate locks to serialize executions, which may introduce performance bugs and deadlocks. Synthesized context-aware gate locks (by *Grail*) require complex source code synthesis. We propose *αFixer* to adaptively fix atomicity violations. It firstly analyses the lock acquisitions of an atomicity violation. Then it either adjusts the existing lock scope or inserts a gate lock. The former addresses cases where some locks are used but fail to provide atomic accesses. For the latter, it infers the visibility (being global or a field of a class/struct) of the gate lock such that the lock only protects related accesses. For both cases, *αFixer* further eliminates new lock orders to avoid introducing deadlocks. Of course, *αFixer* can produce both kinds of fixes on atomicity violations with locks. The experimental results on 15 previously used atomicity violations show that: *αFixer* correctly fixed all 15 atomicity violations without introducing deadlocks. However, *GLA* and *Grail* both introduced 5 deadlocks. *HFix* (that only targets on fixing certain types of atomicity violations) only fixed 2 atomicity violations and introduced 4 deadlocks. *αFixer* also provides an alternative way to insert gate locks (by inserting gate locks with proper visibility) considering fix acceptance.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging • Theory of computation→Program verification.

KEYWORDS

Atomicity violations, fix, repair, concurrency bugs, deadlock, multithreaded program, lock order

ACM Reference format:

Yan Cai, Lingwei Cao, and Jing Zhao. 2017. Adaptively Generating High Quality Fixes for Atomicity Violations. In *Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8 2017 (ESEC/FSE'17)*, 12 pages. *http://dx.doi.org/10.1145/3106237.3106239*

† Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ESEC/FSE'17, September 04-08, 2017, Paderborn, Germany
© 2017 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
http://dx.doi.org/10.1145/3106237.3106239

1. INTRODUCTION

Concurrency bugs widely exist in multithreaded programs [3][10][19][22][25][30][32][38][48][57]. They are difficult to detect and to reproduce [8][47][53][60], as well as to correctly fix [29].

Manual bug fixing not only takes a long time [22] but also is error-prone [58]. Recently, automated bug fixing becomes popular [9][14][15][16][28][36][40][43]. However, almost all existing techniques on fixing concurrency bugs insert new locks (known as gate locks) statically or dynamically to serialize all executions of threads involved in a concurrency bug, including *AFix* [22][23], *Axis* [34], *Grail* [37], *Gadara* [51], and [41]. As the inserted gate locks prevent two or more threads from executing concurrently, the original incorrect thread interleaving is eliminated. We refer to the techniques that insert gate locks as *Gate Lock Algorithms* (*GLA*). However, introducing gate locks may introduce performance bugs [21] as they always serialize threads of the targeted concurrency bugs. To solve it, *Grail* inserts synthesized gate locks: it maps the hash values of the variables from the concurrency bugs to unique gate locks. However, the synthesized gate locks may sometimes reduce fix acceptance.

Besides, introducing gate locks (e.g., *GLA*) or modifying lock scopes (e.g., *HFix* [35]) may introduce various deadlocks [34][37][41]. This is common even for manual bug fixing (e.g., 16.4% incorrect fixes indeed introduced new deadlocks [58]). If deadlocks are introduced, *Axis* [34] further iteratively fixes these introduced deadlocks by inserting more gate locks. *Grail* [37] improves *AFix* and *Axis* by adopting Petri-net analysis to avoid introducing deadlocks [51]. However, *Grail* is limited to analyse two threads only [37]. Hence, *Grail* fails to avoid introducing deadlocks involving other threads out of the targeted concurrency bugs [6]. *HFix* [35] targets on fixing a subset of atomicity violations by modifying lock scopes. It may also introduce performance issues.

A recent work named *DFixer* [6] introduces lock pre-acquisitions to fix deadlocks. As deadlocks involve high-level lock acquisitions, it is possible to avoid introducing deadlocks by eliminating new lock orders [6]. However, atomicity violations involve low level memory accesses. They may involve lock acquisitions protecting some of their accesses or may involve no lock. In the latter case, gate locks might be necessary; however, in the former case, gate locks together with existing lock acquisitions may form deadlocks. Hence, it is more difficult to correctly fix atomicity violations. Hence, many existing techniques differentiate concurrency bugs as deadlock and non-deadlock bugs [38][42][49][61] as they require different techniques to detect and to fix.

In this paper, we focus on atomicity violations. An atomicity violation occurs if an expected atomic set of memory accesses fails to be atomic [38][52]. For example, Figure 1(a) shows an atomicity violation: two accesses to a pointer *p* from thread t_1 can be interleaved by a write access from thread t_2 (as indicated by two solid arrows), resulting in a NPE error (*NULL Pointer Exception*). To fix such atomicity violations, existing works may introduce various problems (as discussed above, also see Section 2).

We propose an adaptive approach to fix atomicity violations, known as *AlphaFixer* or *αFixer* for short. Given an atomicity violation, *αFixer* firstly identifies all involved lock acquisitions in it. If most of the accesses of the atomicity violation are protected by the same lock, *αFixer* then tries to fix it by either extending an existing lock scope or combining two existing lock scopes to provide an atomic region to put all accesses under the protection of the same lock. Otherwise, *αFixer* inserts a gate lock to fix it. In the latter case, *αFixer* further infers the visibility[1] of the gate lock in order to exactly protect the related accesses that may be involved in the atomicity violations. To guarantee a deadlock-free fix, *αFixer* conservatively restricts its lock scope extension or combination into three cases (see Section 3). If necessary, *αFixer* adopts multiple lock acquisitions (i.e., to acquire multiple locks) at a time to eliminate new lock orders.

We have implemented *αFixer* for C/C++ programs and evaluated it on 15 atomicity violations. We compared *αFixer* with *GLA*, *Grail*, and *HFix* where *HFix* is designed to fix certain types of concurrency bugs. The evaluation is on the following three aspects: correctness[2], performance and code readability. For performance, we followed the approach [37] for comparison purpose. The experiment results show that: (1) *αFixer* fixed all 15 atomicity violations correctly without introducing any deadlock. However, both *GLA* and *Grail* introduced 5 deadlocks; *HFix* was only applicable to fix 6 atomicity violations but introduced 4 deadlocks. (2) *αFixer* only incurred 21.1% overhead on average; however, *GLA*, *Grail*, and *HFix* incurred a significantly larger overhead: 120.2%, 82.9% and 32.5%, respectively. (3) *αFixer* also provided a new way to insert gate lock considering the visibility of the variables in the given atomicity violation, which is an alternative way to improve fix acceptance. The main contributions of this paper are as follows:

- It proposes a new strategy *αFixer* to adaptively fix atomicity violations by producing more effective and efficient fixes. *αFixer* can insert gate locks to fix atomicity violations. However, it is able to infer the visibility of the gate lock to provide an alternative way to improve the acceptance of fixes.
- We have implemented a prototype tool to evaluate *αFixer* (http://lcs.ios.ac.cn/~yancai/alphafixer). The experiment results demonstrate the effectiveness and efficiency of *αFixer* compared with existing works.

[1] In this paper, the visibility of a variable means that whether it is a *global* variable or a class/struct *field*. For the latter case, a variable is accessed based on an instance of its class/struct.

[2] To fix concurrency bugs, thread interleaving space is usually reduced to be a subset of that before fixing if no deadlock is introduced. Therefore, the correctness here refers to whether any deadlock is introduced.

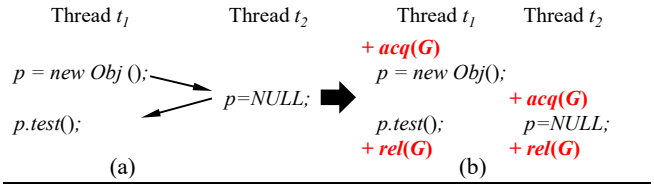

Figure 1. An atomicity violation (a) and its fix by *GLA* (b).

2. BACKGROUND AND MOTIVATIONS

2.1 Preliminaries

We focus on two kinds of events in multithreaded programs: *Memory accesses* and *Lock operations*. The later includes lock acquisitions *acq*(*m*), *tryAcq*(*m*), *acq*(*m*, *n*) and lock release *rel*(*m*). Note that (1) *acq*(*m*) blocks its executing thread if lock *m* is acquired by another thread; but *tryAcq*(*m*) does not; (2) *acq*(*m*, *n*) indicates that a thread tries to acquire two locks at the same time.

If a thread firstly acquires a lock *m* and then acquires another lock *n* before releasing lock *m*, there is a *lock order* from lock *m* to lock *n*, denoted by $m \rightsquigarrow n$. If another lock order exists $n \rightsquigarrow m$ (or $n \rightsquigarrow \ldots \rightsquigarrow m$ for multiple threads), we say it is a *reversed lock order* of the lock order $m \rightsquigarrow n$. There is a special case that, if a thread acquires two locks *m* and *n* at the same time via *acq*(*m*, *n*), then there is no lock order produced between the two locks. This is because a thread performing *acq*(*m*, *n*) immediately releases any acquired lock if one of them cannot be acquired [6].

A (resource) deadlock occurs if a lock order and its reversed lock order occurs at the same time [7][25]. But, the absence of a lock order and its reversed lock order indicates no deadlock on these locks.

2.2 Motivations

GLA fixes an atomicity violation by inserting a gate lock to serialize the executions of the involved threads. It could reduce the parallelism of executions from different threads due to over synchronization (known as performance bugs [21]) and may also introduce deadlocks. *Grail* may improve performance but may produce fixes with low acceptance due to its lock synthesis. Overall, these works focus on the correctness of their fixes but seldom consider the quality (e.g., whether the fix code is acceptable and understandable to developers) of their fixes. This point is extremely important when a program is developed by many developers and is developed continually to produce different versions (e.g., MySQL). We show these limitations in the next two subsections with examples.

2.2.1 Performance and Acceptance of Fixes. **Atomicity violation *AV1*:** Figure 2 shows an atomicity violation *AV1*. It involves two threads (t_1 and t_2) and two variables (*buf->output* and *buf->outcnt*). The variable *buf->output* is a fixed size buffer and the variable *buf->outcnt* points to the end of *buf->output*. Please ignore the four highlighted lines starting with "+" (i.e., lines 3, 7, 11, and 15) for now. The function *ap_bufferred_log_writer*() buffers characters into *buf->output* and then increments *buf->outcnt*. However, these two operations are not protected by any lock. As a result, if two or more threads concurrently call the function and the executions of two threads could be interleaved as what the

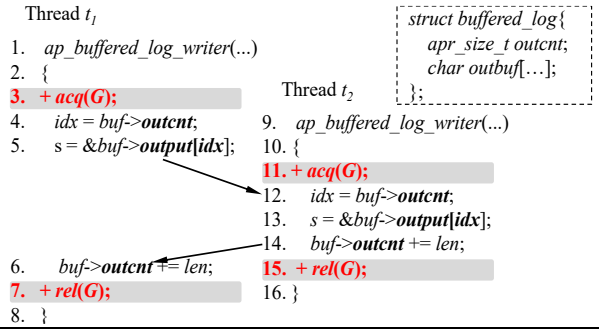

Figure 2. An atomicity violation AV_1 from *apache* with bugID=25520, and a fix to it by *GLA*.

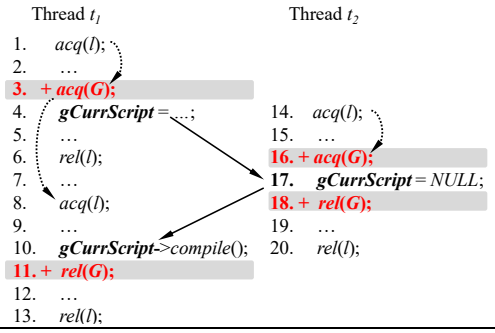

Figure 3. An atomicity violation AV_2 from *mozilla* [59] and its fix by *GLA*.

two solid arrows indicate, an atomicity violation will occur, corrupting both *buf->output* and *buf->outcnt*.

Atomicity violation AV_2: Figure 3 shows an atomicity violation AV_2, involving one variable *gCurrScript*. The atomicity violation occurs when thread t_2 writes a NULL value to *gCurrScript* in between the write to *gCurrScript* (at line 4) and the invocation of *compile*() on *gCurrScript* (at line 10) by thread t_1, as indicated by the two solid arrows. Although the original program contains a lock *l* (at lines 1, 6, 8, 13, 14, 20) to protect accesses to *gCurrScript*, this protection is only on the individual access. It fails to provide an atomic region for two accesses to *gCurrScript* from thread t_1.

To fix AV_1, *GLA* inserts a lock *G* to serialize two threads. This fix is shown in lines starting with "+" (i.e., lines 3, 7, 11, and 15). To fix AV_2, *GLA* inserts a lock *G* (at lines 3, 11, 16, and 18) to prevent the write to *gCurrScript* (at line 17) from occurring in between the two accesses by thread t_1.

GLA may introduce high runtime overhead. For example, on AV_1, if the variable *buf*s of two threads are different, then the two variables *buf->output* of two threads are also different. Hence, no atomicity violation may occur and the two threads can be executed concurrently. However, after fixed by *GLA*, the two threads always execute sequentially due to the unique global gate lock *G*, incurring runtime overhead.

The latest work *Grail* [37] follows *GLA*, but synthesizes a context-aware gate lock *G* according to all variables of the targeted atomicity violation as follows:

G = *contextL*(*hash*(v_1), *hash*(v_2) ...), where v_1, v_2, ... are variables from the atomicity violation and the function *contextL*(...) returns a unique lock corresponding to the inputs (i.e., the hash values of all variables). Thus, if the actual variables of two threads are different, *Grail* computes two different gate locks. Hence, the two threads are able to execute concurrently. In this way, *Grail* does not reduce parallelism if no atomicity violation may occur. Figure 4 shows the two gate locks generated by *Grail* to fix AV_1 and AV_2, respectively. However, there are three main limitations of *Grail*.

Firstly, the readability of fix by *Grail* might be worse than that by *GLA*. For example, on fixing AV_1 and AV_2, the inserted lock acquisition by *GLA* is simply "*acq*(*G*)" where the lock *G* is globally defined once. Whereas, the gate lock inserted by *Grail* are: "*G* = *contexL*(*hash*(&(*buf->outcnt*)), *hash*(&(*buf->output*))); *acq*(*G*);" and "*G* = *contextL*(*hash*(&(*gCurrScript*))); *acq*(*G*);", respectively. These fixes may be difficult for developers to understand.

Secondly, a synthesized context-aware lock may not be always required. If an atomicity violation involves only global variables (e.g., on AV_2), a global lock is enough. In this case, even if the gate lock is produced by *Grail*, the produced gate locks will always be the same. Otherwise, if an atomicity violation involves class (including struct) level variables, a gate lock of the same class level will be enough (e.g., on AV_1).

Thirdly, the implementation of *contextL*(...) might be complex. For example, the original implementation [37] uses *String.intern*() function provided by native code of JDK. This implementation maintains a *HashTable* structure and a lock to protect operations on it. On each call to *String.intern*(), the *HashTable* is iterated to search for a unique String object (which is taken as a lock in Java, see the function *JVM_InternString* in *jvm.cpp*). For C/C++, a similar pair of map/table and lock is also required.

Considering above discussions, the fixes generated by *Grail* may have a low acceptance to developers. Of course, different developers may hold different views on what kind of fixes they may prefer to accept.

2.2.2 Introducing Deadlocks. On AV_1, *GLA* inserts a lock *G* to serialize two threads. This is a correct fix. However, on AV_2, after *GLA* inserts a lock, it also introduces three new lock orders: two $l \rightsquigarrow G$ (i.e., from line 14 to line 16 and from line 1 to line 3) and one $G \rightsquigarrow l$ (from line 3 to line 8), as shown in three dotted arrows. The two lock orders $l \rightsquigarrow G$ (i.e., from line 14 to line 16) and $G \rightsquigarrow l$ (from line 3 to line 8) form a deadlock. (Actually, another deadlock is introduced from two locks orders $l \rightsquigarrow G$ (from line 1 to line 3) and $G \rightsquigarrow l$ (from line 3 to line 8) if they can be formed by multiple threads at the same time.)

Grail further relies on Petri-net analysis to prevent introducing deadlocks, which is limited to two threads only [37].

For some special atomicity violations like AV_2 where the same lock (e.g., lock *l*) is used to protect part of accesses, a recent work *HFix* [35] suggests a fix: move either an acquisition or a release

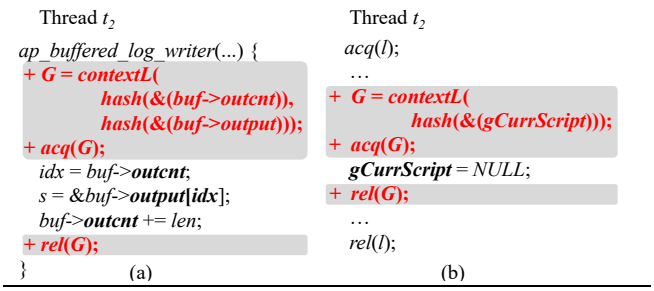

Figure 4. The two gate locks genearted by *Grail* to fix AV_1 (a) and AV_2 (b).

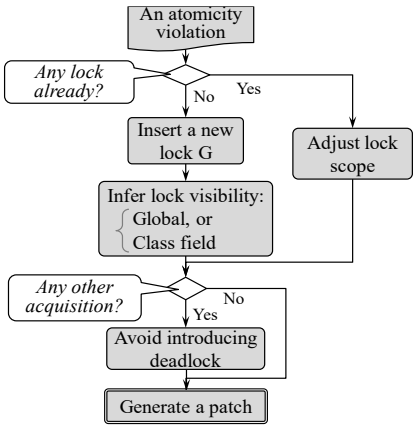

Figure 5. An overview of our *αFixer*.

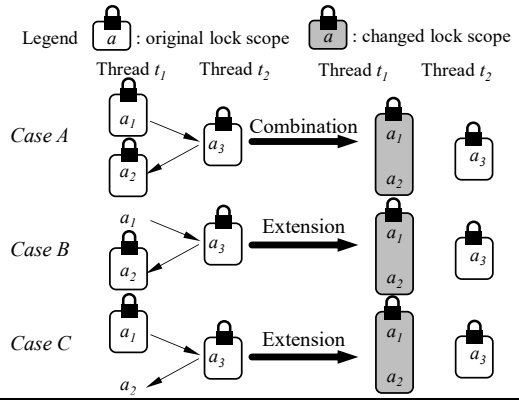

Figure 6. The three cases to fix certain atomicity violations (where the arrows also indicate the error interleaving).

statement to protect all other accesses not protected by the same lock. On AV_2, *HFix* may either move $rel(l)$ at line 6 to a location right after line 10 or move $acq(l)$ at line 8 right before line 4. However, this fix actually introduces a self-deadlock as thread t_1 will acquire lock l twice where the second acquisition is blocked. Besides, it may still introduce other deadlocks. For example, if there is a lock acquisition $acq(m)$ between lines 4 and 10, a new lock order $l \rightsquigarrow m$ will be introduced. Then a deadlock is introduced if another thread forms a lock order $m \rightsquigarrow l$.

3. OUR APPROACH

3.1 Rationale and Overview

An atomicity violation involves at least three accesses to a set of shared variables. It is possible that these accesses are protected by some locks (e.g., on AV_2). But it is also possible that no lock protects the involved accesses (e.g., on AV_1).

Therefore, our insight is: it is not always necessary to introduce new locks to serialize threads to fix atomicity violations. If there are already some locks protecting most of the involved accesses, the locks could be slightly adjusted to fix these atomicity violations. For example, in Figure 3, AV_2 could be fixed by combining the two separated locking regions (i.e., removing "$rel(l)$;" at line 6 and "$acq(l)$;" at line 8). By doing so, the two accesses to *gCurrScript* from thread t_1 are fully protected by lock l; hence, the access at line 17 by thread t_2 cannot be interleaved in between the two accesses. And AV_2 is fixed.

On the other hand, no lock may protect any access from an atomicity violation. In this case, a new lock is necessary. However, when introducing a new lock, the introduced lock orders (if any) must be carefully handled to avoid introducing deadlocks.

Besides, if a new lock is required, the visibility of the lock should also be carefully determined. Unlike *Grail* that synthesizes a gate lock, it would be better if we could insert a lock with the same visibility as that of the involved variables.

Although the first step is to adjust any existing lock protection, a gate lock can also be inserted to fix an atomicity violation. It is difficult to say which fix is better. For example, if the two accesses of an atomicity violation are far away to each other and if the same lock protects the two accesses separately, then adjusting the two lock scopes may incur high runtime overhead. Therefore, if an atomicity violation can be fixed by adjusting its lock scopes, *αFixer* further produces a second fix by inserting a gate lock. The second fix can also be an option to developers.

Overall, as shown in Figure 5, *αFixer* firstly analyses the given atomicity violation to identify all involved lock acquisitions and then determines whether to adjust the lock scope or to insert a gate lock. For the latter, as a new lock is required, *αFixer* infers the visibility of the involved variables to determine whether the new lock should be a global one or a class field one. Next, *αFixer* analyses the involved locks to avoid introducing deadlocks.

3.2 Adjust Lock Scopes to Fix Atomicity Violations

For atomicity violations that already involve some locks protecting the accesses, they might be fixed by slightly adjusting the lock scopes. In this paper, we only focus on three scenarios as shown in Figure 6 (where a box with a lock indicates a pair of lock acquisition and release):

- ***Case A***: all accesses (e.g., a_1, a_2, and a_3) are separately protected by the same lock. In this case, the atomicity violation could be fixed via *Combination*: Combine two separated lock scopes of the same lock of the corresponding thread (i.e., thread t_1 in *Case A*).
- ***Case B*** and ***Case C***: only part of accesses from a thread is protected by a lock and other accesses from the second thread are all protected by the same lock. Then, the atomicity violation could be fixed via *Extension*: Extend the lock scope of the first thread to also protect the remaining accesses from this thread (i.e., thread t_1 in *Cases B* and *C*).

There might be other cases where the lock scopes can be changed to fix atomicity violations. Our criterion is that there must be the same lock protecting at least one access of each thread. Hence, for other cases, we fix them by inserting gate locks (see the next subsection). Note that, *HFix* [35] has a similar suggestion as *Case B* and *Case C*. However, *HFix* does not distinguish *Case A* from *Cases B* and *C*. Hence, on *Case A*, *HFix* can introduce self-deadlocks as discussed in Section 0 (also see our experiment in Section 5.3).

3.3 Introduce New Locks to Fix Atomicity Violations

A new lock is necessary to fix an atomicity violation via gate lock strategy. Unlike *GLA* that introduces a global gate lock, *αFixer* tries to introduce a context-aware gate lock. Unlike *Grail* that introduces a synthesized lock, *αFixer* automatically infers the lock visibility of the new locks to avoid synthesizing gate locks. As a result, *αFixer* provides an alternative way to insert gate locks.

3.3.1 Infer Visibility of Gate Locks. We found that, for an atomicity violation, the visibility of the involved variables is usually determined: either global variables or class field variables. Here, the fields refer to the variable member of class in object-oriented languages (e.g., C++) or struct (e.g., C). For example, the variables *outcnt* and *output* in Figure 2 are two fields of the struct *buffered_log*; and the variable *gCurScript* in Figure 3 is a global variable.

Hence, given an atomicity violation, if all its variables are global, an explicit global lock is enough; otherwise, if all involved variables are fields of the same class instance, a field lock within the same class is also enough. In these two cases, even if we follow *Grail* to synthesize gate locks, the synthesized gate locks are always the same global lock (for the former case) or are always the same lock of the same class instance. Hence, there is no need to additionally synthesize gate locks dynamically; a unique global gate lock or a class level gate lock is enough.

For single-variable atomicity violation [38], the involved variable is deterministically a global one or a class field. However, for multi-variable atomicity violations [38], the involved variables may contain:

(1) Global variables only, or
(2) Field variables of the same class instances, or
(3) Both global ones and class fields, or
(4) Multiple fields of different class, or
(5) Fields of the same class but different class instances.

The first two cases can be handled in the same way as handling single-variable atomicity violations because all involved variables are either global ones or fields of the same class instance. However, the last three cases are more complex. To fix them, *αFixer* simply inserts a global lock to serialize two threads. Admittedly, the synthesized gate locks by *Grail* may perform better than the global locks theoretically. Note that, the fields of a class/struct may also be global (e.g., declared to be *static*). Such cases can be easily handled and hence are not discussed in this paper.

3.3. 2 Insert Gate Locks. Once the visibility of a gate lock is determined, it is straightforward to insert the gate lock to serialize the two threads of the given atomicity violation. This step is the same as what *GLA* performs (see Figure 1(b) where the gate lock is the lock *G*).

3.4 Avoid Introducing Deadlocks

3.4.1 Why can deadlocks be introduced? αFixer fixes an atomicity violation by either adjusting lock scope of an existing lock or inserting a gate lock. In both cases, deadlocks may be introduced. We discuss the two cases below.

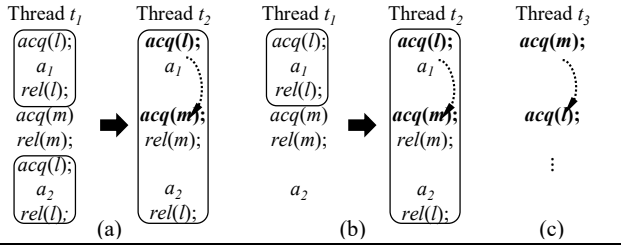

Figure 7. New lock orders are introduced if lock scopes are adjusted.

Deadlocks may be introduced by adjusting lock scopes. Recall that adjusting lock scopes consists of either combination or extension. The combination of two separated lock scopes may introduce new lock orders if, in between the two separated scopes, there are other lock acquisitions. Consider the example in Figure 7(a) where a pair of acquisition and release on lock *m* exists in between two lock scopes on lock *l*. After combining the two lock scopes on lock *l* into one, a new lock order $l \rightsquigarrow m$ is introduced. Similarly, the extension of a lock scope to protect more accesses may also introduce new lock orders, as shown in Figure 7(b). Then, for above two scenarios, deadlocks are introduced if a different thread has the lock order $m \rightsquigarrow l$, as shown in Figure 7(c).

If a gate lock is introduced, deadlocks may also be introduced. We have demonstrated this in Figure 3(b).

To ease the presentation, we refer to locks that are nested in combined or extended lock scopes or inserted gate lock scopes as ***inner locks*** (i.e., lock *m* in Figure 7 and lock *l* at line 8 in Figure 3); and we refer to the corresponding new lock orders as ***inner lock orders*** (i.e., $l \rightsquigarrow m$ in Figure 7 and $G \rightsquigarrow l$ in Figure 3). Similarly, we refer to new lock orders from other existing locks to adjusted locks or to inserted gate locks as ***outer lock orders*** ($l \rightsquigarrow G$ in Figure 3); and we refer to the former existing locks as ***outer locks*** (lock *l* at line 1 in Figure 3).

3.4.2 How to Avoid Introducing Deadlocks. For the inner lock orders (e.g., $l \rightsquigarrow m$ or $G \rightsquigarrow m$), if the inner locks (i.e., lock *m*) can be identified, then these lock orders can be eliminated by acquiring two locks together (i.e., *acq*(*l*, *m*) or *acq*(*G*, *m*)). This is because the inner locks, if any, exist in between the two accesses of one thread; and the two accesses usually have a short distance in term of source code lines. Otherwise, if the inner locks cannot be acquired together with the adjusted locks or the inserted gate locks, *αFixer* gives up fixing the atomicity violation.

When an inner lock *m* is acquired together with the adjusted lock *l*, it is possible that this lock *m* is actually the lock *l*. In this case, a self-deadlock is introduced as the lock will be acquired twice. However, it is difficult to statically know whether the two locks *l* and *m* are the same one, especially when the class/struct instances are involved. To solve this challenge, we change the property of lock *m* to be *reentrant* (i.e., *recursive* lock) because a reentrant lock can be acquired and released multiple times in a nested manner by the same thread. For the inserted gate lock *G*, we also set both lock *G* and any inner lock *m* to be a reentrant lock considering recursive function calls.

If an inner lock (e.g., lock *m* in Figure 7) is acquired together with adjusted locks or inserted gate locks (e.g., *acq*(*l*, *m*) or *acq*(*G*, *m*)), to avoid introducing new lock orders, we do not remove the original lock acquisition (e.g., *acq*(*m*)) on inner locks. Because the

original lock acquisition may exist in a different function which can be called from a different control branch. (Otherwise, we have to adopt ad-hoc synchronization to fix program control as adopted in *DFixer* to fix deadlocks, which is usually harmful [56].)

For the second type of outer lock orders introduced due to inserted gate locks (i.e., $k \rightsquigarrow G$ where lock k is acquired before the acquisition on lock G), if all inner lock orders are eliminated (i.e., $G \rightsquigarrow m$), no deadlock will be introduced. This is because outer lock orders alone without their reversed lock orders cannot form any deadlocks.

However, for the second type of outer lock orders introduced by lock adjusting (i.e., $k \rightsquigarrow l$ where lock k is acquired before the acquisition on lock l), even if we eliminate their reversed lock order $l \rightsquigarrow k$, a third thread may still form their reversed lock orders $l \rightsquigarrow k$. This is because, unlike inserted gate locks, these locks already exist before fixing. In this case, we give up fixing such atomicity violations by adjusting locks; instead, we try to fix it by introducing a gate lock, as shown in Figure 5. Of course, in this case, the fixing approach via adjusting locks can be a suggestion.

Besides lock acquisitions and releases, other additional lock synchronizations (e.g., *wait*(*l*) and *notify*(*l*)) may be involved in one or more threads of atomicity violations. Such cases are complicated. Therefore, we only consider one case where a lock is adjusted to fix an atomicity violation, and the additional synchronization(s) is within the lock scope (protected region) of the adjusted lock before and after adjusting. In this case, no new lock order is introduced. For other cases, *αFixer* gives up its fixing.

Limitations. *αFixer* may fail to fix an atomicity violation in two cases: (1) there is other inner lock in between the two accesses of a thread and such lock(s) cannot be acquired together with the inserted gate lock or the adjusted lock; and (2) synchronizations except lock acquisitions and releases will be contained within the adjusted lock scope or the scope of the inserted gate locks except the case discussed in the last paragraph. To guarantee a theoretical correctness, *αFixer* gives up fixing such atomicity violations.

3.4.3 Guarantee of αFixer. *αFixer* does not guarantee to fix all atomicity violations as discussed in the last subsection. However, if it generates a fix, it guarantees to fix the atomicity violation without introducing deadlocks as Theorem 1.

Theorem 1. *Given an atomicity violation AV, if αFixer generates a fix, it does not introduce any deadlock.*

Proof Sketch. We prove the theorem via three cases showing that the fix does not introduce resource deadlocks, communication deadlocks, and self-deadlocks, respectively. And we mainly prove the scenarios where a gate lock is inserted. The scenarios where a lock is adjusted can be proved similarly.

A) Suppose that *αFixer* introduced a gate lock G to fix *AV*. If no new lock order is introduced, there is no way for *αFixer* to introduce a resource deadlock. Now, suppose that there are other locks within the scope of gate lock acquisitions, these locks are acquired together with the gate lock G by following *αFixer* approach (see the first paragraph of Section 0). (Note, if any inner lock cannot be acquired together with gate lock, *αFixer* does not produce any fix, see the last paragraph of Section 0.) Hence, any potential inner lock orders are eliminated. Next, suppose that some outer lock orders from an outer lock is introduced, say $k \rightsquigarrow G$ where k is an outer lock. However, as no any inner lock order (e.g., $G \rightsquigarrow l$ where the lock l may be the lock k) is introduced. Hence, no reversed lock order of the introduced outer lock order $k \rightsquigarrow G$ is introduced. Therefore, no resource deadlock is introduced by *αFixer*.

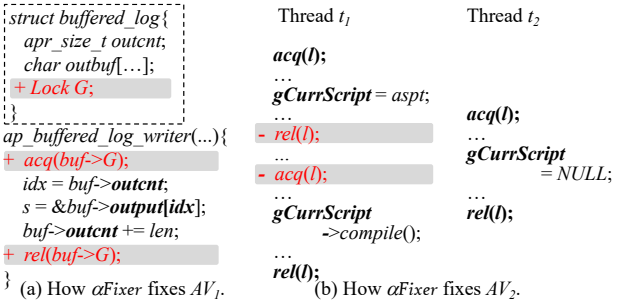

Figure 8. One way to fix AV_1 and AV_2 by *αFixer*. (On AV_1, we only show one thread as two threads share the same code lines. On AV_2, the fix is to combine two separated lock scope, corresponding to *Case A* in Figure 6. The second fix to AV_2 by *αFixer* (i.e., inserting a gate lock) is omitted.)

B) If there are other synchronizations (e.g., *wait*() and *notify*)), by following *αFixer* approach (see the 6th paragraph of Section 0), *αFixer* only adjusts lock scope if: before and after adjusting, the synchronizations are always within the original scope. That is, after adjusting, the lock orders remain the same as that before adjusting the lock scope. Hence, *αFixer* does not block any communication (i.e., not introduce communication deadlocks).

C) If any lock is inserted or adjusted, *αFixer* changes it to be re-entrant lock (see the 1st paragraph of Section 0). This enables a thread to acquire the same lock multiple times without blocking itself. Hence, *αFixer* does not introduce self-deadlocks.

Based on the above analysis, Theorem 1 is proved. □

3.5 Our Approach on Examples

The fix to AV_1 by *αFixer* is shown in Figure 8(a) (where we only show one thread as two threads share the same code lines). On AV_1, no lock is found. Then, *αFixer* introduces a gate lock G to fix it. This fix is the same as *GLA*. However, *αFixer* firstly inserts a lock G to the struct *buffered_log* before inserting lock acquisition and release *acq*(*buf->G*) and *rel*(*buf->G*). This brings an alternative fix code lines besides that by *Grail* (see Figure 4).

To fix AV_2, *αFixer* combines the two lock scopes of lock l for thread t_1 as shown in Figure 8(b). This fix is obviously simple and different from that by *GLA* and *Grail*. Besides, on AV_2, no deadlock is introduced by *αFixer*; whereas, *HFix* introduces a self-deadlock (see the last paragraph of Section 0).

3.6 Fix Program Control Flow

Like other approaches, *αFixer* also needs to fix program control flows, which is similar to existing works [6][22][23]. For example, when an acquisition *acq*(*G*) is inserted, its corresponding release *rel*(*G*) should be inserted at each exit branch containing the inserted *acq*(*G*).

4. USER STUDY OF GRAIL AND αFIXER

This section presents our user study on the fixes to the atomicity violations AV_1 and AV_2. We decided to conduct the user studies on these two atomicity violations as they are representatives among all benchmarks in our experiment (see Section 5). The questionnaire firstly offered a brief introduction to atomicity violations. The second parts were the two original pieces of code with two atomicity violations and two short descriptions on how they could occur, as well as the two fixes of *Grail* and *αFixer* (which were referred to as *Tool1* and *Tool2*, respectively). We designed four selection-questions: between *Tool1* and *Tool2*, (1) which fix is more understandable (Understandability), (2) which fix is more readable (Readability), (3) which fix may incur larger overhead; and (4) which fix do you prefer? For each question, we offered an "*Any*" option to indicate an equal or an unclear preference. We also requested participants to fill their occupations and any additional comments. Our questionnaire was distributed via the social network community WeChat (see our tool website). There was no time limit for participants to answer the questions before we collected the results at the paper submitting time.

Totally, there were 40 participants: 3 from Mathematics, 4 from Finance, 33 from IT companies. Table 1 presents the results. From the table, it shows that, more than 85% participants held a preference on *αFixer* other than on *Grail* in terms of understandability and readability. On overhead, more than 67% participants regarded that *Grail* incurs larger overhead. Finally, *αFixer* was well accepted (preferred) by more than 80% participants.

Among the 40 participants, only 8 of them filled their comments; and 7 comments obviously pointed out that *Tool2* (i.e., *αFixer*) should be more readable, straightforward, adaptive and simpler, (e.g., different fix for different variable visibility). The remaining one did not point out which one is better.

Table 1. Statistics on user studies.

		Tool1 (*Grail*)	**Tool2** (*αFixer*)	**Any**
AV_1	Understandability	2.5%	87.5%	10.0%
	Readability	0.0%	90.0%	10.0%
	Larger Overhead	67.5%	12.5%	20.0%
	Preferred Fix	5.0%	95.0%	0.0%
AV_2	Understandability	7.5%	92.5%	0.0%
	Readability	2.5%	85.0%	12.5%
	Larger Overhead	75.0%	10.0%	15.0%
	Preferred Fix	10.0%	82.5%	7.5%

5. EXPERIMENT

5.1 Benchmarks

We selected a set of real-world benchmarks [2][59] including 20 benchmarks. we excluded 6 of them: 1 deadlock, 1 duplicated bug (i.e., *cherokee*), 3 order violations, and 1 atomicity violation involving Java code. Including our two motivating examples (where AV_1 is from *apache*$_{25520}$), there are 15 benchmarks. Some benchmarks cannot be correctly compiled in our experiment environment. We followed an existing work [28] to extract the source code containing atomicity violations. All these atomicity violations are listed in Table 3 including whether we used the original benchmarks or the extracted ones (under the column "Original").

5.2 Implementation and Experimental Setup

We have implemented *αFixer*, *GLA* (i.e., the AFix [22] algorithm) *Grail*, and *HFix* within LLVM 3.6 framework [1][31]. LLVM IR does not support class/struct information. We modified Clang frontend to generate the information for *αFixer* to infer lock visibility. *Grail* synthesizes context-aware gate locks which is based on *String.intern*() from Java library [37]. We extracted the OpenJDK implementation of *String.intern*().

After applying the four techniques to all benchmarks, we ran each fixed program by each technique for 1,000 times and collected the results. During this 1,000 runs, we inserted a set of random sleep before and after each lock acquisition of the fixed programs to amplify the probabilities for any introduced deadlock to occur. We also ran them for 1,000 additional times without sleep to collect their execution time except on those where deadlocks frequently occurred after fixing.

To evaluate the performance scalability of the fixed programs, we followed [37] to amplify the overhead introduced by each technique for comparison purpose. We configured the number of threads to be 2, 4, 8, 16, 32, 64, and 128. Note that, this amplification only applies to execution of the code lines involved in atomicity violations. This is the same as the previous work [37].

Our experiments were conducted on a ThinkPad workstation with a processor i7-4710MQ, installed with Ubuntu 14.04.

5.3 Result Analyses

In this section, we firstly present the summary of fixing results. Next, we separately compare *αFixer* with *HFix* first, and then compare *αFixer* with *GLA* and *Grail*. This is because *HFix* is only applicable to atomicity violations with some locks by merging two lock scopes. Among our benchmarks, only 6 out of 15 benchmarks can be handled by *HFix*.

Table 2. Fixing summary of all techniques.

#	#of fixed atom. violations				Avg. overhead			
Total	*GLA*	*Grail*	*HFix*	*αFixer*	*GLA*	*Grail*	*HFix*	*αFixer*
15	10 (67%)	10 (67%)	2 (13%)	15 (100%)	120.2%	82.9%	32.5%	21.1%

5.3.1 Fixing Summary. Table 2 summarizes the fixing results by all four techniques on all 15 atomicity violations. Overall, both *GLA* and *Grail* correctly fixed 10 (i.e., 67%) atomicity violations. *HFix* only correctly fixed 2 (i.e., 13%) atomicity violations. Our *αFixer* correctly fixed all 15 (i.e., 100%) atomicity violations.

Besides, on performance scalability testing with 128 threads, *GLA* incurred the largest overhead: 120.2% on average, followed by *Grail* incurring 82.9% average overhead. *HFix* incurred an average overhead of 32.5%. *αFixer* only incurred an average overhead of 21.1%. From the summary, *αFixer* outperforms all other techniques considering both effectiveness and efficiency.

5.3.2 Comparisons on Effectiveness. Table 3 (A) and (B) show the fixing results. The first three columns show the benchmark information. The fourth major column shows the fixes by *αFixer*. The table also shows the results of *αFixer* and *HFix* on the 6 atomicity violations involving locks. In the table, "*AdjL-A*" means that *αFixer* fixed the benchmark by adjusting an existing lock

Table 3.

(A) Detailed comparisons of adjusting lock fixes by *HFix* and *αFixer*.

Benchmark	Original?	Loc	*αFixer*		# of new (outer/inner) lock orders		# of Deadlocks		Average overhead	
			Case	L-type	*HFix*	*αFixer*	*HFix*	*αFixer*	*HFix*	*αFixer*
mozilla	×	106	AdjL-*A*	Global	0/1	0/0	1	0	-	12.80%
apache$_{21285}$	✓	45.34K	AdjL-*A*	Field	0/1	0/0	1	0	-	36.50%
apache$_{45605}$	✓	43.86K	AdjL-*B*	Field	0/0	0/0	0	0	26.80%	25.80%
mysql$_{12228}$	×	122	AdjL-*A*	Global	0/1	0/0	1	0	-	15.40%
mysql$_{12848}$	×	181	AdjL-*C*	Field	0/1	0/0	1	0	-	7.10%
mysql$_{169}$	×	145	AdjL-*C*	Field	0/0	0/0	0	0	38.20%	37.80%
				Sum:	**0/4**	**0/0**	**4**	**0**	**Avg.** **32.50%**	**22.57%**

(B) Detailed comparisons of gate lock fixes by *GLA*, *Grail*, and *αFixer*.

Benchmark	Original?	Loc	*αFixer*	# of new (outer/inner) lock orders			# of Deadlocks			Average overhead		
			L-type	*GLA*	*Grail*	*αFixer*	*GLA*	*Grail*	*αFixer*	*GLA*	*Grail*	*αFixer*
mozilla	×	106	Global	1/1	1/1	0/0	1	1	0	-	-	14.70%
aget$_{0.4}$	✓	0.32K	Global	1/0	1/0	1/0	0	0	0	5.70%	16.00%	8.06%
apache$_{21285}$	✓	45.34K	Field	1/1	1/1	0/0	1	1	0	-	-	44.36%
apache$_{21287}$	✓	45.61K	Field	0/1	0/1	0/0	0	0	0	133.20%	72.70%	3.35%
apache$_{25520}$	✓	45.61K	Field	0/0	0/0	0/0	0	0	0	163.00%	50.80%	25.63%
apache$_{45605}$	✓	43.86K	Field	0/0	0/0	0/0	1	1	0	-	-	31.66%
cherokee$_{0.9.2}$	✓	12.76K	Field	1/0	1/0	1/0	0	0	0	198.00%	108.60%	25.33%
memcached$_{127}$	✓	1.27K	Global	0/2	0/2	0/0	0	0	0	100.70%	118.30%	100.89%
mysql$_{12228}$	×	122	Global	1/1	1/1	0/0	1	1	0	-	-	17.70%
mysql$_{12848}$	×	181	Field	1/1	1/1	0/0	1	1	0	-	-	7.21%
mysql$_{169}$	×	145	Field	1/1	1/1	0/0	0	0	0	211.70%	288.10%	77.53%
mysql$_{2011}$	×	126	Field	3/0	3/0	3/0	0	0	0	54.90%	22.90%	5.63%
mysql$_{3596}$	×	122	Field	0/0	0/0	0/0	0	0	0	90.40%	31.60%	9.52%
mysql$_{644}$	×	118	Field	0/0	0/0	0/0	0	0	0	124.40%	95.40%	1.00%
mysql$_{791}$	×	125	Field	2/0	2/0	2/0	0	0	0	120.40%	24.50%	1.59%
			Sum:	**12/8**	**12/8**	**7/0**	**5**	**5**	**0**	**Avg.** **120.20%**	**82.90%**	**24.90%**

(*AdjL*) according to *Case A* (as shown in Figure 6). The sub-column "*L-type*" shows the lock visibility (*Global* or *Field*) that *αFixer* adjusted or inserted. The remaining columns of Table 3 (A) show (1) the number of new lock orders introduced by each technique (in form of "*outer*/*inner*"), (2) the number of deadlocks introduced by each technique, (3) the average overhead of each technique at the number of threads to be 128. In the last row, we also show the summation values or the average values for the last three major columns. Table 3 (B) shows the results of *αFixer*, *GLA*, and *Grail* on fixing all atomicity violations. Table 3 (B) can be read in the same way as Table 3 (A) except the fourth column which shows whether *αFixer* inserts a *Global* or a *Field* gate lock. In Table 3, the marks "-" under the last major column ("*Average overhead*") indicate that no data was collected because, for *GLA*, *Grail*, and *HFix*, after fixing, deadlocks frequently occurred (explained below). Note that, both kinds of fixes by *αFixer* were listed in Table 3 for comparison purpose.

HFix was only applicable to 6 atomicity violations. Among these 6 atomicity violations, three of them fall into *Case A*, one of them falls into *Case B*, and the remaining two fall into *Case C*. From Table 3 (A), it is observed that *HFix* introduced 4 inner lock orders and they formed 4 self-deadlocks.

From Table 3 (B), we observed that both *GLA* and *Grail* introduced 20 new lock orders on 11 benchmarks; *αFixer* introduced 7 new lock orders on 5 benchmarks. However, *αFixer* only introduced outer lock orders but no inner lock orders; and it did not introduce any deadlocks. This is consistent with its guarantee. But both *GLA* and *Grail* introduced 5 deadlocks, respectively.

Subsection 5.3.4 will discuss why *HFix*, *GLA*, and *Grail* introduced deadlocks.

5.3.3 Comparisons on Efficiency. From Table 3 (A), we observed that, on the only two benchmarks that *HFix* was able to fix correctly, both *HFix* and *αFixer* introduced almost the same overhead (i.e., 26.80% vs 25.80% and 38.20% vs 37.80%). This is because the two techniques produced the same fix except some fix code by *αFixer* to avoid introduce deadlocks.

From Table 3 (B), it is observed that, *αFixer* incurred significantly lower overhead than that by *GLA* and *Grail*, even on benchmarks that all the three techniques handled correctly. Compared with *GLA*, *Grail* incurred lower overhead. This is because *αFixer* is able to infer the lock visibility and can insert a class field gate lock; however, *GLA* always inserts global locks and *Grail* always inserts synthesized locks which may take effect but may introduce additional overhead on maintaining the map from a hash value to a unique lock.

Figure 9 shows the performance scalability of all techniques with increasing number of threads. The *x*-axis of each sub-figure shows the number of threads from 2 to 128; and the *y*-axis shows the time (in microsecond μs). Particularly, Figure 9(A) shows the scalability comparison of *HFix* and *αFixer*; and Figure 9(B) shows the scalability comparison of *GLA*, *Grail*, and *αFixer*. Note: if *HFix*, *GLA*, or *Grail* failed to correctly fix an atomicity violation (and no data was collected), the time of *αFixer* is still shown for comparison with that from the original runs.

The advantage of *αFixer* on inferring lock visibility is clearly reflected in Figure 9, where we highlight the sub-figures in gray background if a *Field* lock was adjusted or inserted by *αFixer*.

Figure 9(A) shows that, on two benchmarks that *HFix* were able to fix correctly, both *HFix* and *αFixer* introduced almost the

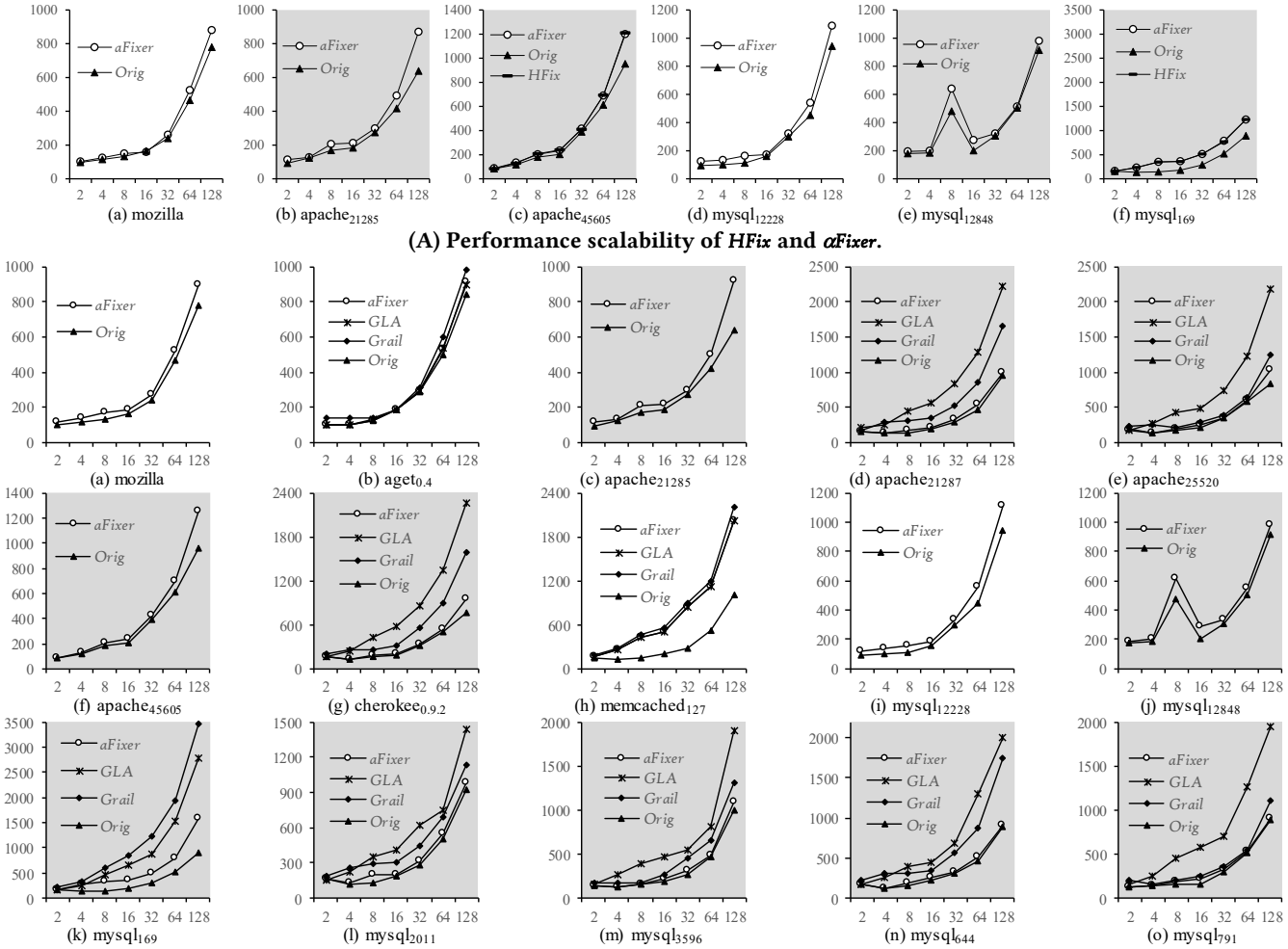

Figure 9. Performance scalability where the *x*-axis shows the increasing number of threads from 2 to 128 and *y*-axis shows the execution time (µs).

same overhead. Figure 9(B) shows that *αFixer* obviously incurred the least overhead compared with that by *GLA* and *Grail*. On 2 sub-figures not highlighted (i.e., Figure 9 (B.b) and (B.h)), all techniques incurred the similar overhead. On 2 remaining sub-figures not highlighted (i.e., Figure 9 (B.a) and (B.i)), all techniques except *αFixer* failed to fix the 2 atomicity violations, and no data was collected for *GLA* and *Grail*.

On the other hand, from Figure 9 (B), among most of sub-figures, *GLA* incurred the largest overhead; and *Grail* incurred less overhead than that by *GLA*. This is consistent with the previous experimental result [37]. However, on three atomicity violations *aget0.4*, *memcached127*, and *mysql169* (i.e., Figure 9 (B.b), (B.h), and (B.k)), *GLA* incurred less overhead than that by *Grail*. We have identified that, on the first two, the involved variables are global ones. Hence, *Grail* always synthesized the same gate locks. On the last one, although the variables are class fields, there is a global lock (named *LOCK_OPEN*) that is firstly acquired by both threads. Hence, *Grail* gained no advantage by synthesizing a gate lock. Instead, its synthesizing process increased its overhead.

5.3.4 Case Studies and Discussions. One of the main contributions of *αFixer* is, if a gate lock is inserted, the ability to infer lock visibility to reduce potential fixing overhead. We have presented how our *αFixer* inserted a class/struct field gate lock to fix *AV1* from *apache25520*. The case on *mysql791* is almost the same as *AV1*, where an atomicity violation occurs between two writes to *log_type* from a thread and a read to it from a different thread. And this variable *log_type* is from a class *MYSQL_LOG*.

To fix this atomicity violation, *Grail* also inserted a synthesized hash lock: "*G = contextL*(*hash*(&(*this->log_type*))); *acq*(*G*);", which unintentionally indicates that the lock *G* has nothing to do with the class *MYSQL_LOG*. However, *αFixer* identified that *log_type* is a member of class *MYSQL_LOG* and then inserted a lock *G* as a member of this class. Finally, it only inserted an acquisition "*acq*(*this->G*);" (and "*acq*(*mysql_log->G*);" in another thread). This fix might be more understandable as it is clearly reflected that the lock *G* is used to protect its neighbor member *lock_type* of the same class *MYSQL_LOG*. Hence, such kind of fixes provides an alternative way to improve fix acceptance to developers.

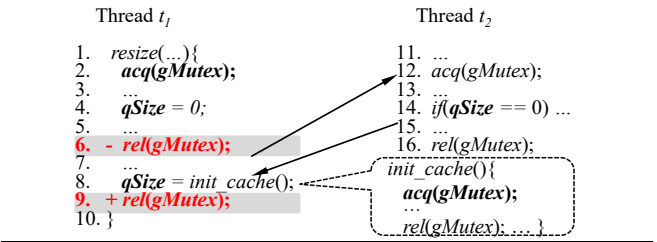

Figure 10. The atomicity violation from *mysql*$_{12848}$.

Study on Deadlock Introduction. From Table 3, *GLA* and *Grail* both introduced 5 deadlocks. On *mozilla*, the introduced deadlock is shown in Figure 3 and we have analysed the reason. The other 4 deadlocks (on *apache*$_{21285}$, *apache*$_{45605}$, *mysql*$_{12228}$, and *mysql*$_{12848}$) are similar to that on *mozilla*.

Among our benchmarks, *HFix* is only applicable to fix 6 atomicity violations but introduced 4 deadlocks (see Table 3 (A)). These deadlocks are self-deadlocks. We have analysed the introduced deadlock on *mozilla*.

Figure 10 shows another case from *mysql*$_{12848}$. In Figure 10, the atomicity violation occurs if, in between line 4 and line 8 (two writes to *qSize*), a second thread reads the value of *qSize*. This program contains a lock *gMutex* protecting the two accesses (at lines 4 and 14); hence, *HFix* is applicable to fix it by enlarging the lock scope of *gMutex* (between lines 2 and 6) to protect the write to *qSize* at line 8. That is, *HFix* moves the lock release *rel*(*gMutex*) to line 9 (i.e., "+ *rel*(*gMutex*)"). Then, all three accesses to *qSize* are protected by lock *gMutex*. However, at line 8, there is a call to function *init_cache*() which also contains a pair of lock acquisition and release on lock *gMutex*. Hence, thread t_1 is blocked when it enters function *init_cache*() to acquire lock *gMutex* as which has been acquired by itself at line 2. Thus, a self-deadlock occurs.

αFixer is able to correctly fix this atomicity violation. According to Figure 6, this atomicity violation falls into *Case C*. Hence, *αFixer* also tries to fix it by extending the lock scope of *gMutex*, which is the same as *HFix*. Next, *αFixer* has to ensure that no new (inner) lock order is introduced from lock *gMutex* to other locks in between line 6 and line 9. Then, *αFixer* found a pair of lock acquisition and release on the same lock *gMutex* after lock scope extension. Finally, *αFixer* put this lock acquisition together with lock acquisition at line 2 (i.e., in form of *acq*(*gMutex*, *gMutex*) and modified the property of this lock to be reentrant. In this way, this potential self-deadlock is avoided.

Discussion on HFix. *HFix* can also be adapted to change the locks to be re-entrant locks to avoid introducing self-deadlock. However, it still cannot avoid introducing other deadlocks as it may introduce new lock orders (see Section 0). Besides, *HFix* only targets on fixing atomicity violations involving locks. From our benchmarks, we see that there are still many atomicity violations (e.g., 9 out of 15) not involving locks; *HFix* fails to fix these atomicity violations. What's more, in some cases, even if an atomicity violation can be fixed by adjusting a lock, gate lock strategy might be better. For example, on *mysql*$_{169}$, a global lock *LOCK_open* is used to protect a class field variable. In such cases, a field lock under the same class might be better. It is difficult to say which fix is better without deeply understanding the source code. However, *αFixer* can produce both kinds of fixes.

6. RELATED WORK

Concurrency bugs widely exist in multithreaded programs [4][5][44]. Many techniques [11][39][18][22][23][34][37][50][52][54][62] have been proposed to fix them automatically. Many of these techniques insert gate locks to serialize executions of threads involved in the bug. The inserted gate lock may introduce performance bugs and deadlocks, as already noticed [22][23][41]. Although deadlocks could be theoretically detected via reachability analysis [27] or model checking [20], they cannot scale up to large-scale programs.

We have extensively discussed *GLA*, *Grail*, and *HFix*. *DFixer* [6] adopts lock pre-acquisiton to fix deadlocks by eliminating the hold-and-wait condition that is a necessary condition for a deadlock to occur. However, *DFixer* is not applicable to fix concurrency bugs involving memory accesses. *αFixer* is specially designed to fix atomicity violations.

Flint [36] tries to fix linearizability violation in concurrent compositions (i.e., Map data structure). *ConcBugAssist* [28] automatically infers wrong interleaving and then applies constraints (i.e., gate locks, *wait* and *notify* operations) to fix concurrency bugs. Unlike *αFixer*, *ConcBugAssist* may introduce deadlocks.

Concurrency bugs can also be prevented or avoided [12][13][17][26][51]. *Gadara* [51] and *Dimmunity* [26] prevent previously occurred deadlocks by invoking gate locks depending on whether a deadlock may occur based on execution context matching. *αFixer* could be adopted into these techniques to infer the visibility of gate locks to be inserted to improve runtime overhead.

Recovery techniques could be considered once a concurrency bug occurs. *ConAir* [61] tries to recover most concurrency bugs with low overhead. *Sammati* [45] and [46] aim to provide deadlock recovery by rolling back executions. Lin et al. [33] propose to change lock acquisition primitives (i.e., from *acq*() to *tryAcq*()). However, recovery might be infeasible as discussed in [33] considering unrecoverable operations (e.g., file IO operations).

7. CONCLUSION

Concurrency bugs are difficult to be fixed correctly. We presented *αFixer* to fix atomicity violations adaptively. It analyses the lock acquisitions involved in a given atomicity violation to determine whether to adjust existing lock acquisitions or to insert gate locks to fix atomicity violations. For the latter case, unlike existing approaches that insert global or synthesized gate locks, *αFixer* tries to insert either global locks or class/struct field locks to generate fixes that are more efficient. Besides, *αFixer* guarantees deadlocks-free fixes. We demonstrated the effectiveness and the efficiency of *αFixer* over a set of 15 real-world benchmarks.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their invaluable comments and suggestions on improving this work. This work is supported in part by National Natural Science Foundation of China (NSFC) (grant No. 61502465 and 61572150), National 973 program of China (2014CB340702), and the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (2017151).

REFERENCES

[1] LLVM Compiler Infrastructure, version 3.6, http://llvm.org.

[2] Concurrency Bugs, https://github.com/jieyu/concurrency-bugs.

[3] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multithreaded programs. IBM Journal of Research and Development, Vol. 54 (5), 520–534, 2010.

[4] E. Bodden and K. Havelund. Aspect-oriented race detection in Java. IEEE Transactions on Software Engineering (TSE), 36(4), 509–527, 2010.

[5] E. Bodden and K. Havelund. Racer: effective race detection using aspectj. In Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08), 155–166, 2008.

[6] Y. Cai and L.W. Cao. Fixing deadlocks via lock pre-acquisitions. In Proceedings of the 38th International Conference on Software Engineering (ICSE'16), 1109–1120, 2016.

[7] Y. Cai and W.K. Chan. MagicFuzzer: scalable deadlock detection for large-scale applications. In Proceedings of the 34th International Conference on Software Engineering (ICSE'12), 606−616, 2012.

[8] Y. Cai, S. Wu, and W.K. Chan. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In Proceedings of the 36th International Conference on Software Engineering (ICSE'14), 491–502, 2014.

[9] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus. NPEFix: Automatic Runtime Repair of Null Pointer Exceptions in Java. Technical Report 1512.07423, Arxiv, 2015.

[10] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09), 121–133, 2009.

[11] Q. Gao, Y.F. Xiong, Y.Q. Mi, L. Zhang, W.K. Yang, Z.P. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for C programs. In Proceedings of the 37th International Conference on Software Engineering (ICSE'15), 459–470, 2015.

[12] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. In Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'11), 15–28, 2011.

[13] P. Gerakios, N. Papaspyrou, K. Sagonas, and P. Vekris. Dynamic deadlock avoidance in systems code using statically inferred effects. In Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS'11), Article No. 5, 2011.

[14] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. 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 (ICSE'12), 3–13, 2012.

[15] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. Software Quality Journal, 21(3): 421–443, 2013.

[16] C. Le Goues, T. Nguyen, S. Forrest and W. Weimer. GenProg: A generic method for automated software repair. IEEE Transactions on Software Engineering (TSE), 38(1): 54-72, 2012.

[17] M. Grechanik, B.M. M. Hossain, U. Buy, and H. Wang. Preventing database deadlocks in applications. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13), 356–366, 2013.

[18] M. Grechanik, B.M. M. Hossain, and U. Buy. Testing database-centric applications for causes of database deadlocks. In Proceedings of the 2013 IEEE 6th International Conference on Software Testing, Verification and Validation (ICST'13), 174–183, 2013.

[19] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In Proceedings of the 30th International Conference on Software Engineering (ICSE'08), 231–240, 2008.

[20] K. Havelund. Using runtime analysis to guide model checking of java programs. In Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification (SPIN'00), 245–264, 2000.

[21] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12), 77–88, 2012.

[22] G. Jin, L.H, Song, W. Zhang, S. Lu, B. Liblit. Automated atomicity-violation fixing. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11), 389–400, 2011.

[23] G. Jin, W. Zhang, D. Deng, B. Liblit, S. Lu. Automated concurrency-bug fixing. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12), 221–236, 2012.

[24] P. Joshi, M. Naik, K, Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10), 327–336, 2010.

[25] P. Joshi, C.S. Park, K. Sen, amd M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09), 110–120, 2009.

[26] H. Jula, D. Tralamazza, C. Zamfir, and G.e Candea. Deadlock immunity: enabling systems to defend against deadlocks. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08), 295–308, 2008.

[27] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05), 505–518, 2005.

[28] S. Khoshnood, M. Kusano, and C. Wang. ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs. In Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15), 165–176, 2015.

[29] M. Kim, S. Sinha, C. Görg, H. Shah, M. J. Harrold and M. G. Nanda. Automated bug neighborhood analysis for identifying incomplete bug fixes. In Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST'10), 383–392, 2010.

[30] Z. Lai, S. C. Cheung and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), 235–244, 2010.

[31] C. Lattner and B. Adve. 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 (CGO'04), 75–86, 2004.

[32] D. Li, W. Srisa-an, and M. B. Dwyer. SOS: saving time in dynamic race detection with stationary analysis. In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11), 35–50, 2011.

[33] Y. Lin and S. S. Kulkarni. Automatic repair for multi-threaded programs with Deadlock/Livelock using maximum satisfiability. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14), 237–247, 2014.

[34] P. Liu and C. Zhang. Axis: automatically fixing atomicity violations through solving control constraints. In Proceedings of the 34th International Conference on Software Engineering (ICSE'12), 299–309, 2012.

[35] H.P. Liu, Y. Chen, and S. Lu. Understanding and generating high quality patches for concurrency bugs. In Proceedings of the 2016 24th

ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16), 715–726, 2016.

[36] P. Liu, O. Tripp, and X.Y. Zhang. Flint: fixing linearizability violations. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14), 543–560, 2014.

[37] P. Liu, O. Tripp, and C. Zhang. Grail: context-aware fixing of concurrency bugs. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14), 318–329, 2014.

[38] S. Lu , S. Park , E. Seo , Y.Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08), 329–339, 2008.

[39] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek. Detecting deadlock in programs with data-centric synchronization. In Proceedings of the 2013 International Conference on Software Engineering (ICSE'13), 322–331, 2013.

[40] M. Martinez and M. Monperrus. Mining repair actions for guiding automated program fixing. Technical report 1311.3414, Arxiv, 2012.

[41] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from exhibiting to healing. In Proceedings of 8th Workshop on Runtime Verification (RV'08), 104–118, 2008.

[42] S. Park. Debugging non-deadlock concurrency bugs. In Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA'13), 358–361, 2013.

[43] Y. Pei, C. A. Furia, M. Nordio, and B. Meyer. Automatic program repair by fixing contracts. In Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE'14), 8411:246–260, 2014.

[44] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12), 521–530, 2012.

[45] H. K. Pyla and S. Varadarajan. Avoiding deadlock avoidance. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10), 75–86, 2010.

[46] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies---a safe method to survive software failures. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05), 235–248, 2005.

[47] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10), 37–46, 2010.

[48] R.E.K. Stirewalt, R. Behrends, and L. K. Dillon. Safe and reliable use of concurrency in multi-threaded shared-memory systems. In Proceedings of the 29th Annual IEEE/NASA on Software Engineering (SEW'05), 201–210, 2005.

[49] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar. Test-driven repair of data races in structured parallel programs. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), 15–25, 2014.

[50] S.H. Tian and A. Roychoudhury. relifix: automated repair of software regressions. In Proceedings of the 37th International Conference on Software Engineering (ICSE'15), 417–482, 2015.

[51] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: dynamic deadlock avoidance for multithreaded programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08), 281–294, 2008.

[52] D. Weeratunge, X.Y. Zhang, and S. Jaganathan. 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 (OOPSLA'11), 19–34, 2011.

[53] D. Weeratunge, X.Y. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In Proceedings of the 15th edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10), 155–166, 2010.

[54] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen. Automatic program repair with evolutionary computation. Communications of the ACM (CACM), 53(5): 109–116, 2010.

[55] A. Williams, W. Thies, and M.D. Ernst. Static deadlock detection for java libraries. In Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05), 602–629, 2005.

[56] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10), 163–176, 2010.

[57] C.Y. Ye, S.C. Cheung, W.K. Chan, and C. Xu. Detection and resolution of atomicity violation in service composition. In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering (ESEC/FSE'07), 235–244, 2007.

[58] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11), 26–36, 2011.

[59] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA'09), 325–336, 2009.

[60] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In Proceedings of the 5th European Conference on Computer Systems (EuroSys'10), 321–334, 2010.

[61] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam. 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 (ASPLOS'13), 113–126, 2013.

[62] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - More accurate information-retrieval-based bug localization based on bug reports. In Proceedings of the 34th International Conference on Software Engineering (ICSE'12), 14–24, 2012.