# Understanding and Generating High Quality Patches for Concurrency Bugs

Haopeng Liu Yuxi Chen Shan Lu
University of Chicago, USA
{haopliu, chenyuxi, shanlu}@cs.uchicago.edu

## ABSTRACT

Concurrency bugs are time-consuming to fix correctly by developers and a severe threat to software reliability. Although many auto-fixing techniques have been proposed recently for concurrency bugs, there is still a big gap between the quality of automatically generated patches and manually designed ones. This paper first conducts an in-depth study of manual patches for 77 real-world concurrency bugs, which provides both assessments for existing techniques and actionable suggestions for future research. Guided by this study, a new tool HFix is designed. It can automatically generate patches, which have matching quality as manual patches, for many concurrency bugs.

## CCS Concepts

•**General and reference** → *Reliability;* •**Computer systems organization** → *Reliability; Availability;* •**Software and its engineering** → *Software maintenance tools;*

## Keywords

concurrency bugs, automated bug fixing, multi-threaded software, empirical study

## 1. INTRODUCTION

### 1.1 Motivation

Concurrency bugs are caused by synchronization problems in multi-threaded software. They have caused real-world disasters [20, 35] in the past, and are a severe threat to software reliability with the pervasive use of multi-threaded software. The unique non-determinism nature has made them difficult to avoid, detect, diagnose, and fix by developers. Previous studies of open-source software [27] have shown that it often takes developers several months to correctly fix concurrency bugs. Furthermore, concurrency bugs are the most difficult to fix correctly among common bug types [40], with many incorrect patches released. Consequently, automated tools that help fix real-world concurrency bugs are well desired.

```
  //child thread                    //parent thread
  if (...) {                    + thread_join(...);
    unlock(fifo->mut);//A         if (...) {
    return;                          fifo->mut=NULL;//B1
  }                               }
                                  fifo=NULL;//B2
```

(a) Manual and HFix patch

```
  //child thread
  if (...) {                      //parent thread
    unlock(fifo->mut);//A       + lock(L);
+   lock(L);                    +   while(...){
+   signal(con);                +     wait(con, L);
+   unlock(L);                  +   }
    return;                     + unlock(L);
  }                               if (...) {
+ lock(L);                          fifo->mut=NULL;//B1
+ signal(con);                    }
+ unlock(L);                     fifo=NULL;//B2
```

(b) Simplified CFix patch

**Figure 1: A simplified concurrency bug in PBZIP2.**

Recently, many automated fixing techniques have been proposed. Some are for general bugs [7, 16, 17, 18, 19, 25], and some are for specific type of bugs [8, 10, 33]. Particularly, several tools dedicated to concurrency bugs have been proposed [5, 13, 15, 22, 23, 24, 38].

Existing concurrency-bug fixing tools can handle all common types of concurrency bugs leveraging a unique property of concurrency bugs — since concurrency bugs manifest non-deterministically, the correct computation semantics already exist in software. Consequently, these tools work not by changing computation semantics, which is required for most non-concurrency-bug fixing, but by adding constraints to software timing. They mostly achieve this by adding synchronization operations, including locks [13, 22, 24, 38] and condition variable signal/waits [15], into software.

Figure 1a illustrates a simplified version of a real-world bug in PBZIP2, a parallel file-compression software. Here, the lack of synchronization causes the parent thread to occasionally nullify objects ($B_1$ and $B_2$ in Figure 1) that are still used by the child thread ($A$ in figure), making PBZIP2 crash. An existing tool, CFix [15], can automatically fix this bug by adding condition variable signals and waits, as illustrated by the '+' lines in Figure 1b[1].

Although great progress has been made, patches generated by existing tools are mostly *different* from patches manually designed by developers. Existing auto-patches mostly insert

locks and lock-related synchronization operations into software [13, 22, 24, 38]; yet, less than one third of real-world concurrency bugs are fixed by developers through adding or changing lock operations [27].

The state-of-the-art auto-patches often lack simplicity comparing with manual patches, which we will discuss in details in Section 2. For example, developers simply fix the PBZIP2 bug by adding the missing thread-join operation, as shown in the Figure 1a. Instead, automatically generated patches are much more complicated, adding five sets of condition-variable signal/wait operations, as well as corresponding lock/unlock operations, and counter/flag maintenance and checking operations [15][1].

Clearly, we need a better understanding of the gap between automatically generated patches and manually generated patches, so that we can eventually design auto-fixing tools that generate not only correct but also simple and well-performing patches, appealing to developers.

## 1.2 Contributions

In this paper, we first conduct an in-depth study of manual patches for real-world concurrency bugs. Guided by this study, we then build a new bug fixing tool HFix that can automatically generate patches with matching quality as manual patches for many concurrency bugs.

**Study manual patches** We have checked the manual patches of 77 real-world concurrency bugs collected from a set of open-source C/C++ multi-threaded software. It tells us what synchronization primitives and fix strategies are used by developers to fix concurrency bugs. We list a few findings below, with more details in Section 2.

1. Lock is indeed the dominant synchronization primitive for enforcing mutual exclusion (atomicity); condition variable signals and waits are **not** the dominant primitive for enforcing pairwise ordering in patches.

2. Although adding extra synchronization operations is indeed a common fix strategy, leveraging existing synchronization in software is as common. Unfortunately, the latter has **not** been explored by previous work that fixes concurrency bugs in large-scale software.

3. More than 40% of the studied bugs are **not** fixed by constraining the timing of program execution. In fact, about 30% of bugs are fixed by changing computation, not synchronization, semantics in a thread, **deviating** from the fundamental assumptions taken by existing concurrency-bug fixing tools. These patches are not ad-hoc. They follow certain patterns and are promising to get automatically generated in the future.

Our findings provide assessment for existing techniques, both justifying their directions and identifying their limitations, and provide actionable suggestions for future research.

**HFix** Guided by the above study, we design HFix that can fix many real-world concurrency bugs in large software by two strategies that have not been well explored before.

The first, $\text{HFix}_{\text{join}}$, enforces ordering relationship by adding thread-join operations, instead of signal/waits (Section 4).

[1]Figure1b does not show all signals and waits in auto-patches; it also omits counter/flag operations associated with each signal or wait operation for illustration purpose.

**Table 1: Applications and bugs in study**

| App. | Description | # Bugs AV | # Bugs OV |
|---|---|---|---|
| **Ap**ache | Web Server | 8 | 5 |
| **Mo**zilla | Browser Suite | 26 | 15 |
| **My**SQL | Database Server | 10 | 2 |
| **Op**enOffice | Office Suite | 3 | 2 |
| **Misc.** | Cherokee web server, Click router, FFT benchmark, PBZIP2 compressor, Transmission bittorrent client, and X264 encoder | 1 | 5 |
| **Total** | | 48 | 29 |

$\text{HFix}_{\text{join}}$ can produce simple patches just like the manual patch shown in Figure 1a.

The second, $\text{HFix}_{\text{move}}$, enforces ordering and atomicity relationship by leveraging synchronization operations that already exist in software (Section 5).

HFix works for a wide variety of concurrency bugs. Evaluation using real-world bugs shows that it can indeed automatically generate patches that have matching quality with manual patches and are much simpler than those generated by previous state of the art technique (Section 7).

# 2. STUDYING MANUAL PATCHES

Our study aims to answer three sets of questions.

**What are manual patches like?** What are the fix strategies and synchronization primitives used by manual patches? Are all concurrency bugs fixed by constraining the timing? Does any patch change sequential semantics? How do patches vary for different types of concurrency bugs?

**How are existing techniques?** How do existing tools work, particularly compared with patches manually developed by developers?

**How about the future?** How might future tools generate patches that match the quality of manual patches?

## 2.1 Patch Study Methodology

To answer the above questions, we review the manual patches of 77 bugs. These bugs come from two sources. The first is the real-world concurrency-bug benchmark suite created by previous work [27]. Among the 74 non-deadlock bugs in that suite[2], a couple of them are not completely fixed by developers and hence are excluded from our study. The remaining 71 are shown in the top half of Table 1. The second part includes all the bugs evaluated by a few recent concurrency bug detection and fixing papers [13, 15, 41] that have available manual patches and are not included in the first source, shown in the "Misc." row of Table 1.

These bugs come from a broad set of C/C++ open-source applications, that include both big server applications (e.g., MySQL database server and Apache HTTPD web server) and client applications (e.g., Mozilla web-browser suite, and many others). These applications are all widely used, with a long software development history.

Among these bugs, 48 of them are atomicity violation (**AV**) bugs and 29 of them are order violation (**OV**) bugs.

[2]Our study focuses on non-deadlock concurrency bugs.

**Table 2: Synchronization primitives in patches**

| | Lock | Con.Var. | Create | Join | Misc. | None |
|---|---|---|---|---|---|---|
| **AV** | 18 | 1 | 0 | 0 | 2 | 27 |
| **OV** | 4 | 3 | 6 | 4 | 5 | 7 |
| **Total** | 22 | 4 | 6 | 4 | 7 | 34 |

Atomicity violations occur when the atomicity of a code region in one thread gets violated by interleaving accesses from another thread(s). Previous research [15, 28, 30] often denotes an atomicity violation by a *p*-*c*-*r* triplet, where $p$ and $c$ represent two operations forming the expect-to-be-atomic code region in one thread and $r$ represents the interleaving operation from another thread. We will use these symbols when discussing AV bugs.

Order violations occur when an operation $B$ unexpectedly execute before, instead of after, an operation $A$ (e.g., the bug shown in Figure 1). Previous research [15] uses $AB$ to represent an order violation bug, and we will also use these symbols when discussing OV bugs.

We carefully study the final patch of each bug. We also read developers' discussion on the on-line forums [1, 2, 3, 4] and software source code to obtain a deep understanding of each patch. Every bug was reviewed by all authors, with the patch categorization cross-checked by all authors.

**Threats to Validity** Like all empirical studies, our study cannot cover all concurrency bugs. Our study only looks at C/C++ user-level client and server applications, and does not cover Java programs, operating systems software, or high-performance computing software. Our study does not look at deadlock bugs, and also does not cover bugs that are related to the newly minted C++11 concurrency constructs. Our study does not and cannot cover all concurrency bugs in Apache, MySQL, Mozilla, and other software in our study. Our main bug source, the benchmark from previous work [27], is based on fixed concurrency bugs randomly sampled from the above applications' bug databases. All our findings should be interpreted with our methodology in mind.

## 2.2 What Are Manual Patches Like?

**Synchronization Primitives** As shown in Table 2, a big portion of bugs are fixed *without* using any synchronization primitives (about half). Most of these bugs are fixed without disabling the buggy timing, which will be explained later.

Among patches that leverage synchronization primitives, there is a clear distinction between atomicity violation and order violation patches. In AV patches, lock is the single dominant synchronization primitive; rarely, condition variables, interrupt disabling, and atomic instructions are used. In OV patches, condition variable signal-waits, thread creates, and thread joins are about equally common. Occasionally, customized synchronizations like spin loops are used.

**Fix Strategies** Concurrency bugs are caused by instructions that access shared variables under unexpected timing. Patches can prevent these bugs in three ways: (1) change the timing among those instructions (*Timing* in Table 3), which can be achieved by either adding new synchronization ($Add_S$) or moving around existing synchronization ($Move_S$); (2) bypass some instructions under the original buggy context (*Instruction Bypass*); (3) make some shared variables private under the original buggy context (*Data Private*). Patches could also tolerate the effect of concurrency bugs, instead of preventing them (*Tolerance*). The break-down of these strategies is shown in Table 3.

**Table 3: Fix Strategies (break-downs across root caused and applications are both shown, above and below the line)**

| | Prevention | | | | Tolerance |
|---|---|---|---|---|---|
| | Timing | | Instruction | Data | |
| | $Add_S$ | $Move_S$ | Bypass | Private | |
| **AV** | 15 | 6 | 13 | 8 | 6 |
| **OV** | 10 | 14 | 1 | 0 | 4 |
| **Ap** | 2 | 5 | 3 | 0 | 3 |
| **My** | 1 | 3 | 2 | 3 | 2 |
| **Mo** | 14 | 10 | 8 | 5 | 5 |
| **Op** | 4 | 0 | 1 | 0 | 0 |
| **Misc.** | 4 | 2 | 0 | 0 | 0 |
| **Total** | 25 | 20 | 14 | 8 | 10 |

```
                                  //parent thread
                                  void tr_sessionInit (...) {
                                    h = malloc(...);
                                +   h->band = bdNew(h);
                                    tr_eventInit(...);
                                    ...
                                -   h->band = bdNew(h); //A
                                  }

                                  void tr_eventInit (...) {
//child thread                      pthread_create(...);
assert(h->band); //B              }
```

**Figure 2: A bug in Tranmission, with '+' and '-' denoting its manual/HFix patch.**

Overall, as shown in Table 3, constraining the timing through new or existing synchronization is the most common fix strategy, applied to almost 60% of bugs in study. Other fix strategies are not as common, but still non-negligible, each applied to at least 10% of studied bugs.

Among patches that use the *Timing* fix strategy, about half add new synchronization operations into the software, and the other half leverage existing synchronization operations. For the latter type, the patch is always done by code movement. For example, the real-world bug illustrated in Figure 2 is fixed by moving variable initialization ($A$) before child-thread creation in the parent thread, so that the child thread is guaranteed to read an already-initialized value ($B$).

8 bugs are fixed by making some instructions involved in the bug access local, instead of shared, variables. We will discuss them in more details in Section 2.3.

Patches with instruction-bypassing and bug-tolerance strategies change the sequential computation semantics (24 out of 77). Note that all previous concurrency-bug fixing work [5, 13, 15, 17, 22, 23, 24] uses an *opposite* assumption and only produce patches that preserve sequential semantics.

## 2.3 How About Existing Techniques?

**Adding locks and condition variables** Recently, several tools have been built to automatically fix concurrency bugs by adding locks, such as AFix, Grail, and Axis [13, 22, 24], and condition variables, such as CFix [15]. These techniques provide general fixing capability that applies for a wide variety of concurrency bugs.

Our empirical study shows that these techniques indeed emulate the most common type of manual fix strategies – add new synchronization ($Add_S$), as shown in Table 3.

However, there are still many bugs that are **not** fixed through $Add_S$ by developers (>40% in our study). In many cases, other fix strategies can produce much simpler patches and introduce fewer synchronization operations into software than $Add_S$, such as those in Figure 1 and 2.

Another limitation for this series of tools is that they only look at two types of synchronization primitives: locks and condition variables. Locks are indeed the most dominant primitive for fixing AV bugs. However, condition variables are **not** the most dominant primitive for fixing OV bugs, as shown in Table 2. In fact, among the 10 OV bugs that are fixed by adding new synchronizations, only 3 of them are fixed by adding condition variable signal/waits. Most of them are in fact fixed by adding thread-join operations, such as that in Figure 1a.

**Data privatization** Another fix strategy automated by recent research is data privatization [12]. Previous technique targets two types of AV bugs, where the atomicity of read-after-write (RAW) accesses or read-after-read (RAR) accesses can be violated. Its patch creates a temporary variable to buffer the result of an earlier write access (in case of RAW) or read access (in case of RAR) to or from shared variables, and let a later read instruction from the same thread to directly access this temporary variable, instead of the shared variable.

Our empirical study shows that data privatization is indeed a common fix strategy for AV bugs in practice, taken by developers to fix 8 out of 48 AV bugs in our study.

However, our study also found that the data privatization scheme used by developers goes beyond what used by existing research. First, some write-after-write (WAW) atomicity violations are also fixed by data privatization by developers. For example, Mozilla-52111 and Mozilla-201134 are both fixed by making the first write outputs to a temporary local variable, so that an interleaving read will not see the intermediate value. Second, in several cases, the bugs are fixed not by introducing a temporary local variable, but by changing the declaration of the original shared variable to make it a local variable. For example, in MySQL-7209, Mozilla-253786 and MySQL-142651, developers realize there are in fact no need to make the buggy variables shared. Only 3 bugs are fixed by developers following exactly the same way as existing research proposes.

## 2.4 How About The Future?

Our study points out directions for future research in automated concurrency-bug fixing. Specifically, future work can further refine existing auto-fix strategies, such as data privatization, following our study above; future research can also try to automate manual fix strategies that have not been well explored before, which we will discuss below.

**Automating $Add_{join}$ for OV bugs** Although many recent research tools apply $Add_S$ to fix concurrency bugs [13, 15, 22, 24], they only add lock-related synchronization into software, including locks and condition variables. This is particularly problematic for OV bugs, as many manual OV patches are unrelated to locks or condition variables. Our work along this direction will be presented in Section 4.

**Automating $Move_S$ for concurrency bugs** $Move_S$ leverages existing synchronization in software to fix concurrency bugs. It is one of the most common manual fix strategies for both AV (6 out of 48) and OV bugs (14 out of 29). Unfortunately, it has never been automated by previous research to fix real-world bugs in large applications. Our work along this direction will be presented in Section 5.

**Table 4: Semantic-changing patches (B: Bypass strategy; T: Tolerance strategy).**

| | Patch Location | | | | | Patch Structure | | |
|---|---|---|---|---|---|---|---|---|
| | $AV_c$ | $AV_r$ | $OV_A$ | $OV_B$ | Misc | Skip | UnSkip | Misc |
| B | 2 | 11 | 1 | 0 | 0 | 13 | 0 | 1 |
| T | 2 | 3 | 1 | 3 | 1 | 6 | 4 | 0 |

**Semantic changing fix for concurrency bugs** *Bypass* and *Tolerance* are two intriguing concurrency-bug fix strategies, as they change the sequential semantics and were never explored by previous research. They are common enough to deserve attention – together, they are chosen for 24 out of 77 real-world bugs in our study. Their patches are often simple, mostly between 1–3 lines of code changes.

Our in-depth study shows that these patches are not ad-hoc. Instead, they follow common patterns that can be leveraged by automated tools, as shown in Table 4.

First, the patch location is almost always around key operations in the bug report, as shown in Table 4.

Second, the patch structure is mostly simple. Naturally, all bypass patches add condition checks to bypass code. Interestingly, almost all tolerance patches are also about control flow changes. Some add condition checks to bypass failure-inducing operations, such as a NULL-pointer dereference, after the unsynchronized accesses. Others change existing condition checking, so that some code that was originally skipped under the unsynchronized accesses would now get executed under the patch.

Of course, there are still challenges, such as figuring out the expressions used for condition checking and refactoring following the control flow changes. Overall, our empirical study shows that automating bypass and tolerance strategies are not only intriguing, but also important and promising. We leave this direction to future research.

# 3. BUG FIX BACKGROUND & OVERVIEW

HFix reuses the general bug-fixing framework proposed by CFix [15] (Figure 3). The inputs to the bug-fixing framework are bug reports, which can be automatically generated by bug detection tools [21, 31, 34, 41, 42]. Given a bug report, some checkings are conducted to see which fix strategies might be suitable for the bug (Fix-Strategy Design). Then, program analysis and code transformation are conducted to enforce the desired synchronization following the fix strategy (Synchronization Enforcement). After that, the generated patches go through testing and merging, with final patches produced.

HFix will modify and extend three key components of the fixing framework, highlighted by gray background in Figure 3. That is, different fix strategies will be considered; different types of program analysis and code transformation will be conducted following the different fix strategies; finally, the patch merging will also be different.

HFix reuses some CFix techniques. Specifically, HFix reuses the function cloning technique used in CFix when it

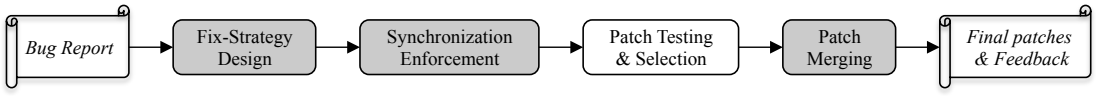



**Figure 3: Bug fixing process**

generates patches that take effect under specific calling contexts. CFix makes best effort to avoid introducing deadlocks, but does not prove deadlock free. Instead, it provides the option to instrument patches for light-weight patch-deadlock monitoring, which is reused in HFix.

HFix reuses an important philosophy of CFix — bug fixing works only when correct bug reports are provided. HFix re-uses the bug-report format requirement of CFix. An AV report needs to specify three statements, $p$, $c$, and $r$. As discussed in Section 2.1, an AV bug manifests when one thread unexpectedly executes $r$ between another thread's execution of $p$ and $c$. An OV report needs to specify two statements, $A$ and $B$. As also discussed in Section 2.1, the OV bug happens when $A$ unexpectedly executes after, instead of before, $B$. Since OV bugs are sometimes context sensitive, CFix also requires an OV bug report to contain the calling contexts of $A$ and $B$, including (1) a call stack that executes the buggy instruction in the thread, and (2) a chain of call stacks (referred as thread stack) indicating how that thread has been created. For an instruction $i$, we call its call stack and thread stack together as *stack* of $i$, and we call the thread that executes $i$ as thread of $i$, or $\texttt{thread}_i$.

# 4. HFix$_{\text{JOIN}}$

A thread join operation (i.e., `join`), such as `pthread_join`, can enforce all operations after `join` in a parent thread to wait for all operations in the joined thread. Adding `join` is a common way to fix OV bugs, as discussed in Section 2. This section presents HFix$_{\text{join}}$ that automatically identifies suitable OV bugs and fixes them through Add$_{\text{join}}$ strategy.

HFix$_{\text{join}}$ takes as input the OV bug report that includes the stack of $A$ and the stack of $B$ with $A$ expected to execute before $B$, as defined in Section 3. HFix$_{\text{join}}$ will then go through two main steps.

1. Suitability checking (Section 4.1). Not all OV bugs can be fixed by adding `join`. HFix checks whether $\texttt{thread}_A$ is a never-joined child of $\texttt{thread}_B$ and other conditions to decide whether Add$_{\text{join}}$ is a suitable strategy for the given bug.

2. Patching (Section 4.2). Once the suitability is decided, HFix conducts code transformation to fix the bug.

## 4.1 Patch Suitability Checking

**Is $\texttt{thread}_A$ a joinable child thread of $\texttt{thread}_B$?** HFix patches follow the common practice and only join a thread from its parent thread. To achieve this, HFix checks whether $\texttt{thread}_A$ is a child of $\texttt{thread}_B$ by examining the stacks of $A$ and $B$. From the stack of $A$, HFix identifies the statement $C$ that creates $\texttt{thread}_A$, and then easily tells whether $C$ comes from $\texttt{thread}_B$ by comparing the stacks. In addition, HFix checks the stack of $B$ to make sure there will be only one instance of $\texttt{thread}_B$ (i.e., HFix checks to make sure that the thread-creation statements are not in loops). Otherwise, inserting `join` cannot guarantee every instance of $B$ to wait for all instances of $A$.

**Is $\texttt{thread}_A$ already joined?** When the software already contains a `join` for $\texttt{thread}_A$, adding an extra `join` is often not a good strategy. This analysis goes through two steps. We first go through all functions in software to identify `join` statements. For every existing `join`, we then check whether it is joining $\texttt{thread}_A$. In our implementation, this is done by checking whether the first parameter of the `pthread_create` statement for $\texttt{thread}_A$ may point to the first parameter of the `pthread_join` under study.

**Will there be deadlock?** An added `join` will force $B$ to wait for not only $A$, but also all operations following $A$ in $\texttt{thread}_A$. This extra synchronization is not required by bug fixing. Therefore, we need to check whether it may lead to deadlocks or severe performance slow-down. Specifically, we conduct inter-procedural analysis to see if any blocking operations (i.e., `pthread_cond_wait`, `pthread_join`, and `pthread_mutex_lock` in our implementation) may execute after $A$ in thread-$A$. If any of these are found, HFix aborts the Add$_{\text{join}}$ fix strategy, as adding `join` will bring a risk of potential deadlocks and/or severe performance slowdowns. Furthermore, HFix also aborts the patch if $B$ is inside a critical section, because inserting a blocking operation (i.e., `join`) inside a critical section may lead to deadlocks.

## 4.2 Patch Generation

To generate an Add$_{\text{join}}$ patch, we need to first decide the location of the new `join` — right before $B$. If $B$ is inside a loop, we will use a flag to make sure that the `join` executes only once. Note that, an alternative and simpler patch is to insert the `join` before the loop, which does not require any flags. The current implementation of HFix does not take this option, fearing that the early execution of `join` may hurt performance or introduce deadlocks.

The parameter of `join` needs to contain an object that represents the child thread (i.e., the `thread_t` object returned by `pthread_create` in POSIX standard). To obtain this object, our patch creates a global vector; pushes every newly created `thread_t` object into this vector right after an instance of $\texttt{thread}_A$ is created; and invokes `join` for each object in the vector right before $B$.

Our current implementation targets POSIX standard and POSIX functions. Small modifications can make HFix work for non-POSIX, customized synchronization operations.

# 5. HFix$_{\text{MOVE}}$

As shown in Section 2 and Table 3, many concurrency bugs, including both AV bugs and OV bugs, can be fixed by leveraging existing synchronization in software, instead of adding new synchronization. Specifically, a Move patch rearranges the placement of memory-access statements, which

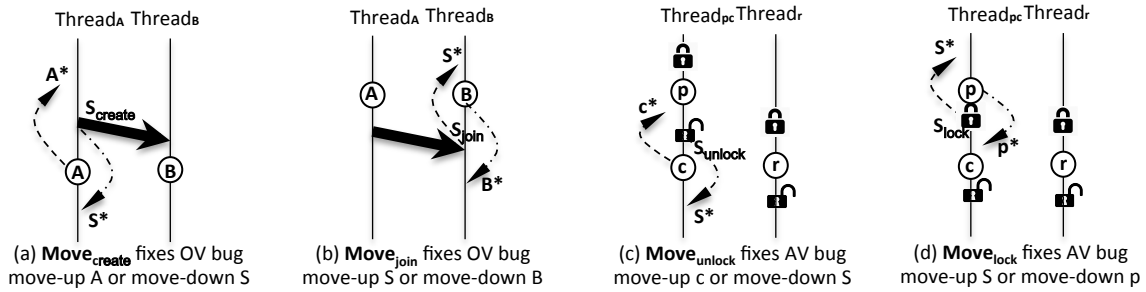



**Figure 4: Fixing OV bugs and AV bugs through Move (thick arrows represent happens-before relationship enforced by a synchronization operation $S$, such as signal/wait, `create`, and `join`; the dashed arrows demonstrate move directions)**

are involved in a concurrency bug, and synchronization operations, which already exist in the same thread, so that the memory-access statements will be better synchronized, as illustrated in Figure 4. This section will present $\text{HFix}_{\text{move}}$ that automates the Move fix strategy.

## 5.1 Overview Of $\text{HFix}_{\text{move}}$

Applying a Move patch is more complicated than applying an $\text{Add}_{\text{join}}$ patch. The suitability checking and patch generation are conducted together in four steps.

1. Identify two operations in one thread, so that flipping their execution order can fix the reported bug. How to conduct this step varies depending on the type of the bug and the type of synchronization nearby.

2. Control flow checking. We need to make sure the movement does not break control dependencies, causing a statement to execute for more or fewer times than it should be. At the end of this step, a candidate patch will be generated and go through the next two steps.

3. Data flow checking. We need to make sure the movement does not break existing define-use data dependency within one thread, causing the patched software to deviate from the expected program semantic.

4. Deadlock and performance checking. We need to check whether the movement could bring risks of deadlocks or severe performance slowdowns.

## 5.2 Identifying Move Opportunities

**$\text{Move}_{\text{join}}$ opportunities for OV bugs** Since `join` can enforce ordering between operations in parent and child threads, we can leverage existing `join` to fix OV bugs, as shown in Figure 4b. Given an OV bug ($AB$), we check whether $\texttt{thread}_A$ is `join`-ed by $\texttt{thread}_B$ in the buggy software. If such a `join` exists, we know that this bug can potentially be fixed by moving $B$ after the `join` (Move-Down) or moving the `join` before $B$ (Move-Up), as shown in Figure 4b. Of course, if we want to make sure all instances of $A$ will execute before $B$, we need to check the stack of $B$ to make sure there is only one dynamic instance of $\texttt{thread}_B$.

**$\text{Move}_{\text{create}}$ opportunities for OV bugs** A thread-creation operation, denoted as `create`, forces all operations before `create` inside the parent thread to execute before all operations inside the child thread. Consequently, `create` can be leveraged to fix some OV bugs, as shown in Figure 4a.

Specifically, given an $AB$ order violation bug, we will check the stack of $B$ to see if $\texttt{thread}_B$ is created by $\texttt{thread}_A$. If so, a $\text{Move}_{\text{create}}$ opportunity is identified: the bug can potentially be fixed by moving $A$ to execute before the `create` (Move-Up) or moving the corresponding `create` to execute after $A$ (Move-Down). Of course, like that in $\text{Move}_{\text{join}}$, if the patch wants to force all dynamic instances of $A$ to execute before $B$, we also need to check the stack of $A$ to make sure that there could be only one dynamic instance of $\texttt{thread}_A$.

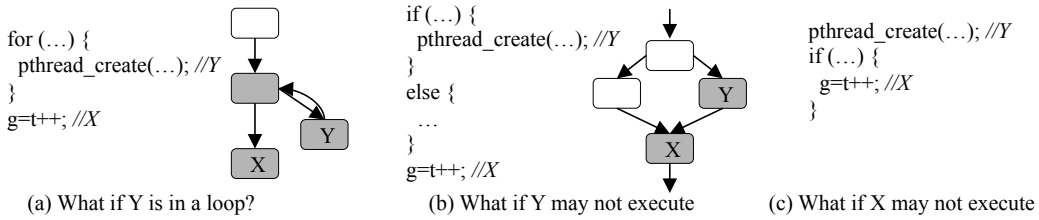
**$\text{Move}_{\text{lock}}$ and $\text{Move}_{\text{unlock}}$ opportunities for AV bugs** Given an AV bug report $p$-$c$-$r$, the patch needs to provide mutual exclusion between the $p$–$c$ code region and $r$. If $r$ and part of the $p$–$c$ code region are already protected by a common lock, the patch can leverage existing `lock` or `unlock`, as illustrated by Figure 4d and Figure 4c.

Specifical, HFix identifies $\text{Move}_{\text{lock}}$ or $\text{Move}_{\text{unlock}}$ opportunities for an AV bug in two cases. In the first case, $p$ and $r$ are inside critical sections of lock $l$, but $c$ is not. As shown in Figure 4c, this type of bugs can potentially be fixed by re-ordering $c$ and a corresponding `unlock` operation ($\text{Move}_{\text{unlock}}$[3]). In the second case, $c$ and $r$ are inside critical sections of lock $l$, but $p$ is not. As shown in Figure 4d, this type of bugs can potentially be fixed by re-ordering $p$ and a corresponding `lock` operation ($\text{Move}_{\text{lock}}$). Below, we describe our analysis algorithm focusing on the $\text{Move}_{\text{unlock}}$ case. The algorithm for $\text{Move}_{\text{lock}}$ patches is similar.

To identify the above two cases, we first identify all the critical sections that contain $p$, $c$, and $r$, respectively, and then compare the locks that are used for these critical sections. Note that, accurately identifying all enclosing critical sections and comparing lock sets are very challenging, as it involves inter-procedural and pointer alias analysis. Our current implementation only considers critical sections that are in the same function as $p$, or $c$, or $r$, not in their callers, and are protected by global locks, not heap locks. This way, although we may miss some fix opportunities, we keep our analysis simple and accurate.

**Generalize $\text{Move}_{\text{join}}$, $\text{Move}_{\text{create}}$, $\text{Move}_{\text{lock}}$, $\text{Move}_{\text{unlock}}$** For the ease of discussion, we generalize all these strategies, and use the following terms and symbols in the remainder of this section. We use *Move-Up* to refer to patches that make an operation execute earlier by moving it up in its thread, denoted by up-pointing dashed arrows in Figure 4; we use *Move-Down* to refer to patches that make an opera-

[3]Sometimes, we use $\text{Move}_{\text{lock}}$ to represent both lock-moving patches and unlock-moving patches.




(a) What if Y is in a loop? (b) What if Y may not execute (c) What if X may not execute

**Figure 5: Control flow challenge for code movement.**

tion execute later by moving it down in its thread, denoted by down-pointing dashed arrows in Figure 4.

We use $X$ to denote the operation in the original program that gets moved; use $X*$ to denote its counter-part in the patched program; and use $Y$ to denote the operation that $X$ needs to move around in order to fix the bug. For example, Figure 4a demonstrates $\text{Move}_{\text{create}}$ strategy. In the Move-Up version of a $\text{Move}_{\text{create}}$ patch, $X$ represents operation $A$, and $Y$ represents synchronization $S$ (i.e. `create`); in the Move-Down version of $\text{Move}_{\text{create}}$, $X$ represents synchronization $S$ and $Y$ represents operation $A$.

## 5.3 Control Flow Checking

We conduct several rounds of control flow checking to identify the correct location of $X^*$, while make sure that $X^*$ will execute for exactly the same number of times as $X$.

First, check if $X$ and $Y$ are inside the same function. We do not want to make the movement when $X$ and $Y$ are inside different functions, as that would break the original code encapsulation. For the cases of $\text{Move}_{\text{join}}$ and $\text{Move}_{\text{create}}$, when $X$ and $Y$ are inside different functions, we find a common function $f$ on their call stacks, change the operations under interests from $X$ and $Y$ to their call sites in $f$, and then conduct all the control-flow analysis and movement inside one function, $f$. For example, for the bug shown in Figure 2, the two operations that we initially want to reorder are `pthread_create` inside function `tr_eventInit` and the write access to `h`→ `band` inside function `tr_sessionInit`. Since they are not inside the same function, we instead identify their common caller `tr_sessionInit`, and shift our focus to flipping the order between `tr_eventInit` and the write access. The same strategy cannot be directly applied to $\text{Move}_{\text{lock}}$ or $\text{Move}_{\text{unlock}}$. Imagine we want to move a `lock` operation in function $f_1$ so that the corresponding critical section can be extended to contain not only $c$ in $f_1$ but also $p$ in function $f_0$, the caller of $f_1$. We cannot simply move the callsite of $f_1$ in $f_0$, because that would likely move the whole critical section inside $f_1$ and still leave $p$ outside the critical section. Consequently, HFix simply makes a clone of the callee function, makes the cloned function inlined, and then conducts code movement inside one function.

Second, check if $X$ is inside a loop. If it is, we abort the fix. The reason is that it would be too difficult to move $X$ without changing the number of times $X$ executes.

Third, identify the location $X^*$. Note that, this is easy when $X$ and $Y$ are inside the same basic block — $X^*$ would be simply right after or before $Y$. However, in reality, there could be several challenges. For example, as demonstrated in Figure 5a, if $Y$ is inside a loop, simply moving $X$ to be right after or before it could cause $X^*$ to execute for many more times in the patched program than $X$ does in the original program. More importantly, this movement cannot guarantee that $X$ will execute before *all* instances of $Y$, a property that is needed to fix many OV bugs (e.g., we may want a variable to be initialized before any child thread can read it). As another example demonstrated in Figure 5b, when $X$ and $Y$ are not inside the same basic block, a naive movement could also cause $X^*$ to execute for fewer times in the patched program than $X$ does in the original program.

Our algorithm addresses all these challenges. When we want to move $X$ up to execute before all instances of $Y$, we will do the following. We first identify all control-flow graph nodes in function $f$ that are reachable from $Y$, referred to as `AfterY` nodes, as shown by the gray boxes in Figure 5. We then delete $X$, and make $X^*$ appear on every edge that goes from a non-`AfterY` node $n$ to an `AfterY` node $n_2$. This arrangement guarantees that $X^*$ will execute before all instances of $Y$, because otherwise $n$ would have been an `AfterY` node; it also guarantees that $X^*$ will execute at most once in its function, because once the execution goes from non-`AfterY` nodes to `AfterY` nodes, it can never go back to nodes non-reachable from Y before the function exits.

When we want to move $X$ down to execute after all instances of $Y$, the handling is similar. We first identify all control-flow graph nodes inside function $f$ that can reach $Y$, referred to as `BeforeY` nodes. We then delete $X$, and place $X^*$ on every edge that goes from a `BeforeY` node $n$ to a non-`BeforeY` node $n_2$. Similarly, this arrangement guarantees that that $X^*$ will execute at most once in function $f$ and will execute before all instances of $Y$.

Finally, we need to make sure $X^*$ will execute for the same number of times $X$ does. Note that, since $X$ is not inside a loop, it will execute at most once. Our arrangement above guarantees the same property for $X^*$. In case of Move-Down, our patch would insert a flag setting statement in the original location of $X$, and a flag checking before every $X^*$. This way, we guarantee that $X^*$ is executed only when the original $X$ would have been executed. The Move-Up case could be complicated, as illustrated in Figure 5c. When $X^*$ is executed, we may not be able to predict whether $X$ would be executed or not later. If later execution decides to by-pass $X$, we will have to revert the effect of $X^*$. The revert could be impossible within one thread, if $X^*$ creates a thread or writes to a shared variable. Given this complexity, in our implementation, we apply the Move-Up strategy only when (1) $X$ executes exactly once in function $f$; or (2) the execution of $X$ does not require revert or can be reverted easily (e.g., some moved-up `join` does not require revert; some moved-up `lock` can be easily reverted).

The complexity of the above analysis is linear to the size of the control flow graph of function $f$.

## 5.4 Data Flow Checking

We use data flow analysis to check whether the code movement could affect thread-local data dependency and mistakenly change program semantics.

Specifically, we identify all instructions $I$ that might execute between the original location of $X$ and the new location of $X^*$, and check if they might access the same memory locations as $X$. We abort the patch, if either (1) $I$ may access a memory location that is written by $X$ or (2) $I$ may write to a memory location that is read by $X$. In these cases, the code movement could break the write-after-write, write-after-read or read-after-write dependency in the original program, incorrectly changing program semantics, and hence should not be used in bug fixing. In our current prototype, we use the default MAY-pointer-alias analysis in LLVM to decide whether two sets of instructions may read/write the same memory object. We consider `create`, `join`, and `(un)lock` functions to only access its parameter objects.

## 5.5 Deadlock And Performance Checking

To avoid introducing deadlocks or severe performance slowdowns into the program, we also check and abort some patches to minimize the risk of introducing circular waits among threads. For $\text{Move}_{\text{create}}$, we make sure the patch does not push extra blocking operations to execute before `create`; for $\text{Move}_{\text{join}}$, we make sure the patches does not delay some unblock operations to execute after `join`. In case of $\text{Move}_{\text{lock}}$ and $\text{Move}_{\text{unlock}}$, we make sure not to move any blocking operations into a critical section. In our implementation, the black list of blocking operations includes lock, condition-variable wait, and thread-join operations.

**Summary** $\text{HFix}_{\text{move}}$ is sound but not complete. Its patches are guaranteed not to introduce new bugs that violate control dependency or thread-local data dependency of the original software, as discussed in Section 5.3 and 5.4. $\text{HFix}_{\text{move}}$ guarantees not to introduce deadlocks, as long as its list of (un)blocking operations is complete. However, $\text{HFix}_{\text{move}}$ may miss the opportunity to generate Move patches for corner-case Move-suitable bugs, as we will see in our evaluation (Section 7.2.2).

# 6. PATCH MERGING

A single synchronization mistake, such as forgetting to join a child thread, can often lead to multiple related bug reports. Fixing these related bugs separately would hurt the patch simplicity and performance. For example, bug detectors report five highly related OV bugs in PBZIP2 (two of them are illustrated by $AB_1$ and $AB_2$ in Figure 1). Naively fixing these bugs separately would add five `join` within a few lines of code, which is unnecessarily complicated.

Fortunately, both $\text{Add}_{\text{join}}$ and Move strategies are naturally suitable for patch merging. For example, if multiple OV bugs between a parent thread and a child thread are reported, we may fix one bug through $\text{Add}_{\text{join}}$, and the remaining ones through $\text{Move}_{\text{join}}$, leveraging the newly added `join`. We could also fix each bug report separately, and then analyze the patches to see if we can merge them, which is exactly what we have implemented in this work.

We only merge patches with the same fix strategy. We only discuss how to merge two patches below; merging more than two patches is similar. We do not discuss how to merge $\text{Move}_{\text{lock}}$ patches below, because it can be addressed by merging technique proposed in previous work [15].

## 6.1 Merge $\text{Add}_{\text{join}}$ Patches

Imagine that two $\text{Add}_{\text{join}}$ patches are generated for two OV bugs $A_1B_1$ and $A_2B_2$. Our merger explores merging these two patches if $A_1$ and $A_2$ are from the same thread, and $B_1$ and $B_2$ are from the same thread. This checking is conducted based on the stack of $A_1$, $A_2$, $B_1$, and $B_2$.

If the two bugs/patches pass the first checking, our merger will try to create a merged patch. That is, the merger tries to decide where to add `join` in the merged patch. We use $j_1$ to denote the location of the added `join` in the patch for $A_1B_1$ (i.e., right before $B_1$ in Figure 1a), and $j_2$ to denote the location of the added `join` in the patch for $A_2B_2$ (i.e., right before $B_2$ in Figure 1a). The merger will identify the nearest-common-dominator of $j_1$ and $j_2$, denoted as $j_{12}$, and put the `join` there in the merged patch (i.e., the '+' line in Figure 1a). This merged patch can guarantee to fix both $A_1B_1$ and $A_2B_2$ OV bugs, because $B_1$ and $B_2$ are guaranteed to execute after their common dominator `join` $j_{12}$, which in turn is guaranteed to execute after $A_1$ and $A_2$.

To avoid introducing deadlocks or severe performance slowndowns, the merger stops the merging attempt if there exist any signal or unlock operations along the paths that connect $j_{12}$ and $j_1$, and $j_{12}$ and $j_2$. Note that, we decide to put the merged `join` at the nearest, instead of any, common-dominator, exactly because we want to minimize the risk of introducing deadlocks or severe perforfance slowdowns.

## 6.2 Merge $\text{Move}_{\text{join}}$ And $\text{Move}_{\text{create}}$ Patches

Imagine that two $\text{Move}_{\text{join}}$ patches are generated for two OV bugs $A_1B_1$ and $A_2B_2$. There is clearly no chance for merging, if one patch uses Move-Up (i.e., moving `join` to execute before $B_1$ or $B_2$) and the other patch uses Move-Down (i.e., moving $B_1$ or $B_2$ to execute after `join`); there is also no benefit of merging, if both patches use the Move-Down strategy. When both patches use Move-Up, our merger explores merging these two patches if $A_1$ and $A_2$ are from the same thread, and $B_1$ and $B_2$ are from the same thread, just like that in $\text{Add}_{\text{join}}$ patch merging.

Once the two patches pass the above checking, HFix creates a merged patch in a similar way as $\text{Add}_{\text{join}}$ patch merging. That is, the merged patch keeps only one of the `join` from the two patches, at the nearest common dominator of the original locations in the two patches. Similar checking is conducted to make sure the merge does not bring extra risks of deadlocks or severe slowdowns.

HFix merges $\text{Move}_{\text{create}}$ patches in a similar way as merging $\text{Move}_{\text{join}}$ patches. We skip the discussion here.

# 7. EXPERIMENTAL EVALUATION

## 7.1 Methodology

HFix is implemented using LLVM 3.6.1. All the experiments are conducted on eight-core Intel Xeon machines.

**Benchmark Suite** To evaluate HFix, we use two sets of real-world concurrency bugs. The first is the bug-fixing benchmark suite set up by CFix [15]. It contains 7 OV and 6 AV bugs. These 13 bugs are all representative benchmarks used by many previous works [14, 36, 41, 42]. They come from publicly released versions of 10 open-source C/C++ multithreaded applications, most of which contain tens to

**Table 5: Patch comparison for OV bugs (A: Add; M: Move; #Sync: number of new synchronization operations added by the patch; j: join; s: signal; w: wait; l: lock; u: unlock; -: patch not generated or not available)**

| BugID | App. | HFix | | Manual | | CFix | |
|---|---|---|---|---|---|---|---|
| | | Strategy | #Sync | Strategy | #Sync | Strategy | #Sync |
| OV1 | FFT | A $_{\text{join}}$ | 1j | A $_{\text{join}}$ | 1j | A $_{\text{s.w.}}$ | 4s,1w |
| OV2 | HTTrack-20247 | - | - | - | - | A $_{\text{s.w.}}$ | 1s,2w |
| OV3 | Mozilla-61369 | - | - | A $_{\text{lock}}$ | 2l,2u | A $_{\text{s.w.}}$ | 1s,1w |
| OV4 | PBZIP2 | A $_{\text{join}}$ | 1j | A $_{\text{join}}$ | 1j | A $_{\text{s.w.}}$ | 4s,1w |
| OV5 | Transmission-1818 | $M_{\text{create}}$ | 0 | $M_{\text{create}}$ | 0 | A $_{\text{s.w.}}$ | 1s,1w |
| OV6 | X264 | $M_{\text{join}}$ | 0 | $M_{\text{join}}$ | 0 | A $_{\text{s.w.}}$ | 6s,1w |
| OV7 | ZSNES-10918 | $M_{\text{create}}$ | 0 | - | - | A $_{\text{s.w.}}$ | 2s,1w |

hundreds of thousands lines of code. They lead to severe crashes and security vulnerabilities. They are fixed by developers through a wide variety of strategies, which we will discuss in Section 7.2. For each bug, CFix suite contains the following information: (1) the original buggy software, (2) bug reports that can be used as inputs to auto-fixing tools, following the format discussed in Section 3, and (3) scripts for patch performance and correctness testing. Particularly, there is a slightly modified buggy program that contains random `sleep`s to make a bug manifests more frequently and hence more suitable for rigorous correctness testing.

The second set includes *all* the AV bugs in the concurrency-bug benchmark suite composed by Lu et. al. [27] that are fixed by developers through moving synchronization operations. There are six bugs in this set as shown in Table 3. For each bug, there is a bug report prepared by us using the format discussed in Section 3.

We use the first set of benchmarks, because it enables direct comparison between HFix and the state-of-the-art concurrency-bug fixing tool, CFix. We use the second set, because it allows a targeted evaluation about how well HFix can automate the Move fix strategy for AV bugs.

**Evaluation Metrics** We evaluate the quality of HFix patches mainly by comparing them with manual patches. We will explain what are the differences, if any, and whether/how the differences affect patch quality. It is infeasible to prove the correctness of a big multi-threaded software. Fortunately, HFix already provides soundness guarantees for its patches (discussed at the end of Section 5), and we will use comparison with manual patches to further demonstrate the correctness of HFix patches.

Since the main goal of HFix is to improve the patch simplicity of the state of the art, we will quantitatively measure patch simplicity by counting the number of new synchronization operations added by each patch.

In addition to comparing with manual patches, we will also compare HFix with CFix using CFix benchmark suite. We will use the simplicity metric discussed above, and also run the performance and correctness testing provided by CFix benchmark suite to show that HFix patches do not hurt performance or correctness.

## 7.2 Experimental Results

### 7.2.1 Overall Results For OV Bugs

As shown in Table 5, HFix correctly identifies OV bugs that are suitable for Add or Move fix strategies, and effectively generates patches that are as simple as manual patches, well complementing the state of the art.

Among all the 7 OV bug benchmarks, HFix automatically and correctly generates simple patches for five. HFix also makes correct decisions for the other two that indeed cannot be fixed by $\text{Add}_{\text{join}}$ or Move. Specifically, OV3 happens when a child thread unexpectedly reads a variable before it is initialized in the parent thread. As correctly pointed out by HFix, OV3 cannot be fixed by $\text{Add}_{\text{join}}$ or Move, because `join` cannot force parent-thread operations to execute before child-thread operations and data-dependency prevents Move from being applied. The situation in OV2 is opposite: the parent thread could read a variable before it is initialized in the child thread. $\text{Move}_{\text{join}}$ is tried and correctly aborted by HFix due to failed deadlock checking.

Comparing with manual patches, HFix performs very well. HFix produces *exactly the same* patches as developers manually did for OV1, OV4, and OV5. HFix patch for OV6 has a trivial control-flow difference from the corresponding manual patch — HFix patch conducts "`if(..){..; X;}else{X;}`", while manual patch does "`if(..){..}; X;`".

Comparing with CFix, HFix can generate much simpler patches than CFix does. For OV1 and OV4 — OV7, CFix introduces about four signal operations and one wait operation in *each* patch. Instead, HFix only introduces 0 – 1 synchronization operation. Note that, the CFix patch complexity goes beyond the synchronization operations listed in Table 5. For example, it also contains the declarations of new global synchronization variables; every signal or wait operation also comes with one lock, one unlock, and some corresponding flag setting/checking operations. Of course, CFix can fix all the seven OV bugs, including OV2 and OV3 that cannot be fixed by HFix. This result matches the different design goals of CFix, which emphasizes generality, and HFix, which emphasizes specialization and simplicity.

### 7.2.2 Overall Results For AV Bugs

HFix can patch not only OV bugs, but also AV bugs, through the Move strategy. However, manual Move patches are not as common for AV bugs than for OV bugs in real world (Table 3). Our evaluation shows a consistent trend.

Among the six AV bugs in CFix benchmark suite, none of them is suitable for Move strategy, because there is no lock-/unlock operations around the buggy code. HFix correctly figures this out, and did not generate patch for any of them. CFix can fix all these six bugs, introducing one new global lock variable and 2–9 lock/unlock operations in each patch. Developers fixed these bugs by a mix of strategies, including adding lock/unlock operations, data privatization, changing reader-lock to writer-lock, and tolerating buggy timing.

**Table 6: Patch comparison for AV bugs**

| Bug ID | App. | HFix | | Manual | |
|---|---|---|---|---|---|
| | | Stra | #Sync | Stra | #Sync |
| AV1 | Ap-21287 | - | - | $M_{unlock}$ | 0 |
| AV2 | Mo-18025 | $M_{lock}$ | 0 | $M_{lock}$ | 0 |
| AV3 | Mo-141779 | $M_{lock}$ | 0 | $M_{lock}$ | 0 |
| AV4 | My-169 | $M_{unlock}$ | 0 | $M_{unlock}$ | 0 |
| AV5 | My-14262 | $M_{unlock}$ | 0 | $M_{unlock}$ | 0 |
| AV6 | My-14931 | $M_{unlock}$ | 0 | $M_{unlock}$ | 0 |

Among the six AV bugs in our second benchmark suite (Table 6), HFix correctly generates simple Move patches, involving no new synchronization operations or variables, for five of them. HFix patch for AV4 is exactly the same as the manual patch. HFix patches for AV2 and AV3 only have trivial difference from the manual patches, exactly like the case of OV6 discussed above. HFix patches for AV5 and AV6 are very similar with manual patches. The difference is that manual patches directly moved code between a caller function $f_0$ and a callee function $f_1$. Instead, HFix first inlines the corresponding invocation of $f_1$ inside $f_0$ before the movement. HFix did not generate Move patches for AV1, because none of the atomicity-violation related statements (`p`, `c`, and `r`) are inside any critical sections in the buggy software. The manual patch moves unlock statements to extend a critical section that did not contain `p`, `c`, or `r` in the buggy version to contain all three of them.

### 7.2.3 Other Detailed Results

**Alternative patches** For every evaluated bug, HFix tries all fix strategies and generates as many patches as possible. Having said that, OV6 is the only case where HFix generates more than one patch: one moves `join` up and one moves a memory access down. They only have a trivial difference and are semantically equivalent with each other: there are a few lines of local-variable computation that are executed after `join` in one patch, yet before `join` in the other.

**Patch testing using CFix benchmark suite** We conducted the patch testing provided by CFix benchmark suite. HFix patches passed the correctness testing: the failure rates of unpatched programs range between 10% and 60%, measured through 1,000 testing runs with random `sleep`s; the failure rates of HFix patched programs are all 0 under the same setting. HFix patches also passed the performance testing: the overhead of HFix patched programs is always under 0.5%, which is each averaged upon 1,000 failure-free runs, similar with that of CFix patches [15].

**HFix static analysis** HFix static analysis is efficient. It takes less than 5 seconds to generate patches for each bug.

**Merging** Our patch merging is effective. Among all the bugs, OV1 and OV4 are the two cases that contain multiple related bug reports. Without patch merging, HFix patches would have contained as many as 10 and 5 join operations for these two bugs. Fortunately, after the merging, only one `join` is needed for each, as illustrated in Figure 1a.

### 7.2.4 Threats To Validity

We evaluated HFix on two sets of representative real-world bugs: the CFix benchmark suite [15] and all AV bugs with Move-style manual patches in the benchmark suite composed by Lu et. al. [27]. We should not over-generalize the evaluation results. It would be wrong to speculate that HFix can generate patches for more than two thirds of OV bugs or 90% of Move-suitable AV bugs in the world.

What we *can* generalize is that HFix is a useful complement and can effectively improve the state of the art of auto-fixing for concurrency bugs. HFix is designed to produce high-quality patches for many, but not all, concurrency bugs. that HFix has achieved this goal. Users can use HFix together with other auto-fixing tools, and pick the best patches produced.

Our benchmark suite does not include any deadlock bugs. HFix currently does not handle deadlock bugs. Theoretically, some deadlock bugs could be fixed by moving lock-acquisition statements, which we leave as future work.

We should also note that the set of bugs used to evaluate HFix in this section is a subset of the 77 bugs studied in Section 2. This methodology is potentially a source of threats to validity. We decide to focus HFix on Move and Join fix strategies exactly because of the study of those 77 bugs. However, the exact design did not target any specific bug. We have tried our best to make HFix algorithm generally applicable.

Another potential threat to validity is that, for big multi-threaded software, it is infeasible to prove the correctness of patched software. Following the methodology of previous work [13, 15], we made our best effort in correctness checking through manual examination and anecdotal patch testing. We are confident in the correctness of HFix patches because of both the soundness guarantee of HFix (discussed at the end of Section 5) and the similarity between HFix patches and manual patches.

Just like CFix, HFix takes bug-report inputs, which specify static program statements involved in bugs. At run time, each static statement may have multiple dynamic instances. The current design of HFix, just like CFix, generates patches for all dynamic statement instances, in case of AV bugs, and for all dynamic statement instances that match the reported callstacks, in case of OV bugs. This treatment could lead to incomplete patches for OV bugs, or overly synchronized patches for OV and AV bugs. The callstack-sensitive treatment of OV bugs requires HFix to conduct function cloning, which may cause the resulting patches to be more complicated than necessary. Ultimately, it is impossible to decide what is the best fix strategy, unless we know which exact dynamic statement instances are buggy.

Finally, HFix relies on the bug-report inputs to work effectively. If the bug reports are incomplete or incorrect, HFix cannot guarantee the quality of its patches. Fortunately, as shown by our experiments and earlier work [15], a lot of concurrency bugs can indeed be correctly and automatically detected and reported by existing concurrency-bug detectors.

## 8. RELATED WORK

**Empirical study of concurrency bugs** Past studies looked at real-world concurrency bugs [9, 27] and synchronization-related code changes [11, 29, 32, 39]. They provide important guidance for concurrency bug detection.

Several empirical studies have looked at concurrency-bug patches, but with different focuses and findings from our study. One focuses on the correctness of intermediate patches [40]; one focuses on how transactional memory might help simplify concurrency bug patches [37]; one studies patches to understand how file systems evolve [26].

Lu et. al. summarize concurrency bug patches into five categories: condition check, code switch, design change, add/change locks, others [27]. Their overall study focuses on bug understanding, and we cannot directly use their results to help automated concurrency-bug fixing. First, that study does not provide break-downs among synchronization primitives used in patches. It does not discuss which specific synchronization primitives are used other than locks. Second, their categories are not aligned with fix strategies. For example, their "condition check" category actually includes both bypassing cases and spin-loop synchronization cases, which require completely different techniques to automatically generate. Third, they do not provide enough information for each category of patches to help understand how patches might be automatically generated. Specifically, they do not provide category break-downs across different bug root causes; they do not explain how components of a patch might be related to components of a bug report; they do not discuss how many patches change sequential semantics.

Another study conducted by Cerny et. al. [5] checked patches of concurrency bugs in Linux device drivers. Comparing our findings with theirs, there are both commonalities and differences between concurrency-bug patches in user-level applications and kernel code. For examples, there are similar portions of concurrency bugs fixed by existing synchronization operations ($\text{MOVE}_S$), adding lock synchronization ($\text{Add}_{\text{lock}}$), and bypassing respectively in both user-level applications and kernel. On the other hand, there are many kernel-level bugs fixed by atomic instructions and synchronization upgrading, which are rare in user-level applications; there are many user-level bugs fixed by data privatization, bug tolerance, and thread-create/join synchronization, which are rare or not mentioned in the kernel study. Furthermore, with a different fix framework in mind, their study does not tie fix strategies with bug root causes, and hence cannot directly guide CFix/HFix-style bug fixing.

**Fixing general software bugs** Many techniques have been proposed to automatically fix general bugs [7, 8, 16, 18, 19, 25, 33]. Since they neither leverage unique features of concurrency bugs nor assume knowledge about concurrency-bug reports, they cannot be directly applied for concurrency bugs. Interestingly, our study shows that a non-negligible portion of concurrency-bug patches actually do change the sequential computation semantics. Therefore, future research can leverage these general techniques to help generate semantic-changing concurrency-bug patches.

Past work has studied real-world patches of general bugs to guide bug fixing [18, 43]. Without targeting concurrency bugs, the findings do not directly guide concurrency-bug fixing. Note that, since there are much fewer concurrency bugs that are clearly documented, studies for general bugs and patches usually check many more samples than studies for concurrency bugs. For example, the main bug set used in this paper comes from Lu et. el. [27]. Containing around 70 non-deadlock real-world concurrency bugs, it is already among the largest concurrency-bug study.

**Concurrency bug fixing** Here we focus on techniques that have not been well discussed. ConcurrencySwapper and ConRepair are designed for fixing concurrency bugs in Linux device drivers [5, 6]. Different from many previous techniques, they could fix a bug by reordering statements, similar with the Move strategy in HFix. However, due to the different design goal, their techniques are different from HFix and cannot be applied to bugs handled by HFix. Specifically, they do not use an atomicity-violation or order-violation bug report as input; they try all possible statement reordering within basic blocks; they rely on model checking. They are evaluated on simplified skeleton programs (hundreds of lines of code) written in a simplified language which supports a subset of C. Even the simplified bug could takes 30 minutes to fix. The simplified language provides `atomic` and `await` constructs, but not explicit locks, condition variables, thread-create/joins, etc. Although inspiring, at this point, this technique cannot handle concurrency bugs in large software, like the ones HFix targets.

# 9. CONCLUSION

Automated bug fixing is both challenging and important. Many automated fixing techniques have been proposed recently for concurrency bugs. This paper provides an in-depth understanding of this research direction, through a thorough study of manual patches for 77 real-world concurrency bugs. Our study provides both endorsement for existing techniques and actionable suggestions for future research to further improve the quality of automatically generated patches. Our design of HFix leverages some of these findings, and our evaluation shows that HFix can indeed produce high-quality patches for many real-world concurrency bugs in large applications. We believe future research can further improve the quality of auto-patches following the guidance provided by our patch study and extending HFix.

# 10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their tremendous feedback and comments. We also would like to thank Guoliang Jin and Linhai Song for their continuing supports. This material is based upon work supported by the NSF (grant Nos. IIS-1546543, CNS-1514256, CCF-1217582, CCF-1439091, CCF-1514189) and generous supports from Huawei, Alfred P. Sloan Foundation, and CERES Research Center. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

## 11. REFERENCES


[1] Apache bugzilla. https://issues.apache.org/bugzilla/index.cgi.

[2] Mozilla bugzilla. http://bugzilla.mozilla.org/.

[3] Mysql bugs. http://http://bugs.mysql.com/.

[4] Welcome to apache openoffice (aoo) bugzilla. https://bz.apache.org/ooo/.

[5] P. Černỳ, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *Computer Aided Verification*, 2013.

[6] P. Černỳ, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Regression-free synthesis for concurrency. In *Computer Aided Verification*, 2014.

[7] S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic debugging. In *ICSE*, 2011.

[8] M. Fazzini, P. Saxena, and A. Orso. Autocsp: Automatically retrofitting CSP to web applications. In *ICSE*, 2015.

[9] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *DSN*, 2010.

[10] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for c programs. In *ICSE*, 2015.

[11] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu. What change history tells us about thread synchronization. In *FSE*, 2015.

[12] J. Huang and C. Zhang. Execution privatization for scheduler-oblivious concurrent programs. In *OOPSLA*, 2012.

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

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

[15] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[16] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: automated synthesis of repair hints. In *ICSE*, 2014.

[17] S. Khoshnood, M. Kusano, and C. Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *ISSTA*, 2015.

[18] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, 2013.

[19] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, 2012.

[20] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[21] D. Li, W. Srisa-an, and M. B. Dwyer. SOS: saving time in dynamic race detection with stationary analysis. In *OOPSLA*, 2011.

[22] P. Liu, O. Tripp, and C. Zhang. Grail: Context-aware fixing of concurrency bugs. In *FSE*, 2014.

[23] P. Liu, O. Tripp, and X. Zhang. Flint: Fixing linearizability violations. In *OOPSLA*, 2014.

[24] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ICSE*, 2012.

[25] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA*, 2012.

[26] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of linux file system evolution. In *FAST*, 2013.

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

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

[29] S. Okur and D. Dig. How do developers use parallel libraries? In *FSE*, 2012.

[30] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their finding places. In *ASPLOS*, 2009.

[31] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *ICSE '10*, 2010.

[32] C. Sadowski, J. Yi, and S. Kim. The evolution of data races. In *MSR*, 2012.

[33] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE*, 2012.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[35] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[36] 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.

[37] H. Volos, A. J. Tack, S. Lu, and M. Swift. Applying transactional memory to concurrency bugs. In *ASPLOS*, 2012.

[38] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlk. Gadara: dynamic deadlock avoidance for mult-threaded programs. In *OSDI*, 2008.

[39] R. Xin, Z. Qi, S. Huang, C. Xiang, Y. Zheng, Y. Wang, and H. Guan. An automation-assisted empirical study on lock usage for concurrent programs. In *ICSM*, 2013.

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

[41] 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.

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

[43] H. Zhong and Z. Su. An empirical study on real bug fixes. In *ICSE*, 2015.