Automated Concurrency-Bug Fixing

Guoliang Jin Wei Zhang Dongdong Deng Ben Liblit Shan Lu
University of Wisconsin–Madison
{aliang,wzh,dongdong,liblit,shanlu}@cs.wisc.edu

Abstract

Concurrency bugs are widespread in multithreaded programs. Fixing them is time-consuming and error-prone. We present CFix, a system that automates the repair of concurrency bugs. CFix works with a wide variety of concurrency-bug detectors. For each failure-inducing interleaving reported by a bug detector, CFix first determines a combination of mutual-exclusion and order relationships that, once enforced, can prevent the buggy interleaving. CFix then uses static analysis and testing to determine where to insert what synchronization operations to force the desired mutual-exclusion and order relationships, with a best effort to avoid deadlocks and excessive performance losses. CFix also simplifies its own patches by merging fixes for related bugs.

Evaluation using four different types of bug detectors and thirteen real-world concurrency-bug cases shows that CFix can successfully patch these cases without causing deadlocks or excessive performance degradation. Patches automatically generated by CFix are of similar quality to those manually written by developers.

1 Introduction

1.1 Motivation

Concurrency bugs in multithreaded programs have already caused real-world disasters [27, 46] and are a growing threat to software reliability in the multi-core era. Tools to detect data races [12, 50, 68], atomicity violations [7, 13, 30, 31], order violations [16, 33, 66, 69], and abnormal inter-thread data dependencies [53, 70] have been proposed. However, finding bugs is just a start. Software reliability does not improve until bugs are actually fixed.

Bug fixing is time-consuming [38] and error-prone [54]. Concurrency bugs in particular bring unique challenges, such as understanding synchronization problems, selecting and using the right synchronization primitives in the right way, and maintaining performance and readability while adding synchronization into multi-threaded software. A previous study of open-source software [32] finds that it takes 73 days on average to correctly fix a concurrency bug. A study of operating-system patches [65] shows that among common bug types, concurrency bugs are the most difficult to fix correctly. 39% of patches to concurrency bugs in released operating system code are incorrect, a ratio 4 – 6 times higher than that of memory-bug patches.

Fortunately, concurrency bugs may be more amenable to automated repair than sequential bugs. Most concurrency bugs only cause software to fail rarely and nondeterministically. The correct behavior is already present as some safe subset of all possible executions. Thus, such bugs can be fixed by systematically adding synchronization into software and disabling failure-inducing interleavings.

Prior work on AFix demonstrates that this strategy is feasible [20]. AFix uses static analysis and code transformation to insert locks and fix atomicity violations detected by CTrigger [43]. Although promising, AFix only looks at one type of synchronization primitive (mutex locks) and can fix only one type of concurrency bug (atomicity violations) reported by one specific bug detector (CTrigger). In addition, AFix cannot fix a bug when CTrigger reports bug side effects instead of bug root causes.

1.2 Contributions

CFix aims to automate the entire process of fixing a wide variety of concurrency bugs, without introducing new functionality problems, degrading performance excessively, or making patches needlessly complex. Guided by these goals, CFix system automates a developer's typical bug fixing process in five steps, shown in Figure 1.

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

The second step is *fix-strategy design* (Section 2). CFix designs a set of fix strategies for each type of bug report. Each fix strategy includes mutual-exclusion/order relationships[1] that, once enforced, can disable the failure-inducing interleaving. By decomposing every bug report into mutual-exclusion and order problems, CFix addresses the diversity challenge of concurrency bugs and bug detectors. To extend CFix for a new type of bugs, one only

[1] A mutual-exclusion relationship requires one code region to be mutually exclusive with another code region. An order relationship requires that some operation always execute before some other operation.

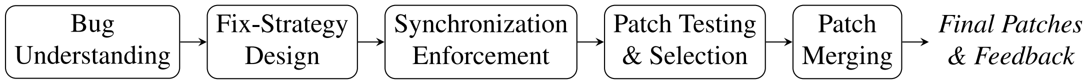


Figure 1: CFix bug fixing process

needs to design new fix strategies and simply reuses other CFix components.

The third step is *synchronization enforcement* (Section 3). Based on the fix strategies provided above, CFix uses static analysis to decide where and how to synchronize program actions using locks and condition variables, and then generates patches using static code transformation. Specifically, CFix uses the existing AFix tool to enforce mutual exclusion, and a new tool OFix to enforce order relationships. To our knowledge, OFix is the first tool that enforces basic order relationships to fix bugs with correctness, performance, and patch simplicity issues all considered. This lets CFix use more synchronization primitives and fix more types of bugs than previous work.

The fourth step is *patch testing and selection* (Section 4). CFix tests patches generated using different fix strategies, and selects the best one considering correctness, performance, and patch simplicity. In this step, CFix addresses the challenge of multi-threaded software testing by leveraging the testing framework of bug detectors and taking advantage of multiple patch candidates, as the testing result of one patch can sometimes imply problems of another. This step also addresses the challenge of bug detectors reporting inaccurate root causes: patches fixing the real root cause are recognizable during testing as having the best correctness and performance.

The fifth step is *patch merging* (Section 5). CFix analyzes and merges related patches. We propose a new merging algorithm for order synchronization operations (i.e., condition-variable signal/wait), and use AFix to merge mutual-exclusion synchronizations (i.e., locks). This step reduces the number of synchronization variables and operations, significantly improving patch simplicity.

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

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

Overall, this paper makes two major contributions:

Firstly, we design and implement OFix, a tool that enforces two common types of order relationship between two operations identified by call stacks tailored for fixing concurrency bugs. Specifically, OFix focuses on two basic order relationships: either (1) an operation B cannot execute until *all* instances of operation A have executed; or (2) an operation B cannot execute until at least *one* instance of operation A has executed, if operation A executes in this run at all. We refer to these respectively as all*A*–*B* and first*A*–*B* relationships. See Sections 3 and 5 for details.

Secondly, we design and implement CFix, a system that assembles a set of bug detecting, synchronization enforcing, and testing techniques to automate the process of concurrency-bug fixing. Our evaluation shows that CFix is effective for a wide variety of concurrency bugs.

2 Fix Strategy Design

The focus of CFix is bug fixing; we explicitly do not propose new bug-detection algorithms. Rather, CFix relies on any of several existing detectors to guide bug fixing. We refer to these detectors as CFix's *front end*. We require that the front end provide information about the failure-inducing interleaving (i.e., a specific execution order among bug-related instructions). We do *not* require that the bug detector accurately report bug root causes, which we will demonstrate using real-world examples in Section 6.

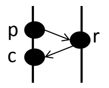
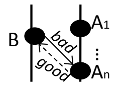
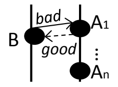
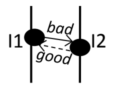
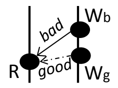
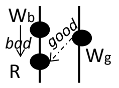
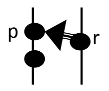
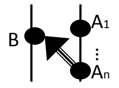
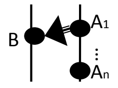
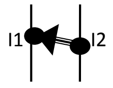
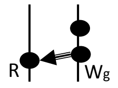
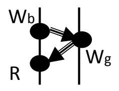
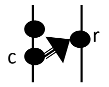
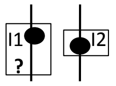
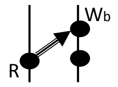
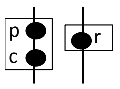
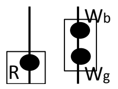
2.1 Mutual Exclusion and Ordering

To effectively handle different types of concurrency bugs and bug detectors, CFix decomposes every bug into a combination of mutual-exclusion and order problems. The rationale is that most synchronization primitives either enforce mutual exclusion, such as locks and transactional memories [17, 18, 48], or enforce strict order between two operations, such as condition-variable signals and waits. Lu et al. [32] have shown that atomicity violations and order violations contribute to the root causes of 97% of real-world non-deadlock concurrency bugs.

In this paper, a *mutual-exclusion relationship* refers to the basic relationship as being enforced by AFix [20] among three instructions p, c, and r. Once mutual exclusion is enforced, code between p and c forms a critical section which prevents r from executing at the same time.

An *order relationship* requires that an operation A always execute before another operation B. Note that A and B may each have multiple dynamic instances at run

Table 1: Bug reports and fix strategies. Rectangles denote mutual exclusion regions, wide arrows denote enforced order, and circles illustrate instructions. Vertical lines represent threads 1 and 2. A_n is the nth dynamic instance of A.

	(a) Atomicity Violation	(b) Order Violation all*A–B*	(b) Order Violation first*A–B*	(c) Race	(d) Def-Use Remote-is-Bad	(d) Def-Use Local-is-Bad
Reports:						
Strategy (1):						
Strategy (2):		N/A	N/A			N/A
Strategy (3):		N/A	N/A	N/A		N/A

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

Figure 2: Concurrency bug simplified from FFT. Making Thread 1 mutually exclusive with Thread 2 cannot fix the bug, because r can still execute after p and c.

```
// Thread 1                       // Thread 2
while (...) {                     free(buffer); // B
  tmp = buffer[i]; // A
}
```

Figure 3: Order violation simplified from PBZIP2

time; the desired ordering among these instances could vary in different scenarios. We focus on two basic order relationships: all*A–B* and first*A–B*. When bug fixing needs to enforce an order relationship, unless specifically demanded by the bug report, we try all*A–B* first and move to the less restrictive first*A–B* later if all*A–B* causes deadlocks or timeouts.

2.2 Strategies for Atomicity Violations

An atomicity violation occurs when a code region in one thread is unserializably interleaved by accesses from another thread. Many atomicity-violation detectors have been designed [13, 15, 30, 31, 35, 43, 57, 64]. CFix uses CTrigger [43] as an atomicity-violation-detecting front end. Each CTrigger bug report is a triple of instructions $(\mathsf{p}, \mathsf{c}, \mathsf{r})$ such that software fails almost deterministically when r is executed between p and c, as shown in Table 1(a).

Jin et al. [20] patch each CTrigger bug by making code region p–c mutually exclusive with r. However, this patch may not completely fix a bug: CTrigger may have reported a side effect of a concurrency bug rather than its root cause. Figure 2 shows an example.

Instead of relying on CTrigger to point out the root cause, which is challenging, CFix explores all possible ways to disable the failure-inducing interleaving, as shown in Table 1(a): (1) enforce an order relationship, making r always execute before p; (2) enforce an order relationship, making r always execute after c; or (3) enforce mutual exclusion between p–c and r.

2.3 Strategies for Order Violations

An order violation occurs when one operation A should execute before another operation B, but this is not enforced by the program. Order violations contribute to about one third of real-world non-deadlock concurrency bugs [32]. Figures 2 and 3 show two examples simplified from real-world order violations.

Many existing tools detect order violation problems and could work with CFix. This paper uses ConMem [69] as a representative order-violation detector. ConMem discovers buggy interleavings that lead to two types of common order violations: dangling resources and uninitialized reads. For dangling resources, ConMem identifies a resource-use operation A and a resource-destroy operation B, such as those in Figure 3. The original software fails to enforce that all uses of a resource precede all destructions of the same resource. For uninitialized reads, ConMem finds a read operation B and a write operation A. The original software fails to enforce that at least one instance of A occur before B, leading to uninitialized reads as in Figure 2.

For each of these two types of bugs, CFix has one corresponding fix strategy. As shown in Table 1(b), we enforce an all*A*–*B* order relationship to fix a dangling resource problem, and we enforce a first*A*–*B* order relationship to fix an uninitialized-read problem.

2.4 Strategies for Data Races

Race detectors [8, 12, 14, 36, 41, 47, 50, 68] report unsynchronized instructions, including at least one write, that can concurrently access the same variable from different threads. Race-guided testing [22, 39, 51] can identify a race-instruction pair $(\mathsf{I1}, \mathsf{I2})$ such that the software fails when I2 executes immediately after I1. For CFix we implement a widely used lock-set/happens-before hybrid race-detection algorithm [2, 52] coupled with a RaceFuzzer-style testing framework [51]. This front end can identify failure-inducing interleavings as shown in Table 1(c).

Table 1(c) also illustrates two possible strategies for fixing a typical data-race bug: (1) force an order relationship, making I2 always execute before I1; or (2) force a mutual-exclusion relationship between I2 and a code region that starts from I1 and ends at a to-be-decided instruction. We use the second strategy only when the front end also reports a data race between I2 and a third instruction I3, I3 comes from the same thread as I1, and software fails when I2 executes right before I3. If all of these constraints hold, we consider both the first strategy and the second strategy that makes I1–I3 mutually exclusive with I2. In all other cases, we only consider the first strategy.

2.5 Strategies for Abnormal Def-Use

Some bug detectors identify abnormal inter-thread data-dependence or data-communication patterns that are either rarely observed [33, 53, 66] or able to cause particular types of software failures, such as assertion failures [70]. CFix uses such a tool, ConSeq [70], as a bug-detection front end. Each ConSeq bug report includes two pieces of information: (1) the software fails almost deterministically when a read instruction R uses values defined by a write Wb; and (2) the software is observed to succeed when R uses values defined by another write Wg. Note that ConSeq does not guarantee Wg to be the *only* correct definition of R. Therefore, CFix only uses Wg as a hint.

We refer to the case when R and Wb come from different threads as Remote-is-Bad. Depending on where Wg comes from, there are different ways to fix the bug by enforcing either mutual exclusion or ordering. Table 1(d) shows one situation that is common in practice. We refer to the case when R and Wb come from the same thread as Local-is-Bad. Enforcing orderings is the only strategy to fix this case, as shown in Table 1(d).

2.6 Discussion

CFix does not aim to work with deadlock detectors now, because deadlocks have very different properties from other concurrency bugs. Furthermore, there is already a widely accepted way to handle deadlocks in practice: monitor the lock-wait time and restart a thread/process when necessary.

The goal of this work is not to compare different types of bug detectors. In practice, no one bug detector is absolutely better than all others. Different detectors have different strengths and weaknesses in terms of false negatives, false positives, and performance. We leave it to software developers and tool designers to pick bug detectors. CFix simply aims to support all representative types of detectors. CFix can also work with other bug detectors, as long as failure-inducing interleavings and the identity of the involved instructions are provided. We leave extending CFix to more detectors to future work.

3 Enforcing an Order Relationship

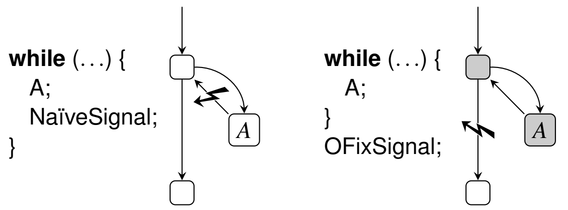
CFix uses AFix [20] to enforce mutual exclusion during patch generation. This section presents OFix, the static analysis and patch-generation component of CFix that enforces orderings, specifically all*A*–*B* and first*A*–*B* orderings. It enforces the desired ordering while making a strong effort to avoid deadlock, excessive performance loss, or needless complexity.

OFix expects bug detectors to describe a run-time operation using two vectors of information: (1) a call stack in the thread that executes that instruction, and (2) a chain of call stacks indicating how that thread has been created, which we call a *thread stack*. We refer to the instruction that performs *A* as an *A instruction*. The thread that executes the *A* instruction is a *signal thread*. All ancestors of the signal thread are called *s-create threads*. Conversely for *B* we have the *B instruction* executed by a *wait thread* whose ancestors are *w-create threads*. We write a *call stack* as $(f_0, i_0) \rightarrow (f_1, i_1) \rightarrow \cdots \rightarrow (f_n, i_n)$. f_0 is the starting function of a thread, which could be main or any function passed to a thread creation function. Each i_k before the last is an instruction in f_k that calls function f_{k+1}. In a signal or wait thread, the last instruction i_n is the *A* instruction or *B* instruction respectively. In s-create or w-create threads, the last instruction i_n calls a thread-creation function.

In Section 7 we discuss limitations of OFix and the impact on CFix as a whole system, especially those caused by the decision to try only all*A*–*B* and first*A*–*B* orderings and the choice to use call stack as operation identity.

3.1 Enforcing an all*A*–*B* Order

It is difficult to statically determine how many instances of *A* will be executed by a program, and hence it is difficult to find the right place to signal the end of *A* and make *B* wait



(a) May signal any number of times, waking the wait thread too early or never

(b) OFix signals exactly once; gray nodes are reaching nodes

Figure 4: Naïve signals versus OFix signals for all*A*–*B*. "↯" marks signal operations on edges.

for all instances of A. Consider that A could be executed an unknown number of times in each signal thread; signal threads could be created an unknown number of times by each s-create thread; and s-create threads could also be created an unknown number of times. We address these challenges in four steps: (1) locate places to insert signal operations in signal threads, (2) locate places to insert signal operations in s-create threads, (3) locate places to insert wait operations, and (4) implement the signal and wait operations to coordinate all relevant threads.

Finding Signal Locations in Signal Threads A naïve solution that inserts signal operations right after the A instruction could lead to many problems, as shown in Figure 4a. OFix aims to place signal operations so that each signal thread executes exactly one signal operation as soon as it cannot possibly execute more A.

Assume that A has call stack $(f_0, i_0) \rightarrow \cdots \rightarrow (f_n, i_n)$ where i_n is the A instruction. OFix analysis starts from f_0. When the program can no longer execute i_0 in f_0, we know that the problematic A call stack can no longer arise, and it is safe to signal. Thus OFix first analyzes the control flow graph (CFG) of f_0 to obtain the set of *reaching nodes*: those CFG nodes that can reach i_0 in zero or more steps. OFix then inserts a signal operation on each CFG edge that crosses from a reaching node to a non-reaching node.

After placing signals in f_0, we may need to continue down the stack to f_1, f_2, and so on. The critical question here is whether i_0 can call f_1 multiple times, i.e., whether i_0 is in a loop. If so, then f_1 cannot determine when A will be executed for the last time in the thread. Rather, that decision needs to be made in f_0, outside of the outermost loop that contains i_0. The signal-placement algorithm stops, without continuing down the stack.

Conversely, if f_1 can be invoked at most once at i_0, we suppress the signal operation that would ordinarily be placed on the edge from i_0 to its successor. Instead, we delegate the signaling operation down into f_1 and repeat the signal-placement analysis in f_1. This process continues, pushing signals deeper down the stack. Signal placement stops when it reaches the end of the call stack or when it finds a call inside a loop.

Note that a function f_k could be invoked under many different call stacks, while we only want f_k to execute a signal operation under a particular call stack. We solve this problem using a function-cloning technique discussed in Section 3.3. All OFix-related transformations are actually applied to cloned functions.

The above strategy has two important properties. We omit proofs due to space constraints. First, each terminating execution of a signal thread signals exactly once, as shown in Figure 4b, because thread execution crosses from reaching nodes to non-reaching nodes exactly once. Second, the signal is performed as early as possible within each function, according to the CFG, and interprocedurally, benefiting from our strategy of pushing signals down the stack. These properties help ensure the correctness of our patches. They also help our patches wake up waiting threads earlier, limiting performance loss and reducing the risk of deadlocks.

Finding Signal Locations in s-create Threads Signal operations also need to be inserted into every s-create thread that could spawn more signal threads. Otherwise, we still would not know when we had safely passed beyond the last of all possible A instances. The procedure for s-create threads matches that already used for signal threads. We apply the signal-placement analysis and transformation to each s-create thread in the thread-creation ancestry sequence that eventually leads to creation of the signal thread. This algorithm ensures that each of these ancestral threads signals exactly once immediately when it can no longer create new s-create threads or signal threads.

Finding Wait Locations in Wait Threads OFix must insert wait operations that can block the execution of B. Assuming that (f_n, i_n) is the last level on the call stack of B, OFix creates a clone of f_n that is only invoked under the bug-related call stack and thread stack. We then insert the wait operation immediately before i_n in the clone of f_n.

Implementing Wait and Signal Operations Our implementation of the signal and wait operations has three parts:

Part 1: track the number of threads that will perform signal operations. OFix creates a global counter C in each patch. C is initialized to 1, representing the main thread, and atomically increases by 1 immediately before the creation of every signal thread and s-create thread.

Part 2: track how many threads have already signaled. Each signal operation atomically decrements C by 1. Since each signal or s-create thread executes exactly one signal operation, each such thread decreases C exactly once. Figure 5a shows pseudo-code for the signal operation.

Part 3: allow a wait thread to proceed once all threads that are going to signal have signaled. This is achieved by

```
mutex_lock(L);
if (--C == 0)
  cond_broadcast(con);
mutex_unlock(L);
```

(a) Signal operation

```
mutex_lock(L);
if (C > 0)
  cond_timedwait(con, L, t);
mutex_unlock(L);
```

(b) Wait operation

Figure 5: Pseudo-code for all*A*–*B* operations

condition-variable broadcast/wait and the checking of C, as shown in Figure 5b.

These signal and wait operations operate on three variables: one counter, one mutex, and one condition variable. Each of these is statically associated with the order relationship it enforces.

OFix's synchronization operations are not merely semaphore operations. Since we cannot know in advance how many signal operations each wait operation should wait for, OFix relies on a well-managed counter and a condition variable to achieve the desired synchronization.

Correctness Assessment OFix makes a best effort to avoid introducing deadlocks by signaling as soon as possible and starting to wait as late as possible. However, it is impossible to statically guarantee that a synchronization enforcement is deadlock free. To mask potential deadlocks introduced by the patch, OFix allows every wait operation to timeout. In fact, deadlocks mostly occur when the bug requires a different fix strategy entirely, making them an important hint to guide CFix patch selection (Section 4).

Barring timeouts, the wait operation guarantees that no *B* executes before C reaches 0. The signal operations guarantee that C reaches 0 only when all existing signal and s-create threads have signaled, which implies that no more signal or s-create threads can be created, and therefore no more *A* instances can be executed. Thus, if no wait times out, OFix patch guarantees that no instance of *B* executes before any instance of *A*.

Simplicity Optimization We use the static number of synchronization operations added by a patch as the metric for patch simplicity. In the current implementation, OFix's patches operate on LLVM bitcode [25], but we envision eventually using similar techniques to generate source-code patches. The simplicity of OFix bitcode patches would be a major factor affecting the readability of equivalent source-code patches. OFix's simplicity optimization attempts to reduce the static number of synchronization operations being added.

In the algorithm described above, OFix signal operations are inserted based solely on the calling context of *A*. In fact, a signal operation *s* is unnecessary if it never executes in the same run as *B*, such as in Figure 6. Since identifying all such *s* is too complicated for multi-threaded software, OFix focuses on two cases that we find common and especially useful in practice.

```
void main() {            void foo() { // f_1
  if (...)                 pthread_create(Bthread, ...); // iB_1
    foo(); // i_0          pthread_create(Athread, ...); // iA_1
  else                   }
    ~~OFixSignal~~; // s
}
```

(a) Optimization case 1

```
void main() {
  if (...) {
    ~~OFixSignal~~; // s
    exit(1);
  }
  pthread_create(Bthread, ...); // iB_0
  pthread_create(Athread, ...); // iA_0
}
```

(b) Optimization case 2

Figure 6: Removing unnecessary OFix signal operations

To ease our discussion, we use $C = (\mathsf{main}, i_0) \rightarrow \cdots \rightarrow (f_k, i_k)$ to denote the longest common prefix of the calling contexts of *A* and *B* where none of the call sites $i_0, \ldots, i_k$ is inside a loop. When i_k is a statically-bound call, the next function along the contexts of *A* and *B* is the same, denoted as f_l. The next level of *B*'s context is denoted as (f_l, iB_l).

Case 1: OFix signal operations in *C* can all be removed, because they never execute in the same run as *B*. To prove this, assume *s* to be a signal operation inserted in f_k. The rationale for optimizing *s* away is as follows. First, no instance of *B* will be executed any more once *s* is executed: program execution cannot reach *C* any more once it executes *s*, according to how OFix inserts signal operations. Second, no instance of i_k, and hence *B*, has executed yet when *s* is executed. If this were not true, then the signal thread would signal multiple times (once inside the callee of i_k and once at *s*). But this is impossible in OFix patches. Therefore, when *s* is executed, *B* cannot have executed yet and will not execute any more. Removing *s* does not affect the correctness of our patch, as shown in Figure 6a.

Case 2: An OFix signal operation *s* in f_l can be removed if *s* cannot reach iB_l and iB_l cannot reach *s*. The rationale of this optimization is similar to that of Case 1. Figure 6b shows an example of applying this optimization.

3.2 Enforcing a first*A*–*B* Order

Basic Design To guarantee that *B* waits for the first instance of *A*, we insert a signal operation immediately after the *A* instruction, and a wait operation immediately before the *B* instruction. Figure 7 shows the code for first*A*–*B* synchronization operations. A Boolean flag and a condi-

```
if (!alreadyBroadcast) {
  alreadyBroadcast = true;
  mutex_lock(L);
  cond_broadcast(con);
  mutex_unlock(L);
}
```

(a) Signal operation

```
mutex_lock(L);
if (!alreadyBroadcast)
  cond_timedwait(con, L, t);
mutex_unlock(L);
```

(b) Wait operation

Figure 7: Pseudo-code for first*A*–*B* operations. It contains a benign race on alreadyBroadcast.

tion variable work together to block the wait thread until at least one signal operation has executed.

Safety-Net Design The basic design works if the program guarantees to execute at least one instance of A. However, this may not be assured, in which case forcing B to wait for A could hang the wait thread. To address this problem, OFix enhances the basic patch with a safety net: when the program can no longer execute A, safety-net signals release the wait thread to continue running.

OFix first checks whether A is guaranteed to execute: specifically, whether i_k post-dominates the entry block of f_k in each level (f_k, i_k) of the A call stack. A safety net is needed only when this is not true.

When a safety net is needed, OFix inserts safety-net signal operations using the all*A*–*B* algorithm of Section 3.1. That algorithm maintains a counter C as in Figure 5a and guarantees that C drops to 0 only when the program can no longer execute A. To allow safety-net signal operations to wake up the wait thread, these operations share the same mutex L and the same condition variable con as those used in the basic patch in Figure 7. Whichever thread decrements C to 0 executes pthread_cond_broadcast to unblock any thread that is blocked at con. That thread also sets alreadyBroadcast to **true** so that any future instances of B will proceed without waiting.

OFix checks whether B is post-dominated by any safety-net signal operation. In that case, the safety net can never help wake up B and is removed entirely. Lastly, OFix applies the two simplicity-optimization algorithms presented in Section 3.1 to remove unnecessary safety-net signals.

Even with the safety net, OFix does not guarantee deadlock freedom. As for all*A*–*B* patches, OFix uses a timeout in the first*A*–*B* wait operation to mask potential deadlocks.

3.3 Function Cloning

OFix clones functions to ensure that each OFix patch only takes effect under the proper call stack and thread-creation context. All patch-related transformations are applied to cloned functions.

Consider a failure-related call chain $(main, i_0) \rightarrow (f_1, i_1) \rightarrow \cdots \rightarrow (f_n, i_n)$, which chains together the call stacks of all s-create and signal threads (or all w-create and wait threads) through thread-creation function calls. Function cloning starts from the first function f_k on this call chain that can be invoked under more than one calling context. OFix creates a clone f'_k for f_k and modifies i_{k-1} based on the kind of invocation instruction at i_{k-1}. If i_{k-1} is a direct function call to f_k, OFix simply changes the target of that call instruction to the clone f'_k. If i_{k-1} calls a thread-creation function with f_k as the thread-start routine, OFix changes that parameter to f'_k. If f_k is invoked through a function pointer, OFix adds code to check the actual value of that function pointer at run time. When the pointer refers to f_k, our modified code substitutes f'_k instead. OFix proceeds down the stack in this manner through each level of the call chain to finish the cloning. It is straightforward to prove that the above cloning technique inductively guarantees that OFix always patches under the right context.

OFix repeatedly uses the above cloning technique in various parts of the patching process, with one f-clone created for each unique bug-related calling context of f.

3.4 Discussion

In the algorithms given above, we always consider the calling context of a bug report. We believe this is necessary, especially for bug-detection front-ends that are based on dynamic analysis. OFix can also enforce the ordering between two static instructions, regardless of the call stack. This option can be used when call stacks are unavailable.

OFix achieves context-awareness by cloning functions, which could potentially introduce too much duplicated code. The strategy to start cloning from the first function that can be invoked under more than one calling context ensures that all clones are necessary to achieve context-awareness. Our experience shows that OFix bug fixing in practice does not introduce excessive code duplication, affecting only a small portion of the whole program.

Our current implementation uses POSIX condition variables. We could also use other synchronization primitives, such as pthread_join. Our function cloning technique and the analysis to find signal/wait locations would still be useful: by design, placement and implementation of the synchronization operations are largely orthogonal.

4 Patch Testing and Selection

After fix strategies are selected and synchronization relations are enforced, CFix has one or more candidate patches for each bug report. CFix tests these patches as follows.

CFix first checks the correctness of a patch through static analysis and interleaving testing. Considering the huge interleaving space, correctness testing is extremely challenging. CFix first executes the patched software once without external perturbation, referred to as an *RTest*, and then applies a guided testing where the original bug-detection front end is used, referred to as a *GTest*. A patch is rejected under any of the following scenarios:

Correctness check 1: Deadlock discovered by static analysis. If an OFix wait operation is post-dominated by an OFix signal operation that operates on the same condition variable, the patch will definitely introduce deadlocks.

Correctness check 2: Failure in RTest. Since multi-threaded software rarely fails without external perturbation, this failure usually suggests an incorrect fix strategy. For example, according to the CTrigger bug report shown in Figure 2, CFix will try a patch that forces r to always execute after c, which causes deterministic failure.

Correctness check 3: Failure in GTest. This usually occurs when the patch disables the failure-inducing interleaving among some, but not all, dynamic instances of bug-related instructions.

Correctness check 4: Timeout in RTest. A patch timeout could be caused by a masked deadlock or a huge performance degradation of the patch; our timeout threshold is 10 seconds. CFix rejects the patch in both cases, and provides deadlock-detection result to developers.

Correctness check 5: Failures of related patches. Interestingly, we can use the correctness of one patch to infer that of a related patch. Consider Figure 2. If an order patch where r is forced to execute after c fails, we infer that a patch that makes r mutually exclusive with p–c is incorrect, because mutual exclusion does not prohibit the interleaving encountered by the order patch.

Rarely, CFix may not find any patch that passes correctness checking. We discuss this in Section 7.

When multiple patches pass correctness checking, CFix compares performance impacts. In our current prototype, CFix discards any patch that is at least 10% slower than some other patch. When a bug has multiple patches passing both correctness and performance checking, CFix picks the patch that introduces the fewest synchronization operations. Note that some patches can be merged and significantly improve simplicity, which we discuss in Section 5. Therefore, given two patches for the same bug report, CFix chooses the one that can be merged with other patches.

CFix includes run-time support to determine whether a timeout within CFix-patched code is caused by deadlock or not. Traditional deadlock-detection algorithms cannot discover the dependency between condition-variable wait threads and signal threads, because they cannot predict which thread will signal in the future. Inspired by previous work [20, 28], CFix addresses this challenge by starting monitoring and deadlock analysis after a timeout. By observing which thread signals on which condition variable as the post-timeout execution continues, CFix can discover circular wait relationships among threads at the moment of the timeout. This strategy imposes no overhead until a patch times out. It can be used for both patch testing and production-run monitoring. The general idea and much of the detail are the same as in the run-time for AFix [20], but we extended the run-time for AFix to support signal and wait on condition variables. Unlike Pulse [28], CFix does not require kernel modification, because it focuses only on deadlocks caused by its own patches.

CFix currently only conducts *RTest* and *GTest* using the failure-inducing inputs reported by the original bug detectors. In practice, this is usually enough to pick a good patch for the targeted bug, as demonstrated by our evaluations. In theory, this may overlook some potential problems, such as deadlock-induced timeout under other inputs and other interleavings. In such cases, we rely on our low-overhead run-time to provide feedback to refine patches.

5 Patch Merging

The goals of patch merging are to combine synchronization operations and variables, promote simplicity, and potentially improve performance. Merging is especially useful in practice, because a single synchronization mistake often leads to multiple bug reports within a few lines of code. Fixing these bugs one-by-one would pack many synchronization operations and variables into a few lines of the original program, harming simplicity and performance. Jin et al. [20] presented mutual-exclusion patch merging. In this section, we describe how OFix merges order-enforcing patches.

5.1 Patch Merging Guidelines

Patch merging in OFix is governed by four guidelines. Guideline 1: The merged patch must have statically and dynamically fewer signal and wait operations than the unmerged patches. Guideline 2: Each individual bug must still be fixed. Guideline 3: Merging must not add new deadlocks. Guideline 4: Merging should not cause significant performance loss. Note that signal operations cannot be moved earlier, per guideline 2. However, delaying signals too long can hurt performance and introduce deadlocks.

5.2 Patch Merging for all$A$–B Orderings

Figure 8 shows a real-world example of merging all$A$–B patches. To understand how the merging works, assume that we have enforced two all$A$–B orderings, A_1–B_1 and A_2–B_2, using patches P_1 and P_2.

OFix considers merging only if A_1 and A_2 share the same call stack and thread stack, except for the instruction at the last level of the call stacks, denoted as (f_n, i_n^1) and (f_n, i_n^2) for A_1 and A_2 respectively. Our rationale is to avoid moving signals too far away from their original locations, as this could dramatically affect performance (guideline 4). We do not initially consider B_1 and B_2: each patch includes just one wait operation, with little simplification potential.

Next OFix determines the locations of signal operations in the merged patch, assuming a merge will take place. To fix the original bugs (guideline 2), a signal thread must

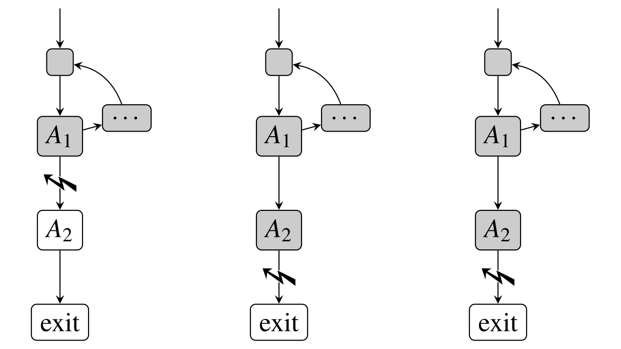
```
  while (1) {
    mutex_lock(L); // $A_1$
    if (...) {
-     $OFixSignal_1$;
      mutex_unlock(L); // $A_2$
-     $OFixSignal_2$;
+     $OFixSignal_\cup$;
      return;
    }
    ...
  }
```

(a) Signal thread

```
+ $OFixWait_\cup$;
- $OFixWait_1$;
- $OFixWait_2$;
  mutex_destroy(L); // $B_1, B_2$
```

(b) Wait thread

Figure 8: all*A*–*B* merging, simplified from PBZIP2. + and – denote additions and removals due to patch merging.



(a) *ReachSet*$_1$ and *OFixSignal*$_1$ (b) *ReachSet*$_2$ and *OFixSignal*$_2$ (c) *ReachSet*$_\cup$ and *OFixSignal*$_\cup$

Figure 9: CFG of the signal thread in Figure 8a. "↯" marks signal operations on edges.

execute a merged signal operation exactly once when it can no longer execute either A_1 or A_2. This leads to the same signal locations as those in patch P_1 and patch P_2, except for in f_n. If P_1 and P_2 do not actually place any signal operations in f_n, merging is done. Otherwise, we place the merged signal operations in f_n, so that f_n signals once it can no longer execute either i^1_n or i^2_n. Let *ReachSet*$_1$ and *ReachSet*$_2$ be the sets of nodes in f_n that can reach i^1_n and i^2_n respectively. Let *ReachSet*$_\cup$ be union of *ReachSet*$_1$ and *ReachSet*$_2$. Merged signal operations should be inserted on every edge that crosses from the inside to the outside of *ReachSet*$_\cup$. Figure 9 shows CFGs corresponding to the code in Figure 8a, with the various reaching sets highlighted in gray.

Now that we know where signal operations would be placed if the patches were merged, we reject a merged patch if it cannot reduce the signal operation count, per guideline 1. In fact, one can prove that merging improves simplicity only when *ReachSet*$_1$ overlaps with *ReachSet*$_2$.

A final check rejects merging if it delays signal operations so much that deadlocks could be introduced (guideline 3). OFix merges only when there is no blocking operation on any path from where unmerged signal operations are to where merged signal operations would be, thereby guaranteeing not to introduce new deadlocks. Our implementation considers blocking function calls (such as lock acquisitions) and loops conditioned by heap or global variables as blocking operations. To determine whether a function call may block, CFix keeps a whitelist of non-blocking function calls.

OFix merges P_1 and P_2, when all of the above checks pass. OFix removes the original signal operations in P_1 and P_2, and inserts merged signal operations into locations selected above. The simplicity optimizations described in Section 3.1 are reapplied: a merged signal is removed if neither B_1 nor B_2 would execute whenever it executes.

To merge wait operations, OFix changes the two wait operations, *OFixWait*$_1$ and *OFixWait*$_2$, in P_1 and P_2 to operate on the same synchronization variables as those in the merged signal operations. OFix also has the option to replace *OFixWait*$_1$ and *OFixWait*$_2$ with a single wait, *OFixWait*$_\cup$, located at their nearest common dominator. This option is taken only when *OFixWait*$_1$ and *OFixWait*$_2$ share the same call stack and thread stack, and when this replacement will not introduce deadlocks. For example, *OFixWait*$_1$ and *OFixWait*$_2$ in Figure 8 pass the above checks and are merged into *OFixWait*$_\cup$.

This ends the merging process for a single pair of all*A*–*B* patches. The merged patch may itself be a candidate for merging with other patches. Merging continues until no suitable merging candidates remain.

```
  if (...) {
    Gend = end; // $A_1, A_2$
-   $OFixSignal_1$; // $i^1$
-   $OFixSignal_2$; // $i^2$
+   $OFixSignal_\cup$; // $i^\cup$
  }
```

(a) Signal thread

```
+ $OFixWait_\cup$;
- $OFixWait_1$;
  printf("%d\n", Gend); // $B_1$
- $OFixWait_2$;
  printf("%d\n", Gend-init); // $B_2$
```

(b) Wait thread

Figure 10: first*A*–*B* merging, simplified from FFT. + and – denote additions and removals due to patch merging.

5.3 Patch Merging for first*A*–*B* Orderings

Figure 10 provides a real-world example of merging first*A*–*B* patches, highlighting the changes made by merging.

Given two first*A*–*B* orderings, A_1–B_1 and A_2–B_2, OFix considers merging their patches only if A_1 and A_2 share the same call stack and thread stack, except for the last instruction on the call stack. This reflects the same performance concern discussed for all*A*–*B* merging. We denote the basic signal operations used to separately enforce these two orderings as i^1 and i^2, as shown in Figure 10a.

Table 2: CFix Benchmarks. Not all benchmarks have a report ID. A report ID could be a bug report ID in their corresponding bug database, which is the case for Apache, cherokee, Mozilla, MySQL and Transmission, or a forum post ID, which is the case for HTTrack and ZSNES.

ID	App.[-ReportID]	LoC	Description
Root Cause: Order Violations			
OB1	PBZIP2	2.0 K	Several variables are used in one thread after being destroyed/nullified/freed by main.
OB2	x264	30 K	One file is read in one thread after being closed by main.
OB3	FFT	1.2 K	Several statistical variables are accessed in main before being initialized by another thread.
OB4	HTTrack-20247	55 K	One pointer is dereferenced in main before being initialized by another thread.
OB5	Mozilla-61369	192 K	One pointer is dereferenced in one thread before being initialized by main.
OB6	Transmission-1818	95 K	One variable is used in one thread before being initialized by main.
OB7	ZSNES-10918	37 K	One mutex variable is used in one thread before being initialized by main.
Root Cause: Atomicity Violations			
AB1	Apache-25520	333 K	Threads write to the same log buffer concurrently, resulting in corrupted logs or crashes.
AB2	MySQL-791	681 K	The call to create a new log file is not mutually exclusive with the checking of file status.
AB3	MySQL-3596	693 K	The checking and dereference of two pointers are not mutually exclusive with the NULL assignments.
AB4	Mozilla-142651	87 K	One memory region could be deallocated by one thread between another thread's two dereferences.
AB5	Cherokee-326	83 K	Threads write to the same time string concurrently, resulting in corrupted time strings.
AB6	Mozilla-18025	108 K	The checking and dereference of one pointer are not mutually exclusive with the NULL assignment.

To maintain the A_1–B_1 and A_2–B_2 orderings (guideline 2), we should guarantee that the merged basic signal executes when both A_1 and A_2 have executed. This location, denoted as $i^{\cup}$, is the nearest common post-dominator of i^1 and i^2 that is also dominated by i^1 and i^2. OFix abandons the merge if this location $i^{\cup}$ does not exist.

After locating $i^{\cup}$, OFix checks whether merging could cause new deadlocks, checks whether the wait operations can also be merged, inserts safety-net signal operations for a merged patch, and continues merging until no suitable merging candidates remain.

6 Experimental Evaluation

6.1 Methodology

CFix includes two static analysis and patching components: (1) AFix by Jin et al. [20]; and (2) OFix, newly presented in this paper. Both AFix and OFix are built using LLVM [25]. They apply patches by modifying the buggy program's LLVM bitcode, then compiling this to a native, patched binary executable.

We evaluate CFix on 13 real-world bug cases, representing different types of root causes, from 10 open-source C/C++ server and client applications, as shown in Table 2. We collect similar numbers of bug cases from two categories: (1) bug cases that require atomicity enforcement and (2) bug cases that require order enforcement. For the first category, we use exactly the same set of bug cases as in the AFix [20] evaluation. For the second category, we gather bug cases that have been used in previous papers on ConMem [69], ConSeq [70], and DUI [53]. These three previous papers altogether contain nine bug cases that require order enforcement; we randomly select seven out of these nine. We believe that our bug case set is representative to some extent, although we cannot claim that it represents all concurrency bugs in the real world.

To patch these buggy applications, we follow the five steps described in Section 1. We first apply the four bug-detection front-ends discussed in Section 2 to these applications using the bug-triggering inputs described in the original reports. We refer to these four detectors as the *atomicity-violation front end* (AV), the *order-violation front end* (OV), the *data-race front end* (RA), and the *definition-use front end* (DU). In each case, at least one front-end detects bugs that lead to the failures described in the original reports. CFix then generates, tests, and selects patches for each of the 90 bug reports generated by AV, OV, RA, and DU. It also tries patch merging for cases with multiple bug reports before generating the final patches.

Our experiments evaluate the correctness, performance, and simplicity of CFix's final patches. Our experiments also look at the original buggy program and the program manually fixed by software developers, referred to as *original* and *manual* respectively. For fair comparison, all binaries are generated using identical LLVM settings. All experiments use an eight-core Intel Xeon machine running Red Hat Enterprise Linux 5.

6.2 Overall Results

The "Number of Bug Reports" columns of Table 3 show the number of reports from different bug detectors. The reports for each case from each detector are mostly very different from each other and are not just multiple stack traces of the same static instructions.

The "Overall Patch Quality" columns of Table 3 summarize our experimental results. In this table, "✓" indicates clear and complete success: the original concurrency-bug problem is completely fixed, no new bug is observed, and performance degradation is negligible. "-" indicates that

Table 3: Results of OFix patches. The a and f subscripts indicate all$A$–B and first$A$–B bug reports respectively. The L and R subscripts indicate Local-is-Bad and Remote-is-Bad definition-use bug reports respectively. ✓ a good patch; - good patch is not generated; blank: not applicable.

	Number of Bug Reports				Overall Patch Quality						Failure Rates		Overhead		# of CFix
ID	AV	OV	RA	DU	AV	OV	RA	DU	CFix	Manual	Original	CFix	CFix	Manual	Sync Ops
OB1	2	5_a	4		✓	✓	✓		✓	✓	43%	0%	-0.3%	1.6%	5
OB2		1_a				✓			✓	✓	65%	0%	-0.1%	3.6%	7
OB3	7	4_f	10	4_L	✓	✓	✓	✓	✓	✓	100%	0%	0.2%	0.0%	5
OB4	1	1_f	2		✓	✓	✓		✓		97%	0%	0.5%		2
OB5	1		1		✓		✓		✓	✓	64%	0%	0.0%	0.0%	2
OB6	1	1_f	2	1_L	✓	✓	✓	✓	✓	✓	93%	0%	0.3%	-0.3%	2
OB7		1_f				✓			✓	✓	97%	0%	0.2%		3
AB1	6		6		✓		✓		✓	✓	52%	0%	-0.9%	-0.4%	3
AB2	1		2	1_R	✓		✓	✓	✓	✓	39%	0%	0.7%	0.5%	5
AB3	2		4	2_R	✓		✓	-	✓	-	53%	0%	-0.0%	1.0%	9
AB4	1		2		✓		✓		✓	-	55%	0%	-0.5%	0.0%	3
AB5	4		5	1_R	✓		✓	✓	✓	✓	68%	0%	-0.2%	0.4%	2
AB6	1		2	1_R	✓		✓	✓	✓	✓	42%	0%	0.7%	0.5%	5

CFix fails to generate a patch that passes its own internal testing. Blanks are cases where no bug is reported by the corresponding front end. For manual patches, "-" means developers submitted intermediate patches that were later found to be incomplete by developers or testers (i.e., the original software failure can still occur with the patch applied); blanks mark cases where developers have not yet provided any patch for the corresponding bug.

Overall, CFix is highly effective. Across all four bug-detection front ends and all 13 benchmarks, CFix successfully fixes all bugs except two reports for one case (AB3) under one front end (DU). CFix's final patches are all of high quality regarding correctness, performance and simplicity, comparable with the final patches designed by software developers. In several cases, CFix patches are even better than the first few patches generated by developers. In the case of HTTrack, CFix creates good patches while the developers have yet to propose any patches at all. Note that Jin et al.'s work on automated atomicity-bug fixing [20] only works with the AV front end and can only fix 6 cases, AB1 through AB6.

6.3 Patching for Different Bug Detectors

CFix generates one or more patches using the fixing strategies described in Section 2. Among all front ends, OV has the most straightforward fixing process. OV only detects order violations. It reports six all$A$–B violations and seven first$A$–B violations. CFix generates one ordering patch for each report. All of these patches pass correctness testing.

Fixing AV bug reports is much more challenging. As shown in Table 4, AV finds 27 atomicity violations for 11 benchmarks. Among these, 12 reports from OB1 through OB6 are side effects of order violations, akin to Figure 2. These are examples of bug reports that are different from

Table 4: Patch testing and selection for AV front end. Subscripts aO, fO, and A indicate all$A$–B ordering, first$A$–B ordering, and mutual exclusion patches respectively. C1–C5 count correctness rejections as in Section 4. P and R respectively count performance and simplicity rejections.

	# AV	Rejected Patches							Final
ID	Bugs	C1	C2	C3	C4	C5	P	R	Patch
OB1	2	0	0	0	4	0	1	1	2_{aO}
OB3	7	0	7	0	0	7	0	0	7_{aO}
OB4	1	1	1	0	0	1	0	0	1_{fO}
OB5	1	0	0	0	2	0	0	1	1_{aO}
OB6	1	0	1	0	0	1	0	0	1_{aO}
AB1	6	12	0	5	7	0	0	0	6_A
AB2	1	2	0	1	1	0	0	0	1_A
AB3	2	4	0	2	2	0	0	0	2_A
AB4	1	2	0	0	2	0	0	0	1_A
AB5	4	8	0	2	6	0	0	0	4_A
AB6	1	0	0	0	2	0	1	0	1_A

actual root cause. The software would still fail if we merely enforced mutual exclusions based on these 12 bug reports.

CFix successfully picks patches that match the root causes and completely fixes the 11 benchmarks. Table 4 summarizes this process. For each AV bug report, CFix tries to generate 3–5 patches: one mutual-exclusion patch and two all$A$–B ordering patches, as shown in Table 1(a). If an all$A$–B patch is rejected due to timeouts or deadlocks, the corresponding first$A$–B patch is generated and tested instead. Although testing multi-threaded software is challenging, the static (C1) and dynamic (C2–C5) correctness checks complement each other and help CFix identify and reject bad patches. Most patches that do not reflect root causes are rejected this way, as shown by the C1–C5 columns of Table 4. In a few cases in OB1, OB5, and AB6, the buggy software can be fixed by either mutual-exclusion or order synchronization. CFix generates both patches and selects the one with the best performance and simplicity.

Fixing RA bugs is also challenging, because a data-race report itself contains no root cause information. CFix successfully generates final patches for RA bug reports as follows. RA finds 19 data races for 5 benchmarks with order-violation root causes. Following Section 2.4, CFix decides that no mutual-exclusion patch is suitable for any of these bugs. The ordering patches generated by OFix all pass correctness testing and are selected as CFix's final patches. RA also finds data races for 6 benchmarks that have mutual-exclusion root causes (AB1–AB6). CFix generates mutual-exclusion patches, all of which pass correctness testing. However, all ordering patches for AB2, AB3, AB4 and some ordering patches for AB1 and AB5 are rejected, because OFix statically determines that these patches would cause deadlocks. Ordering patches for AB6 are rejected due to failures under guided testing. CFix does not generate any ordering patch for 6 data-race bugs in AB1 and AB5, because the RA front-end indicates that software fails as long as the race instructions execute one right after the other regardless of the order between them.

The DU patch-generation process is similar to that for OV in the case of Local-is-Bad reports, and is similar to that for AV in the case of Remote-is-Bad reports. However, CFix fails to generate any patch for AB3. The problem in AB3 is that two reads R1 and R2 should not read values defined by different write instructions. Unfortunately, DU simply reports R1 should not read values from a particular write instruction. CFix statically determines that disabling this data dependence, without considering R2, causes deadlocks, and therefore does not generate a patch.

CFix's final patches are generally identical or have only trivial differences as we switch from one front end to another. The two exceptions are in OB3 and OB4. In OB3, CFix generates a first*A*–*B* patch for an OV-reported bug following the fix strategy design, but an all*A*–*B* patch for three other front ends. In fact the program can only execute one instance of *A* at run time, so these two patches only differ in simplicity. In OB4, the final patch generated for RA and the one generated for AV and OV both correctly fix the reported failure without perceivable performance differences. The RA patch also fixes unreported failures under different inputs. Unless otherwise specified, we use majority vote to select final patches for evaluation results in Table 3 and Section 6.4.

6.4 Quality of CFix's Final Patches

Correctness Results As discussed in Section 4, CFix's final patches have all passed the guided testing of CFix bug-detection front end, without triggering the previously reported failures. To further test the correctness of CFix's final patches, we insert random sleeps in code regions that are involved in each bug report. The "Failure Rates" columns of Table 3 show the failure rates of the original buggy software and the CFix-patched software through 1,000 testing runs with the same random sleep patterns. As we can see, CFix patches eliminate all of the failures. In addition, the timeouts CFix inserted in its wait operations have never fired in our experiments with these final patches. Thus, our deadlock-avoidance heuristics, while imperfect in theory, perform extremely well in practice.

Manual inspection confirms that these fixes are correct and nontrivial. For example, half of the benchmarks cannot be fixed using locks alone. In OB1 and OB2, either the number of signal threads or the number of *A* instances per thread is not statically bound. Without OFix's careful analyses, naïve patches could easily lead to deadlocks or fail to fix the problem. A naïve first*A*–*B* patch without the safety net would cause FFT to hang nondeterministically.

Performance Results The "Overhead" columns of Table 3 show the overheads of both CFix and manual patches relative to the original buggy software. All CFix overheads are below 1%, which is comparable with correct manual patches. These results are averages across 100 non-failing runs of each version of the software with potentially-bug-triggering inputs.

For ordering patches, good performance stems from OFix's efforts to signal as soon as possible and wait as late as possible. This leads to little or no unnecessary delay in the program. For mutual-exclusion patches, good performance is due to short critical sections.

The static analyses in OFix perform well, taking less than one second to generate one patch. We anticipate no scalability problems for larger code bases.

Simplicity Results Jin et al. [20] have shown that AFix can provide mutual-exclusion patches with good simplicity. OFix uses patch optimization (Section 3.1) and patch merging (Section 5) to simplify ordering patches. To evaluate these two techniques, Table 5 presents detailed counts of signal operations (numbers followed by "s") and wait operations (numbers followed by "w") in CFix's final patches that are generated by OFix, under different optimization and merging strategies. In the few cases where different final patches are generated under different front ends, we present worst-case results for the final patch with the most synchronization operations.

The "All Opt" column of Table 5 counts synchronization operations with both simplicity optimizations enabled. We find that these numbers are quite moderate when patch merging is also enabled, highlighted in **bold**. Out of seven benchmarks, six can be fixed with no more than five synchronization operations. In the worst case, OB2 can be fixed with six signal operations and one wait operation. Manual inspection confirms that all of the synchronization operations in these final patches are genuinely necessary for our fixing strategy.

Table 5: Number of synchronization operations in patches

ID	All Opt	Only 1st	Only 2nd	No Opt
OB1 merged	**4s, 1w**	19s, 1w	4s, 1w	19s, 1w
OB1 unmerged	20s, 5w	95s, 5w	20s, 5w	95s, 5w
OB2	**6s, 1w**	8s, 1w	20s, 1w	22s, 1w
OB3 merged	**4s, 1w**	17s, 1w	4s, 1w	17s, 1w
OB3 unmerged	40s,10w	170s,10w	40s, 10w	170s,10w
OB4 merged	**1s, 2w**	1s, 2w	1s, 2w	1s, 2w
OB4 unmerged	2s, 2w	2s, 2w	2s, 2w	2s, 2w
OB5	**1s, 1w**	1s, 1w	4s, 1w	4s, 1w
OB6 merged	**1s, 1w**	1s, 1w	1s, 1w	10s, 1w
OB6 unmerged	2s, 2w	2s, 2w	2s, 2w	20s, 2w
OB7	**2s, 1w**	2s, 1w	36s, 1w	36s, 1w

OB1 and OB3 respectively have five and ten bugs reported by the corresponding front-end. As a result, their unmerged OFix patches contain twenty five and fifty synchronization operations respectively, severely hurting the code simplicity. Fortunately, only five synchronization operations remain in the merged patches: a very modest number considering the number of bugs fixed.

The "Only 1st" and "Only 2nd" columns of Table 5 show how many synchronization operations would be required if each of the two simplicity optimizations of Section 3.1 were used in isolation, while "No Opt" shows the effect of disabling both optimizations. These large numbers on OB1 and OB3 are caused by unnecessary signal operations before **return** statements as in Figure 6b. Many such statements appear in the command-line option parsing code for these two applications. OB4 is the only benchmark that does not benefit from simplicity optimization. In fact, OFix initially adds 103 safety-net signal operations into OB4. Further analysis of OFix finds that the B operation in OB4 is post-dominated by a safety-net signal. OFix therefore removes the entire safety net before optimization is applied, per Section 3.2. For the other benchmarks, the optimizations are quite effective; with neither in place, our patches would have 2.5 to 12 times as many synchronization operations. Each of the two optimizations has its own strengths. OB5 and OB7 benefit from the first optimization; OB1 and OB3 benefit from the second; and OB2 and OB6 benefit from both.

Our current CFix implementation operates directly on LLVM bitcode, not source code. However, these simplicity results suggest that CFix patches are a good starting point for generating clean, readable source-level patches as well.

7 Limitations of CFix

Although CFix correctly fixes all bugs that require order enforcement in our evaluation using OFix, OFix is not a universal fixer for all possible bugs that require order enforcement. OFix is restricted by the fact that it only tries two different orderings, as well as by its use of call stacks to identify operations. Therefore, OFix cannot fix bugs that require all$A$–B or first$A$–B relationships between some, but not all, instances of one call stack and some instances of another call stack.

```
char *buffer[10];                      void main() {
                                         for (i = 0; i < 10; ++i) {
void child(...) { // child-i               pthread_create(child, ...);
  ...                                      buffer[i][0] = 'a'; // B
  buffer[i] = malloc(32); // A             free(buffer[i]);
  ...                                    }
}                                      }
```

Figure 11: Example that presents a challenge to OFix

Figure 11 shows a bug that OFix cannot fix. Operations A and B require order enforcement, but share the **for** loop in function main. The ideal way to fix this bug is to force each dynamic instance of B ("buffer[i][0] = 'a'" in main) to wait until after the corresponding instance of A ("buffer[i] = malloc(32)" in child). The all$A$–B strategy cannot fix this bug, because it would make the main thread wait during the first iteration of the loop for all potential instances of A, which causes a deadlock-induced timeout. The first$A$–B strategy cannot fix this bug either: after the first B instance, a later instance of B could still happen before the corresponding instance of A.

To fix the above bug requires using loop indexes as part of operations' identities and enforcing order relationships accordingly. This type of code pattern is common in server applications with dispatch loops. As a result, OFix has certain limits in those applications.

The bug cases evaluated in Section 6 include bugs whose buggy code regions are contained in one loop: AB1–AB6. The patch testing and selection process of CFix has correctly judged that OFix cannot fix any of these bugs. These six bugs happen to all require atomicity enforcement and are correctly fixed by AFix.

The other two benchmarks in papers on ConMem [69], ConSeq [70], and DUI [53] require order enforcement but are not included in our evaluation. Our preliminary results show that one of them can be patched correctly by CFix. The other cannot, as it is a true order violation but the two operations share a loop as discussed earlier in this section.

CFix may fail to fix a bug in several other scenarios. First, the software may have deep design flaws and not be fixable through synchronization enforcement alone. Second, the bug detector may provide insufficient information for bug fixing, such as AB3 under DU. Third, OFix makes a best effort to avoid deadlocks, by signaling as soon as possible in signal threads and waiting as late as possible in wait threads. When OFix patches suffer deadlocks, CFix concludes that the bug should not be fixed through ordering enforcement. However, this could be wrong in rare cases. For example, complex branch conditions may cause infeasible paths and prevent OFix from identifying earlier opportunities to signal. While possible in theory, this never occurs in our experiments: OFix successfully generates final patches without deadlocks detected during patch testing. Lastly, CFix patch testing cannot guarantee to catch all problems in a patch, as this is infeasible for any large multi-threaded application. The CFix run-time supports production-run patch monitoring and could potentially be extended to avoid deadlocks at run time [21].

Based on our experience, CFix can work with most concurrency bug detectors that report failure inducing interleavings. Adding a new detector as CFix front end mainly requires a corresponding fix-strategy design, as discussed in Section 2. CFix currently uses four front ends that report no false positives. If a future front end reports false positives or benign races, CFix may enforce some unnecessary synchronizations in the program. This may result in a patch with poor performance or that cannot pass the testing stage in the first place.

In some cases, CFix patches are more complicated than manual patches. These manual patches use non-lock/condition-variable synchronization primitives and sometimes leverage developers' knowledge of special program semantics. For example, some order violations are fixed by swapping the order of original program statements and some are fixed by using pthread_join. Future work can further simplify CFix patches in this direction.

CFix's patch currently operate in terms of LLVM bitcode. This enables quick deployment but makes developers' involvement difficult in the long term. Our evaluation shows CFix can generate compact patches containing few new synchronization operations. This lays a good foundation for eventual production of simple source-level patches. Such a transformation tool will need to consider additional source-level syntax issues that we have not addressed here. We leave such an extension to future work.

8 Related Work

As discussed in Section 1, many concurrency-bug detectors have been proposed. These tools aim to identify problems, not to fix them. Therefore, they inevitably leave many challenges for bug fixing, such as figuring out root causes and inserting synchronization operations correctly without unnecessary degradation in performance or simplicity. As a bug-fixing tool, CFix has considered and addressed these challenges, complementing bug detectors.

Techniques have been proposed to insert lock operations into software based on annotations [37, 57], atomic regions inferred from profiling [61], and whole-program serialization analysis [56]. QuickStep [23] automatically selects functions to put into critical sections based on race-detection results during loop parallelization. Recent work by Navabi et al. [40] parallelizes sequential software based on future-style annotations. It automatically inserts barriers to preserve sequential semantics during parallelization.

Compared with the above techniques, CFix is unique in fixing concurrency bugs reported by a wide variety of bug detectors and in synchronizing using both locks and condition variables. CFix addresses unique challenges such as fix-strategy design, simplicity optimization, patch merging, and patch testing. The static analysis conducted by OFix differs from that of Navabi et al. [40] by considering additional issues such as simplicity and performance.

Program synthesis [11, 55, 58] uses verification techniques to generate synchronized programs that satisfy certain specifications. The nature of the problem makes it hard to scale to large, real-world applications. CFix does not try to understand all synchronizations in a program and therefore avoids the associated scalability problems.

Hot-patching tools fix running software. ClearView [45] patches security vulnerabilities by modifying variable values at run time. Its design is not suitable for concurrency bugs. The LOOM system [62] provides a language for developers to specify synchronizations they want to add to a running software and deploys these synchronization changes safely. Similar to CFix, LOOM also does CFG reachability analysis for safety, and has a run-time component to recover from deadlocks. Since LOOM has different design goals from CFix, it does not need to consider issues like working with bug detectors, fix-strategy design, locating synchronization operations, handling statically-unknown numbers of signals, simplicity concerns, patch merging and testing. Tools like CFix can potentially complement LOOM by automatically generating patches for LOOM to deploy.

Run-time tools can help survive some concurrency bugs [7, 21, 24, 26, 34, 49, 59, 66, 67]. Since CFix aims to permanently fix bugs, it has different design constraints and must address unique challenges, such as fixing a wide variety of bugs completely, instead of statistically, with unknown root causes, statically locating synchronization operations while lowering the risk of deadlock, maintaining simplicity, patch testing, and patch selection.

Many record-replay tools [1, 44, 60, 63] and production-run bug detectors [6, 19, 36, 59] have been proposed. They can enable CFix to fix bugs discovered in production runs.

Deterministic systems [3–5, 9, 10, 29, 42] can make some concurrency bugs deterministically happen and some other bugs never occur. This promising approach still faces challenges, such as run-time overhead, integration with system non-determinism, language design, etc. In general, these tools address different problems than CFix. Even for software executed inside a deterministic environment, fixing bugs still requires manual intervention. CFix and these tools can complement each other. Tern [9] and Peregrine [10] proposed precondition computation to enforce specific interleavings for selected inputs. This technique can potentially be used to enable or disable CFix patches for selected inputs.

9 Conclusion

CFix is a framework for automatically fixing concurrency bugs. For concurrency bugs reported by a wide variety of detection tools, CFix automatically inserts synchronization operations to enforce the desired orderings/mutual-exclusion and fix the bugs. CFix uses testing to select the best patch among patch candidates, and incorporates optimization and merging algorithms to keep patches simple. Experimental evaluation shows that CFix produces high-quality patches that fix real-world bugs while exhibiting excellent performance. CFix is a significant step forward toward relieving software developers of the time-consuming and error-prone task of fixing concurrency bugs. It can be used to generate patches or patch candidates for developers. Its analysis, testing, and run-time monitoring results can also provide useful feedback to both developers and bug detection tools.

Acknowledgments

We thank the anonymous reviewers for their invaluable feedback, and our shepherd, Jason Flinn, for his guidance in preparing the final version. We thank the Opera group from UCSD for sharing with us their bug benchmarks. This work is supported in part by DoE contract DE-SC0002153; LLNL contract B580360; NSF grants CCF-0701957, CCF-0953478, CCF-1018180, CCF-1054616, and CCF-1217582; and a Claire Boothe Luce faculty fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] C. Armour-Brown, J. Fitzhardinge, T. Hughes, N. Nethercote, P. Mackerras, D. Mueller, J. Seward, B. V. Assche, R. Walsh, and J. Weidendorfer. *Valgrind User Manual*. Valgrind project, 3.5.0 edition, Aug. 2009. http://valgrind.org/docs/manual/manual.html.

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.

[7] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys*, 2010.

[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[9] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, 2010.

[10] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[11] J. Deshmukh, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Logical concurrency control from sequential proofs. In *ESOP*, 2010.

[12] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[13] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[14] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[15] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[16] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2nd-Strike: toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, 2011.

[17] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[19] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[20] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[21] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[22] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with Portend. In *ASPLOS*, 2012.

[23] D. Kim, S. Misailovic, and M. Rinard. Automatic parallelization with statistical accuracy bounds. Technical Report MIT-CSAIL-TR-2010-007, MIT, 2010. URL http://hdl.handle.net/1721.1/51680.

[24] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD*, 2007.

[25] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[26] Z. Letko, T. Vojnar, and B. Křena. AtomRace: data race and atomicity violation detector and healer. In *PADTAD*, 2008.

[27] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162.

[28] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *USENIX*, 2005.

[29] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

[30] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. In *ASPLOS*, 2006.

[31] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, 2007.

[32] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

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

[34] B. Lucia, J. Devietti, L. Ceze, and K. Strauss. Atom-Aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29(1), 2009.

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

[36] D. Marino, M. Musuvathi, and S. Narayanasamy. Effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[37] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.

[38] MySQL. Bug report time to close stats. http://bugs.mysql.com/bugstats.php, Dec. 2011.

[39] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.

[40] A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static scheduling for safe futures. In *PPOPP*, 2008.

[41] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[42] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[43] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[44] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[45] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[46] K. Poulsen. Software bug contributed to blackout. http://www.securityfocus.com/news/8016, Feb. 2004.

[47] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *PLDI*, 2006.

[48] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

[49] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP*, 2009.

[50] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15, 1997.

[51] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[52] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. In *WBIA*, 2009.

[53] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition?: DefUse: definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[54] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *HotDep*, 2007.

[55] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[56] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Automatic atomic region identification in shared memory SPMD programs. In *OOPSLA*, 2010.

[57] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[58] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[59] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, 2011.

[60] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *ASPLOS*, 2011.

[61] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the positive: Atomicity inference and enforcement using correct executions. In *OOPSLA*, 2011.

[62] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *OSDI*, 2010.

[63] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *FSE*, 2010.

[64] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[65] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[66] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[67] J. Yu and S. Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *MICRO*, 2010.

[68] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[69] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, 2010.

[70] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.