Fixing Deadlocks via Lock Pre-Acquisitions

Yan Cai† and Lingwei Cao

State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
{ycai.mail, lingweicao}@gmail.com

ABSTRACT

Manual deadlock fixing is error-prone and time-consuming. Existing generic approach (*GA*) simply inserts gate locks to fix deadlocks by serializing executions, which could introduce various new deadlocks and incur high runtime overhead. We propose a novel approach *DFixer* to fix deadlocks without introducing any new deadlocks by design. *DFixer* only selects one thread of a deadlock to pre-acquire a lock `w` together with another lock `h`, where before fixing, the deadlock occurs when the thread holds lock `h` and waits for lock `w`. As such, *DFixer* eliminates a hold-and-wait necessary condition, preventing the deadlock from occurring. The thread performing pre-acquisition is carefully selected such that no other synchronization exists in between the two original acquisitions. Otherwise, *DFixer* further introduces a context-aware conditional protected by above lock `w` to guarantee the correctness of *DFixer*. The evaluation is on 20 deadlocks, including 17 from widely-used real-world C/C++ programs. It shows that *DFixer* successfully fixed all deadlocks. Whereas *GA* introduced 9 new deadlocks; a latest work *Grail* failed to fix 8 deadlocks and introduced 3 new deadlocks on others. On average, *DFixer* incurred only 2.1% overhead, where *GA* and *Grail* incurred 15.8% and 11.5% overhead, respectively.

CCS Concepts

• Software and its engineering➡Deadlocks • Software and its engineering➡Software testing and debugging.

Keywords

Deadlock, fixing, multithreaded program, lock order

1. INTRODUCTION

Deadlock [39] occurrence prevents a program execution from making further progress. In general, there are two kinds of deadlocks [28]: *resource deadlock* [7][29] and *communication deadlock* [28][34]. A *resource deadlock* occurs when a set of threads are holding some locks and are waiting for the other locks held by the threads in the same set. A *communication deadlock* occurs when some threads wait for some messages but they never receive these messages. In this paper, we focus on fixing resource deadlocks as two kinds of deadlocks are caused by different mechanisms and cannot be handled by the same technique [28].

† Corresponding author.

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884819

Manual bug fixing not only takes a long time [26] but is also error prone [60]. Recently, automated bug fixing become popular [19][20][21][33][44][57][64]. However, almost all existing techniques on concurrency bugs fixing insert new locks (known as gate locks) statically or dynamically to serialize all executions of threads involved in a concurrency bug, including *AFix* [26][27], *Axis* [36], *Grail* [37], *Gadara* [55], and [42]. By introducing new locks, new deadlocks may also be introduced [36][37][42]. Even manual fixing may also introduce deadlocks (e.g., 16.4% incorrect fixing indeed introduced new deadlocks [60]). *Axis* [36] further iteratively fixes introduced deadlocks by adding more new gate locks. *Grail* [37] adopts Petri-net analysis to eliminate such introduced deadlocks [55] which, however, is only applicable to deadlocks with two threads [37].

Introducing gate locks might be necessary to fix other concurrency bugs except deadlocks as fixing the former requires serialization of memory accesses from all threads of such bugs. However, deadlock is a kind of high level concurrency bugs caused by incorrect synchronization orders; whereas others (e.g., atomicity violations) are usually caused by missing synchronizations to protect the involved memory accesses from occurring in wrong orders. For example, many techniques differentiate concurrency bugs as deadlock bugs and non-deadlock bugs [33][39][43][54][62] as they require different techniques to detect and fix. *ConcBugAssist* [33] focuses on data races, atomicity violations, and order violations. Even among above listed fixing techniques, *AFix* cannot fix deadlocks [26][37] and *Grail* only targets to fix deadlocks of two threads which further uses Petri-net analysis to avoid introducing new deadlocks.

In this paper, we propose a novel strategy known as *DFixer* toward deadlock fixing. The key insight of *DFixer* is that a deadlock can be fixed by breaking a necessary condition for this deadlock to occur: the hold-and-wait condition of one thread involved in this deadlock. Suppose that if a deadlock *D* occurs, one of its thread `t` is waiting for a lock (denoted by `wLock` of thread `t`) while holding another lock (denoted by `hLock` of thread `t`) and this `hLock` is waited by another thread in the same deadlock *D*. Our fixing is, for the thread `t` of the deadlock *D*, its `wLock` should be acquired (i.e., *pre-acquired*) together with its acquisition on the `hLock`. This fixing strategy exactly breaks the hold-and-wait condition of a thread (e.g., holding a `hLock` and waiting for a `wLock` by above thread `t`) in a deadlock. Hence *DFixer* is able to fix the deadlock. The advantages of this strategy are that (1) it does not introduce any new lock by its design; (2) if a thread is properly selected (see Section 3) to perform its pre-acquisition on its `wLock`, no new deadlock is introduced; and (3) it exactly fixes a deadlock without serializing the executions from other threads that execute the same program code but do not participate in the deadlock, avoiding performance downgrade.

We have implemented *DFixer* for C/C++ programs and evaluated it on 20 deadlocks, including 17 real-world deadlocks and 13 of them are from three versions of widely-used large-scale `MySQL` database. We compared *DFixer* with the generic approach (denoted by *GA* that fixes a deadlock by inserting gate locks) and a latest concurrency bug fixing technique *Grail* (that is based on *GA* but inserts context-aware gate locks). The experiment result shows that *DFixer* was able to fix all these deadlocks without introducing any new deadlock; whereas *GA* fixed all deadlocks but also introduced 9 new deadlocks, and *Grail* not only failed to fix 8 deadlocks but also introduced 3 deadlocks on fixing other deadlocks. After fixing, *DFixer* incurred the least overhead (i.e., about 2% on average) while both *GA* and *Grail* incurred a significantly larger overhead (i.e., 15.8% and 11.5%, respectively).

The main contributions of this paper are as follows:

- It proposes a novel deadlock fixing strategy *DFixer* that introduces neither new locks nor new deadlocks.
- *DFixer* fixes a deadlock by selecting only one thread to pre-acquire a lock. This allows parallel executions of threads not from the deadlocks, avoiding performance downgrade.
- We implemented *DFixer* as a prototype tool (see http://lcs.ios.ac.cn/~yancai/dfixer) to evaluate *DFixer* with comparison to the generic approach *GA* and a latest technique *Grail*. The experiment results demonstrate the effectiveness and efficiency of *DFixer* compared to *GA* and *Grail*.

2. BACKGROUND AND MOTIVATIONS

2.1 Preliminaries

A deadlock occurrence involves a subset of the following events:

- $acq(t, m)$: A thread t acquires a lock m.
- $tryAcq(t, m)$: A thread t tries to acquire a lock m and it returns *true* if this try succeeds or *false* otherwise.
- $rel(t, m)$: A thread t releases a lock m.
- $wait(t, m)$: A thread t firstly releases a lock m and then waits to acquire it again on a notification (i.e., a communication message) from a different thread (see below).
- $notify(t, m)$: A thread t sends a notification to a different thread t' that is blocked on $wait(t', m)$. If there is no such a thread t', the notification is discarded.

In the rest of this paper, we may not mention thread t or even lock m when we discuss above kinds of events if they are implied by the context (e.g., we may refer to $acq(t, m)$ as $acq(m)$ or $acq()$).

If a thread firstly acquires a lock m and then acquires another lock n before releasing lock m, we say there is a ***lock order*** from lock m to lock n, denoted by $m \rightsquigarrow n$. If there exists another lock order $n \rightsquigarrow m$ (or $n \rightsquigarrow \ldots \rightsquigarrow m$ for multiple threads), we say it is a ***reversed lock order*** of the lock order $m \rightsquigarrow n$. Existence of a lock order and its reversed lock order indicates a potential deadlock depending on whether they can be formed at the same time in an execution; however, the absence of a lock order and its reversed lock order indicates the absence of any deadlock on these two locks.

To simply our analysis on lock orders, we assume that "*a thread can only release the lock that it acquired last*" [31]. Or at least, we assume this kind of lock acquisitions within deadlocks.

Formally, we adopt the lock dependency relation [12][29] to define deadlocks. A lock ***dependency*** $\tau = \langle t, w, h, L\rangle$ denotes that a thread t acquires a lock w while holding lock h and all locks in set L. Besides, each event occurs at a program location which is referred to as a *Site*. A sequence of k ($k > 1$) dependencies $D = \langle \tau_1, \tau_2 \ldots \tau_k\rangle$, where $\tau_i = \langle t_i, w_i, h_i, L_i\rangle$, forms a resource ***deadlock***, if:

(1) for $1 \le i \le k-1$, $w_i \notin L_i$, $w_i = h_{i+1}$ ($w_k = h_1$), and,

(2) for $1 \le i < j \le k$, $t_i \ne t_j$, $w_i \ne w_j$, and $L_i \cap L_j = \emptyset$.

The above definition describes that a set of threads wait mutually for a set of locks that are held by other threads in the same set. That is, each lock dependency τ_i is a necessary condition for a deadlock D to occur. For example, the deadlock shown in Figure 1(a) is described as $D_1 = \langle\langle t_1, n, m, \{\}\rangle, \langle t_2, m, n, \{\}\rangle\rangle$.

For a lock dependency $\tau = \langle t, w, h, L\rangle$ of a deadlock D, we refer to lock w as a ***wLock*** of thread t and lock h as a ***hLock*** of thread t as when deadlock D occurs, thread t is **w**aiting on lock w while **h**olding lock h. For example, on above deadlock D_1 in Figure 1(a), for thread t_1, its $hLock$ and $wLock$ are lock m and lock n, respectively; and for thread t_2, its $hLock$ and $wLock$ are lock n and lock m, respectively.

2.2 Generic Approach

A generic approach (*GA*) to deadlock fixing serializes the executions of all threads in the deadlock by inserting a gate lock. *GA* is widely adopted by existing works and is also adopted to fix other concurrency bugs [26][27][36][37][55][42]. As discussed in Section 1, *GA* could fix a deadlock; but it may easily introduce various new resource or communication deadlocks, and may further reduce the parallelism of executions from different threads due to over synchronization (i.e., introducing performance bugs [25]). We firstly illustrate *GA* on three deadlocks D_1 to D_3 as well as how it introduces various new deadlocks. For simplicity, we may not show lock releases if they are not related to our discussion.

Deadlock *D₁*: Figure 1(a) shows a program P_1 with a deadlock D_1 on two threads t_1 and t_2 as they acquire two locks m and n in reversing lock orders (denoted by two dotted arrows). To fix deadlock D_1, *GA* inserts a gate lock G to prevent two threads from acquiring two locks m and n concurrently as shown in Figure 1(b). *GA* correctly fixes D_1.

Deadlock *D₂*: Figure 1(c) shows a program P_2 with three threads t_1 to t_3 executing lock acquisitions and releases on locks m and n in three functions $f_1()$ to $f_3()$, respectively. Program P_2 contains a deadlock D_2 between threads t_1 and t_2 if the value of $need\_m$ at site s_{22} is *true*. (The variable $need\_m$ is used to prevent a second lock acquisition by thread t_3 via its call to $f_2()$ at site s_{33}).

Figure 1(d) shows the program fixed by *GA* on deadlock D_2. After fixing, deadlock D_2 never occurs due to the insertion of a gate lock G. However, considering three threads together, we could observe that a new deadlock is introduced between threads t_1 and t_3: right after thread t_1 acquires lock G and thread t_3 acquires lock m (at site s_{31}), thread t_1 cannot further acquire lock m (at site s_{11}) as which is held by thread t_3; next, thread t_3 cannot acquire lock G (at site s_{Ga2}) on its call to function $f_2()$ (at site s_{33}) as lock G is held by thread t_1. As a result, GA fixes deadlock D_2 but introduces a new resource deadlock on locks G and m.

Deadlock *D₃*: Figure 1(e) shows a program P_3 with two threads t_1 and t_2 to acquire locks m and n. Similar to program P_1, program P_3 contains a deadlock D_3. The difference is that program P_3 contains a pair of events $wait(n)$ and $notify(n)$ at sites s_{22} and s_{13}, respectively. However, the deadlock D_3 is not related to this pair of events. It occurs if (1) thread t_1 acquires lock m and is about to acquire lock n (at site s_{12}) and (2) thread t_2 acquires lock n and is about to acquire lock m at site s_{23} without executing $wait(n)$ at site s_{22} (i.e., the value of v is a $false$).

Figure 1(f) shows the program fixed by *GA* on deadlock D_3. After fixing, deadlock D_3 never occurs. However, a new communication deadlock is introduced: if thread t_2 acquires both locks G and n and then executes $wait(n)$ (i.e., the value of v is $true$), then the corresponding notification message will never be received by thread t_2. It is because thread t_1 is prevented from sending out the message at site s_{13} by executing $notify(n)$ at site s_{Ga1}, as lock G is

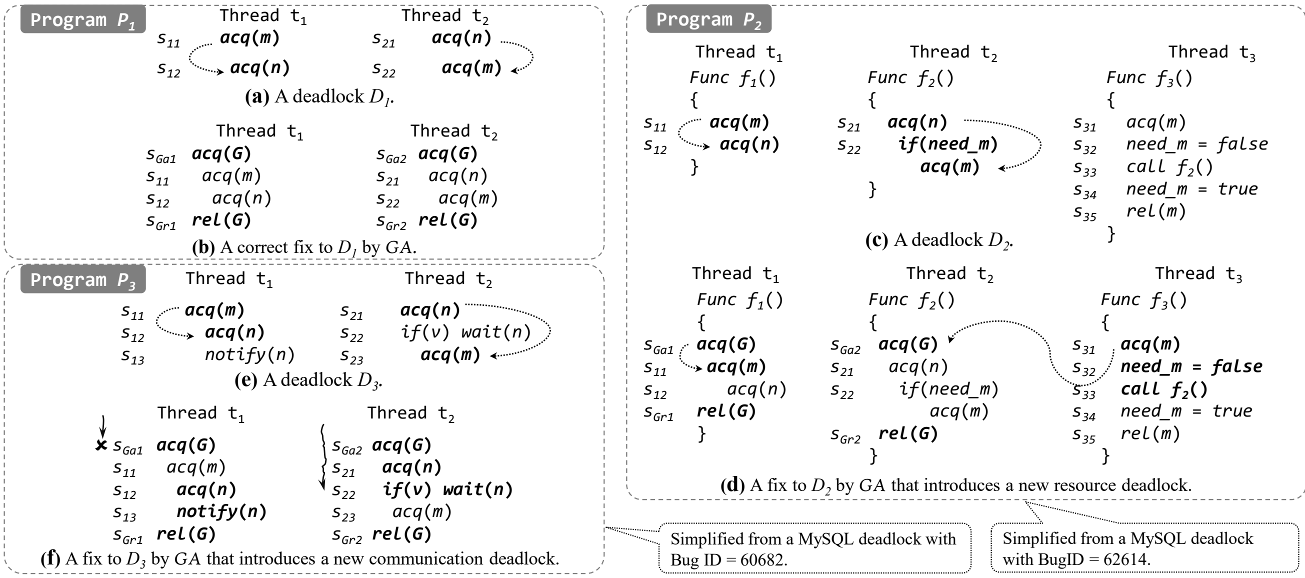


Figure 1. Three deadlocks (D_1 to D_3) and their fixing by *GA*.

already held by thread t_2. As a result, *GA* fixes deadlock D_3 but introduces a new communication deadlock.

Besides introducing new deadlocks, *GA* also introduces performance bugs because it inserts a global lock as a gate lock. For example, on D_1, if the two locks `m` and `n` of thread t_1 are different from the locks `m` and `n` of thread t_2, no deadlock occurs; hence, the two thread could execute in parallel. However, after fixing by *GA*, the two threads always execute sequentially due to a global gate lock, incurring runtime overhead.

The latest work *Grail* [37] follows *GA* approach, but inserts a context-aware gate lock (determined by both locks `m` and `n`). Thus, *Grail* does not reduce parallelism if no deadlock may occur. However, as *Grail* still adopts the gate lock strategy, it cannot avoid introducing new deadlocks like *GA* (e.g., on fixing deadlock D_2 and D_3); hence, *Grail* has to rely on other analyses (e.g., Petri-net model) to further prevent newly introduced deadlocks. Besides, as *Grail* needs to compute a context-aware lock involving all locks of a deadlock [37], it may fail on complex programs as some locks cannot be determined before some statements are executed. Due to these reasons, *Grail* failed to fix 8 out of 20 deadlocks in our experiment (in Section 5).

3. OUR APPROACH

3.1 Rationales and Overview of DFixer

GA fixes a deadlock by inserting new gate locks to serialize executions of the targeted deadlocks. Introducing new locks must introduce new lock orders from the introduced gate locks to the locks involved in targeted deadlocks. These newly introduced lock orders may form new deadlocks if their reversed lock orders are also introduced. For example, on fixing deadlock D_2 in Figure 1(c), the two newly introduced lock orders $G \rightsquigarrow m$ and $m \rightsquigarrow G$ form

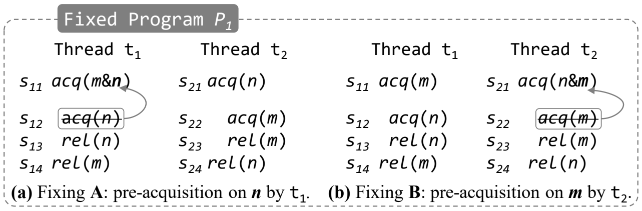


Figure 2. Two ways to fix deadlock D_1 in program P_1 by *DFixer*.

a new deadlock. Besides, the introduced new global locks are inserted to prevent all threads of a deadlock from executing concurrently, which may (1) block communication messages from sending out (e.g., on fixing deadlock D_3) or (2) introduce performance bugs by preventing other threads from executing the same program code concurrently.

Therefore, the key insights of deadlock fixing strategy are (1) to avoid introducing new lock orders and (2) to fix the executions exactly involved in the targeted deadlocks, but not to globally serialize all the involved program code. Based on above insights, we propose a novel strategy to fix deadlocks, known as *DFixer*. We note that a necessary condition for a deadlock *D* to occur is that each thread of *D* has to hold a `hLock` and then waits for a `wLock` (i.e., the hold-and-wait condition). *DFixer* exactly breaks such a necessary condition of one thread by fixing this thread to acquire its `wLock` together with its `hLock`, denoted by `acq(hLock&wLock)` which is formally defined in Section 3.2.1. That is, the selected thread by *DFixer* should either acquire the two locks at the same time or not acquire any one of them, breaking a hold-and-wait condition of the thread. We refer to this early acquisition by a selected thread on its `wLock` together with the acquisition on its `hLock` as a lock ***pre-acquisition***.

For example, Figure 2 shows program P_1 (see Figure 1) with deadlock D_1 fixed by *DFixer*. There are two ways for *DFixer* to fix deadlock D_1: (1) thread t_1 pre-acquires its `wLock n` (i.e., `acq(m&n)`), and (2) thread t_2 pre-acquires its `wLock m` (i.e., `acq(n&m)`, where the two pre-acquisitions are highlighted and also depicted by ▭→ from the original acquisition on the corresponding `wLock` to its pre-acquisition.

However, not all deadlocks could be fixed like the way to fix D_1. For example, if there is another lock acquisition `acq(p)` in between `acq(m)` and `acq(n)` of thread t_1 in P_1, pre-acquisition on lock `n` also introduces a new lock order $n \rightsquigarrow q$. Hence, such other synchronization events may also introduce various new deadlocks. To address such challenge, we carefully analyze these cases and further propose context-aware conditionals to guarantee the fixing correctness of *DFixer* via pre-acquisition.

Overall, the novelties of *DFixer* are: (1) neither new lock nor new lock order is introduced, introducing no resource deadlocks. (2)

DFixer only selects one thread to pre-acquire a lock and if any conditionals are also introduced, they are made to be context-aware (i.e., specified by both `hLock` and `wLock`). This allows all other threads to execute concurrently (if they are not involved in deadlock) and to execute without preventing communications from sending out, introducing no communication deadlocks.

3.2 Lock Pre-acquisitions and Context-aware Conditionals

In this subsection, suppose that for each thread in a deadlock, the acquisition on its *hLock* dominates its acquisition on *wLock* (i.e., if *acq*(*hLock*) is executed, *acq*(*wLock*) must be executed; and if not, the latter is not executed). Section 3.3 discusses how to handle the opposite cases.

3.2.1 Implement Lock Pre-acquisition

DFixer requires that the two locks *wLock* (*w* for short) and *hLock* (*h* for short) of a selected thread should be acquired at the same time. However, if the two statements are simply placed together (i.e., "*acq*(*h*); *acq*(*w*)" or "*acq*(*w*); *acq*(*h*)"), there always exists a lock order between two acquisitions (i.e., *h* ⇝ *w* or *w* ⇝ *h*, respectively), which either is the same as that before fixing (i.e., *h* ⇝ *w*) or may introduce a new deadlock as a new lock order is introduced (i.e., *w* ⇝ *h*).

To eliminate both lock orders, the two acquisitions must be performed at the same time. This could be implemented by re-writing locking mechanism. However, we propose to use the existing locking primitive *tryAcq*() (e.g., *pthread_mutex_trylock()* from Pthread) to implement *acq*(*h*&*w*) as follows:

```
acq(h&w) =
while( (tryAcq(h) && tryAcq(w)) == false)
 { rel(h); rel(w); }
```

That is, if a thread cannot acquire both locks, it immediately releases the acquired one if any. Although this implementation still introduces a lock order *h* ⇝ *w* which, however, does not introduce any new deadlocks even if there exists a reversed lock order (i.e., *w* ⇝ *h*). The reason is that the thread involved in above pre-acquisition immediately releases its lock *h*, which never results in a hold-and-wait condition on locks *h* and *w*. From this viewpoint by not introducing any deadlock, we regard that this implementation does not introduce a lock order *h* ⇝ *w*. In the rest of this paper, we directly use "*acq*(*h*&*w*)" to denote the pre-acquisition on a *wLock w* together with a *hLock h*.

Note that *tryAcq*() may introduce livelocks [35]. In theory, such a livelock cannot be eliminated. In practice, it can be easily resolved by inserting a random sleep (e.g., from 0 to 5 milliseconds as adopted in our experiment) right after two release operations.

3.2.2 Avoid Introducing Resource Deadlocks

Simply let a thread to pre-acquire its *wLock* may also introduce new (resource) deadlocks as it may introduce new lock orders. Let us consider a general case. Suppose for a deadlock *D* shown in Figure 3(a), thread t_1 is selected to pre-acquire its lock *w* together with its lock *h* (i.e., *acq*(*h*&*w*)) as shown in Figure 3(b).

After pre-acquisition, a challenge is that: if there exists other lock acquisitions, say on a lock *p*, between the original two acquisitions, a new lock order *w* ⇝ *p* is then introduced as denoted in a dotted arrow in Figure 3(b). For such a lock order *w* ⇝ *p*, if its reversed lock order *p* ⇝ *w* also exists (e.g., Figure 3(c)), a new deadlock is introduced.

Therefore, a straightforward approach for *DFixer* is to only select a thread of a deadlock such that, in between its *acq*(*h*) and *acq*(*w*),

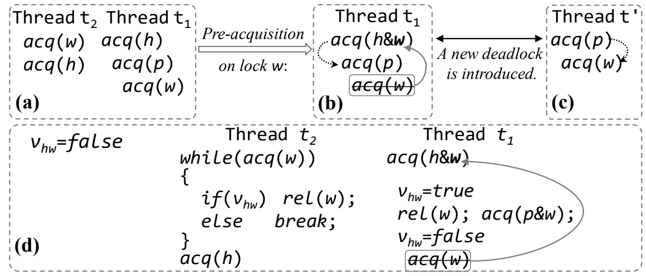


Figure 3. Fixing via lock pre-acquisition fails (above) and a conditional is required (below).

no other lock acquisition exists. For such a thread, its pre-acquisition on *w* not only fixes the deadlock but also introduces no new lock orders, hence introducing no new deadlocks.

However, above approach may fail on fixing some deadlocks as, for a deadlock, all its threads may acquire other locks in between their two acquisitions. We further propose *context-aware conditionals* (specified by both *hLock* and *wLock*) to handle such cases where a thread of a deadlock acquires other locks in between its two acquisitions, together with lock pre-acquisition. This fixing is shown in Figure 3(d) where the original deadlock is the one in Figure 3(a). Our proposal is, after pre-acquisition, if there is any other lock acquisition, say *acq*(*p*):

(1) *DFixer* firstly releases the pre-acquired lock *w* right before the acquisition on lock *p* and then re-acquires lock *w* together with the acquisition on lock *p* (i.e., from "*acq*(*p*)" to "*rel*(*w*); *acq*(*p*&*w*)").

(2) *DFixer* further guarantees that the second thread of the deadlock could not acquire lock *w* if the thread in (1) has released its pre-acquired lock *w* but not re-acquired it together with lock *p*.

The first step guarantees no new lock order *w* ⇝ *p* is introduced. However, the re-acquisition on lock *w* of *acq*(*p*&*w*) recovers the lock order *h* ⇝ *w* (formed by "*acq*(*h*&*w*) … *rel*(*w*); *acq*(*p*&*w*)"), failing to fix the deadlock considering its reversed lock order *w* ⇝ *h* from the second thread of the deadlock (or *w* ⇝ ... ⇝ *h* if the deadlock contains more than two threads). Therefore, *DFixer* has to guarantee that such a lock order *h* ⇝ *w* does not form a deadlock from the other thread that forms the lock order *w* ⇝ *h*. This is guaranteed in (2) that prevents two lock orders forming at the same time. This guarantee could be implemented by adding new locking mechanism or even communications (e.g., a pair of *wait*() and *notify*() primitives). However, this makes *DFixer* much more complex.

We then introduce a context-aware conditional v_{hw}, specified by both *hLock h* and *wLock w*, to provide the guarantee. Specifically, as shown in Figure 3(d), thread t_1 sets a v_{hw} to be *true* right before it releases its pre-acquired lock *w* and recovers it to be *false* after it re-acquires lock *w*. For thread t_2, after it acquires lock *w* (i.e., the *hLock* of thread t_2), it checks whether thread t_1 requires to re-acquire lock *w* (i.e., v_{hw} = *true*?); if so, it does not actually acquire lock *w* but waits until v_{hw} becomes *false*. As such, although thread t_1 forms a lock order *h* ⇝ *w*, it cannot be formed with the lock order *w* ⇝ *h* by thread t_2 at the same time. Besides, this conditional does not prevent either thread t_2 acquiring lock *w* at other sites or other threads acquiring lock *w*. Note that, this conditional is different from an ad-lock synchronization [59] as accesses to v_{hw} are always protected by the same lock *w*.

The cases where more than one other lock acquisitions exist in between *acq*(*h*) and *acq*(*w*) are handled in the same way.

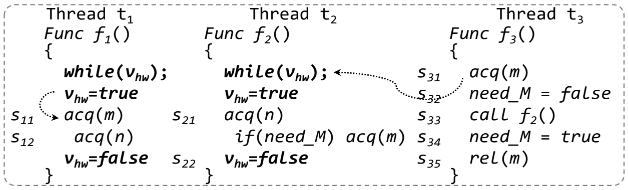


Figure 4. Deadlock fixing via a conditional without pre-acquisitions.

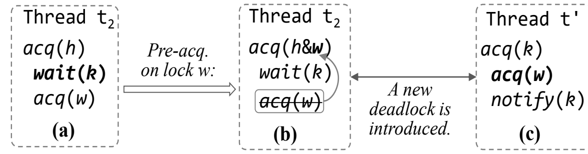


Figure 5. A communication deadlock introduced after pre-acquisition.

Discussion. To avoid introducing new deadlocks, *DFixer* fixes a given thread via lock pre-acquisitions and context-aware conditionals. A question is that: *without any pre-acquisition, could a deadlock be fixed directly by any conditionals alone*? We believe a deadlock could be fixed by conditionals only. However, it may involve complex control logic among two threads (e.g., considering protections on conditionals, two cases considering which thread firstly acquire their first lock); otherwise, hangs (like deadlock) may occur, prevent the threads from making any progress. For example, Figure 4 shows that a conditional v_{hw} is used to allow only one thread of a deadlock (e.g., deadlock D_2 in Figure 1(c)) to execute acquisitions on two locks at a time.

Then, a hang occurs as follows: after thread t_1 changes v_{hw} to be *true* and thread t_3 acquires lock *m*, t_1 cannot acquire *m* at site s_{11} and thread t_3 always executes *while*(v_{hw}) after it calls f_2() at site s_{33}. For deadlock D_3, if a conditional is applied to fix it, the result is similar as a gate lock is applied (i.e., a communication deadlock is introduced). Besides, the conditional has to be protected by a common lock. Introducing such a lock further brings a potential to introduce deadlocks; whereas, our conditional is rightly protected by the existing *wLock* of a selected thread.

3.2.3 Avoid Introducing Communication Deadlocks

Although *DFixer* aims to fix resource deadlocks, it should introduce neither resource deadlocks nor communication deadlocks. If *DFixer* fixes a deadlock without considering communications among all threads, a communication deadlock may also be introduced as shown in Figure 5. Figure 5(a) shows a general case: a thread t_2 (we use the symbol "t_2" not "t_1" to be consistent with deadlock D_3 in Figure 1) of a deadlock executes a *wait*(*k*) between its two acquisitions (where lock *k* is acquired before *wait*(*k*) and may be the same as lock *h*). After pre-acquisition (as shown in Figure 5(b)), a communication deadlock occurs if (1) thread t_2 is blocked on executing *wait*(*k*) while it is holding lock *w* and (2) a thread *t'* that should execute *notify*(*k*) is then blocked as it cannot acquire lock *w* as shown in Figure 5(c). The cases where a *notify*() eixsts is similar; we only discuss *wait*() below as its solution also applies to cases of *notify*().

Fortunately, our solution in the last subsection (to address other lock acquisitions *acq*(*p*)) also applies to the existence of above *wait*(*k*) in Figure 5(a). This is because an event *wait*(*k*) consists of three setps: release lock *k* (denoted by $rel_w(k)$), *wait* for a message related to lock *k*, and re-acquire lock *k* (denoted by $acq_w(k)$). As $rel_w(k)$ does not produce lock orders, we do not consider it. However, the *wait* requires that pre-acquisition on lock *w* should not prevent other threads sending a message via *notify*(*k*); and $acq_w(k)$ requires that no new lock order from the pre-acquired lock *w* is introduced. Hence, in both cases, the pre-acquired lock *w* should be released, which is similar with the case on avoiding introducing resource deadlocks and our above solution also applies to this case.

The only difference between *acq*(*k*) and $acq_w(k)$ of a *wait*(*k*) is that, the latter is implicitly included in the *wait*(*k*). That is, right after *wait*(*k*), the re-acquisition on lock *k* (i.e., $acq_w(k)$) has been done. Hence, we insert a *rel*(*k*) right after a *wait*(*k*) and then let the thread acquire both locks together:

```
wait(k) and acq(w) =
wait(k); rel(k); acq(k&w);
```

However, as we mentioned before, the lock *k* in *wait*(*k*) might be the lock *h*. This does not affect the fixing correctness of *DFixer* except one special case: the corresponding *notify*(*k*) (i.e., *notify*(*h*)) is expected to be executed by thread t_1 (i.e., thread t_1 is the same as thread *t'*) in between its acquisition and release on lock *w* (i.e., *hLock* of thread t_1). This case is actually the deadlock D_3 in Figure 1. For this case, above fixing fails as three threads (if they are likely to form a deadlock) are expected to execute by following the below orders according to our solution, resulting a controdiction:

1) Thread t_2 pre-acquires lock *w* together with lock *h* and then releases lock *h* right before *wait*(*k*).
2) thread *t'* firstly acquires lock *w* (***acq*(*w*)**) and then executes *notify*(*k*).
3) thread t_2 re-acquires lock *w* together with lock *k* (*acq*(*w*&*k*)).
4) thread t_1 (i.e., thread *t'*) should acquire lock *w* (***acq*(*w*)**).

When thread *t'* is actually the thread t_1 and the lock *k* is the lock *h*, their acquisitions on lock *w* (highlighted in 2) and 4)) are the same one, making above execution order infeasible. Actually, after executing the first three steps, there is no fourth step as it is included in step 2). As the step 4) is forced by our context-aware conditional, we then remove this conditional. That is, to fix deadlocks of this special case, the pre-acquisition alone is enough (on the thread where a *wait*(*k*) exists and lock *k* is its *hLock*).

Figure 6 shows fixing of deadlock D_3 on program P_3 if thread t_2 is selected. This fixing only involves pre-acquisition of *wLock m*.

3.2.4 Fix Multiple Deadlocks

A program may contain multiple deadlocks. These deadlocks could be incrementally (i.e., one by one) fixed by *DFixer*. However, *DFixer* could also be optimized to fix multiple deadlocks by selecting a shared thread, if these deadlocks share the thread as well as its two acquisitions (i.e., share a lock dependency).

3.3 Handle Program Control Flows

In Section 3.2, we assume that *acq*(*h*) dominates its *acq*(*w*) for a thread selected by *DFixer*. However, this is not always the case due to the complexity of program controls (e.g., an early return may exist in between *acq*(*h*) and *acq*(*w*)).

There are five basic cases according to whether the code lines between two acquisitions on *hLock* and *wLock* of a thread involve (1) single or multiple entries and single or multiple exits and (2) loop structures, as shown in Figure 7. To ease our following

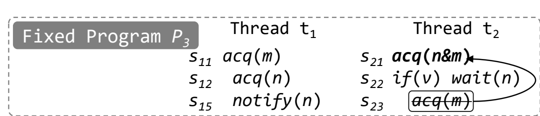


Figure 6. Fixing on deadlock D_3 by *DFixer*.

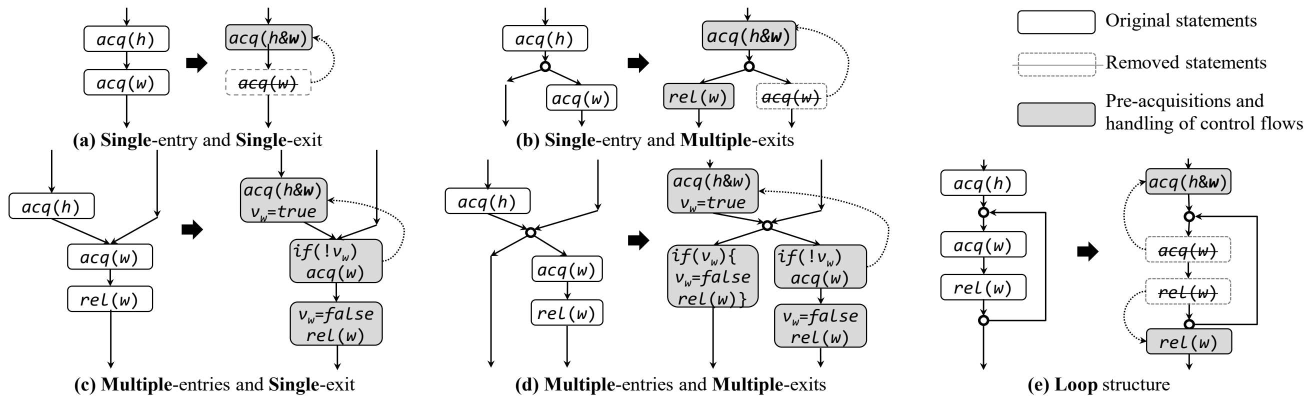


Figure 7. Five basic cases of control flows.

presentation, we suppose that the two locks *h* and *w* are the *hLock* and the *wLock* of a selected thread, respectively.

- **Single**-entry and **Single**-exit. In this case, *DFixer* directly inserts an *acq*(*w*) into the pre-acquisition block, as the execution of *acq* (*h*) always results in the execution of the original *acq* (*w*); and the original *acq*(*w*) should be removed.
- **Single**-entry and **Multiple**-exits. If there are more than one branch between the two lock acquisitions, *DFixer* has to insert a lock release statement (i.e., *rel*(*w*)) at the beginning of all other branches that do not contain the original *acq*(*w*).
- **Multiple**-entries and **Single**-exit. If there are multiple entries between the two lock acquisitions (e.g., *acq*(*w*) and *acq*(*h*) are in two different functions), *DFixer* adds a lock *w* specified conditional (i.e., v_w in Figure 7(c)) to indicate whether the lock *w* is previously acquired at its pre-acquisition site.
- **Multiple**-entries and **Multiple**-exits. This case is a combination of the last two cases. Therefore, *DFixer* not only inserts release statements on lock *w* to all other branches not containing the original *acq*(*w*), but also inserts a lock *w* specified conditional. For this case, the inserted release statements should also be executed conditionally.
- **Loop** structure. We firstly note that if the original *acq*(*w*) is within a loop, its corresponding *rel*(*w*) should also be in the same loop; otherwise, a self-deadlock exists. As *DFixer* requires that the lock *w* should be pre-acquired, it has to take the acquisition on lock *w* out of the loop body. Otherwise, the originally protected executions become unprotected during the second and later executions of the loop.

Among our example deadlocks, only deadlock D_2 involves multiple-exits on thread t_2. If thread t_2 is selected, the program control flow is fixed as shown in Figure 8 according to Figure 7(b).

3.4 DFixer Algorithm

Algorithm 1 outlines *DFixer*. Given a program *P* and a deadlock *D* from program *P*, *DFixer* firstly (Step 1) analyzes the program statements[1] involved in each thread of *D*. This analysis is based on a Depth-First-Search, for each thread *t*, to explore all possible paths from the statement of its *hLock* (i.e., *site*(*h*)) to the statement of its *wLock* (i.e., *site*(*w*)). Within this search, *DFixer* keeps all other locks *p* of *acq*(*p*) in $L_p(t)$ and all locks *k* of *wait*(*k*) or *notify*(*k*) in $WN_k(t)$.

[1] These statements should be extracted when the deadlock occurs as it is difficult for Object-oriented programs (e.g., C++) to statically extract the concrete calls between the two sites *site*(*h*) and *site*(*w*) for a thread.

Next (Step 2), *DFixer* tries to select a thread *t* such that the size of $L_p(t)$ and $WN_k(t)$ is the smallest one among all not selected (see Step 3) threads of *D*. If the size of $L_p(t)$ and $WN_k(t)$ is 0, *DFixer* directly applies pre-acquisition fixing alone; otherwise, it applies both pre-acquisition and a context-aware conditional to fix *D*. It then handles program control follows as said in Section 3.3.

After applying fixing, *DFixer* (Step 3) compiles the fixed program. If the compilation fails, *DFixer* returns to Step 2 to select another thread to fix deadlock *D* again. (This compilation failure is usually caused as some *wLocks* cannot be pre-acquired). If no thread is selected in Step 2, *DFixer* fails to fix the deadlock *D*.

3.5 Guarantee of DFixer

DFixer guarantees to fix a given deadlock *D* without introducing new resource or new communication deadlocks as Theorem 1.

Theorem 1. *Given a deadlock D from a program P, after fixing deadlock D by DFixer according to Algorithm 1: (1) the events in D do not form any deadlock occurrence, and (2) no other resource or communication deadlock is introduced.*

Proof Sketch. Suppose that the deadlock $D = \langle \ldots \langle t_i, w_i, h_i, L_i \rangle \ldots \rangle$ and *DFixer* selects the thread t_i to pre-acquire its *wLock* w_i.

Case 1: $|L_p(t_i) + WN_k(t_i)| = 0$. This case is straightforward. Before fixing, there are two lock orders: $h_i \rightsquigarrow w_i$ for thread t_i and $w_i \rightsquigarrow \ldots \rightsquigarrow h_i$ for other threads in *D*. After fixing, the lock order $h_i \rightsquigarrow w_i$ is removed due to pre-acquisition of w_i (i.e., *acq*(h_i & w_i)). Therefore, the events in *D* cannot form a deadlock occurrence. On the other hand, as $|L_p(t_i) + WN_k(t_i)| = 0$, no other lock acquisitions or *wait*() /*notify*() exist in between the original *acq*(h_i) and *acq*(w_i). Therefore, after pre-acquisition of the lock w_i, no new lock order is introduced and the pre-acquisition does not prevent any *wait*() or *notify*() from occurring. Hence, no new resource deadlock or communication deadlock is introduced.

Case 2: $|L_p(t_i) + WN_k(t_i)| \neq 0$. In this case, as the original lock order $h_i \rightsquigarrow w_i$ is eliminated after fixing, the events in *D* cannot form a deadlock occurrence. After fixing, no other lock order is

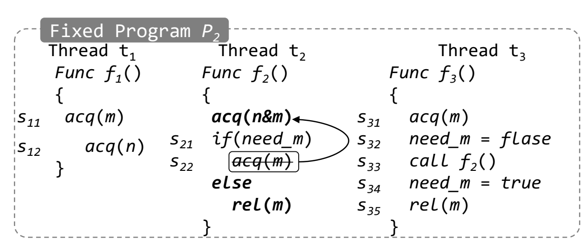


Figure 8. Fixing on deadlock D_2 by *DFixer* if thread t_2 is selected.

introduced except one for each lock p: $h_i \rightsquigarrow w_i$ due to the three fixing statements (i.e., $acq(\underline{h_i}$ & $w_i)$; $rel(w_i)$; $acq(p$ & $\underline{w_i})$) from thread t_i. However, there is a context-aware conditional v_{hw} is introduced (see line 14 of Algorithm 1) to determine whether the lock order $h_i \rightsquigarrow w_i$ is formed. The lock order only occurs when v_{hw} = $true$ (see thread t_1 in Figure 3(d)). But the original lock order $w_i \rightsquigarrow \ldots \rightsquigarrow h_i$ only occurs when $v_{hw} = false$ (see thread t_2 in Figure 3(d)). Hence, the two lock orders cannot be formed at the same time. Therefore, after fixing, the events in D as well as the introduced lock orders cannot form a deadlock occurrence. Besides, in this case, right before any other acq() or $wait$()/$notify$(), the pre-acquired lock w_i is released, introducing no new lock order and does not prevent thread t_i from executing $notify$(). Hence, neither new resource deadlock nor communication deadlock is introduced.

Based on the above two cases, Theorem 1 is proved. □

4. DISCUSSIONS AND LIMITATIONS

In practice, some *wLocks* depend on data structures which cannot appear together with the acquisition of their *hLocks*. Of course, *DFixer* is able to fix a deadlock via multiple ways. If a thread could not perform its pre-acquisition, another thread is then selected. However, the worst case is that no thread of a deadlock could perform a pre-acquisition on its *wLock*. In theory, this case does exist. Note that, this challenge is also suffered by *Grail*; however, *Grail* fails on fixing deadlocks with at least one such thread. The reason is that *Grail* requires exactly all *hLocks* and *wLocks* to abstract a context-aware gate locks. In our experiment, it failed on 7 deadlocks from `MySQL` due to this reason.

DFixer may also introduce more runtime overhead than *Grail* and *GA*. For example, after pre-acquisition of a *wLock*, the thread may take a long time before reaching the original acquisition and release of the *wLock*; and this may prevent other threads (not from the deadlock) acquiring the *wLock*. However, *Grail* and *GA* do not suffer this limitation as their inserted new locks only affect the executions of threads from the deadlock.

5. EXPERIMENT

5.1 Benchmarks

We collected a set of nine benchmarks: `DB Maintain`, `Bank Trans.`, `Dining Philo.`, `HawkNL`, `SQLite`, `OpenLDAP`, and three different versions of large-scale `MySQL Database Server`. The first three are used for deadlock research purpose and the rest are widely-used real-world programs. They totally include 20 deadlocks and each involves two or three threads, covering most of deadlocks cases [39]. All these benchmarks have been used in previous works multiple times [11][17][18][30][32][55] and are available either online [1][3][4][6] or from the previous works [30][55]. These benchmarks including their test cases are also

Algorithm 1: *DFixer*

1. **Input**: P and $D = \langle\langle t_1, w_1, h_1, \{\}\rangle, \langle t_1, w_2, h_2, \{\}\rangle \ldots\rangle$
2. //Step 1: identify program information
3. **for** each $\langle t, w, h, \{\}\rangle \in D$ {
4. let $L_p(t) = \emptyset$, $WN_k(t) = \emptyset$ //other $acq(q)$ and $wait(k)$ in between
5. Analyze P from $site(h)$ to $site(w)$ and update $L_p(t)$, $WN_k(t)$
6. //this analysis is based on DFS exploring
7. }
8. //Step 2: select a thread and apply fixing
9. **let** $t \in D$ be a thread with smallest $|L_p(t) + WN_k(t)|$ from all threads
10. in D except those threads previously selected.
11. **if** no such a thread t,
12. **print** "*DFixer* failed to fix deadlock D from program P." **halt**.
13. **if** $|L_p(t) + WN_k(t)| = 0$: Apply pre-acquisition on thread t.
14. **else** Apply pre-acquisition and context-aware conditional on thread t.
15. **handle** program control flow according to Section 3.3.
16. //Step 3: verify fixing
17. **let** *the fixed program P as program* P'.
18. **compile** P', if failed, **goto** Step 2.

available at http://lcs.ios.ac.cn/~yancai/dfixer.

Table 1 shows the statistics of all benchmarks, including benchmark names with version numbers (if available), Bug IDs (if available), program size (SLOC [5]), the number of threads of each benchmark ("`prog`"), the number of threads involved in each deadlock ("`dlk`"), the number of deadlocks ("`# of dlks`") in each benchmark. The next five columns show the statistics related to *DFixer*, including the number of other lock acquisitions ("L_q") and the number of $wait$()/$notify$() event ("WN_k") of each thread in each deadlock, respectively, whether there are *multi-entries*, *multi-exits*, and *loops* structures. We show the five metrics for each thread of each deadlock, where a single value or symbol is shown if they are same for all threads of a deadlock. The eleventh column shows the depth from $acq(hLock)$ to $acq(wLock)$ of each thread in each deadlock, in terms of the number of functions and the code lines (SLOC). For example, the first such value is "`0 (1) / 0 (3)`", indicating that the two acquisitions of both threads are within the same function and there are 1 and 3 code lines between them, respectively. Note that, some benchmarks include multiple deadlocks. These deadlocks from the same benchmark involve the same set of locks but occur in different scenarios (i.e., from different set of threads and in different functions), we treat them as different deadlocks as each of them should be fixed. However, due to page limit, the statistics only show the data of one deadlock for each benchmark; and the full statistics are also available at our online benchmark page. The last column shows whether deadlocks from a benchmarks could be fixed by lock pre-acquisition only (i.e., without a context-aware conditional).

5.2 Implementation and Experimental Setup

We implemented *DFixer* (as well as *GA* and *Grail*) on top of LLVM framework [2][38]. *DFixer* extends the *ModulePass* class

Table 1. Statistics of benchmarks and deadlocks.

Benchmark	Bug ID	SLOC	# of threads (prog/dlk)	# of dlks	L_p	WN_k	Multi-entries?	Multi-exits?	Any Loops?	Depth Func. (SLOC)	Pre-acq only?
DB Maintain	n/a	0.1K	3 / 2	1	0	0	✗	✗	✗	0 (1) / 0 (3)	✓
Bank Trans.	n/a	0.1K	3 / 2	1	0	0	✗	✗	✗	0 (3) / 0 (3)	✓
Dining Philo.	n/a	0.1K	5 / 5	1	0	0	✗	✗	✗	0 (1) / 0 (1)	✓
Hawknl (1.6b3)	n/a	9.3K	3 / 2	1	0	0	✗	✗	✗	0 (5) / 0 (6)	✓
SQLite (3.3.3)	1672	74.0K	3 / 2	2	0	0	✗	✗	✗	0 (1) / 1 (4)	✓
OpenLDAP (2.2.20)	3494	167.3K	5 / 2	1	1 / 0	0	✗ / ✓	✗/ ✓	✗ / ✓	1 (36) / 1 (29)	✓
MySQL-1 (6.0.4a)	34567	1,093.6K	16 / 2	4	1 / 0	0	✓ / ✗	✓/ ✗	✓ / ✗	8 (26) / 0 (2)	✓
MySQL-2 (6.0.4a)	37080	1,093.6K	17 / 2	1	1 / 3	0	✓/ ✓	✓/ ✓	✓ / ✗	4 (43) / 4 (15)	✗
MySQL-3 (5.5.17)	62614	1,282.7K	22 / 2	2	1 / 0	0	✗ / ✓	✗ / ✓	✗ / ✓	0 (1) / 1 (13)	✓
MySQL-4 (5.1.57)	60682	1,146.7K	19 / 3	6	1 / 0 / 0	1 / 1 / 0	✓	✓	✓ / ✗/ ✓	1 (23) / 4 (116) / 2 (18)	✓

Table 2. Detailed comparisons of *GA*, *Grail*, and *DFixer* (The avg. overhead in last row is on real-world deadlocks only).

Benchmark	# of deadlocks occurrences with random sleep				# of new deadlocks						Average overhead		
					Potential			*Triggered*					
	Native	*GA*	*Grail*	*DFixer*	*GA*	*Grail*	*DFixer*	*GA*	*Grail*	*DFixer*	*GA*	*Grail*	*DFixer*
DB Maintain	51	0	0	0	0	0	0	0	0	0	0.0%	0.0%	0.0%
Bank Trans.	53	0	0	0	0	0	0	0	0	0	184.4%	312.5%	3.1%
Dining Philo.	31	0	-	0	0	-	0	0	-	0	106.8%	-	0.6%
Hawknl	73	0	0	0	0	0	0	0	0	0	11.2%	27.1%	1.9%
SQLite	56	100	100	0	1	1	0	1	1	0	-	-	2.7%
OpenLDAP	47	0	0	0	0	0	0	0	0	0	2.0%	2.9%	0.5%
MySQL-1	62	0	0	0	0	0	0	0	0	0	7.1%	9.5%	0.5%
MySQL-2	37	0	-	0	0	-	0	0	-	0	19.4%	-	2.9%
MySQL-3	70	0	0	0	2	2	0	2	2	0	11.5%	6.3%	1.8%
MySQL-4	28	47	-	0	≥6	-	0	6	-	0	43.5%	-	4.1%
				Sum:	≥9	3	0	9	3	0	**Avg.**: 15.8%	11.5%	2.1%

of LLVM to perform a Depth-First-Search as shown in Algorithm 1 on Bitcode files. It firstly extracts synchronizations and controls (i.e., Control-Flow-Graphs and Call-Graphs) for each thread in between its *acq*(*hLock*) and *acq*(*wLock*). Next, based on extracted information, it applies pre-acquisition or/and context-aware conditionals. However, *DFixer* does not directly modify any Bitcode files; instead, it generates a fixing guide file (e.g., where and what should be inserted). We built a small program to translate this file into a Linux patch file. The patch file can be patched into the source code of the given program to fix a deadlock.

For the *DFixer* and *Grail*, some *wLocks* were not visible at the acquisition of their *hLocks*. Trying to solve the visibility issue might be difficult (as which requires C/C++ source file inclusion and could easily introduce compilation errors). Therefore, we set a pointer and assign the value of the *wLock* to the pointer. This pointer is used for pre-acquisition by *DFixer* or for computation of context-aware gate locks by *Grail*. We implemented context-aware conditionals for *DFixer* via a map structure from two given locks (*hLock* and *wLock*) to a Boolean value.

Grail is based on context-aware gate locks. It firstly generates a string by concatenating the addresses of *hLock* and *wLock*. Next, a second constant string (mapped in Java String Pool in Java 7, see `String.intern()`) with same values is returned as a gatelock object. This constant string is unique in each Java execution. We implemented this via a map structure in C++, which maps a concatenation of two lock addresses to a gate lock. There is no essential difference between our implementation in C++ and the *Grail* original implementation in Java.

After applying three techniques to all benchmarks, we ran each fixed program by each technique for 100 times and collected the cases where deadlock occurred. During this 100 runs, we inserted a set of random sleep before and after each original and fixing lock acquisition of each deadlock to amplify deadlock occurrence probabilities. (Note, without the random sleep, the 100 runs were not enough.) We also ran them for 10 additional times without sleep to collect their execution time. As all versions of `MySQL` are servers, we only collected their processing time on SQL queries (i.e., test cases) but not the whole program execution time.

We conducted the experiment on a ThinkPad workstation W540 with a 2.5 GHz (up to 3.4GHz) i7-4710MQ processor, installed with Ubuntu 14.04 and GCC 4.8.

Table 3. Summary of fixing on real-world deadlocks.

# of total real deadlocks	# of fixed			With overhead < 5%?		
	GA	*Grail*	*DFixer*	*GA*	*Grail*	*DFixer*
17	7 (41%)	6 (35%)	17 (100%)	✗	✗	✓

5.3 Result Analysis

5.3.1 Overall Effectiveness

Table 2 shows the detailed fixing results. The second major column shows that, before ("`Native`") and after fixing by each technique, how many deadlocks occurred in 100 runs (with random sleep). The third major column shows, after fixing, whether any new deadlock was introduced. We adopted manual inspection into the fixed source code firstly ("`Potential`") and then ran each fixed program to see whether any new deadlocks could be triggered ("`Triggered`"). The mark "-" indicates that no data was collected (e.g., a technique failed to fix a deadlock or a new deadlock always occurred after fixing). The last major column shows the average fixing overhead of the 10 additional runs (no sleep).

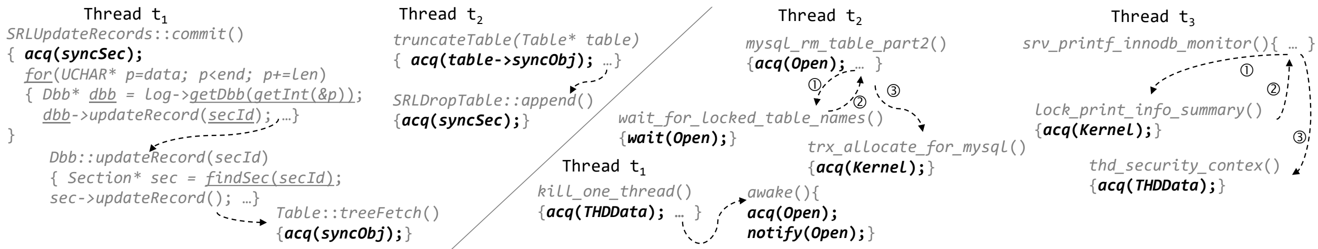
From the second major column of Table 2, we see that all deadlocks were likely to occur with random sleep. After fixing, no deadlocks occurred except on two benchmarks where new deadlocks were introduced. However, no deadlock occurrences did not indicate that no new deadlocks were introduced by three techniques. The second major column of Table 2 then shows that, after fixing, many potential deadlocks were introduced by both *GA* and *Grail*; and these potential deadlock were also triggered. However, *DFixer* did not introduce any potential deadlocks and no deadlock was triggered.

We further summarized the fixing results of three techniques on 17 deadlocks from real-world benchmarks in Table 3 (summarized from Table 2), including the number of deadlocks successfully fixed by each technique and whether there is any significant performance downgrade (e.g., more than 5% overhead). From the table, we observe that *Grail* and *GA* only successfully fixed 7 and 6 deadlocks out of 17 deadlocks, respectively; on other deadlocks, they either failed or/and introduced new deadlocks. However, *DFixer* fixed all 17 deadlocks correctly. Besides, both *Grail* and *GA* incurred larger than 5% overheads on average; whereas, *DFixer* did not incur such a large overhead across all benchmarks.

5.3.2 Overall Efficiency

The last major column of Table 2 shows the fixing overhead. Averagely, on real-world deadlocks, *GA* incurred 15.8% overhead, *Grail* incurred 11.5% overhead, but *DFixer* only incurred 2.1% overhead.

We note the following: *Grail* fixes a deadlock by inserting a context-aware gate lock, which could reduce fixing overhead compared to *GA* that inserts a global gate lock. Previous experiments [37] also verified this point. In our experiment, *Grail* incurred the largest overhead on four of six benchmarks. This, however, does not contradict the previous results [37]. The reason is that our



(a) Deadlock on MySQL-2 (BugID: 37080) (b) Deadlock on MySQL-4 (BugID: 60682)

Figure 9. Two deadlocks simplified from MySQL-2 **and** MySQL-4**.**

experiments focused on scenarios where deadlocks were likely to occur before fixing, while the previous experiments focused on scenarios where deadlocks (and atomicity violations) were not likely to occur (see the deadlock benchmark Log4j-bugID-24159 [37]). Therefore, for our cases, all three techniques have to serialize part or all executions in each deadlock as our test cases are designed to trigger deadlock occurrences. As both *GA* and *Grail* completely serialized the executions in our cases, they incurred more overhead than that by *DFixer* which not only serialized part of executions via per-acquisitions but also released pre-acquired locks if no deadlocks may occur (i.e., by fixing program control flows). On the other hand, *GA* simply inserted gate locks; whereas *Grail* had to compute context-aware gate locks by matching context in a map structure (even in its original implementation in Java, see Section 5.2). As a result, *Grail* may incur a larger overhead than *GA* on some benchmarks.

5.3.3 Detailed Discussion

DB Maintain and Bank Trans. These two benchmarks are simple ones like our example deadlock D_1. All three techniques correctly fixed them. On DB Maintain, no additional overhead was incurred by all techniques. However, on Bank Trans, the two threads acquire their first locks twice before they acquire their second lock, i.e., "*acq*(*m*); *acq*(*m*); *acq*(*n*)" (here no self-deadlock exists as two threads use recursive locking of Pthread). Besides, the two threads only acquire their second lock in less than half of all cases. As a result, the deadlock seldom occurs before fixing (without sleep). However, *GA* and *Grail* completely serialized two threads from their first acquisitions, resulting in heavy overhead (184.4% by *GA* and 300% by *Grail*). *DFixer* only selected one thread to pre-acquire its second lock together with its first acquisition; and if the thread takes another branch, it immediately releases the pre-acquired locks, only incurred 3.1% overhead.

Dining Philo. This benchmark includes five threads t_1 to t_5 (to simulate five philosophers) and each thread t_i acquires two locks l_i and l_{i+1} ($l_6 = l_0$). The deadlock occurs when each thread t_i acquires lock l_i and waits for lock l_{i+1}. Both *GA* and *DFixer* were able to fix it. However, *Grail* only targets to fix deadlocks with two threads. Therefore, it was unable to fix this deadlock. (Note that, for deadlocks involving more than two threads, *Grail* might generate a gate lock based on all these locks, which is feasible on Dining Philo. but may fail on other deadlocks (e.g., MySQL-4 discussed later)). After fixing, *GA* incurred 106.8% overhead as it completely serialized all executions of five threads; whereas, *DFixer* only incurred 0.6% overhead as it serialized only two threads by selecting only one thread to pre-acquire a lock.

Hawknl and OpenLDAP. On these two real-world benchmarks, all three techniques were able to fix them correctly. Note that, although one thread from OpenLDAP involves a lock acquisition on a third lock as shown in Table 1, *DFixer* fixed it by selecting the second thread to perform a pre-acquisition only. Actually, by selecting the first thread, *DFixer* was also able to fix it. On the performance, on Hawknl, *Grail* incurred the largest overhead (i.e., 27.1%), followed by *GA* incurring 11.2% overhead; *DFixer* incurred only 1.9% overhead. On OpenLDAP, both *Grail* and *GA* incurred larger overheads (i.e., 2.9% and 2.0%, respectively) than that by *DFixer* (i.e., 0.5%).

SQLite. The two deadlocks from this benchmark occur when a data race occurs on a variable *inMutex*. *DFixer* correctly fixed both deadlocks. However, both *GA* and *Grail* failed to fix them. On their 100 runs, there were exactly 100 occurrences of a new deadlock. The original deadlock is like our example deadlock D_2. In the original program, two threads of each deadlock acquire two locks *mutex1* and *mutex2* in a reversed order; however, a thread sometimes does not release lock *mutex2* (controlled by the variable *inMutex*). After fixing by *GA* and *Grail*, a gate lock *G* was inserted, resulting in two lock orders $G \rightsquigarrow$ *mutex1* and $G \rightsquigarrow$ *mutex2*. However, when one of two threads does not release lock *mutex2* and if it later re-acquires lock *mutex1*, it has to acquire the inserted gate lock *G*, resulting in a lock order *mutex2* $\rightsquigarrow G$. This lock order, together with the lock order $G \rightsquigarrow$ *mutex2* from another thread, forms a new deadlock. *DFixer* successfully fixed two deadlocks via pre-acquisition. Of course, it had to fix the control flows as its fixing on deadlock D_2 shown in Figure 8.

MySQL-1 and MySQL-2. There are totally 5 deadlocks within MySQL-6.0.4a. All three techniques correctly fixed the first four. However, for the last one (BugID=37080), both *DFixer* and *GA* fixed it but *Grail* failed. We simplified this deadlock in Figure 9(a). The thread t_2 firstly acquires a lock from a table *table->syncObj* and then acquires a global lock *syncSec*. The thread t_1 acquires the two locks in a reversed order. However, for thread t_1, after it acquires the global lock *syncSec*, it has to iteratively explore a linked structure data via a pointer p in a for-loop (underlined). From the pointer p, a *Dbb* pointer *dbb* is fetched (via function *getDbb*()); then a *Section* pointer *sec* is fetched (via function *findSec*()). The pointer *sec* points to a memory containing a table pointer *table* and its lock *table->syncObj*. As a result, before executing *findSec*(), the table is unknown and hence, the lock *syncObj* is also unknown. Therefore, *Grail* failed to compute a gate lock from both locks *syncSec* and *syncObj* (which is unknown). However, for *DFixer*, although the lock *syncObj* cannot be pre-acquired with lock *syncSec* by thread t_1, the lock *syncSec* can be pre-acquired with lock *table->syncObj* by thread t_2. Hence, *DFixer* fixed this deadlock.

MySQL-3. The two deadlocks from this benchmark are actually our example deadlock D_2. There are two locks *thread_count* and *index*. One of two threads acquires lock *index* if the value of the variable *need_mutex* is *true* in a function *purge_logs*(). However, this function may also be called from another function `purge_first_log()`. In this case, the lock *index* is acquired in `purge_first_log()`. Therefore, although *Grail* and *GA* fixed the original deadlock, they introduced a new deadlock if function `purge_logs()` is called in `purge_first_log()`. This produces a lock order *index* ↝ *GateLock*. Together with its reversed lock order *GateLock* ↝ *index* formed by another thread, a new resource deadlock is introduced. *DFixer* fixed this deadlock like its fixing to deadlock D_2 without introducing new deadlocks. On this benchmark, *GA* incurred the largest overhead (i.e., 11.5%), followed by *Grail* (i.e., 6.3%). *DFixer* only incurred 1.8% overhead.

`MySQL-4`. The deadlocks from this benchmark are complex. Figure 9(b) shows one of them. This deadlock involves three threads and three locks as highlighted. However, like our deadlock D_3, there is a pair of *wait*() and *notify*() on lock *Open*. *Grail* failed to fix this deadlock as locks *Open* and *THDData* are specified by a database; *Grail* failed to compute a gate lock. For *GA*, like its fix on D_3, it introduced a communication deadlock as, after fixing, once thread t_2 executes *wait*(*Open*), it holds the gate lock which prevents thread t_1 from executing both *acq*(*Open*) and *notify*(*Open*). This newly introduced communication deadlock was identified by our manual inspection and was also triggered. For *DFixer*, like its fixing on D_3, it fixed this deadlock by selecting thread t_2 to perform its pre-acquisition on lock *Kernel*. Note, *DFixer* could not select thread t_1 or thread t_3 to perform a pre-acquisition as both locks *THDData* and *Open* are specified by a database. *GA*, by serializing all three threads, incurred 43.5% overhead; whereas *DFixer* incurred only 4.1% overhead.

On `MySQL-4`, we manually identified 6 potential deadlocks introduced by *GA* which were also triggered. We suspect that more deadlocks were introduced by *GA* as there were many other parallel executions like threads t_1 and t_2, which could result in deadlock occurrences with the gate locks inserted by *GA*. However, these potential deadlocks were not triggered in our experiment as they may require different test cases. We use symbol "≥" to indicate this case in Table 2.

6. RELATED WORK

6.1 Deadlock Detection

Detection of deadlocks is mainly based on detection of either cycles in lock order graphs [7][8][9][15][24][40][41][52][58] or cyclic lock dependencies on lock dependency relation [11][12][29] statically or dynamically [7][15][41][48][52] [58].

Static ones may report many false positives [58] compared to dynamic ones, even with various filters [41]. Although dynamic one are relatively precise, they also report false positives. Kahlon et al. [31] theoretically analyze whether two threads may form a deadlock occurrence through reachability checking. Other works, recently, focus on how to actually trigger occurrences of real-world deadlocks by searching for possible scheduling [10][12][13][29][49]. *DFixer* focuses on how to fix deadlocks. It could be integrated with these techniques to fix their detected and triggered deadlocks as a subsequent action.

There are also many works on synthesizing concurrency bugs once observed. *ESD* [61] synthesizes an execution from a core dump file of an execution with a deadlock occurrence. *PENELOPE* [53] also synthesizes part of execution to replay an observed atomicity violations or deadlocks. These techniques may fail due to the lack of thread interleaving and test cases.

ConTeGe [45] targets to generate concurrent test cases so as to trigger an expected concurrency bug. *OMEN* [50] further synthesizes executions for deadlock triggering based on *ConTeGe*. *Sherlock* [16] actively infers test cases based on interleaving constraints of threads involved in a targeted deadlock via concolic executions [51].

Synthesis of executions and concurrent test cases may also help to verify the existence of a potential deadlock introduced by deadlock fixing approaches. However, unlike existing approaches, *DFixer* avoids introducing any new deadlocks by its design, no matter what test cases are given.

Deadlocks may easily exist in database applications (e.g., most of deadlocks in our benchmarks were taken from MySQL Database Servers). These deadlocks could also be detected and prevented by analyzing hold-and-wait relations (i.e., cycles) among threads and locks [22][23]. *DFixer* also breaks such a hole-and-wait relation to fix a deadlock.

6.2 Concurrency Bug Fixing and Recovery

Many techniques have been proposed to fix concurrency bugs [14][26][27][30][36][37][55][56][63]. However, almost all these techniques insert gate locks dynamically or statically to serialize executions of threads in a deadlock, which could introduce new deadlocks as discussed in this paper. *DFixer* distinguishes itself from all these works by its design to avoid introducing any new deadlocks.

Among above techniques, both *Gadara* [55] and *Dimmunity* [30] aim to prevent previously detected deadlocks occurring. They adopt a strategy like *GA* except that they may not always invoke acquisitions on the inserted gate lock via context matching. However, context matching may introduce false positives, which fails to prevent a deadlock occurring.

Recovery techniques could be integrated with deadlock detection and fixing. *Sammati* [46] aims to provide deadlock recovery by rolling back the executed operations, once a deadlock is detected. *ConAir* [62] tries to recover most concurrency bugs including deadlock. Lin et al. [35] propose to change lock acquisition primitives (i.e., from *acq*() to *tryAcq*() or from *tryAcq*() to *acq*()) to partially fix a deadlock. They further propose to recover program executions once a deadlock occurs [47], which may incur high runtime overhead. Besides, recovery from deadlock occurrence might be infeasible as discussed in [35] (e.g., when a thread involves file IO operations or accesses shared variables). Once *DFixer* fixes a deadlock, the deadlock never occurs. Therefore, there is no need for *DFixer* to adopt any recovery techniques.

7. CONCLUSION

Existing deadlock fixing strategies may easily introduce new deadlocks and may also incur high runtime overhead. We propose *DFixer* toward deadlock fixing without introducing any new deadlocks via lock pre-acquisition. We have evaluated *DFixer* on a set of widely used benchmarks including 20 deadlocks and also compared it with existing approaches. The experimental result shows that, compared to existing ones, *DFixer* not only fixed all deadlocks but also introduced no new deadlocks; besides, *DFixer* only incurred about 2% overhead on average which is significantly lower than that of compared approaches.

8. ACKNOWLEDGEMENT

We thank anonymous reviewers for their invaluable comments and suggestions on improving this work. We also thank Dr. Chao Wang and Dr. Lingming Zhang for their suggestions on this work. This work is supported in part by National 973 program of China (2014CB340702) and National Natural Science Foundation of China (NSFC) (grant No. 61502465, 91418206).

9. REFERENCES

[1] HawkNL, http://hawksoft.com/hawknl.

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

[3] MySQL, http://www.mysql.com.

[4] MySQL Bugzilla, http://bugs.mysql.com.

[5] SLOCCount 2.26. http://www.dwheeler.com/sloccount.

[6] SQLite, http://www.sqlite.org.

[7] 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.

[8] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In PADTAD, 2005.

[9] S. Bensalem, J.C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potential detected by runtime analysis. In Proc. PADTAD, 41−50, 2006.

[10] Y. Cai, C. Jia, K. Zhai, and W.K. Chan. ASN: A Dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs. IEEE Transactions on Parallel and Distributed Systems, 26(01), 13−23, 2015.

[11] Y. Cai and W.K. Chan. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. IEEE Transactions on Software Engineering (TSE), 40(3), 266–281, 2014.

[12] Y. Cai and W.K. Chan. MagicFuzzer: scalable deadlock detection for large-scale applications. In Proc. ICSE, 606−616, 2012.

[13] Y. Cai, S. Wu, and W.K. Chan. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In Proc. ICSE, 491–502, 2014.

[14] X. Chang, Z. Zhang, P. Zhang, J. Xue, and J. Zhao. BIFER: a biphasic trace filter approach to scalable prediction of concurrency errors. Frontiers of Computer Science (FCS), 9(6), 944–955, 2015.

[15] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In Proc. ASE, 480–491, 2009.

[16] M. Eslamimehr and J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In Proc. FSE, 353–365, 2014.

[17] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. In Proc. TLDI, 15–28, 2011.

[18] P. Gerakios, N. Papaspyrou, K. Sagonas, and P. Vekris. Dynamic deadlock avoidance in systems code using statically inferred effects. In Proc. PLOS, Article No. 5, 2011.

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

[20] C. L. 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 Proc. ICSE, 3–13, 2012.

[21] C. L. 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.

[22] M. Grechanik, B.M. M. Hossain, U. Buy, and H. Wang. Preventing database deadlocks in applications. In Proc. ESEC/FSE, 356–366, 2013.

[23] M. Grechanik, B.M. M. Hossain, and U. Buy. Testing database-centric applications for causes of database deadlocks. In Proc. ICST, 174–183, 2013.

[24] K. Havelund, Using runtime analysis to guide model checking of java programs. In Proc. SPIN, 245–264, 2000.

[25] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In Proc. PLDI, 77–88, 2012.

[26] G. Jin, L.H, Song, W. Zhang, S. Lu, B. Liblit. Automated atomicity-violation fixing. In Proc. PLDI, 389–400, 2011.

[27] G. Jin, W. Zhang, D. Deng, B. Liblit, S. Lu. Automated concurrency-bug fixing. In Proc. OSDI, 221 - 236, 2012.

[28] P. Joshi, M. Naik, K, Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In Proc. FSE, 327–336, 2010.

[29] P. Joshi, C.S. Park, K. Sen, amd M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In Proc. PLDI, 110–120, 2009.

[30] H. Jula, D. Tralamazza, C. Zamfir, and G.e Candea. Deadlock immunity: enabling systems to defend against deadlocks. In Proc. OSDI, 295–308, 2008.

[31] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In Proc. CAV, 505–518, 2005.

[32] T. Kelly, Y. Wang, S. Lafortune, and S. Mahlke. Eliminating concurrency bugs with control engineering. Computer, 42(12), 52–60, 2009.

[33] S. Khoshnood, M. Kusano, and C. Wang. ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs. In Proc. ISSTA, 165–176, 2015.

[34] E. Knapp. Deadlock detection in distributed database systems. ACM Computing Surveys, 19(4):303−328, 1987.

[35] Y. Lin and S. S. Kulkarni. Automatic repair for multi-threaded programs with Deadlock/Livelock using maximum satisfiability. In Proc. ISSTA, 237–247, 2014.

[36] P. Liu and C. Zhang. Axis: automatically fixing atomicity violations through solving control constraints. In Proc. ICSE, 299–309, 2012.

[37] P. Liu, O. Tripp, and C. Zhang. Grail: context-aware fixing of concurrency bugs. In Proc. FSE, 318–329, 2014.

[38] C. Lattner and B. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In Proc. CGO, 75–86, 2004.

[39] S. Lu , S. Park , E. Seo , Y.Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Proc. ASPLOS, 329–339, 2008.

[40] Z.D. Luo, R. Das, and Y. Qi,. MulticoreSDK: a practical and efficient deadlock detector for real-world applications. In Proc. ICST, 309–318, 2011.

[41] M. Naik, C.S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In Proc. ICSE, 386–396, 2009.

[42] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from exhibiting to healing. In Proc. RV, 104–118, 2008.

[43] S. Park. Debugging non-deadlock concurrency bugs. In Proc. ISSTA, 358–361, 2013.

[44] Y. Pei, C. A. Furia, M. Nordio, and B. Meyer. Automatic program repair by fixing contracts. In Proc. FASE, 8411:246–260, 2014.

[45] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In Proc. PLDI, 521–530, 2012.

[46] H. K. Pyla and S. Varadarajan. Avoiding deadlock avoidance. In Proc. PACT, 75–86, 2010.

[47] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies---a safe method to survive software failures. In Proc. SOSP, 235–248, 2005.

[48] R. Raman, J.S. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In Proc. PLDI, 531–542, 2012.

[49] M. Samak and M.K. Ramanthan. Trace driven dynamic deadlock detection and reproduction. In Proc. PPoPP, 29–42, 2014.

[50] M. Samak and M.K. Ramanathan. Multithreaded test synthesis for deadlock detection. In Proc. OOPSLA, 473–489, 2014.

[51] K. Sen and G. Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In Proc. CAV, 419–423, 2006.

[52] V.K. Shanbhag. Deadlock-detection in java-library using static-analysis. In Proc. APSEC, 361–368, 2008.

[53] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In Proc. FSE, 37–46, 2010.

[54] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar. Test-driven repair of data races in structured parallel programs. In Proc. PLDI, 15–25, 2014.

[55] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: dynamic deadlock avoidance for multithreaded programs. In Proc. OSDI, 281–294, 2008.

[56] D. Weeratunge, X. Zhang, and S. Jaganathan. Accentuating the positive: Atomicity inference and enforcement using correct executions. In Proc. OOPSLA, 19–34, 2011.

[57] 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.

[58] A. Williams, W. Thies, and M.D. Ernst. Static deadlock detection for java libraries. In Proc. ECOOP, 602–629, 2005.

[59] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In Proc. OSDI, article No. 1–8, 2010.

[60] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In Proc. FSE, 26–36, 2011.

[61] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In Proc. EuroSys, 321–334, 2010.

[62] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam. ConAir: featherweight concurrency bug recovery via single-threaded idempotent execution. In Proc. ASPLOS, 113–126, 2013.

[63] L. Zheng, X. Liao, S. Wu, X. Fan, and H. Jin. Understanding and identifying latent data races cross-thread interleaving. Frontiers of Computer Science (FCS), 9(4), 524–539, 2015.

[64] 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 Proc. ICSE, 14–24, 2012.