Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

9-2018

PFix: Fixing concurrency bugs based on memory access patterns

Huarui LIN

Zan WANG

Shuang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Dongdi ZHANG

See next page for additional authors

Citation

Author

Huarui LIN, Zan WANG, Shuang LIU, Jun SUN, Dongdi ZHANG, and Guangning WEI

PFɪx: Fixing Concurrency Bugs Based on Memory Access Patterns

Huarui Lin
School of Computer Software
Tianjin University
Tianjin, China
linhuaruitju@tju.edu.cn

Zan Wang*
School of Computer Software
Tianjin University
Tianjin, China
wangzan@tju.edu.cn

Shuang Liu*
School of Computer Software
Tianjin University
Tianjin, China
shuang.liu@tju.edu.cn

Jun Sun
Singapore University of Technology
and Design
Singapore
sunjun@sutd.edu.sg

Dongdi Zhang
School of Computer Software
Tianjin University
Tianjin, China
zhangdongdi@tju.edu.cn

Guangning Wei
School of Computer Software
Tianjin University
Tianjin, China
weiguangning@tju.edu.cn

ABSTRACT

Concurrency bugs of a multi-threaded program may only manifest with certain scheduling, i.e., they are heisenbugs which are observed only from time to time if we execute the same program with the same input multiple times. They are notoriously hard to fix. In this work, we propose an approach to automatically fix concurrency bugs. Compared to previous approaches, our key idea is to systematically fix concurrency bugs by inferring locking policies from failure inducing memory-access patterns. That is, we automatically identify memory-access patterns which are correlated with the manifestation of the bug, and then conjecture what is the intended locking policy of the program. Afterwards, we fix the program by implementing the locking policy so that the failure inducing memory-access patterns are made impossible. We have implemented our approach in a toolkit called PFɪx which supports Java programs. We applied PFɪx to a set of 23 concurrency bugs and are able to automatically fix 19 of them. In comparison, Grail which is the state-of-the-art tool for fixing concurrency bugs in Java programs can only fix 3 of them correctly.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

Multi-threading, Concurrency bugs, Memory-access pattern, Locking policy, Automatic fixing

---

*corresponding author

1 INTRODUCTION

Multi-threading is ubiquitous nowadays with the development of multi-core and many-core processors. Concurrency bugs (of a multi-threaded program) are bugs which may only manifest with certain scheduling, i.e., they are heisenbugs which may only be observed if we execute the same program with the same input multiple times. They are known to be hard to debug [55]. The difficulty in fixing concurrent bugs is at least threefold. Firstly, it is challenging to replay a concurrency bug. Even with the right test input, we must find a failure-inducing scheduling as the bug may manifest only with certain scheduling. In general, there could be exponentially many schedulings in the number of schedulable points. Researchers have tackled this problem by recording the scheduling during a test execution so that the bug can be replayed [12, 13]. Secondly, even with the recorded scheduling, debugging the multi-threaded bug may still be challenging due to the large number of steps and context switches in the test execution (i.e., the execution of the test case with particular scheduling), many of which may not be relevant to the bug. A programmer must be able to "abstract" the test execution so as to identify the root cause of the bug. Recently, there have been several proposals on abstracting a test execution using memory-access patterns. It has been shown that memory-access patterns are often correlated to the presence of multi-threaded bugs [50]. Thirdly, it is challenging to fix concurrency bugs as a fix must be able to avoid the bug with all possible scheduling.

In this work, we investigate the problem of fixing concurrent bugs. There have been multiple methods and tools proposed for fixing concurrency bugs [25, 29, 32, 34]. Existing approaches fix concurrent bugs resulting in either atomicity violations [10, 25, 26, 33], deadlocks [9, 53], or data races [48]. Roughly speaking, existing approaches design their fixes based on a few concrete execution traces which are either obtained from user-provided bug reports or

from runtime monitoring. We refer the readers to Section 5 for a detailed discussion on existing approaches.

In most of the cases, the fix in existing approaches amounts to inserting additional locks and synchronization to inhibit the bad concrete executions. In contrary, our approach is designed to systematically fix concurrency bugs based on inferring the intended locking policy (which allows us to reuse existing locks). As stated in [45] and other places, *the key to reduce concurrency bugs and ensure thread-safety is to design a locking policy according to the program specification.* For instance, a well-designed locking policy must guard the same variable with the same lock throughout the program and must guard related variables with the same lock throughout the program. Only by implementing a well-designed locking policy systematically throughout the program, we can guarantee thread-safety and thus free of concurrency bugs. Ideally, if the locking policy is explicitly documented, we can fix a concurrency bug by examining where the locking policy is not correctly implemented and fix it accordingly. In practice, however software engineers often do not document the locking policies properly. The challenge is then how to infer the locking policy and subsequently fix concurrency bugs systematically.

Our approach is designed as follows. Firstly, we apply existing approaches to systematically identify failure-inducing memory-access patterns with regard to a concurrency bug. The idea of correlating bugs with memory-access patterns has been explored in [40, 54, 57]. It has been found that memory-access patterns are often correlated with the root cause of concurrency bugs [50]. In our setting, the failure-inducing memory-access patterns represent violation of the locking policy which ought to be implemented systematically to prevent the bug. Afterwards, we automatically conjecture what is the intended locking policy of the program. The idea is to identify a well-designed locking which makes the failure-inducing memory-access patterns impossible. The last step of our approach is to systematically fix the program by consistently implementing the conjectured locking policy throughout the program. Our approach is different from those existing approaches on fixing concurrency bugs as our fixes are based on a comprehensive set of failure-inducing memory-access patterns (which have been shown to be complete [50]), whereas existing approaches are often based on concrete executions or particular patterns like single-variable atomicity violation. Furthermore, our fixes work through consistently implementing well-formed locking policies and thus not only fix those program executions which have been observed but also potentially those unseen ones. It is our belief that locking policies should play a central role in building thread-safe programs and thus should be the basis of fixing concurrency bugs.

Our approach has been implemented in a self-contained toolkit called PFix [4] (short for pattern-based fix) for Java programs. PFix is implemented based on existing frameworks including Java Pathfinder [19] and Soot [6]. We have experimented PFix with a set of 23 concurrency bugs, which we collect from previously published repositories. PFix is able to automatically fix 19 of them. On average, PFix spends 33.7 seconds to fix a bug, which we consider is reasonably efficient. For baseline comparison, we apply Grail [34] to the same set of bugs and it is only able to fix 3 of the bugs. We remark that other previously reported tools are either not maintained or target different programming languages (e.g., AFix [25],

```
public synchronized StringBuffer append(StringBuffer sb){
  if (sb == null) { sb = NULL; }
  //fix: synchronized (sb) {
1. int len = sb.length();
  int newcount = count + len;
  if (newcount > value.length) {expandCapacity(newcount);}
2. sb.getChars(0, len, value, count);
  //fix: }
  count = newcount;
  return this;
}

public synchronized StringBuffer delete(int start,int end){
  if (start < 0)
    throw new StringIndexOutOfBoundsException(start);
  if (end > count) { end = count; }
  if (start > end)
    throw new StringIndexOutOfBoundsException();
  int len = end - start;
  if (len > 0) {
    if (shared) { copy(); }
    System.arraycopy(value,start+len,value,start,count-end);
4. count -= len;
  }
  return this;
}

public synchronized void getChars(int srcBegin,
       int srcEnd, char dst[], int dstBegin) {
  if (srcBegin < 0)
    throw new StringIndexOutOfBoundsException(srcBegin);
  if ((srcEnd < 0) || (srcEnd > count))
3.  throw new StringIndexOutOfBoundsException(srcEnd);
  if (srcBegin > srcEnd) throw
    new StringIndexOutOfBoundsException("srcBegin>srcEnd");
  System.arraycopy(value,srcBegin,dst,
            dstBegin,srcEnd-srcBegin);
}
```

Figure 1: A concurrency bug in JDK1.4.2

AXIS [35] and CFix [26]). Furthermore, it has been reported that Grail is stricter better than AFix and AXIS in [34].

The remainders of the paper are organized as follows. Section 2 illustrates how our approach works through an example. Section 3 presents the details of our approach step-by-step. Section 4 evaluates our approach. Section 5 discusses related work and Section 6 concludes.

2 MOTIVATING EXAMPLE

In this section, we show how our approach works with an illustrative example. The example is a concurrency bug in the *StringBuffer* class in JDK1.4.2 [1]. Figure 1 shows the relevant part of the program under test, i.e., three methods of the *StringBuffer* class. Method *append* appends a given string buffer to the end of *this* string buffer; method *delete* deletes a substring (from index *start* to index *end*) from *this* string buffer; and method *getChars* copies from *this* string buffer into the destination character array *dst* with offset *dstBegin*. Note that method *append* calls *getChars* through the input string buffer *sb*. All three methods are synchronized.

A test case which potentially reveals the bug is shown in Figure 2. In the test case, two string buffer objects *a* and *b* are created and two threads are created, one executing *a.append(b)* while the other executing *b.delete(0, b.length())*. Executing the test case many times, we might observe a *StringIndexOutOfBoundsException(srcEnd)* due to line 3 in method *getChars*. One concrete execution which generates this exception is as follows. First, one thread executes line 1 in method *a.append(b)* to get string buffer *b*'s length, which is 5. Afterwards, the other thread executes method *b.delete()* to delete every char in *b*. As a result, *b*'s *count* becomes 0. Next, the thread

Table 1: Memory-access patterns in the example

Memory-Access Pattern	Susp.
$(t1, l1, \{count\}, \{\}), (t2, l4, \{\}, \{count\}), (t1, l3, \{count\}, \{\})$	1.00
$(t1, l1, \{count\}, \{\}), (t2, l4, \{\}, \{count\})$	0.50
$(t2, l4, \{\}, \{count\}), (t1, l3, \{count\}, \{\})$	0.49

executing method *a.append(b)* calls method *b.getChars()* at line 2 with *len* being 5. When method *b.getChars()* is executed, condition *srcEnd > count* (which is 5 > 0) is satisfied and thus the exception is thrown.

Although there could be many concrete executions which generate the exception, we can abstractly see that the exception occurs as long as the following memory-access pattern is present: thread 1 executes line 1, then thread 2 executes line 4, and then threads 1 executes line 3 to update the same variable *count*. In our approach, PFix systematically analyzes multiple failing and passing concrete executions in order to automatically identify a list of ranked abstract memory-access patterns in order to identify the root cause of the bug. There are a total of 17 generic patterns, and PFix scans through each concrete execution to count the number of times that an instance of those 17 patterns occurs. Table 1 shows the three patterns generated by PFix for this example where *l1, l2, l3* and *l4* denote line 1, 2, 3 and 4 respectively. Each pattern is composed of a sequence of steps of the form (t, s, R, W), which reads thread t executes instruction s to read variables in R and write variables in W. Note that the first pattern matches our understanding, whereas the other two are patterns which capture only part of the first pattern. PFix computes a suspicious score for each pattern based on how frequent they appear in failed executions and passed executions. The second column of Table 1 shows the respective suspicious scores. Note that the first pattern is always observed in failed test executions and thus has a suspicious score of 1.

Once we identify the failure-inducing memory-access patterns (i.e., the ones in Table 1), we proceed to generate a fix for the bug, based on the most suspicious pattern first. For different generic patterns, we have designed different fixes. According to our bug fixing algorithm (which will be presented in detail in Section 3), the fix for this 3-step pattern is to add additional synchronization so that the first step and the third step are in the same synchronization block. Intuitively, such a fix would make this memory-access pattern impossible and thus prevent the bug. The question is: which lock object do we use? To answer this question, PFix systematically monitors all concrete executions in order to identify the locking policy. That is, PFix monitors each shared variable and record which lock is held when the variable is accessed (for either reading or writing). In this example, PFix observes that variable *b.count* is accessed always with lock *b* held and variable *a.count* is accessed always with lock *a* held. It thus conjectures that *count* in the class is to be guarded by *this* according to the locking policy.

Based on this locking policy, PFix then proceeds to analyze whether it is possible to introduce a *synchronized* block *synchronized (sb) { }* which encloses both line 1 and 3. Note that the lock object is *sb* since both line 1 and line 3 access *sb.count*. As line 1 and 3 are in different methods, it is infeasible to introduce a *synchronized* block directly. PFix then analyzes the call graph in order to identify a common method where the *synchronized* block can be introduced.

```
public void test() {
    StringBuffer a = new StringBuffer("Hello");
    StringBuffer b = new StringBuffer("World");
    new Thread(new Runnable () {
        public void run() {
            a.append(b);
        }
    }).start();

    new Thread(new Runnable () {
        public void run() {
            b.delete(0, b.length());
        }
    }).start();
}
```

Figure 2: A test case for StringBuffer

In our example, because line 3 is executed due to the function call at line 2 and line 1 and line 2 are in the same function, PFix then proceeds to introduce a *synchronized* block which begins with line 1 and ends with line 2, as shown in Figure 1 in the form of comments.

Lastly, we validate the fixed program through standard means (i.e., extensive testing or using tools like Java pathfinder [19]) and repeat the above-discussed steps if necessary.

3 DETAILS OF THE APPROACH

In this section, we present details of each step in our approach. The input to PFix is a buggy program as well as a set of executions of a given test case (which can be obtained through standard means). We assume that at least one of the test executions results in failure (so that we know there is a concurrency bug).

3.1 Step 1: Identify Memory Access Patterns

The first step is to identify the failure-inducing memory-access patterns. The reason that we focus on memory-access patterns is that memory-access patterns are often correlated to bugs as shown in [37, 43]. An alternative is to focus on failure-inducing scheduling, which is not ideal for multiple reasons. Firstly, there might be a huge number of scheduling and many of them may be failure-inducing. Identifying all of them would be expensive if not infeasible. Secondly, even if we are able to identify all of them, it is not clear how to fix the program so that all of them are prevented. In comparison, concurrency bugs can be always reduced to one or multiple of a total of 17 generic memory-access patterns as shown in [42]. The idea is to design fixes for each and every one of the 17 patterns so that we can fix concurrency bugs systematically.

A memory access pattern is represented in the form of a sequence of steps of a test execution. Each step is a tuple of (t, s, R, W), where t is a thread id, s is a bytecode instruction generated by a statement in the program, R is a set of variables being read and W is a set of variables being written. Given a bytecode instruction s, we write $origin(s)$ to denote the program statement which generates the bytecode. In this work, we adopt the set of 17 memory access patterns defined in [42], shown in Table 2. The second column of the table shows the memory-access pattern. Each memory-access pattern is a sequence of at most four steps in the test execution, which concerns only with two threads and at most two variables. As a result, given a test execution, the number of memory-access patterns is bounded by $C_N^2 * C_M^2 * C_K^4$ where N is the number of shared variables, M is the number of threads and K is the total number of steps in the test execution.

Table 2: The generic memory-access patterns [42]

ID	Memory-Access Pattern
1	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\})$
2	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset)$
3	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\})$
4	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \{x\}, \emptyset)$
5	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \{x\}, \emptyset)$
6	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset), (t_a, s_k, \emptyset, \{x\})$
7	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \emptyset, \{x\})$
8	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \emptyset, \{x\})$
9	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_b, s_k, \emptyset, \{y\}), (t_a, s_l, \emptyset, \{y\})$
10	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{y\}), (t_b, s_k, \emptyset, \{x\}), (t_a, s_l, \emptyset, \{y\})$
11	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{y\}), (t_a, s_k, \emptyset, \{y\}), (t_b, s_l, \emptyset, \{x\})$
12	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset), (t_b, s_k, \{y\}, \emptyset), (t_a, s_l, \emptyset, \{y\})$
13	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{y\}, \emptyset), (t_b, s_k, \{x\}, \emptyset), (t_a, s_l, \emptyset, \{y\})$
14	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_b, s_k, \emptyset, \{y\}), (t_a, s_l, \{y\}, \emptyset)$
15	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{y\}), (t_b, s_k, \emptyset, \{x\}), (t_a, s_l, \{y\}, \emptyset)$
16	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{y\}), (t_a, s_k, \{y\}, \emptyset), (t_b, s_l, \emptyset, \{x\})$
17	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{y\}, \emptyset), (t_a, s_k, \emptyset, \{y\}), (t_b, s_l, \{x\}, \emptyset)$

It has been shown that these memory-access patterns capture the essence of multi-threaded bugs [42]. In addition, it is shown that this set is complete [50], as multi-threaded bugs can be reduced to one or more of these patterns. The memory-access patterns can be viewed as an abstraction of the test execution, which allows us to get rid of irrelevant details and yet preserve the cause of the multi-threaded bug.

Based on the frequency of the memory-access patterns in the test executions, we calculate a suspiciousness score for each pattern using Equation 1, where $\#fail(p)$ is the number of failing test executions in which the pattern p occurs and $\#succ(p)$ is the number of passing test executions in which the pattern p occurs.

$$suspicious(p) = \frac{\#fail(p)}{(\#fail(p) + \#succ(p))} \quad (1)$$

The larger the suspicious score, the more likely that the pattern is failure-inducing. Therefore, we sort the patterns based on their suspicious scores in the descending order.

3.2 Step 2: Identify Locking Policy

Once the failure-inducing memory-access patterns have been identified, we proceed to identify the locking policy on the relevant variables. The idea is to check if there are locking policies designed for the variables and whether the reason of the bug is that the locking policy has not been implemented properly on the part where the failure-inducing memory access pattern is observed. Formally, a locking policy is a function $lockP : V \rightarrow L$ where V is the set of variables and L is the set of locks. Note that we assume that a variable should be guarded by exactly one lock following the discussion in [45]. We use $lockP(x) = y$ to denote that variable x is guarded by lock y. A locking policy is consistently implemented if and only if every access of any x is guarded by a lock on $lockP(x)$ throughout the program[1].

In our approach, we infer the locking policy dynamically. That is, we monitor at runtime when a lock is held and released for each test execution. For any variable x, we then observe whether it is accessed

---
[1]Except the constructors since they are handled differently in Java.

(either for reading or writing) while some lock y is being held. We may observe that multiple locks are held while x is accessed, in which case $lockP(x)$ could be any of the held locks or even none of them if we assume that the locking policy is not consistently implemented. In general, we can obtain a bag of observations in the form of (x, y) where x is a variable and y is a lock. Afterwards, we heuristically conjecture that $lockP(x)$ is y if (x, y) occurs more than any other pairs (x, z) in the bag. For instance, in the example shown in Section 2, we obtain $(count, this)$ for every access of $count$ in the class and thus conclude $lockP(count)$ is $this$.

We remark that another way to obtain the locking policy is through static analysis, i.e., statically analyze when a lock is applied and released, as well as the variables accessed in between. However, compared to dynamic analysis, static analysis may suffer from imprecision due to aliasing, dynamic typing, etc. Therefore, in our work, we adopt dynamic analysis, which provides more accurate results.

3.3 Step 3: Fix Bugs

After obtaining the locking policy, we then examine each failure-inducing memory-access pattern (from the most suspicious to the least) and see whether the bug is due to an inconsistent implementation of the locking policy or rather the locking policy itself is problematic. In the following, we discuss how a bug is fixed for each failure-inducing memory access pattern. The general idea is to design a fix such that the corresponding memory-access pattern would be avoided. The algorithm is presented as Algorithm 1, which takes as input the failure-inducing pattern $pattern$ and the locking policy $lockP$ which we have inferred in the previous step. The algorithm to fix bugs according to memory access pattern 1 to 3, 4 to 8 and 9 to 17 (of Table 2) are shown from line 1 to 9, 10 to 14 and 15 to 28 (of Algorithm 1), respectively.

Line 1 to 9 in Algorithm 1 applies if $pattern$ is memory-access pattern 1 to 3 in Table 2. All of these patterns have two steps, where two different threads read/write on a shared variable in certain order. They are fixed in the same way. In the following, we use the pattern $(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\})$ (pattern 1 in Table 2) as an example to illustrate the fix. To make sure that these two steps are 'separated' in the fixed program, we distinguish two cases in fixing the bug. The first case is that s_i is not the last access of x in $origin(s_i)$ (which is the source code statement containing s_i) or s_j is not the first access of x in $origin(s_j)$. In such a case, we assume that the bug can be avoided if $origin(s_i)$ is finished before thread t_b preempts it or the other way around. To achieve that, we examine whether $lockP(x)$ is defined. If it is the case, we enclose $origin(s_i)$ and $origin(s_j)$ in a synchronization block with lock $lockP(x)$. If $lockP(x)$ is not defined, we introduce a new lock l, and implement the locking policy by enclosing every access of x with a lock on l. Note that the same locking policy should be propagated throughout the program for all accesses of x, not only those statements in the pattern. If however s_i is the last access of x in $origin(s_i)$ or s_j is the first access of x in $origin(s_j)$, we conclude that the bug occurs if state $origin(s_i)$ is followed by $origin(s_j)$ and this can not be fixed with a modified locking policy on x. The fix is then to prevent such ordering. This is achieved by introducing a fresh volatile boolean variable z with initial value false, adding $if(z)$ before $origin(s_i)$ and

---

Algorithm 1: Algorithm $fix(pattern, lockP)$

---

input : $pattern$: a failure-inducing pattern; $lockP$: a locking policy; the buggy program
output: the fixed program based on $lockP$

1 **if** $pattern$ is any of pattern 1-3 in Table 2 **then**
2   **if** $(t_a, s_i, \{x\}, \emptyset)$ is not the last x access of $origin(s_i)$ or $(t_b, s_j, \emptyset, \{x\})$ is not the first x access of $origin(s_j)$ **then**
3   **if** $lockP(x)$ does not exist **then**
4   set $lockP(x) = l$ where l is a fresh lock and enclose $origin(s_i)$ and $origin(s_j)$ with synchronization on l;
5   **else**
6   enclose $origin(s_i)$ and/or $origin(s_j)$ with synchronization on $lockP(x)$;
7   **else**
8   introduce a fresh volatile boolean variable z with initial value false ;
9   add $if(z)$ before $origin(s_i)$ and add $z = true$ after $origin(s_j)$;

10 **if** $pattern$ is any of pattern 4-8 in Table 2 **then**
11   **if** $lockP(x)$ does not exist **then**
12   set $lockP(x) = l$ where l is a fresh lock and enclose $origin(s_j)$ and enclose $origin(s_j = i)$ and $origin(s_k)$ in the same block with synchronization on l;
13   **else**
14   enclose $origin(s_j)$ and enclose $origin(s_j = i)$ and $origin(s_k)$ in the same block with synchronization on $lockP(x)$;

15 **if** $pattern$ is any of pattern 9-17 in Table 2 **then**
16   **if** $lockP(x)$ does not exist and $lockP(y)$ does not exist **then**
17   set $lockP(x) = l$ where l is a fresh lock and enclose the two steps of t_a in the same block with synchronization on l;
18   enclose the two steps of t_b in the same block with synchronization on l;
19   **else if** $lockP(x)$ does not exist **then**
20   set $lockP(x) = lockP(y)$;
21   enclose the two steps of t_a in the same block with synchronization on $lockP(y)$;
22   enclose the two steps of t_b in the same block with synchronization on $lockP(y)$;
23   **else if** $lockP(y)$ does not exist **then**
24   set $lockP(y) = lockP(x)$;
25   enclose the two steps of t_a in the same block with synchronization on $lockP(x)$;
26   enclose the two steps of t_b in the same block with synchronization on $lockP(x)$;
27   **else**
28   set $lockP(x) = lockP(y)$ and and apply the locking policy for every access of x and y;

```
1. void test () throws Exception {
2.   final D d = new D();
3.   Thread d1 = new Thread(){public void run () {d.m1();}};
4.   Thread d2 = new Thread(){public void run () {d.m2();}};
5.   d1.start(); d2.start();
6.   d1.join(); d2.join();
7.   if (d.x<1) { assert(false); } //d.x<2
}
class D {
8.   int x = 0;
9.   void m1() { x++; }
10.  void m2() { x*=2; }
}
```

Figure 3: An example illustrating repairing

```
class D {
  int x = 0; Object obj = new Object();
  void m1() { synchronized (obj) {x++;} }
  void m2() { synchronized (obj) {x*=2;} }
}
```

Figure 4: An example repair (case 1) for program in Figure 3

```
class D {
  int x = 0; volatile bool flag = false;
  void m1() { x++; flag=true; }
  void m2() { if (flag) {x*=2;} }
}
```

Figure 5: An example repair (case 2) for program in Figure 3

adding $z = true$ after $origin(s_j)$. Intuitively, it is then guaranteed that $origin(s_j)$ must be completed before $origin(s_i)$.

To illustrate the difference between these two cases, let us look at the example shown in Figure 3. Note that line 9 and line 10 both have two accesses (i.e., one read followed by one write) of variable

x. Given the assertion at line 7, one failure-inducing pattern which could be identified is: $(d1, l9_1, \{d.x\}, \emptyset), (d2, l10_2, \emptyset, \{d.x\})$. Intuitively, it means that thread $d2$ executes $m2$ first and thread $d1$ reads $d.x$ before $d2$ finishes executing line 10. Since $(d1, l9_1, \{d.x\}, \emptyset)$ is

not the last access of $d.x$ by thread $d1$, applying the above repairing strategy, we fix the program by introducing a fresh lock and surrounding both line 9 and 10 with a synchronization block. This is shown in Figure 4.

If we change the condition at line 7 to be the one in the comment ($d.x < 2$), one failure-inducing pattern which could be identified is: $(d2, l10_2, \emptyset, \{d.x\}), (d1, l9_1, \{d.x\}, \emptyset)$, which is an instance of pattern 2 in Table 2. Intuitively, it means that thread $d2$ executes $m2$ to finish first and thread $d1$ executes $m1$. Note that the memory access pattern in the previous paragraph is no longer failure-inducing. The reason is that the assertion failure will be avoided only if thread $d1$ finishes executing line 9 before thread $d2$ starts executing line 10. In this case, since $(d2, l10_2, \emptyset, \{d.x\})$ is the last access of $d.x$ by thread $d2$, and $(d1, l9_1, \{d.x\}, \emptyset)$ is the first access of $d.x$ by $d1$, applying the above repairing strategy, we fix the program as shown in Figure 5. Note that $flag$ is declared $volatile$ so as to avoid visibility issues (due to caching).

The fixes for patterns 4 to 8 are the same, as shown in line 10 to 14 in Algorithm 1. Intuitively, these failure-inducing patterns can be prevented if we prevent thread t_b from executing in between s_i and s_k. Thus, the idea is to implement a fix such that s_i and s_k are in the same synchronization block. There are two cases to fix the bug. If $lockP(x)$ does not exist, i.e., there lacks a locking policy for x, we introduce a new lock l and enclose s_i and s_k in the same synchronization block with lock l, and enclose s_j with a synchronization block with lock l as well. Note that by right, this new locking policy on x must be propagated throughout the program. If $lockP(x)$ does exist, we apply the same fix using lock $lockP(x)$ instead.

The fixes for pattern 9 to 17 are also the same. Intuitively, these patterns can be prevented if we implement a fix such that the two steps of thread t_a and t_b in these patterns become atomic. The remaining question is then which lock to use. Note that because these patterns are failure-inducing, we would assume that x and y are related and therefore the locking policy should be such that $lockP(x) = lockP(y)$. We distinguish four cases on fixing the bug. If both $lockP(x)$ and $lockP(y)$ are not defined, we use a fresh lock l to fix the bug. Otherwise, if either $lockP(x)$ or $lockP(y)$ is defined, we use the associated lock to fix the bug. Lastly, if both $lockP(x)$ and $lockP(y)$ are defined but $lockP(x) \neq lockP(y)$, we set $lockP(y)$ to be $lockP(x)$ to fix the bug. These four cases are handled accordingly in Algorithm 1, i.e., line 16-18 handle the case where no locking policies for x or y exist; line 19-26 handle the case where one locking policy exists either for x (line 19-22) or y (line 23-26) but not both, and line 27-28 handle the case where locking policies exist for both x and y. Note that PFɪx is designed to fix concurrency bugs using synchronization blocks instead of locks (i.e., $java.util.concurrent.Locks$) as synchronization blocks are easier to maintain.

3.4 Step 4: Fix the Fixed Program

After the last step, we have applied a fix according to the failure-inducing memory-access pattern and transformed the program to get a "fixed" version. Next, we apply a further step to fix the "fixed" program. This step has two main goals. One is to propagate the updated locking policy throughout the program. The other is to

make sure the transformed program is not only compilable but also efficient (e.g., without redundant locking).

To propagate the updated locking policy, for each shared variable x, we analyze the program systematically to identify part of the program which accesses x. For each access, we analyze whether the access is guarded by a lock by monitoring what are the locks which are held (and not released) before executing that part of the program in all the test executions. Let the set of locks held be denoted as L. We then check whether L includes $lockP(x)$, which is the lock for guarding x according to the identified locking policy. If it is, we do nothing. Otherwise, we introduce a $synchronized$ block to enclose the part of the program with a lock object $lockP(x)$. This way, we make sure the locking policy is systematically implemented throughout the program. Note that because our implementation is based on dynamic analysis, we will not propagate the locking policy to the part of the program which is never executed in the test executions.

As demonstrated in Section 2, we may not always be able to apply the fix according to Algorithm 1. For instance, two statements which we would like enclose in the same $synchronized$ block may be scattered in two very different parts of the program. We thus apply an approach similar to [32] to make sure the fixed program is syntactically correct. That is, we use Eclipse AST to check the scope of each synchronization block. If we find that the added synchronization crosses the original block of statements in the program, i.e., if statement, for loop and while loop, we adjust the scope of the added synchronization block so that it can include the entire block of statements. If we need to add synchronization blocks in two different functions, we first use Soot to identify the call function, and then find the right place to add synchronization so that the two statements are in the same block. In order to avoid redundant locking, if we find that the added synchronization block is in a constructor function, we discard the added synchronization.

3.5 Step 5: Test the Fixed Program

As the last step, we test the fixed program by using JPF to run the fixed program 100 times. We repeat the fixing process if there are errors occur during the testing. The number 100 is chosen based on our experience that a program found to have errors during manual inspections can almost consistently expose errors by executing 100 times with JPF random scheduling.

3.6 Overall Algorithm

The overall algorithm of our approach is shown in Algorithm 2. Given a buggy program and a test case, we run the test case many times in order to obtain a set of concrete test executions. In our implementation, we use Java Pathfinder to generate different scheduling. Next, we run the state-of-the-art approach Unicorn [42] to obtain a ranked list of potential failure-inducing memory access patterns (line 3). Then we infer locking policies dynamically as discussed above (line 5). After the locking policy is obtained, we fix the concurrency bug based on our fixing algorithm shown in Algorithm 1 (line 6). After applying the fixing, we apply step 4 to fix the fixed program. Lastly, we test the fixed program (i.e., run the program 100 times with the help of Java Pathfinder) to check whether the bug is indeed fixed (line 8). If all the 100 test executions

Algorithm 2: Overall Algorithm

> **input** : a buggy program and a test case
> **output**: a fixed program
> 1 **while** *true* **do**
> 2   test the buggy program to obtain a set of test executions;
> 3   obtain a ranked list of failure-inducing pattern P (Step 1);
> 4   select (and remove) the most suspicious pattern p;
> 5   identify locking policy $lockP$ for p (Step 2);
> 6 $fix(p, lockP)$ (Step 3);
> 7   fix the "fixed" program (Step 4);
> 8   test the fixed program (Step 5);
> 9   **if** *no failure is generated* **then**
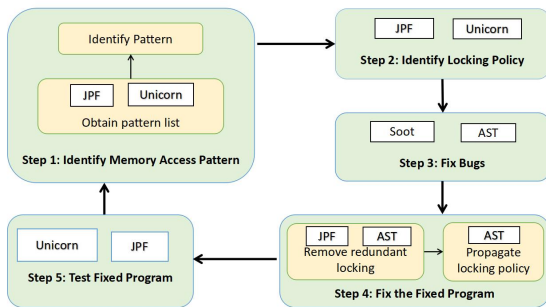> 10   break;

Figure 6: The overall structure of PFɪx

pass, we conclude that the bug is fixed and the algorithm terminates (line 10). If there are still failures observed during the test executions, we repeat the above process.

4 IMPLEMENTATION AND EVALUATION

In this section, we discuss the implementation details of PFɪx and the evaluation results.

4.1 Implementation

The proposed method has been implemented in a self-contained toolkit called PFɪx. PFɪx has a total of 4895 line of codes and the source code is available online at [4]. The overall structure of PFɪx is shown in Figure 6. PFɪx is implemented based on Java Pathfinder [19], Soot [6], Eclipse AST [3] and Unicorn [42]. PFɪx utilizes Java Pathfinder and Unicorn to automatically detect shared variables among multiple threads, and provides shared memory access information at run time in step 1. Unicorn is used to automatically identify a ranked list of suspicious memory-access patterns. Based on the patterns that is obtained from step 1, PFɪx identifies the corresponding locking policies based on Java Pathfinder and Unicorn in step 2. Then PFɪx implements our fixing method by utilizing Soot and Eclipse AST. Soot is a Java optimization framework. In this work, we use it to obtain the function call graph of a given program. The function call graph is necessary in our approach to

handle cases in which statements of a memory-access pattern are in different functions. In such a case, a call graph generated by Soot, which allows us to identify the right scope for introducing synchronized block. Eclipse AST, a part of Eclipse JDT is used to analyze the program syntactically, find the location that the fix patch should be inserted and then fine-tune the patch if the added patch results in compilation errors. In the last step, PFɪx tests the fixed program using Java Pathfinder, Eclipse AST and Unicorn.

4.2 Evaluation Settings

Our evaluation subjects include concurrency bugs in Java programs from multiple existing benchmarks, including the SIR repository [15, 21], Pecan's benchmark programs [1] and JaConTeBe [2]. Note that due to limitations of Java Pathfinder (unable to execute large Java programs), we are not able to apply PFɪx to all the concurrency bugs in the benchmarks. In total, we successfully applied PFɪx to 23 concurrency bugs to test our fixing method. The programs are chosen based on the following reasons. Firstly, our method focuses on fixing concurrency bugs in Java. Therefore, we target programs which are written in Java and are known to have concurrency bugs. Secondly, our approach adopts Java Pathfinder for bug localization and repair. We thus focus on programs which Java Pathfinder can handle[2]. Information on these concurrency bugs are summarized in Table 3. The actual programs are available in our Github repository [5].

In the following, we evaluate PFɪx in term of its effectiveness and efficiency. All our experiment results are obtained on a computer with 3.40GHz CPU, 16 GB memory. We use Windows 10 and JDK 1.8. For each concurrency bug, we first run Java Pathfinder 100 times to obtain test executions (which include both failing ones and passing ones). Unicorn [42] is then executed 100 times to obtain memory-access patterns in the test executions. Afterwards, we apply our approach for fixing the bug as discussed in Algorithm 1 based on the most suspicious pattern. After fixing the bug, we run Java Pathfinder 100 times in order to determine whether the bug has been fixed. In addition, we run random testing 100 times on the fixed program in order to further test it. If no bug is reported, we conjecture that the bug has been fixed. Otherwise, we obtain suspicious patterns and repeat our approach until either we fix the program or run out of suspicious patterns. For each fixed program, we then manually inspect whether the bug is truly fixed. We take the original fix as a reference to check whether PFɪx fixes the program correctly.

4.3 Evaluation Results

The evaluation results are shown in Table 3. The first three columns show the name of the programs, the number lines of the native code (excluding the invoked library) and the bug type (e.g., atomicity violation, data race, consistency bug)[3]. Columns 4 to 7 show the evaluation results with PFɪx, including the time (in seconds) used to identify the potential failure-inducing memory access patterns, the time used to fix the bug, the number of locks added in order to

---

[2]In the future, we plan to replace Java pathfinder with an approach based on code instrumentation so that our approach is more scalable.
[3]We use "unknown" to indicate that the root cause of the bug is complicated and it is hard to classify it into existing types

Table 3: Evaluation Results: time is measured in seconds

program name	#line	type	PFIx				Grail		
			time for patterns	time fixing	#lock	fix status	time fixing (s)	#lock	fix status
account	102	atomicity violation	22.7	238.75	1	success	3709	5	success
accountsubtype	138	atomicity violation	29.4	21.4	1	success	490	4	success
airline	51	atomicity violation	8.35	16.2	1	success	**NA**	**NA**	**NA**
alarmclock	206	unknown	10.75	113.75	**NA**	fail	**NA**	**NA**	**NA**
atmoerror	48	data race	7.3	5.95	1	success	146	1	fail
buggyprogram	258	atomicity violation	9.45	33.55	2	success	143	1	fail
checkfield	41	atomicity violation	7.15	9.8	2	success	**NA**	**NA**	**NA**
consisitency	28	consistency bug	6.75	9.95	1	success	**NA**	**NA**	**NA**
critical	56	atomicity violation	15.4	14.1	2	success	137	2	fail
datarace	90	data race	8.1	51.15	1	success	152	**NA**	fail
even	49	atomicity violation	7.25	91.15	1	success	155	1	fail
hashcodetest	1,258	atomicity violation	8.45	7.45	1	success	178	2	fail
linkedlist	204	atomicity violation	7.95	35.25	1	success	149	**NA**	fail
log4j	18,799	atomicity violation	22.9	20.35	1	success	**NA**	**NA**	**NA**
mergesort	270	unknown	17.95	204.7	**NA**	fail	630	6	fail
pingpong	130	data race	25.2	23.05	1	success	**NA**	**NA**	**NA**
pool	1,815	unknown	10.95	248.05	**NA**	fail	**NA**	**NA**	**NA**
ProducerConsumer	144	unknown	16	108.2	**NA**	fail	221	1	fail
reorder2	135	consistency bug	7.7	11.9	1	success	**NA**	**NA**	**NA**
store	44	atomicity violation	7.2	5.85	1	success	**NA**	**NA**	**NA**
stringbuffer	416	atomicity violation	7	22.2	1	success	172	**NA**	fail
wrongLock	73	atomicity violation	7.15	5.9	1	success	156	3	success
wrongLock2	36	data race	7.3	16.4	1	success	146	1	fail

fix the bug and the fixing status (success or fail). For each fix, we manually check whether the fix is correct or not. The data shows that PFIx is able to fix 19 out of 23 in the benchmark. There are four cases where our method is not able to (completely) fix the bug. We analyze them one by one in the following.

In the case of the *ProducerConsumer* program, the original program tries to guard a static shared variable with a lock on *this* object. Such a locking policy is problematic as there are multiple instances of the class in the program and different threads lock on different *this* objects before accessing the static variable, which is as good as no locking at all. PFIx is able to detect this ill-formed locking policy and successfully fix it using a shared lock. However, there is a further issue in the program. That is, if the consumer threads are very fast, there will be one producer thread waiting forever. The reason for this issue is that the main method proceeds to check the result (i.e., an assertion) without waiting for all producers to finish. Although it is possible for PFIx to find the relevant failure-inducing memory-access pattern for this issue and fix it through line 8 in Algorithm 1, PFIx times out without success. For the case of the *alarmclock* program and the *pool* program, we manually analyse the source code and found that PFIx failed to find the real failure-inducing memory-access pattern. This is possible since the test executions are randomly generated and thus the right failure-inducing pattern may not always be the most suspicious. For the *mergesort* program, the identified pattern is composed of statements from multiple classes. PFIx fixed the program by introducing a shared static object and adding a synchronized block for each statement in the pattern, which unfortunately introduced a deadlock. In general, it is possible to introduce deadlocks as PFIx

sometimes introduces additional synchronization. Such problems can be solved using existing approaches on fixing deadlocks [9], which we leave as future work.

For a baseline comparison, we apply Grail [34], which is the state-of-the-art concurrency bug fixing tool for Java programs, on the same set of benchmarks. We remark that other previously reported tools are either not maintained or target different programming languages (e.g., AFix [25], AXIS [35] and CFix [26]). Furthermore, it has been reported in [34] that Grail is stricter better than AFix and AXIS over a set of benchmarks. The results of Grail are shown in the last three columns of Table 3. Grail is built based on Pecan [20], which is a tool for detecting general access anomalies (AAs) in concurrent programs. AAs are similar to the memory-access patterns. Pecan generates AAs of length 2 to 4. Our inspection of Grail's source code shows that Grail is designed to only fix programs with length 3 AAs (i.e., atomicity violation)[4]. Among all the 23 programs in our benchmark, only 4 programs, i.e., *account*, *accountsubtype*, *wrongLock* and *buggyprogram* result in AAs of length 3. In order to run more programs with Grail, we then manually modify the output of Pecan so that the generated AAs have length of 3. Note that the AAs are modified in a way such that the cause of the bug in the original AA is not tampered. After the modification, Grail successfully runs on 14 benchmarks. The other 9 programs that cannot be executed by Grail are marked **NA** in the third last column of Table 3.

Out of the 14 buggy programs, Grail is able to generate fixes for 11 of them. Our manual inspection, however, shows that only 3 of

---

[4]We tried our best to contact the authors of Grail and got no response.

them are correct. In the following, we investigate why Grail fails on many of the benchmarks.

- Grail is unable to fix bugs which are across multiple classes or methods. We encounter such situations in programs *linkedlist*, *datarace* and *stringbuffer* where errors are reported by Grail during the fixing process.
- Grail sometimes fails to identify the correct scope for adding locks. As a result, bugs remain after the fix. This happens for programs *even* and *critical*.
- Grail has several implementation issues. For instance, if one AA contains multiple statements at the same line, the fixing codes may overwrite each other. This happens for program *wrongLock2*, *buggyprogram*, and *atmoerror*. The result is a program which has a statement for lock release with no matching lock acquire statement. In case of *hashcodetest*, the fix inserts two lock acquire operations but only one lock release operation, which results in a deadlock.
- Lastly, Grail may generate run time exceptions when they are applied to fix certain programs, e.g., program *datarace*.

There are 4 programs, i.e., *alarmclock*, *mergesort*, *ProducerConsumer* and *pool*, that neither PFɪx nor Grail can fix. All the bugs that can be fixed by Grail are fixed by PFɪx.

In terms of efficiency, PFɪx is more efficient. Its execution time ranges from a few seconds to a few minutes, whereas Grail is slower in most cases. For some cases like the program *account*, Grail takes 10 times more execution time than PFɪx. The reason may be because Grail needs to conduct constraint solving during bug fixing, which is very time consuming. Furthermore notice that in most cases, PFɪx generates a fix which uses less locks than Grail. For all benchmark programs, PFɪx introduces at most 2 locks whereas Grail generates as many as 6 locks. The reason is that PFɪx is designed to fix the bug based on the intended locking policy (i.e., using existing locks unless a locking policy is missing for some variables). Introducing excessive locks potentially makes the fix hard to comprehend and increases the likelihood of introducing deadlocks.

4.4 Threats to Validity

In the following, we discuss the threats to validity in our experiments. Firstly, PFɪx is implemented based on Java Pathfinder. As a result, we are not able to evaluate our approach with very large Java programs. While Java Pathfinder provides a good platform for implementing our approach (e.g., for identifying shared variables and obtaining status of locks), our method is not restricted to Java Pathfinder. In the future, we plan to mitigate our implementation entirely to be based on Soot and Eclipse AST, which hopefully will enable us to handle larger programs.

Secondly, in step 1 of our approach, we rely on existing approach to identify a ranked list of failure-inducing memory access patterns. Such ranking is based on simple statistical measurements and thus may not be accurate. Although the most suspicious pattern is usually the correct one in our experiments with the 23 benchmark programs, in general there is no guarantee that the real failure-inducing pattern will be the most suspicious. Furthermore, there may be multiple failure-inducing memory-access patterns. Different patterns might lead to different fixes, some of which may be better than others.

Thirdly, in step 4 of our approach, we verify the fixed program with random testing. Although we test for 100 times, it is possible that there are still concurrency bugs that are not revealed. Therefore, we manually inspect the fix to determine its validity in our experiments. This problem can be potentially solved by adopting approaches like symbolic execution or model checking.

Lastly, PFɪx potentially introduces deadlocks (e.g., the account program) and/or performance overhead. This is particularly the case when PFɪx generates multiple fixes after repeating the fixing process a few times. This problem can be potentially solved by applying existing approaches on fixing deadlocks (e.g., [9, 53]) to the fixed programs.

5 RELATED WORK

Our approach is inspired and related to mainly three groups of existing work, i.e., fault detection and localization, memory access pattern analysis and most importantly, concurrency bug fixing. We review them below.

Concurrency Bug Fixing. Our work is closely related to the line of work on fixing concurrency bug. Different approaches have been proposed to fix concurrency bugs effectively and efficiently. There are many proposals to fix concurrency bugs by eliminating erroneous interleaving patterns, e.g., [22, 25, 26, 35]. In particular, Huang *et al.* [22] proposed to fix concurrency bugs by inserting synchronization. Bradbury et al. [8], inspired by the use of genetic programming in sequential software debugging, proposes to apply genetic programming to fix concurrency bugs.

There are a few approaches for fixing atomicity violation bugs. AFix [25] takes the CTrigger's [41] output as input and adds a mutex lock to the program to fix concurrency bugs. AFix also modifies CTrigger to output all possible combinations of atomicity-violation triples and the complete call stack for each atomicity-violation related statements. It collects each bug report patch and statically identifies patches that can be merged or optimized to improve performance or readability (e.g., by removing redundant patches and merging patches). On the basis of AFix, CFix [26] fixes concurrency bugs due to order violation. By adding synchronization, CFix enforces *all A-B* or *first A-B* order relationships to fix order violation. CFix also enforces mutual exclusion with the same method. Axis [35], similar to AFix, fixes atomicity violations by adding mutual exclusion locks and synchronization measures. Axis additionally takes efforts to reduce the possibility of introducing deadlocks. Axis abstracts the source program into Petri net [49] using the supervisory control theory, i.e., Supervision Based on Place Invariants (SBPI) [24], which turns the problem of program repair into a constraint solving problem. AlphaFixer [11] specializes in fixing atomicity violations, it summarizes previous approaches based on locking. By analyzing the lock acquisitions, AlphaFixer fine-tunes the locking so that it is possible to reduce the introduction of deadlocks.

In the name of generating high quality patches, Liu *et al.* [32] proposed HFix, which designs fix strategies guided by a survey of 77 manual patches of real-world concurrency bugs. In addition to using mutex locks, HFix can also use the create and join operations of threads, while modifying the original locks to achieve the purpose of fix. Grail [34] fixes concurrency bugs by adding locks in ways

similar to AFix and Axis. Compared to AFix, Grail additionally takes measures for deadlock-freedom. Grail builds a Petri net analysis model of the buggy program. The model is context-aware and considers lock alias by adding constraints to the Petri net model. Compared to Axis, Grail guarantees at least the same concurrency level if not higher. Grail can be time consuming due to the use of constraint solving. Besides, Grail fails to consider related variables, and thus cannot fix multi-variable bugs.

Although a lot of work has been proposed on concurrency bug fixing, there is still room for improvement (as demonstrated in our experiments). Our approach distinguishes itself from the above work in multiple ways. Firstly, our fixing is based on memory access patterns, which is the root cause of the concurrency bugs. It helps us accurately identify the statements which cause the bug. Although some existing approaches like AFix and CFix use patterns like *(p, c, r)*, which are similar to some of our memory-access patterns, their approaches are limited to only a few patterns. In comparison, our set of patterns are shown to be complete [50]. Secondly, our work fixes concurrency bugs based on inferred locking policy which allows us to systematically fix bugs throughout the program as well as fixing bugs which involve multiple variables (which are not considered by existing tools). Thirdly, PFɪx is more comprehensive. For instance, AFix and AlphaFixer focus on fixing atomicity violations, whereas Grail cannot fix multi-variable bugs. PFɪx is able to fix order violations, atomic violations, data races, which involve multiple variables.

Memory-access Patterns Analysis. The idea of analyzing memory-access patterns to understand/detect bugs has been explored in multiple settings. In [37], Lu *et al.* presented a tool named AVIO and an empirical study on root cause of 74 real-world concurrency bugs. AVIO focuses on detecting bugs caused by single-variable atomicity violations, i.e., one particular memory-access pattern. Falcon [43] takes multiple test executions as input and computes statistical measurement for memory-access patterns related to atomicity and order violations. It then ranks the patterns according to the measure, i.e., the suspiciousness score. Griffin [40] groups multiple patterns which are found to be related to a bug. It also provides bug graphs to help understand the root cause of the bug. Xu et al. [54] presented an algorithm to identify erroneous event patterns from a failed execution. It first finds the erroneous switch points and then determines the patterns related to this erroneous switch point, which can help users localize the bugs. Unicorn [42] is the first to unify the use of pattern detection and sequencing to locate non-deadlock concurrency bugs. It dynamically collects memory access information and is extended to detect both single-variable and multi-variable concurrency bugs based on Falcon's single-variable concurrency bugs detection. Our approach takes fault related information in the form of memory access patterns as input and generates fixes accordingly.

Fault Detection and Localization. Our work is related to the line of work on fault detection and localization. Extensive research has been conducted on localizing bugs with different strategies. Among them, quite a number are designed for sequential programs [17, 18, 27, 30, 31, 39, 44, 47, 51, 52]. These methods collect and analyze runtime information of statements or predicates. They report the suspicious statements or predicates as final results. There

have been a number of proposals on fault localization for multi-threaded bugs in recent years [7, 23, 36, 38]. CSight [7] generates a communicating finite state machine (CFSM) model by mining program execution logs. Lazy-CSeq [23] works with context-bounded model checking and supports deadlock detection for concurrent C programs. Recon [38] provides information related to bug root causes by showing the scheduling of a test execution. Recon acquires short fragments of inter-thread communications near the bug root causes, and then applies machine learning techniques to identify the bug-related fragments. RaceMob [28] combines static and dynamic bug detection. During the static phase, it uses a static data race detector to find potential data races. Then RaceMob dynamically validates the suspicious data races and updates a list of data races to developers. Similarly, IteRace [46] is also presented for race detection. However, IteRace conducts static race detection in Java parallel loops. There are several approaches [14, 23, 41, 56] try to expose concurrent bugs by inserting random disturbances when concurrent programs are accessing shared memory and synchronizing, or controling the thread scheduling. These methods aim to increase the probability of triggering the rare interleaving executions with the assumption that bugs may be hidden in those executions. However, inserting random delay disturbance may cause high performance overhead. Several techniques, such as model checking [16, 19] use Java Pathfinder to find concurrency bugs. In this work, we apply Java Pathfinder to precisely control thread scheduling. It leverages controlled executions to check whether certain patterns are relevant.

6 CONCLUSION

In this work, we propose an approach to automatically fix concurrency bugs. Our key idea is to systematically fix concurrency bugs by inferring locking policies from failure inducing memory-access patterns. We fix the program by implementing the locking policy systematically and consistently so that the failure inducing memory-access patterns are made impossible. We have implemented our approach in a toolkit called PFɪx which targets Java programs. We applied PFɪx to a set of 23 concurrency bugs (which range from dozens of LOC to thousands of LOC) and are able to automatically fix 19 of them. For future work, with the encouraging results in this work, we aim to re-implement PFɪx so that it can be applied for a variety of real-world programs.

REFERENCES

[1] 2011. The Pecan Benchmarks. http://www.cse.ust.hk/prism/pecan/#Experiment
[2] 2016. JaConTeBe Object Biography. http://sir.unl.edu/portal/bios/JaConTeBe.php
[3] 2018. Abstract Syntax Tree. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html
[4] 2018. The Fix page. https://github.com/PFixConcurrency/Fix
[5] 2018. The FixExamples page. https://github.com/PFixConcurrency/FixExamples
[6] 2018. The Soot GitHub Project. https://github.com/Sable/soot

[7] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 468–479.

[8] Jeremy S Bradbury and Kevin Jalbert. 2010. Automatic repair of concurrency bugs. In *International symposium on search based software engineeringâĂŤfast abstracts*. 1–2.

[9] Yan Cai and Lingwei Cao. 2016. Fixing deadlocks via lock pre-acquisitions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 1109–1120.

[10] Yan Cai, Lingwei Cao, and Jing Zhao. 2017. Adaptively generating high quality fixes for atomicity violations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 303–314.

[11] Y. Cai, L. Cao, and J Zhao. 2017. Adaptively generating high quality fixes for atomicity violations. *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), 303–314.

[12] Yan Cai and W. K Chan. 2012. MagicFuzzer: scalable deadlock detection for large-scale applications. In *International Conference on Software Engineering*. 606–616.

[13] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. A deployable sampling strategy for data race detection. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 810–821. https://doi.org/10.1145/2950290.2950310

[14] Lee Chew and David Lie. 2010. Kivati: fast detection and prevention of atomicity violations. In *European Conference on Computer Systems, Proceedings of the European Conference on Computer Systems, EUROSYS 2010, Paris, France, April*. 307–320.

[15] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.

[16] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *ACM Sigplan-Sigact Symposium on Principles of Programming Languages*. 174–186.

[17] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. 2010. Test Input Reduction for Result Inspection to Facilitate Fault Localization. *Automated Software Engineering Journal* 17, 1 (March 2010), 5–31.

[18] Dan Hao, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. 2009. Interactive Fault Localization Using Test Information. *Journal of Computer Science and Technology* 24, 5 (2009), 962–974.

[19] Klaus Havelund and Thomas Pressburger. 2000. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381.

[20] Jeff Huang and Zhang C. 2011. Persuasive prediction of concurrency access anomalies[C]. *International Symposium on Software Testing and Analysis* (2011), 144–154.

[21] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 337–348.

[22] Jeff Huang and Charles Zhang. 2012. Execution privatization for scheduler-oblivious concurrent programs. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 737–752.

[23] Omar Inverso, Truc L Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. *Automated Software Engineering* (2015), 807–812.

[24] Antsaklis P J Iordache M V. 2006. Supervision Based on Place Invariants: A Survey. *Discrete Event Dynamic Systems* (2006), 451–492.

[25] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. *Acm Sigplan Notices* 46 (2011), 389–400.

[26] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated concurrency-bug fixing. In *Usenix Conference on Operating Systems Design and Implementation*. 221–236.

[27] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Ieee/acm International Conference on Automated Software Engineering*. 273–282.

[28] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced data race detection. In *Twenty-Fourth ACM Symposium on Operating Systems Principles*. 406–422.

[29] Sepideh Khoshnood, Markus Kusano, and Chao Wang. 2015. ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 165–176.

[30] Xiangyu Li, Marcelo DâĂŽAmorim, and Alessandro Orso. 2016. Iterative User-Driven Fault Localization. In *Hardware and Software: Verification and Testing: 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*.

[31] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. In *ACM Sigplan Conference on Programming Language Design and Implementation*. 15–26.

[32] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and Generating High Quality Patches for Concurrency Bugs. In *The International Symposium on the Foundations of Software Engineering*.

[33] P. Liu, J. Dolby, and C. Zhang. 2013. Finding incorrect compositions of atomicity. *In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), 158–168.

[34] Peng Liu, Omer Tripp, and Charles Zhang. 2014. Grail: context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 318–329.

[35] Peng Liu and Charles Zhang. 2012. Axis: Automatically fixing atomicity violations through solving control constraints. In *Proceedings of the 34th international conference on software engineering*. IEEE Press, 299–309.

[36] Shuang Liu, Guangdong Bai, Jun Sun, and Jin Song Dong. 2016. Towards Using Concurrent Java API Correctly. In *Engineering of Complex Computer Systems (ICECCS), 2016 21st International Conference on*. IEEE, 219–222.

[37] S. Lu, Soyeon Park, and Yuanyuan Zhou. 2011. Detecting Concurrency Bugs from the Perspectives of Synchronization Intentions. *IEEE Transactions on Parallel & Distributed Systems* 23 (2011), 1060–1072.

[38] Brandon Lucia, Benjamin P. Wood, and Luis Ceze. 2011. Isolating and understanding concurrency errors using reconstructed execution fragments. *Acm Sigplan Notices* 46 (2011), 378–388.

[39] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. 2012. AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, Washington, DC, USA, 31–40.

[40] Sangmin Park, Mary Jean Harrold, and Richard Vuduc. 2013. Griffin: grouping suspicious memory-access patterns to improve understanding of concurrency bugs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 134–144.

[41] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: exposing atomicity violation bugs from their hiding places. *Acm Sigplan Notices* 44 (2009), 25–36.

[42] Sangmin Park, Richard Vuduc, and Mary Jean Harrold. 2012. A unified approach for localizing non-deadlock concurrency bugs. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 51–60.

[43] Sangmin Park, Richard W Vuduc, and Mary Jean Harrold. 2010. Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 245–254.

[44] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[45] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. 2005. *Java Concurrency in Practice*. Addison-Wesley Professional.

[46] Cosmin Radoi and Danny Dig. 2015. Effective Techniques for Static Race Detection in Java Parallel Loops. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (September 2015), 24:1–24:30.

[47] Abreu Rui, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation, 2007. Taicpart-Mutation*. 89–98.

[48] K. Sen. 2008. Race directed random testing of concurrent programs. *ACM Sigplan Notices* 43, 6 (2008), 11–21.

[49] Murata T. 1989. Petri nets: Properties, analysis and applications. *Proc IEEE* (1989), 541–580.

[50] Mandana Vaziri, Frank Tip, and Julian Dolby. 2006. Associating Synchronization Constraints with Data in an Object-oriented Language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 334–345.

[51] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 1–11.

[52] Shaowei Wang, David Lo, Lingxiao Jiang, Lucia, and Hoong Chuin Lau. 2011. Search-based Fault Localization. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 556–559.

[53] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S Mahlke. 2009. The theory of deadlock avoidance via discrete control. *In ACM SIGPLAN Notices* 44, 1 (2009), 252–263.

[54] Jing Xu, Yu Lei, Richard Carver, and David Kung. 2013. *Dynamic Approach to Isolating Erroneous Event Patterns in Concurrent Program Executions*. Springer Berlin Heidelberg, 97–109.

[55] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 26–36.

[56] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A coverage-driven testing tool for multithreaded programs. *Acm Sigplan Notices*

47 (2012), 485–502.

[57] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: detecting concurrency bugs through

sequential errors. *Architectural Support for Programming Languages and Operating Systems* 46, 3 (2011), 251–264.