# Automated Atomicity-Violation Fixing *

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

## Abstract

Fixing software bugs has always been an important and time-consuming process in software development. Fixing concurrency bugs has become especially critical in the multicore era. However, fixing concurrency bugs is challenging, in part due to non-deterministic failures and tricky parallel reasoning. Beyond correctly fixing the original problem in the software, a good patch should also avoid introducing new bugs, degrading performance unnecessarily, or damaging software readability. Existing tools cannot automate the whole fixing process and provide good-quality patches.

We present AFix, a tool that automates the whole process of fixing one common type of concurrency bug: single-variable atomicity violations. AFix starts from the bug reports of existing bug-detection tools. It augments these with static analysis to construct a suitable patch for each bug report. It further tries to combine the patches of multiple bugs for better performance and code readability. Finally, AFix's run-time component provides testing customized for each patch. Our evaluation shows that patches automatically generated by AFix correctly eliminate six out of eight real-world bugs and significantly decrease the failure probability in the other two cases. AFix patches never introduce new bugs and usually have similar performance to manually-designed patches.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and Debugging; D.4.1 [*Operating Systems*]: Process Management

***General Terms*** Algorithms, Experimentation, Languages, Measurement, Performance, Reliability, Verification

***Keywords*** atomicity violations, automated debugging, concurrency, critical regions, deadlock, mutex locks, mutual exclusion, patching, static analysis

* Supported in part by AFOSR grants FA9550-07-1-0210 and FA9550-09-1-0279; DoE contract DE-SC0002153; LLNL contract B580360; NSF grants CCF-0621487, CCF-0701957, CCF-0953478, CCF-1018180, and CNS-0720565; 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*PLDI'11,* June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06...$10.00

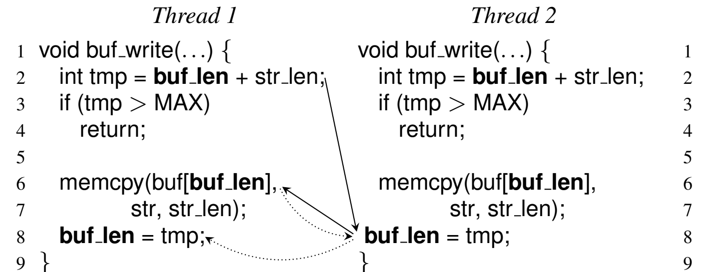



**Figure 1.** Real-world concurrency bug from Apache. Interleaving "⟶" could cause crash. Interleaving "⇢" could corrupt the log.

## 1. Introduction

### 1.1 Motivation

Bug fixing is an indispensable part of software development. It requires developers to understand a bug's root cause, design a patch, implement the patch, and finally validate the patch. This process consumes a huge amount of resources, especially manual effort, during software development. Krebs [15] finds that it frequently takes more than one month to finish the fixing process for one bug. Meanwhile, there are endless bugs waiting to be fixed. Software companies such as Microsoft face pressure to release patches monthly or even more frequently [24]. Furthermore, patches are error-prone. Even after consuming so many development resources, nearly 70% of patches are buggy in their first release [7, 32, 34], often at great financial cost [28]. Self-healing software that fixes its own bugs has long been desired but is yet unrealized [11]. The need for automatic repair techniques remains for concurrency bugs.

Concurrency bugs are synchronization mistakes in multithreaded programs. They are widespread due to software developers' sequential thinking habits. Concurrency bugs have already caused real-world disasters [30]. In the current multi-core era, with multithreaded software becoming pervasive, concurrency bugs are a growing threat to software reliability.

Many advanced techniques have been proposed for bug detection, software testing, and verification to help identify concurrency bugs [9, 26, 38]. However, software reliability does not improve until detected bugs are actually fixed. Unfortunately, fixing concurrency bugs is not trivial and developers are left to themselves to face the enormous pressure of fixing ever-so-many concurrency bugs.

Figure 1 shows an example of a real-world concurrency bug. Existing bug detectors [5, 26, 38] can accurately report two problematic atomicity violations: (1) when lines 2 and 6 are interleaved (⟶) by line 8, Apache could crash; and (2) when lines 6 and 8 are interleaved (⇢) by line 8, Apache could corrupt its log. Unfortunately, even with accurate bug detection, bug fixing is nontrivial:

- To fix the first report, if we simply lock before line 2 and unlock after line 6, the program could deadlock after buf_write exits at line 4.
- To fix the second report, we should not simply put lines 6–8 and line 8 each into one critical region. As line 8 is part of the critical region for lines 6–8, this would lead to deadlock again.
- Two patches that separately fix the above two atomicity violations could deadlock with each other: one thread acquires the first patch's lock at line 2 and waits for the second patch's lock before line 6; a different thread acquires the second patch's lock at line 6 and waits for the first patch's lock before line 8.

Generally speaking, developers face several unique challenges to generate good patches for concurrency bugs. First, concurrency bugs pose unique challenges in understanding their root causes. Their non-determinism makes manual inspection difficult. In addition, understanding their root causes demands non-local and parallel thinking. Even with the help of tools for bug detection [4, 12], failure diagnosis [27], and failure replay [40, 43], it is still non-trivial to understand how to fix a bug's root cause. As a result, it is common that patches released by developers fail to (completely) fix the original concurrency bug. This happened for two of the eight real-world concurrency bugs used in our experimental evaluation.

Second, patches to concurrency bugs often involve synchronization operations that have non-local impact. As a result, these patches can easily introduce additional bugs, such as deadlocks and data races. As a notorious example, the patch to Mozilla concurrency bug #54743 led to new concurrency bugs in the field. Patches for the latter caused further problems, which took more than one year to finally fix [14].

Third, patches to concurrency bugs often constrain program interleavings or introduce serialization bottlenecks, which could cause unexpected and unnecessary performance degradation.

Because of these challenges, long repair times and wrong patches are common for concurrency bugs [20]. Future programming languages may eventually help developers avoid some of these concurrency bugs. For now, though, software developers are in great need of immediate support for automatic repair of concurrency bugs.

### 1.2 Contributions

In this paper, we build a system, AFix, that automates the *whole* process of fixing one common type of concurrency bug: single-variable atomicity violations [9, 19, 20, 26, 38]. AFix leverages existing techniques for bug detection and interleaving testing to bootstrap its fixing process. It uses static analysis and static code transformation to automatically design and implement code patches. It further incorporates run-time monitoring to help developers validate and evaluate each patch it generates.

The design of AFix focuses on the challenges encountered by developers in their manual patching process. Specifically, AFix tries to fix the original bug without introducing new functionality problems, degrading performance excessively, or harming code readability.

Guided by these goals, AFix automates a developer's typical bug fixing process. The first step is *bug understanding*. AFix discovers single-variable atomicity violations using CTrigger [26], an existing bug-detection and testing tool which we review further in Section 2.

The second step is *patching one bug*. AFix maps the dynamically-based bug report to static code structures. It conducts static analysis and static code transformation to fix one problem. Correctness issues are thoroughly analyzed in this step, which we discuss in Section 3.

The third step is *patch merging and optimization* for a set of bugs. AFix collects the patches for each bug report together and statically identifies patches that can be merged or optimized for better performance or readability. This step is addressed in Section 4.

The fourth step is *patch testing*. AFix conducts testing and run-time analysis customized for each patch. Testing checks whether the original problem has been fixed and looks for new problems. The low-overhead run-time analysis targets correctness and performance properties that cannot be statically guaranteed or optimized in preceding steps. It provides the developers an option to compare and refine patches. We discuss this step in detail in Section 5.

Overall, this paper makes the following contributions:

- AFix makes a first step in automating the *whole* process of fixing one common type of concurrency bug. AFix can save developers' manual bug fixing effort by automatically generating patches or patch candidates for concurrency bugs detected during in-house testing or for concurrency failures discovered during production runs. AFix can also help address some tough challenges that usually bother developers. For example, AFix can avoid generating buggy patches. AFix's run-time analysis also helps developers conduct customized patch testing and evaluation.
- Our experience of applying AFix discovers several limitations of state-of-art bug detectors regarding helping bug diagnosis and bug fixing, such as not grouping bugs with the same root cause together and reporting incorrect root causes. We expect this experience to help guide future research in bug detection.
- We evaluate AFix on eight real-world concurrency bugs, with promising results. AFix correctly fixes six out of eight reported bugs. For the remaining two bugs, where the root causes were incorrectly reported by CTrigger, AFix patches significantly decrease failure rates. AFix patches also have good readability based on our manual inspection and do not introduce any new bugs. In comparison, naïve patches cannot fix any of these bugs, and cause deadlocks in seven out of eight cases. Even the original developers made mistakes in fixing two of these eight bugs. AFix patches also show good performance. Software patched by AFix is less than 1% slower than the original buggy software for all but one case.

## 2. Background: Bug Detection Using CTrigger

AFix is a bug fixer, not a bug finder. It depends on problem reports from existing bug-detection tools to guide its repairs. The high-level ideas in AFix are general to all atomicity-violation detectors, but the lower-level details are tuned to the specific bug finder used. This section briefly describes CTrigger: the specific concurrency-bug detector and tester used by our current AFix implementation.

### 2.1 CTrigger General Operation

CTrigger is an in-house concurrency-bug detection and testing framework [26] that targets single-variable atomicity-violation bugs. As shown in Figure 2, when two consecutive accesses that read or write the same shared variable from the same thread are interleaved by another access from a different thread, their execution effect may be different from any serial execution, in which case a single-variable atomicity-violation bug has occurred. An empirical study by Lu et al. [20] finds that single-variable atomicity-violation bugs are one of the most common types of concurrency bug in real code.

CTrigger includes a detection phase and a testing phase. During its detection phase, CTrigger monitors a few executions of the concurrent program, predicts what single-variable atomicity violations could happen in the future under the same input, and outputs the instruction counters for the three instructions involved in each atomicity violation. These three instructions are referred to as *p* (preceding), *c* (current), and *r* (remote), as shown in Figure 2. In its testing phase, CTrigger tries to force each atomicity violation identified above by injecting delays at selected points. If an atomicity

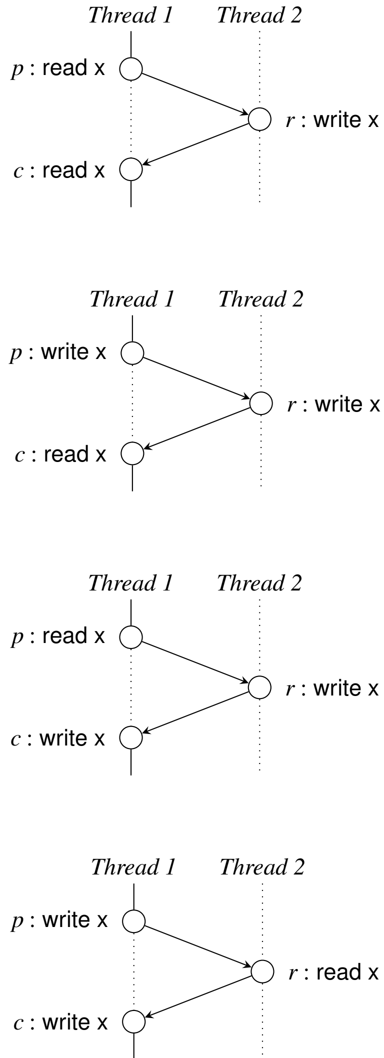



**Figure 2.** Types of atomicity violations that CTrigger detects. "⟶" shows execution order. "x" is a shared memory location.

violation occurs and the program execution fails, CTrigger reports the involved $(p, c, r)$ triple. At the end, a list of atomicity violations that can truly harm the software are reported.

In this project, we use CTrigger bug reports to drive AFix. We will discuss how to extend AFix for general atomicity-violation bugs in Section 3.7. In remainder of this paper, when we talk about atomicity violations, by default, we mean the type of single-variable bugs reported by CTrigger. We continue to use $p$, $c$, and $r$ to represent the three instructions involved in each atomicity violation, as shown in Figure 2.

### 2.2 CTrigger Modifications and Limitations

During the design and evaluation of AFix, we modified two aspects of CTrigger to better drive automated bug fixing. First, the original CTrigger only reports one $(p, c, r)$ triple for each distinct $c$ instruction, to simplify its design. This is probably a good decision for bug detection, but not a good one for bug fixing, because it leads to incomplete patches. We modified CTrigger to output all possible combinations. Second, the original CTrigger only reports instruction counters for $(p, c, r)$, which again is problematic in bug fixing, especially when $p$ and $c$ are inside different functions (discussed in Section 3). Therefore, we modified CTrigger to report the complete call stack for each $p$, $c$, or $r$.

AFix can be no wiser than the bug detector which drives its fixes. Our CTrigger-based implementation of AFix, then, is affected by a few other features of CTrigger. For the most part these are not peculiar to CTrigger, though, but rather are shared with other concurrency-bug detection and testing tools.

First, separate CTrigger bugs sometimes should not be fixed by separate patches. We discuss this further in Section 4.

Second, CTrigger might detect an atomicity violation that is a side effect of another non-atomicity-violation bug. As a result, no matter how well AFix fixes the reported CTrigger bug, the software failure may still occur. We discuss this further in Section 6.

### 2.3 A Naïve Fixing Scheme

Given one CTrigger bug report, a naïve patch is to acquire some lock before $p$ and $r$, then release the same lock after $c$ and $r$. As shown in Figure 1, for many real-world bugs, this naïve patch would cause problems, such as introducing new bugs, introducing significant and unnecessary performance degradation, and hurting code readability. We have implemented this naïve fixing scheme and compare it with AFix using real-world bugs in Section 6.

## 3. Fixing One Bug Report

This section describes how AFix fixes a single atomicity violation detected by CTrigger. The next section will discuss how to fix multiple violations together.

### 3.1 Overview

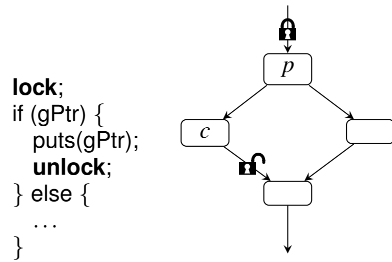
As discussed in Section 2.1, a single CTrigger bug report identifies an instruction $r$ that may non-serializably interleave two other instructions $p$ and $c$, thereby causing failure. Fixing this bug requires changing the code to ensure that the code region from $p$ to $c$ is mutually exclusive with $r$. AFix performs this change in four steps:

1. Put $p$ and $c$ into a critical region. The challenge here is to guarantee that $p$ and $c$ are inside one critical region under all possible control flows, without introducing new bugs. Potential new bugs to avoid include double lock, double unlock, unlock without lock, deadlocks, and others shown in Figure 3. We describe how AFix handles this step in Sections 3.2 to 3.4.
2. Put $r$ into another critical region. This step can be easily done by adding a lock-acquisition operation before $r$ and a lock-release operation after $r$.
3. Make the above two regions mutually exclusive with respect to each other. This step involves more than just assigning the same lock to both critical sections. We discuss hidden traps and how to avoid them in Section 3.5.
4. Select or introduce a lock to protect the two critical regions. Section 3.6 discusses how a lock is chosen.

### 3.2 Single-Function Operation

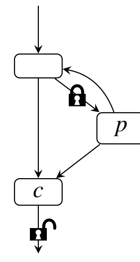
We start with an algorithm that decides where to acquire and release locks when $p$ and $c$ reside in the same function. We further assume that this function is not recursive. Under these restrictions, the algorithm follows a natural strategy: find all nodes that are on any path from $p$ to $c$, and make sure a lock is held at exactly these nodes.

To implement this strategy, AFix first analyzes the control-flow graph to get the set of CFG nodes that are on any intraprocedural path that starts from $p$ and ends at $c$ without touching $p$ or $c$ in



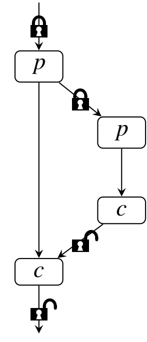

**(a)** Unreleased lock; potential deadlock later

```
while (...) {
  lock;
  ptr = aPtr;
}
puts(ptr);
unlock;
```




**(b)** Potential double lock or unlock without lock

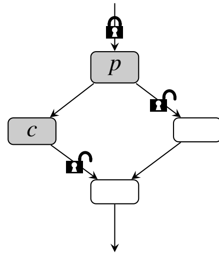
```
void foo() {
  lock;
  ptr = aPtr;
  if (...) foo();
  puts(ptr);
  unlock;
}
```




**(c)** Potential double lock

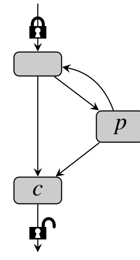
**Figure 3.** Traps in bug fixing. Lock and unlock operations have been placed before and after each $p$ and $c$ node, respectively.

```
lock;
if (gPtr) {
  puts(gPtr);
  unlock;
} else {
  unlock;
  ...
}
```



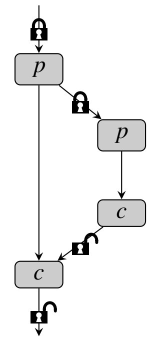

**(a)** Added unlock avoids unreleased lock

```
lock;
while (...) {
  ptr = aPtr;
}
puts(ptr);
unlock;
```




**(b)** Moved lock avoids double lock or unlock without lock

```
void foo() {
  reentrant_lock;
  ptr = aPtr;
  if (...) foo();
  puts(ptr);
  reentrant_unlock;
}
```




**(c)** Reentrant lock avoids double lock

**Figure 4.** AFix's handling of traps from Figure 3. Shaded nodes comprise the $p$–$c$ critical region.

between.[1] We refer to this set of nodes as the *protected nodes*: these are the nodes which will be included in the critical region. The protected nodes may be computed as follows:

1. Search forward from $p$ (either depth- or breadth-first) without ever crossing beyond $c$. That is, temporarily treat $c$ as having no successors for purposes of this search. Call this set of all forward-reachable nodes $P$.
2. Search backward from $c$ (either depth- or breadth-first) without ever crossing beyond $p$. That is, temporarily treat $p$ as having no predecessors for purposes of this search. Call this set of all backward-reachable nodes $C$.
3. The protected nodes are $P \cap C$. This is the set of nodes that are reachable from $p$, and from which $c$ can be reached, without additional crossings through $p$ or $c$ along the way.

It is not difficult to prove that this algorithm correctly identifies the protected nodes which must form a critical region. Any node in $P$ must be reachable from $p$ without crossing any successor edge of $c$. Conversely, any node in $C$ must be able to reach $c$ without crossing any predecessor edge of $p$. Therefore, any node in $P \cap C$ must be along a $p$–$c$ path in which $p$ and $c$ respectively appear only first and last, never as intermediate nodes.

Each depth- or breadth-first search requires time at most linear in the number of nodes and edges. Computing the intersection is likewise at most linear in the number of nodes. Therefore the entire algorithm to identify protected nodes is at most linear in the number of nodes and edges in the CFG for the function containing $p$ and $c$.

Next, AFix inserts lock-acquisition operations on each edge that crosses from an unprotected node to a protected node, and inserts lock-release operations on each edge that crosses from a protected node to an unprotected one.[2]

Figure 4 shows the result of applying this approach to the traps from Figure 3. AFix avoids the first trap by inserting a lock release operation for the else clause, and avoids the second one by moving the lock acquisition operation out of the loop. Handling the third trap requires additional analysis (Section 3.3).

AFix patches thus guarantee two important properties. First, the lock is not held at any unprotected nodes, because it is released whenever execution crosses from protected nodes to unprotected nodes. Second, the lock is held at every protected node, including $p$ and $c$, because it is acquired whenever execution crosses from unprotected nodes to protected nodes. These two properties help us answer the following questions regarding the suitability of the above critical region as part of a bug fix:

***Are $p$ and $c$ inside the critical region on all paths?*** Yes, they are. Given any path that goes from $p$ to $c$, every node on that path must be in the protected node set. The lock is always held throughout any execution along any such path.

[1] No $p$ or $c$ in between is because CTrigger is designed to report atomicity violations between two *consecutive* memory accesses to the same variable.

[2] In practice, to place code "on" an edge from $x$ to $y$, create a new node $n$ and replace the $x \to y$ edge with two edges: $x \to n$ and $n \to y$. Code intended to appear on the original $x \to y$ edge is placed in the newly-created node $n$.

***Can the added code introduce double-lock bugs?*** No, it cannot. AFix only acquires locks on an edge that crosses from an unprotected node to a protected node. Since the lock is not held at unprotected nodes, double-lock bugs never occur in AFix patches. By contrast, the naïve fix suggested in Section 2.3 would easily lead to double-lock problems, as shown in Figure 3b.

***Can the added code introduce double-unlock or unlock-before-lock bugs?*** No, it cannot. AFix only releases locks on an edge that crosses from a protected node to an unprotected node. Since the lock is held at protected nodes, double-unlock or unlock-before-lock bugs never occur in AFix patches. By contrast, the naïve fix suggested in Section 2.3 would easily lead to problems in scenarios like Figure 3b when the while-loop condition is initially false.

***Can the added code introduce new data races?*** No, it cannot. The patch does not introduce any new interleavings into the program. Possible thread schedules in the patched program must be a subset of those that were already possible in the original program.

***Can the added code introduce new deadlocks?*** This is a much tougher question to answer. In general, it is impractical to prove a C/C++ programs deadlock-free. We address this risk using a mixture of static analysis (Section 3.3) and dynamic monitoring (Section 5.1).

***Is this the best policy?*** There could be other policies to decide where to add locks and unlocks in order to protect $p$ and $c$. We have designed and implemented several other schemes. Some of them generate smaller critical regions than the current AFix does for some special control flow structures. We prefer to use the current AFix algorithm presented above, because it has a huge advantage in simplicity. It is far easier to reason about and has better composability in the interprocedural case than other schemes. The patches generated by it also always have good readability. Furthermore, our experiments never observe any excessive performance degradation caused by the current AFix algorithm.

### 3.3 Deadlock Analysis and Avoidance

AFix statically analyzes each critical region to determine whether it includes any potentially-blocking operations. These operations include lock-acquisitions, condition-wait operations, barriers, thread join operations, and some ad-hoc synchronizations like spin loops. The first few of these are easy to identify. To identify ad-hoc spin loops, AFix checks whether there is loop inside the critical region, and whether heap or global variables are accessed inside the loop. If any such loop exists, we conservatively assume that it might constitute an ad-hoc spin loop. This could be made more precise in the future [36, 42], but is sufficient for AFix's current needs.

If this analysis finds no potentially-blocking operations within the critical region, then there is no risk of deadlock and AFix uses standard pthread_mutex_lock calls to acquire locks. This is the case for AFix's handling of the first two traps as shown in Figures 4a and 4b. If potentially-blocking operations are found, then deadlock is a real risk. In this case, AFix instead acquires locks using pthread_mutex_timedlock: this will time out if it is unable to acquire the lock after some maximum delay. AFix's run-time system monitors each timed lock in the patch to identify when our attempted fix has actually introduced a circular wait. This information can help developers refine AFix patches. We discuss this further in Section 5.

Currently, we set the time-out delay to a relatively large value: ten seconds by default. The only disadvantage of long time-out limits is longer latency in discovering any deadlocks that do arise.

One subtle issue that is not covered above is recursion. As shown in Figure 3c, double lock/unlock and deadlocks could arise in the case of recursive calls. AFix identifies all such cases using a simple reachability analysis of the static call graph. When recursion is possible, we use reentrant (a.k.a. counted or nested) locks. In all other cases, we use non-reentrant locks as these can be faster. AFix implements reentrant locks by associating a mutex.count counter and a mutex.owner thread-ID with each reentrant lock mutex. mutex.owner records the thread-ID of the lock's current owner, and mutex.count records the current nesting level in the owner thread. AFix uses reentrant locks in all scenarios when an AFix-added critical region could be called by another AFix-added critical region using the same lock. Figure 4c shows how AFix handles the third trap using a reentrant lock.

```
void newlog()
{
  ...
  close();
  open();
  ...
}

void close() {
  ...
p: log = CLOSE;
}

void open() {
  ...
c: log = OPEN;
}

void insert() {
  ...
r: if (log == OPEN)
     logwrite(...);
  ...
}
```

**Figure 5.** Atomicity violation in MySQL. When $r$ executes between $p$ and $c$, the database log drops an "insert" log entry.

### 3.4 Multiple-Function Operation

AFix's task is more complicated when $p$ and $c$ come from different functions. This is not unusual; Figure 5 shows one example taken from MySQL.

We could use interprocedural analysis in computing the protected node set. For example, context-free language reachability can describe the interprocedurally-valid paths from $p$ to $c$ with only non–$p$-or-$c$ nodes in between. However, it is generally considered taboo to put matched lock and unlock operations in distinct functions due to the poor composability of locks. Examination of manually-designed patches shows that programmers usually add each atomic region's lock acquire and release statements inside the same function. We comply with this practice by adopting an intraprocedural approach that starts and ends each newly-added atomic region inside one function.

The extention to the basic algorithm is inspired by the manual approach a developer might take: study the calling contexts of $p$ and $c$, find a single common function through which both $p$ and $c$ are reached, and add the critical region to this function. For example, for the bug shown in Figure 5, function newlog is a suitable home for the $p$–$c$ critical region.

To carry out the above design, we modify the original CTrigger bug detection tool to output the complete call stack for each atomicity violation. AFix compares the call stacks of $p$ and $c$, and identifies the last (innermost) function $f$ on the common prefix of the two call-stack chains. If $p$ is not already directly in $f$, then we find the call node in $f$ that eventually leads to $p$ in the CTrigger bug report. We treat this call node as the $p$ node to be protected. Similarly, we replace $c$ with the call node in $f$ that eventually reaches $c$. After performing these substitutions, the new $p$ and $c$ must both be contained within $f$, and AFix proceeds as described in Section 3.2.

### 3.5 Harmonizing Two Critical Regions

The process described above yields a critical region protecting all paths from $p$ to $c$, guarded by some lock. A second critical region is created by acquiring and releasing the same lock immediately before and after $r$. However, this alone is not enough: some additional analysis is needed to harmonize these two critical regions so that they cooperate without introducing new bugs.

AFix first checks whether these two critical regions overlap. Let $f$ be the function containing both $p$ and $c$, possibly after performing the substitutions in Section 3.4. If $f$ does not appear on CTrigger's call stack for $r$, then the two regions do not overlap and no additional work is needed. If $r$ appears directly within $f$, then the two regions overlap if $r$ is actually in the set of protected nodes computed earlier. If $r$ is not directly in $f$ but $f$ does appear in the call stack leading to $r$, then identify the specific call node in $f$ that leads to $r$, and check whether this call node is in the protected set. Recursion can lead to multiple $f$ on the call chain. Handling this requires a minor extension that if any of the calls in $f$ which lead to $r$ are themselves inside the $p$–$c$ critical region, then the lock operations protecting $r$ are redundant and may be removed. Recall that AFix is working with each specific dynamic stack trace reported by AFix's front-end bug detector. Therefore, AFix knows the exact call instructions along the call chain, which makes the identification easy.

In all cases, the effect is to determine whether reaching $r$ in the stack configuration reported by CTrigger necessarily implies already being inside the $p$–$c$ critical region. If it does, then the lock operations around $r$ are redundant, and may simply be removed. Otherwise, the lock operations around $r$ are retained.

For example, consider the (line 6, line 8, line 8) atomicity violation depicted in Figure 1. A naïve patch will put lines 6–8 inside one critical region and line 8 inside another. AFix removes the redundant critical region around line 8.

Note that we do not guarantee that there is no other way to reach $r$; we only promise to protect $r$ when reached in the specific stack configuration reported by CTrigger. If other routes to $r$ avoid the $p$–$c$ critical region and thereby cause additional failures, we assume that additional bug reports from CTrigger will eventually mark these as needing fixes as well.

### 3.6 Lock Selection and Reuse

Lastly, AFix decides which lock to use. In our current implementation, when we only face one bug report, AFix simply creates a new global lock to use at the boundaries of the $p$–$c$ and $r$ critical regions.

One might consider reusing an existing lock. This is especially appealing if $p$ and $c$ are already inside a critical region in the original program. Doing this requires identifying some lock that is live and reachable at $r$ and at every edge entering or leaving the $p$–$c$ critical region. This is quite challenging in the general case of heap-allocated locks reached by traversing complex, linked data structures. If $r$ is not even in the same function as $p$ and $c$, and therefore has access to different local variables, the challenge is even greater.

Global locks, however, have fixed names. This makes them inviting targets for reuse. We have implemented a simple scheme that identifies global locks that protect the intended $p$–$c$ critical region in the same function. If some such global lock is identified, AFix uses it instead of creating a new lock, and elides inserting additional lock acquire/release statements. For example, when one $p$–$c$ critical region is already protected by a global lock, AFix does not insert additional lock acquire/release statements to protect it. In practice, though, we never find any reusable global locks for the bugs used in our experimental evaluation.

### 3.7 Implementation Details and Discussion

***Implementation Details*** AFix implements these static analyses in LLVM [17]. After analysis, AFix changes the target software's LLVM byte-code to apply the patch. In our current implementation, AFix uses locks provided by the POSIX threads (pthread) library to enforce mutual exclusion. A subtle implementation issue is that CTrigger describes $(p, c, r)$ triples in terms of executable instruction addresses. To aid patch readability, AFix maps each instruction back to source code lines. We never insert lock or unlock operations in the middle of a source code line, even if this requires slight expansion of critical regions.

```
p: array[1] = 2;
   for (i = 0; i < 2; i++) {
      ...
c:    array[i] = i;
      ...
   }
```

**(a)** Original code

```
   lock(L);
p: array[1] = 2;
   for (i = 0; i < 2; i++) {
      ...
c:    array[i] = i;
      unlock(L);
      ...
      lock(L);
   }
   unlock(L);
```

**(b)** AFix patched

**Figure 6.** Made-up example for the rare case when AFix may fail

***Assessing patch quality*** In terms of correctness, AFix guarantees not to introduce new bugs. In particular, AFix restricts possible interleavings, but never allows any interleaving that was not already possible before patching. Note that AFix patches could cause temporary circular wait, but thanks to timed locks these do not become deadlocks.

AFix can successfully fix bugs reported by CTrigger in all but two scenarios. One is that a lock inside an AFix patch may time out if AFix cannot statically prove deadlock-freedom, and atomicity is no longer guaranteed when the lock does time out. This case is captured by the AFix run time and feedback will be provided for further patch refinement. The other case is very rare. It occurs when $p$ or $c$ has more than one dynamic instance and may access different memory locations. In this situation, the $p$–$c$ pair that requires protection may not be the consecutive ones protected by AFix. This case is theoretically possible, but very rare in reality. Figure 6 shows one made-up example. In this example, $p$ is followed by two dynamic instances of $c$. The first instance accesses a different variable from $p$, while the second instance accesses the same variable as $p$. AFix puts $p$ and the first dynamic instance of $c$ into the critical region, as shown in Figure 6b, and the control flow could leave the protected region between the two instances of $c$. However, what should be in the critical region is $p$ and the second dynamic instance of $c$. In this extremely rare example, AFix patch will mistakenly end the critical region before the second instance of $c$.

In terms of performance, AFix patch strives to avoid introducing unnecessary performance degradation. AFix always ends a critical region immediately when there is no hope to reach $c$. Of course, for specific bugs, there could be faster patches by using lock-free data structures, reader-writer locks, etc. Crafting a general algorithm for using these special tricks is left for future work.

***Extending AFix*** AFix focuses on locks, because they are the most commonly supported and used synchronization primitives in multithreaded programs. AFix could be easily extended to other primitives that support atomicity. For example, if we use transactional memory, then the analysis to determine where to lock and unlock is suitable for deciding where to begin and end transactions. Of course, some detailed concerns will be different. For example, concerns over deadlocks and lock selection go away, while new concerns arise over live-lock and I/O inside transactions.

Although our algorithm description and current AFix implementation is for CTrigger, the AFix algorithm can be easily extended to work with other atomicity-violation bug detectors including those for multi-variable atomicity-violation bugs. Bug reports from many atomicity-violation detectors can be generalized as "code region $X$ needs to be mutually exclusive with code region $Y$." With this knowledge, a patch can be generated using the AFix algorithms

```
lock(L1)          lock(L1)          lock(L1)
  p1                p1                p1
lock(L2)          lock(L2)          lock(L2)
  p2                p2                p2
  c2                c1                c1
unlock(L2)        unlock(L1)        unlock(L1)
  c1                c2              lock(L1)
unlock(L1)        unlock(L2)          r1
                                    unlock(L1)
                                      c2
                                    unlock(L2)
```

**(a)** Eliminating redundant locks **(b)** Improving readability and performance **(c)** Avoiding deadlock

**Figure 7.** Scenarios where patches should or must be merged

discussed above: select one function to add lock/unlock operations, determine where inside that function to add lock/unlock operations, analyze the necessity of reentrant and timed locks, harmonize two critical regions, etc.

# 4. Fixing Multiple Bug Reports

Bug detectors often report multiple bugs that should be fixed by one patch, such as the two atomicity violations in Figure 1 and the scenarios depicted in Figure 7. This section describes how AFix coordinates multiple fixes in order to improve patch quality.

Provided with a set of bug reports, AFix first designs patches for each bug independently. Before applying these patches to the software, AFix considers all patches together. If one patch subsumes another, as shown in Figure 7a, the redundant patch is discarded. If two patches have overlapping critical regions, as shown in Figures 7b and 7c, AFix will further analyze how to merge the patches.

## 4.1 Removing Redundant Patches

AFix creates two critical regions for each bug triple $(p,c,r)$: one containing $p$ and $c$, and one containing $r$ alone. Strictly speaking, a patch is completely redundant if both of its critical regions can be subsumed by another patch. Sometimes, a patch's $p$–$c$ region is subsumed by another patch, but its $r$ region is not. Since the $r$ region is extremely short, AFix still chooses to discard the subsumed $p$–$c$ critical region in this situation. The lock used to protect $r$ will be changed to the lock used in the subsuming patch.

Under AFix's locking policy, a critical region $p$–$c$ is subsumed by another critical region $p'$–$c'$ if and only if the set of CFG nodes in $p$–$c$ critical region is a subset of those in $p'$–$c'$. The reasoning is clear. The lock for $p'$–$c'$ is held at every node inside its critical region. If all $p$–$c$ nodes are contained within $p'$–$c'$, then the $p$–$c$ lock is redundant.

The above conditions are straightforward to check for two critical regions in the same function intraprocedurally. For two regions in different functions, AFix extends the above subsuming conditions interprocedurally as follows. First, let $f$ denote the function containing critical region $p$–$c$, and let $f'$ denote the function containing critical region $p'$–$c'$. Let $n$ be the call node inside $f'$ that eventually leads to $f$ (and therefore to $p$–$c$), if such a node exists. Then $p'$–$c'$ subsumes $p$–$c$ if and only if

1. $f'$ is in the common prefix of $p$ and $c$'s call stacks, and
2. $n$ is inside the $p'$–$c'$ critical region.

Notice that both $f$ and $f'$ are in the common prefix of $p$ and $c$'s call stacks, and $f$ is chosen by our critical region identification algorithm, so we know that $f$ is closer to $p$ and $c$ than $f'$ on the common prefix of $p$ and $c$'s call stacks.

Since we have modified CTrigger to provide the complete call-stack information, the above analysis is not difficult. Note that the simplicity of our interprocedural subsumption analysis benefits from AFix's policy requiring that each critical region's locks and unlocks appear within the same function.

Overall, AFix compares each pair of patches and deletes the redundant ones.

## 4.2 Merging Related Patches

The general problem of patch merging can be described as follows. Given a set of critical regions, some guarded by the same lock and some by different locks, how can we adjust the lock variables and the critical-region boundaries to achieve the best balance of performance, readability, and correctness?

This question rarely has a provably-optimal answer, because performance is affected by many factors. These include but are not limited to the length of each critical region, the cost of acquiring and releasing a lock, how many threads will execute each critical region, and how much contention each lock-acquisition will face. Much of this cannot be decided statically.

Facing this challenge, AFix uses a simple heuristic to decide when and how to merge critical regions. We should note that although we believe this scheme can achieve a good balance between performance, readability, correctness, and analysis simplicity, there is no firm guarantee of performance improvement going from non-merged to merged patches. Developers can make informed decisions based on AFix's run-time performance profiling results. Different merging policies can also be plugged into AFix in the future.

### 4.2.1 When to Merge

AFix merges patches when one patch's lock-protected critical regions include the $p$, $c$, or $r$ nodes of another patch. More formally, consider two sets of nodes for any given patch $i$. Let $Anchors_i$ represent the set of all nodes "anchoring" this patch. Initially this consists of just $\{p_i, c_i, r_i\}$: the set of nodes representing the original bug for which this patch was created. Let $Critical_i$ represent the set of all nodes contained in critical regions guarded by patch $i$'s lock. Initially this will consist of $r_i$ (as a single-node critical region) along with $p_i$, $c_i$, and any other protected nodes along $p_i$–$c_i$ paths as computed in Section 3.2.

For any two patches $i, j$, if $Anchors_i \cap Critical_j \neq \emptyset$, then patches $i$ and $j$ should be merged. In other words, if the critical regions of one patch include any of the anchor nodes of another, then the two patches should be combined into one. Let the *Anchors* and *Critical* sets of this merged patch be the unions of the corresponding sets for the patches being merged. A single lock will guard all nodes in the combined *Critical* set. This merged patch is available for further merging; the process continues until the original patch collection has been collapsed into a (possibly) smaller one in which no patch's $p$, $c$, or $r$ nodes are contained in the lock-guarded critical regions of any other.

### 4.2.2 Merging Two Critical Regions

Once AFix decides to merge, it enacts the merge in two steps:

1. Update the positions of lock and unlock operations. AFix puts an unlock operation on every edge that exits the merged *Critical* set, and a lock operation at every edge that enters the *Critical* set. This has the effect of removing all redundant lock and unlock operations among merged patches.
2. Unite lock variables. AFix arbitrarily chooses one lock variable to use and puts this variable into every lock and unlock operation performed by the merged patch.

For example, AFix merges the two critical regions in Figure 7b, and deletes **unlock**(L1) and **lock**(L2). Similarly, AFix also merges

the three critical regions in Figure 7c, and discards all but the first **lock**(L1) and the last **unlock**(L2), thereby eliminating the potential deadlock. For the bug shown in Figure 1, the final merged patch has just one lock operation inserted before line 2, and two unlock operations inserted before line 4 and after line 8.

The above merging process is only used for critical regions inside the same function. Actually, it covers the intraprocedural case of redundant patch removal from Section 4.1: if a patch is subsumed by another patch in the same function, they will be merged based on AFix's merging policy. Currently, AFix first conducts interprocedural redundant-patch removal and then conducts intraprocedural patch merging. If some other merging policy is adopted in the future, intraprocedural redundant-patch removal may still be needed.

#### 4.2.3 Benefits of Merging

Merging patches as described above has several beneficial effects on patch quality:

- Code readability is improved. In practice, it is common that many atomicity violations are reported within few lines of code, such as the Apache case shown in Figure 1. Using many different locks severely hurts code maintenance and readability.
- Performance can usually improve due to fewer lock and unlock operations without enlarging the critical region too much, such as in Figure 7. Of course, there is no guarantee of performance improvement, because merging can also reduce potential concurrency in certain scenarios.
- Correctness is either the same or improved, because we have larger critical regions now. In fact, Section 6 reports that this helps AFix lower the failure rates of some real-world software bugs that were inaccurately reported by CTrigger.
- Deadlock risk is reduced. It is easy to have one patch deadlock with another patch, as shown in Figure 7c. Merging solves this problem. Under AFix's merging policy, holding one AFix lock and trying to acquire another AFix lock is impossible, because these two locks would have been merged.

## 5. Run-Time Monitoring and Feedback

Not all properties can be guaranteed statically. AFix collects additional information at run time to help developers refine patches.

### 5.1 Deadlock and Performance Monitoring

AFix uses time-outs for lock acquisitions that cannot be guaranteed to be deadlock-free as discussed in Section 3.3. Therefore, deadlocks caused by AFix patches manifest as lock time-outs. Of course, a time-out could also occur without deadlock: a lock may simply encounter too much contention and require a longer waiting period.

AFix implements two run-time deadlock-detection algorithms, through LLVM byte-code rewriting, suitable for different usage scenarios. The first, suitable for in-house patch testing, reports whether a deadlock has occurred immediately after a time-out. It has small overhead at each lock/unlock operation. The second deadlock detector, suitable for production-run deployment, has nearly zero overhead if there is no AFix lock time-out. It takes a little bit longer to complete deadlock diagnosis.

In the first scheme, AFix follows the traditional deadlock-detection algorithm: it maintains a resource graph and looks for cycles when an AFix-added lock times out. To maintain a resource graph, AFix monitors every lock acquisition, lock release, and condition-variable signal and wait, all from the beginning of execution. Its overhead, then, depends on the density of these operations.

In the second scheme, AFix starts its monitoring and analysis only after an AFix lock times out, a moment that we will refer to as $T$. AFix uses information collected after $T$ to recover the resource graph at the moment of $T$, as follows. When some thread $t$ releases a lock $l$, AFix checks whether it also saw $t$ return from a **lock**($l$) call. If not, then $l$ must have been acquired by $t$ sometime before $T$, and therefore was held by $t$ at the moment of $T$. Right after a lock $l$ is acquired by thread $t$, AFix checks whether it has observed $t$ begin a **lock**($l$) call. If not, then $t$ must already have been waiting for $l$ at the moment of $T$. Eventually, AFix will recover the whole resource graph and report whether there was a deadlock. This post–time-out monitoring ends when either a deadlock is identified or the program exits. If the program encounters another AFix lock time-out, AFix will work on recovering multiple resource graphs at the same time. Since AFix lock time-outs are rare after in-house patch testing, this scheme is well-suited to monitoring production runs.

Of course, AFix's run-time system could miss a deadlock if the deadlock involves ad-hoc synchronization, such as spin loops. This is an open problem for all deadlock detection tools.

Currently, AFix relies on developers to refine the patch using its deadlock-detection results. When a time-out is diagnosed as a non-deadlock, developers may want to extend the time-out threshold in related AFix locks. When the time-out is caused by deadlock, developers could discard this patch and choose another, such as a patch that has some critical regions merged. AFix's run-time can also be combined with previous deadlock-prevention systems [13] to automatically fix patch-induced deadlocks in the future.

Other than deadlock monitoring, AFix also supports performance profiling during in-house testing. In profiling mode, AFix measures the waiting time and the number of time-outs for each lock acquisition inside an AFix patch. If excessive waiting time or time-outs are observed at a critical region that merges multiple bug reports' patches, developers may want to split this big critical region to reduce lock contention and improve performance.

### 5.2 Patch Testing

Each patch generated by AFix undergoes two testing phases. The first phase uses the existing CTrigger testing. CTrigger provides a noise injection scheme for each bug it reports. Through a binary instrumentation framework [22], CTrigger deterministically calls sleep before or after specific instructions: before $c$, after $p$, etc. We apply CTrigger testing to AFix-patched software and see whether software failures could still occur.

The second patch testing phase is a more general interleaving test implemented by us. Before executing every instruction inside an AFix critical region, a random number generator decides whether to sleep or not. Similar random delays are also inserted right before the locks and right after the unlocks added by AFix. The sleep probability and the length of the sleep are tunable knobs. This phase can help identify patches that fail to completely fix the bug due to bug-detection limitations of CTrigger.

## 6. Experimental Results

AFix is implemented using LLVM version 2.7. Experiments are performed on an eight-core Intel Xeon machine running Red Hat Linux 5 with kernel version 2.6.18.

We evaluated AFix on eight real-world bugs from six open-source applications. These bugs were all initially reported by software users to each application's bug database or mailing list. We apply CTrigger to these applications using the bug-triggering inputs described in the user bug reports. In each case, CTrigger detects one or more $(p, c, r)$ triples. We have confirmed that atomicity violations described by each triple lead to failure symptoms matching users' descriptions. Table 1 gives additional information about these bugs, applications, and CTrigger detection results.

For each bug listed in Table 1, AFix generates two versions of patched software: one with the patch-merging technique presented in Section 4 applied and one without. We refer to the former as

| Bug ID | Application | LoC | Developer Fix Time | # CTrigger Reports |
|---|---|---|---|---|
| FFT | FFT | 1.2K | N/A | 5 |
| PBZIP2 | PBZIP2 | 2.0K | N/A | 4 |
| Apache | Apache | 333K | 30 days | 2 |
| MySQL1 | MySQL v4.0.12 | 681K | 10 days | 1 |
| MySQL2 | MySQL v4.0.19 | 693K | 13 days* | 2 |
| Mozilla1 | Mozilla-JS v1.4.2 | 87K | 12 days* | 2 |
| Mozilla2 | Mozilla-JS v1.5 | 108K | $> 3$ days$^\sharp$ | 1 |
| Cherokee | Cherokee v0.9.2 | 83K | $> 1$ day$^\sharp$ | 4 |

**Table 1.** Bugs used in experimental evaluation. Developer fix time is the time between developers' first response to a bug report and a correct patch checked in, if known. Mozilla-JS is the JavaScript Engine of Mozilla. *: incorrect patches were submitted during this period. $\sharp$: the bug was reported by developers who suggested a fixing strategy in the initial bug report.

| Bug ID | naïve | unmerged | merged | manual |
|---|---|---|---|---|
| FFT | - | - | ? | ✓ |
| PBZIP2 | - | - | - | ✓ |
| Apache | - | - | ✓ | ✓ |
| MySQL1 | - | ✓ | ✓ | ✓ |
| MySQL2 | - | ✓ | ✓ | ? |
| Mozilla1 | - | ✓ | ✓ | ? |
| Mozilla2 | - | ✓ | ✓ | ✓ |
| Cherokee | - | ✓ | ✓ | ✓ |

**Table 2.** Overall patch quality

the *merged* version and the latter as the *unmerged* version. We also compare AFix with two other versions of patched software: (1) the patch manually generated by developers, referred to as *manual*, and (2) the naïve patch described in Section 2.3, referred to as *naïve*. Our experiments also compare the above patched versions with the original buggy software, referred to as *original*.

Our evaluation considers three aspects of patch quality:

**Correctness.** We use CTrigger testing and intensive random noise injection to check whether the bug has been fixed and whether new bugs are introduced. We report and compare the failure rates of different versions of software. We also use the AFix run time to check whether timed-out locks actually represent deadlocks.

**Performance.** We measure the performance of different versions of patched or unpatched software. We also measure how long it takes AFix to perform its static analysis and patch insertion.

**Code readability.** We manually compare AFix patches with the *manual* patch. We present results through case studies.

### 6.1 Overall Results

Table 2 presents a compact summary of patch quality. "✓" indicates that the original bug is fixed, no new bug is observed, and performance degradation is negligible. "?" indicates that the patch is incomplete, but decreases the failure rate without hurting performance. "-" marks cases where the fix introduces new bugs, does not significantly reduce the failure rate, or imposes an intolerable performance deficiency. For manual patches, "?" means developers submitted intermediate patches that are later determined to be incomplete by developers or testing groups (i.e., the original software failure can still occur with the patch applied). "✓" means that the first developer-submitted patch is complete. For the bugs in our study, developers submitted no patch that could introduce new bugs or intolerable performance deficiencies. We obtained the above information from corresponding Bugzilla records.

| Bug ID | original | naïve | unmerged | merged |
|---|---|---|---|---|
| FFT | 74% | 73% | 87% | 30% |
| PBZIP2 | 94% | 100%* | 66% | 20% |
| Apache | 85% | 100%* | 83% | 0% |
| MySQL1 | 41% | 100%* | 0% | 0% |
| MySQL2 | 53% | 100%* | 0% | 0% |
| Mozilla1 | 41% | 100%* | 0% | 0% |
| Mozilla2 | 48% | 100%* | 0% | 0% |
| Cherokee | 81% | 100%* | 0% | 0% |

**Table 3.** Failure rates under interleaving testing. "100%*" marks cases where the test input deterministically causes deadlock.

Patches generated with merging are highly competitive with manually-generated patches. AFix successfully fixes six out of eight bugs. In two cases, MySQL2 and Mozilla1, merged patches are even better than the first few patches generated by developers. For FFT and PBZIP2, AFix is limited by CTrigger's inaccurate root-cause identification. Even here, the merged patches reduce (but cannot entirely eliminate) failures.

Unmerged patches fix five out of eight bugs. Deadlock prevents this from fixing the Apache bug that merging does correctly fix. The naïve approach fixes no bug. It causes deadlock in all but FFT, and fails to fix FFT due to CTrigger inaccuracy.

### 6.2 Correctness Results

Table 3 shows the failure rates of different versions of software under random noise-injection testing (Section 5.2). We set up exactly the same noise injection environment for all versions of software for each bug: the same program region to inject random noise, which is around the buggy code region; the same sleep probability; and the same sleep length. We use slightly different sleep probabilities and sleep lengths for different bugs, because we want to make sure the testing is intensive enough to make the original unpatched software fail frequently. This lets us effectively evaluate whether the patches are useful. We execute each version of software 100 times.

Merged patches eliminate software failures for six out of eight bugs in our testing. Merging provides incomplete patches for FFT and PBZIP2, but drops failure rates from 74% to 30% in FFT and from 94% to 20% in PBZIP2. The failure rates do not drop to 0% because CTrigger's bug detection is inaccurate: atomicity violation is a side effect but not the root-cause of these two bugs. Specifically, following a CTrigger report $(p,c,r)$, merging correctly ensures that $r$ does not execute between $p$ and $c$. However, the real problem in FFT is that $r$ should not execute after either $p$ or $c$, while the problem in PBZIP2 is that $r$ should execute after both $p$ and $c$. Merging can only partially fix these two problems.

Unmerged patching behaves slightly worse than merged, eliminating failures for five out of eight bugs. It leads to non-deterministic deadlocks in Apache and PBZIP2 due to the reason depicted in Figure 7c. Once the lock times out after a deadlock, atomicity violation and subsequent failure can still occur. For FFT, unmerged patching has a much larger failure rate than merged due to its smaller critical-region size. In FFT, the bug occurs when an instruction $r$ executes after any one of a set of six instructions. Merging puts all six into one critical region, making the bug less likely to occur.

Naïve patching is clearly a very bad choice, leading to deadlock in seven out of eight bugs. There are several different reasons for these deadlocks. For MySQL2 and PBZIP2, deadlocks are caused due to intraprocedural control flows, as depicted in Figure 3. For MySQL1 and Mozilla1, $p$ and $c$ are inside different functions. Locking in one function and unlocking in another easily causes

| Bug ID | naïve | unmerged | merged | manual |
|---|---|---|---|---|
| FFT | -0.02% | -0.07% | -0.02% | 0.19% |
| PBZIP2 | N/A | 89,132% | 181.82% | 0.20% |
| Apache | N/A | 0.45% | -0.97% | -0.26% |
| MySQL1 | N/A | 0.48% | 0.48% | 0.45% |
| MySQL2 | N/A | -0.09% | -0.09% | 1.02% |
| Mozilla1 | N/A | 0.49% | 0.55% | 0.12% |
| Mozilla2 | N/A | -0.40% | -0.40% | -0.20% |
| Cherokee | N/A | -1.02% | -1.04% | 0.39% |

**Table 4.** Performance overheads relative to original

deadlocks when the first function is called twice without the second function in between. In Apache, Cherokee, and Mozilla2, naïve double-lock bugs arise because *r* is inside a *p–c* region. Apache and PBZIP2 also deadlock among different locks added by the patch.

We also reapplied CTrigger to the patched code. According to CTrigger's definitions, both merged and unmerged patches successfully fix all eight bugs. We also manually checked all these patches. Our findings are consistent with the random testing results. In those cases with 0% failure rates, the bugs are all truly fixed.

Overall, merging generates correct patches that not only fix the original bugs but also introduce no new bugs, as long as its front-end bug detector provides a reasonably-accurate bug report. Unmerged patching is also good, but is vulnerable to deadlock.

### 6.3 Performance Results

***Patched application performance*** Table 4 shows that merged and unmerged AFix fixes provide good run-time performance, with negligible difference between them. Note that overheads cannot be measured for most naïve patches as these lead to deterministic deadlocks. In most cases, AFix patches impose no perceivable performance degradation compared with correct manual patches or even the original buggy software. This is because the relevant critical regions are usually small and off performance-critical paths.

Only PBZIP2 suffers from significant performance degradation under AFix. The PBZIP2 bug is caused by a parent thread occasionally destroying shared objects before worker threads finish. The manual patch makes the main thread wait until all worker threads are done, which is correct and lightweight. Due to CTrigger inaccuracy, AFix mistakenly treats this bug as an atomicity-violation bug, and fixes it by putting almost the whole worker thread into a critical region, which causes huge overhead. The unmerged AFix patch is much slower than the merged patch for PBZIP2, because the former suffers from deadlock time-outs. Deadlock time-outs can also occur in the unmerged patch of Apache, but this never happens during performance evaluation without noise injection.

***AFix patch generation performance*** All of AFix's static analyses have been designed with scalability in mind. On every benchmark, AFix takes no more than one second to analyze the program, develop its patches, and inject them into the code. Clearly, AFix has the potential to significantly speed up the bug-fixing process.

### 6.4 Readability Case Studies

AFix's patch merging technique improves patch readability and maintainability. For six out of eight bugs, CTrigger reports two to five atomicity violations related to each bug. The unmerged fix therefore adds two to five new locks and up to ten new critical regions into the software. Merging simplifies this to use just one lock in five out of six cases. Manual inspection shows that all merging decisions made by AFix improve readability.

Taking the Cherokee bug as an example, CTrigger reports four atomicity violations in three different functions. The unmerged fix therefore adds four global lock variables, six lock operations, and seven unlock operations into the software. AFix finds some of these critical regions to be redundant and some to be mergeable. The final patch requires only one lock, one lock operation, and one unlock operation, for excellent code readability and maintainability. In fact, the merged patch closely resembles the manual patch.

### 6.5 Other Results

AFix's run-time deadlock detection gives accurate results for both merged and unmerged patches. For example, in PBZIP2, both merged and unmerged strategies encounter lock time-outs. The AFix run-time system correctly determines that the time-outs in the unmerged patch are caused by deadlocks among patches, but that the time-outs in the merged patch are not. Rather, merging simply has produced large critical regions with much lock contention, and therefore requires longer time-outs.

AFix's post–time-out deadlock-detection algorithm exhibits excellent performance on all bugs. Its overhead is already included in the performance numbers measured in Table 4. AFix's always-on deadlock detection has less than 5% overhead for all bugs except for Mozilla1 and Mozilla2, where the always-on monitoring causes 300% and 97% overhead respectively. This is due to the frequent lock/unlock operations in Mozilla's JavaScript Engine.

## 7. Related Work

### 7.1 Concurrency Bug Detection

Many detection tools have been built to identify interleaving problems in multithreaded programs, such as races [4, 6, 10, 12, 33] and atomicity violations [9, 12, 19, 38]. These tools provide good starting points for automated bug fixing.

Of course, since these tools are designed to identify problems, not to fix problems, they still leave many challenges for bug fixing. For example, many bug detection tools have false positives. "Fixing" false positives leads to over-synchronization and unnecessary performance degradation. Many bug reports do not include complete run-time information, which could cause wrong or incomplete patches. Bug detectors seldom put bug reports that can be fixed by one patch together, which can cause too many locks to be added, harming code maintainability. Furthermore, naïve patches for accurately-described bug reports can easily introduce new bugs.

AFix has considered and addressed the challenges above. In the future, AFix can be extended to work with more concurrency-bug detectors to fix more bugs, which will also help bug-detection tools to get more usage during software development.

### 7.2 Concurrent Program Synthesis

Existing tools that automatically add synchronizations to software mostly have different goals from AFix, and hence different foci. Program synthesis and sketching [8, 35, 39] use smart state-space search and verification techniques to infer synchronization and make concurrent programs satisfy certain specifications. They are powerful in the sense that the specification can be flexible. Unfortunately, the nature of the problem makes them hard to scale to large real-world C/C++ applications such as the ones fixed by AFix.

Some tools [23, 38] encourage programmers to represent their synchronization intentions in non-lock language constructs, such as atomic sets and atomic blocks, and transparently translate these non-lock constructs to lock/unlock operations. In these cases, the critical region boundaries are either directly specified by developers or fixed at the entrances and exits of certain functions; the major challenge is lock assignment. AFix, by contrast, derives critical region boundaries with limited or no human intervention.

TraceFinder [37] performs whole-program synchronization analysis and pointer-alias analysis to identify atomic-block boundaries

that can guarantee conflict-serializability for the whole software. TraceFinder has a different goal from AFix. It cannot scale to large applications. It uses atomic blocks for synchronization, and does not worry about lock assignment or deadlocks.

AFix is unique in fixing bugs reported by automatic bug-detectors. AFix does not face the scalability problems encountered by TraceFinder, because it does not try to figure out all the synchronizations a program needs to use. Rather, AFix faces different challenges.

Finally, AFix is also related to a recently proposed compiler, QuickStep [25]. QuickStep is designed to generate parallel programs that satisfy statistical accuracy guarantees. When presented with a loop parallelization with unacceptable accuracy, QuickStep first identifies data races across loop iterations, then eliminates these data races by replacing unsynchronized versions of methods containing these races with synchronized versions. AFix can potentially complement QuickStep by generating synchronized implementations at finer granularity within each method.

### 7.3 Hot-Patching Concurrency Bugs at Run Time

Some proposed strategies for hot-patching software at run time are not suitable for concurrency bugs [29]. Recent work by Wu et al. [41] provides a framework to deploy hot patches manually designed by developers. AFix can complement this framework by generating patches automatically, instead of completely relying on developers.

Some run-time tools do not try to permanently fix concurrency bugs in the software. Rather, they steer the execution to make failure less likely. Some pay the cost of performance degradation or require non-existing hardware support [21]. Others assume that critical-region boundaries are known [16, 31]. Deterministic execution systems [1–3] can make some concurrency bugs deterministically happen and some other bugs never occur. This promising approach still faces many challenges, such as run-time overhead, integration with system non-determinism, language design, etc. Even for software executed inside a deterministic run-time, fixing bugs still requires manual intervention. In general, these tools look at different problems from AFix. AFix generates patches that can completely fix bugs without unnecessary performance degradation. AFix and these tools can complement each other.

Kivati [5] combines run-time bug detection with temporary patch generation based on hardware watch-points. Its focus is to lower bug detection overheads. However, the limited watch-point resource (four per machine) prevents Kivati from managing many different critical regions at the same time. Kivati does not handle cases where $p$ and $c$ are inside different functions, such as the Cherokee and MySQL1 bugs in Section 6. It causes unnecessarily long critical regions when there are branches between $p$ and $c$, as shown in Figure 3a. It also does not handle critical-region merging or deadlocks. Overall, Kivati lacks the off-line static analyses needed to generate high-quality, permanent fixes.

AtomRace [16, 18] combines run-time bug detection with run-time healing, with emphasis on bug detection. Its healer component assumes that critical-region entrances and exits are all provided as inputs. They do not address those challenges faced by the naïve patches discussed in our paper, and do not consider patch merging.

## 8. Conclusion

We have described AFix, a framework for automatically fixing a common type of concurrency bugs. We have implemented the system and shown AFix to be effective at generating high-quality patches for atomicity-violation bugs detected by an automated bug finder in several large, real-world applications. AFix conducts thorough static analysis to reach a good balance among correctness, performance, and code readability in its automatically generated patches. AFix's testing and monitoring run-time system also provide useful feedback for further patch refinement. In the future we plan to extend AFix to work with more general synchronization primitives and more types of concurrency bug detectors.

## References


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

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

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

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

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

[6] 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.

[7] C. Cowan, H. Hinton, C. Pu, and J. Walpole. The cracker patch choice: An analysis of post hoc security techniques. In *In Proceedings of the National Information Systems Security Conference (NISSC)*, 2000.

[8] J. Deshmukh, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Logical concurrency control from sequential proofs. In *European Symposium on Programming*, 2010.

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

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

[11] M. Harman. Automated patching techniques: the fix is in: technical perspective. *Commun. ACM*, 53(5):108–108, 2010. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1735223.1735248.

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

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

[14] E. Kandrot and B. Eich. Our JavaScript is 3x slower than IE's, Sept. 2000. URL https://bugzilla.mozilla.org/show_bug.cgi?id=54743.

[15] B. Krebs. A time to patch II: Mozilla. *The Washington Post* Security Fix blog, Feb. 2006. URL http://voices.washingtonpost.com/securityfix/2006/02/a_time_to_patch_ii_mozilla.html.

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

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

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

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

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

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

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

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

[24] Microsoft. Revamping the microsoft security bulletin release process, Feb. 2005. URL http://www.microsoft.com/technet/security/bulletin/revsbwp.mspx.

[25] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, 2010. URL http://hdl.handle.net/1721.1/57475.

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

[27] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *ICSE*, 2010.

[28] R. Pegoraro. Apple updates Leopard–again. Washington Post, Feb. 2008.

[29] 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.

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

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

[32] E. Rescorla. Security holes ... who cares? In *USENIX Security Conference*, 2003.

[33] 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.

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

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

[36] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, 2008.

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

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

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

[40] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*, 2010.

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

[42] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI*, 2010.

[43] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys*, 2010.