Grail: Context-Aware Fixing of Concurrency Bugs

Peng Liu\textsuperscript{1} lpxz@ust.hk  
Omer Tripp\textsuperscript{2} otripp@us.ibm.com  
Charles Zhang\textsuperscript{3} charlesz@cse.ust.hk  
\textsuperscript{1}State Key Laboratory of Software Engineering, Wuhan University, China  
\textsuperscript{2}IBM Research, United States  
\textsuperscript{3}HKUST, China

ABSTRACT

Writing efficient synchronization for multithreaded programs is notoriously hard. The resulting code often contains subtle concurrency bugs. Even worse, many bug fixes introduce new bugs. A classic example, seen widely in practice, is deadlocks resulting from fixing of an atomicity violation. These complexities have motivated the development of automated fixing techniques. Current techniques generate fixes that are typically conservative, giving up on available parallelism. Moreover, some of the techniques cannot guarantee the correctness of a fix, and may introduce deadlocks similarly to manual fix, whereas techniques that ensure correctness do so at the expense of even greater performance loss.

We present Grail, a novel fixing algorithm that departs from previous techniques by simultaneously providing both correctness and optimality guarantees. Grail synthesizes bug-free yet optimal lock-based synchronization. To achieve this, Grail builds an analysis model of the buggy code that is both contextual, distinguishing different aliasing contexts to ensure efficiency, and global, accounting for the entire synchronization behavior of the involved threads to ensure correctness. Evaluation of Grail on 12 bugs from popular codebases confirms its practical advantages, especially compared with existing techniques: Grail patches are, in general, \geq 40\% more efficient than the patches produced by other techniques, and incur only 2\% overhead.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming

General Terms

Algorithms, Reliability, Performance

Keywords

Context-aware fixing, concurrency bugs

\textsuperscript{\ast}This research is supported by RGC GRF grant RGC621912.

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.

FSE’14, November 16–22, 2014, Hong Kong, China  
Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...$15.00.

1. INTRODUCTION

Concurrency bugs are elusive. Not only are they hard to reproduce and understand, but also fixing a bug often introduces new bugs. A typical example is that fixing an atomicity violation requires insertion of additional synchronization, which leads to new deadlocks. A recent study of bug fixes in large software systems such as Linux and FreeBSD \cite{tri12} reports that 39\% of the manual fixes committed by programmers are incorrect, and 16.4\% of the incorrect fixes introduce deadlocks. These statistics clearly confirm the hardness of manually fixing concurrency bugs, marking the need for automation.

Several different approaches have been proposed recently to automatically fix concurrency bugs \cite{liu11,liu12,liu19,ari08}. These address concurrency problems resulting from inadequate synchronization, including atomicity violations \cite{liu11,liu20}, deadlocks \cite{ari08}, and data races \cite{sch03}. Roughly speaking, in all of these cases the fix amounts to inserting additional synchronization to inhibit bad interleaving scenarios. The goal, naturally, is to insert “tight” synchronization, which prevents all bad executions (correctness) while permitting most of the good executions (performance).

Existing Approaches. To illustrate the behavior and limitations of existing techniques, we refer to a real-world concurrency bug reported on the NioTcpServer class in the Apache Mina framework.\textsuperscript{1} The buggy code is shown in Figure \ref{fig:example}a. Concurrent execution of methods \texttt{bind} and \texttt{unbindAll} can result in a ConcurrentModificationException if \texttt{bind} adds an element to the \texttt{addresses} collection while \texttt{unbindAll} iterates over the contents of \texttt{addresses}. The bug is likely due to the wrong assumption that a SynchronizedSet instance guarantees atomic iteration over its elements. (The bug report is available as \texttt{bug 860} in JIRA issue tracker.)

Existing fixing techniques are context oblivious. They focus solely on the statements involved in the bug, without accounting for the context, i.e. the aliasing configuration of shared variables and locks that leads to the manifestation of the bug. We illustrate the consequences of context-oblivious fixing by reference to the two most recent and advanced fixing techniques.

The first, AFix \cite{liu11}, disallows concurrent execution of the statements involved in the bug in all contexts, including contexts that do not expose the bug. The AFix patch, shown in Figure \ref{fig:example}b, essentially encloses accesses to \texttt{addresses} inside an atomic region guarded by the \texttt{fixL} lock, which serializes evaluation of the statements even if the threads read different \texttt{addresses} collections and there is no threat of atomicity

\textsuperscript{\ast}\textsuperscript{\ast}http://mina.apache.org
In method bind(localAddr):
       synchronized (
this
) {
A:      ...
B:     
addresses
.add(localAddr);
       ... {
D:        strategy.unbind(addr);
         }
        }
  
addresses 
= Collections.synchronizedSet(new HashSet());

bind(localAddr):
 
 
synchronized(deadlockL){ 
       synchronized (
this
) {
A:      ...
         ...         }
        }
      }
  }
static Object 
fixL 
= new Object();
static Object 
deadlockL 
= new Object();

Any bug that AFix or Axis are able to remedy is also handled
when absolutely necessary. It is nontrivial to realize such
the fixes do not introduce new deadlocks.

The insight underlying our approach
is that quality fixes for concurrency bugs must account not
only for the buggy statements themselves, but also for the
execution context, so that mutual exclusion is enforced only
when absolutely necessary. It is nontrivial to realize such
closest to the execution context — that is, the most permissive synchronization —
that can be synthesized to simultaneously (i) fix the original
bug and (ii) ensure deadlock freedom is to (i) govern
transition into the code blocks accessing addresses with an
instance lock on addresses, which coordinates threads only
when they access the same addresses instance, and (ii) apply
mutual exclusion to avoid the deadlock iff two concurrent
threads have both this and addresses aliased.

Grail enforces this solution by creating synchronization objects of type string that capture the needed conditions, as the implementation of contextL in Figure 1d demonstrates. This factory method computes a hash value for both addresses and this, and concatenates these values. While hash values are not guaranteed to be unique, we rely on System.identityHashCode (abbreviated as hash), which suffices in practice. Note, importantly, the use of the intern method on the resulting string object ensures that if two synchronization objects have the same value, then they also point to the same memory. Otherwise strings representing the same context do not necessarily enforce mutual exclusion.

This solution cannot be gleaned by inspection of the buggy
statements alone. It requires contextual reasoning, which
accounts for the conditions under which the above bug manifests, as well as global reasoning, which accounts for the locks acquired by threads and the constraints they cast on

---

3Throughout this paper, deadlock freedom does not mean the whole-program deadlock freedom, instead, it means that the fixes do not introduce new deadlocks.
concurrency in every possible concurrent execution of bind and unbindAll. Our approach is born out of this insight, and implements this type of analysis model to simultaneously achieve both correctness and efficiency. A comparative study of our approach with two state-of-the-art approaches, AFix [11] and Axis [19], on 12 bugs from 8 popular Java applications (including e.g. Tomcat and Derby) indicates that this mode of reasoning is essential, leading — in most cases — to a performance improvement of over 40% compared to the competing approaches.

This paper makes the following principal contributions:
1. Context-aware bug fixing: We introduce a novel approach for fixing concurrency bugs, which factors into the fix precise modeling of contextual information, yielding a fix that is both optimal and provably correct.
2. Formalization and proofs: We provide full formalization of our approach, including proofs, highlighting interesting aspects of our modeling.
3. Implementation and evaluation: We have implemented a prototype system, Grail, that realizes our approach, and conducted experiments on 12 confirmed bugs in popular real-world applications to evaluate the quality of the fixes. The patches generated by Grail are almost all significantly more efficient than competing approaches while also being provably correct.

2. TECHNICAL OVERVIEW

In this section, we define the scope of our approach, walk the reader through a high-level description of the Grail workflow, and conclude by highlighting points of interest and limitations of our approach. When describing the workflow, we start with a simplified version, and later explain how we deal with challenges that we initially glossed over.

2.1 Scope

Similarly to past studies [6, 12, 19], our work addresses concurrency bugs involving two threads. According to a recent study [21], the vast majority (96%) of real-world concurrency bugs fall into this class. We concentrate on (any combination of) the following three bug categories: (i) data races [16, 33, 18, 34], (ii) atomicity violations [20, 36, 20] and (iii) deadlocks [13, 1]. These are described in more detail, as well as illustrated, in an accompanying technical report (TR) [1].

We assume that patches are in the form of additional mutual-exclusion-based synchronization. We preclude other transformations, such as code reordering, from consideration. In this setting, correctness translates into avoiding from introducing new deadlocks between pairs of threads, and optimality translates into introducing the least amount of synchronization needed to eliminate the reported bug.

2.2 Simplified Flow

We assume as input a multithreaded program along with bug reports given in terms of the involved code statements. (See Section 3.) For the given bugs, the first step of our technique is to model the lexical scopes enclosing the buggy statements as a “concurrent control-flow graph”. We soon explain how the graph is built, but first we start from the scopes it represents. These are essentially the methods containing the buggy statements along with their transitive caller and callee methods. This selected fragment of the entire program suffices for correct fixing of the bug, since all relevant lock acquisitions and releases are contained within this subprogram. By reasoning about them, we avoid from introducing new deadlocks between pairs of threads (Theorem 5.2). According to our experience, the resulting program fragment is relatively small and constrained.

The “flow graph” we mentioned earlier is, more accurately, a Petri net [55]. Stated simply, a Petri net is a graph containing two types of nodes: places, which denote statements; and transitions, which denote control flow between statements. A Petri net is thus similar to a control flow graph, except that control-flow edges are represented as transition nodes. In addition, the Petri net has a mathematical representation that permits formal modeling and reasoning about fix synthesis. To simulate thread execution, the Petri net maintains a (control-flow) token for each of the threads. The token is associated with a place and denotes the current statement a thread is executing. The state of a Petri net is tantamount to the token configuration, capturing a snapshot during concurrent execution of the respective threads.

Reasoning about multithreaded execution under the Petri net model amounts to combining the control-flow graphs of finitely many threads into a unified graph, which is essentially the product of the control-flow graphs. For our needs, it suffices to combine the graphs of two threads, such that every concurrent execution, as modeled by the Petri net state, corresponds to the product of two statements that are being executed by the threads. In the resulting graph, we distinguish between three kinds of concurrent executions:

- **Infeasible** execution refers to the concurrent execution (of a pair of statements) that cannot arise at runtime due to existing synchronization code. As an example, the control locations with labels A and D in Figure 1A cannot be visited simultaneously by two parallel threads when their this references are aliased.
- **Buggy** execution is the concurrent execution that arises during a buggy run, thereby exposing the bug. In our example in Figure 1A, the buggy pairs include (B,C), (B,D) and (A,C). (B,C) and (B,D) directly correspond to the reported bug involving addresses and expose the bug if the addresses variables accessed by threads are aliased. As for (A,C), the reason why this pair is classified as buggy is more subtle, involving its successors: These are (B,C), which is buggy if the addresses references are aliased, and (A,D), which is infeasible if the this references are aliased, and so (A,C) is buggy, if both addresses and this are aliased, by the fact that it can only lead to a buggy execution.
- **Safe** executions are all the remaining executions. That is, the set of all possible executions excluding the buggy and infeasible executions.

In the above, we reason about the buggy executions and the associated contexts (aliasing configurations) in which the bugs manifest. Reasoning about the buggy executions that lead to an input bug, such as (B,C), is relatively easy as they can be determined by analyzing the bug reports. However, reasoning about buggy executions that result from fixing of the input bug, such as (A,C), is less trivial. Based on the Petri net representation, we propose the automatic reasoning that is both contextual, systematically exploring and distinguishing aliasing configurations, and global, accounting for all synchronization concerns arising due to a concurrency bug.
This conveys all the needed information to precisely decide (i) how to avoid buggy executions in different contexts and (ii) how to ensure that the threads are not deadlocked in any context.

At the technical level, we achieve an optimal fix by encoding the safe and buggy executions as vectors and leveraging the theoretical framework of Nazeem et al. [27] for vector separation. As we assert in Theorem 5.1 and prove in the TR, the resulting vectors are linearly separable, and so a solution is guaranteed. Besides, our fix encodes the contexts so that it prohibits the buggy executions only in the relevant contexts. A comprehensive explanation is provided in Section 3 where we establish that our technique eliminates the original bug while guaranteeing deadlock freedom via fine-grained synchronization that is provably optimal.

2.3 Challenges

Our description above hides certain sources of complexity, and in particular, the question of how to synthesize an optimal bug fix given approximate and incomplete knowledge on aliasing between lock references. For example, we mention above that A and D are both governed by the same this lock, and thus the concurrent execution (A,D) is infeasible. This statement is clearly untrue if the this references in bind and unbindAll are not aliased.

In general, the problem of computing a precise aliasing solution is undecidable [31], which appears to stand in our way to a precise fix. Our strategy for addressing this inherent difficulty stems from the observation that in practice, due to the sensitive role of synchronization, a lock variable typically refers to the same lock object throughout the entire execution of the guarded lock region (i.e., it remains unchanged from the entry to the exit). Therefore, aliasing between the lock references accessed by two threads is, in general, immutable during their execution inside lock regions.

Guided by this assumption, we have implemented a divide-and-conquer approach, whereby (i) the baseline model is divided into multiple models, each corresponding to a specific aliasing configuration over the locks, (ii) the flow described above is carried out for each of these models fully precisely, and (iii) the resulting fixes are merged. Roughly speaking, the merging step fuses together two fixes if they both refer to the same set of buggy executions (i.e., statement pairs). The merge operation is equivalence preserving, yielding a bug avoidance condition that is the disjunction of the merged conditions.

2.4 Points of Interest and Limitations

Our technique features several pleasing properties. First, as we prove formally in Lemma 4.1 our technique exploits a small-model property, whereby a fix that is shown correct with respect to concurrent executions involving only a bounded number of threads remains correct in the presence of any number of threads. The intuition underlying this property is that every buggy execution has a corresponding “minimal” buggy execution, where minimality refers to the number of threads. This number can be upper bounded.

A second property that is worthy of mention is that unlike previous approaches that ensure correct fixing [19], our technique is non-iterative. A synchronization solution that both eliminates the original bug and guarantees deadlock freedom is computed at once, without the need to iteratively revisit and correct the solution by inserting more locks. For the code in Figure 4, our analysis simultaneously captures both the missing synchronization in accesses to addresses and the deadlock that would arise once synchronization around addresses is introduced, and addresses these concerns in a single algorithmic step.

We mention above the simplifying assumption of our approach, i.e., the aliasing between the lock references is immutable during the thread executions inside the lock regions. Our ability to guarantee correct fixing leans on this assumption, and so this is a clear limitation of our approach, which restricts its applicability. However, as we confirmed for all the benchmarks in our experimental suite (see Section 6), this assumption appears to be of little practical consequence. Besides, we also default to context-oblivious fixing if the context refers to the variables that are not accessible within the current method/block scope.

Another limitation is that our approach does not apply to ad-hoc synchronization [11] and does not apply to fix communication deadlocks [14].

3. BASIC WORKFLOW

The Grail algorithm accepts as input a program along with one or more concurrency bugs. A bug is specified as a tuple, which captures the involved statements and memory states. As an example, the tuple (s₁: f, s₂: f, s₃: f) would describe an atomicity violation, such that two statements, s₁ and s₂, that access field f are interleaved by statement s₃ that also accesses f.

First, we model the program as a Petri net, which is a well-studied modeling technique [6, 38, 19]. The Petri net model [38] is essentially a bipartite directed graph, as shown in Figure 2a, where a place (denoted as circle) models a statement, a transition between places (denoted as bar) models control flow between statements, and an arc connects a place and a transition. Specifically, the Petri net in Figure 2a models the running example in Figure 1a. The Petri net representation resembles an interprocedural control flow graph (ICFG), except that it models control flow edges explicitly as transition nodes.

As noted above, the Petri-net model we build represents only the relevant fragment of the program. We assume that synchronization is expressed as synchronized(...){ ... } blocks. This means that all lock acquisitions and releases that arise during a buggy execution occur within transitive callers and callees of the methods enclosing the buggy statements. Therefore, modeling only the methods transitively calling, and called from, the buggy methods suffices for the purpose of our analysis. Restricting the scope in this way is a significant performance and scalability optimization, which enables Grail to handle bugs in large-scale real-world programs.

Then, we encode the buggy executions in the Petri net. The Petri net model simulates thread execution using tokens. The token transition from a place to a successor place, e.g., from place A to place B in Figure 2a, represents an execution step by the corresponding thread. The snapshot of all tokens in the Petri net represents the concurrent execution state of all threads. Formally, we refer to the snapshot as a Petri net state, and denote it in (i) vector form, e.g., [1, 0, 1, 0], where each entry records the number of tokens in a place (following the fixed order ABCD), or if the vector is binary, (ii) short form, e.g., [AC], which records only the places that contain a single token. Specifically, the empty state contains zero tokens in each place, denoting the initial execution or
the final execution. We use the terms thread, statement and concurrent execution, and their Petri net counterparts, token, place and state, interchangeably.

We discover all the buggy executions through context-aware analysis, which is specific to a unique execution context. Section 4 explains how we systematically model the contexts. For now, let us assume we already have a context, which fully resolves (i) the memory state conditions, i.e., the aliasing among shared fields (variables), and (ii) the synchronization state conditions, i.e., the aliasing among locks. More concretely, we assume the current context is $\text{addresses}_{\text{th1}} = \text{addresses}_{\text{th2}} \wedge \text{this}_{\text{th1}} = \text{this}_{\text{th2}}$ for our running example. Here, we assume that only two threads $\text{th1}$ and $\text{th2}$ are running. Section 4 discusses thread modeling in more detail. Our context-aware analysis first finds the buggy and infeasible executions, then the executions that would become buggy due to the fixing.

**Buggy Executions.** Given an atomicity violation, we automatically extract the buggy concurrent executions [35], and the memory state conditions associated with them such that the bug manifests. We refer to this condition as the manifesting condition. In our running example (Figure 1a), the buggy executions include $[BC]$ and $[BD]$, denoting that the atomic loop, which includes $C$ and $D$, is interleaved by $B$. Besides, the manifesting condition is $\text{addresses}_{\text{th1}} = \text{addresses}_{\text{th2}}$, which holds in the current context under analysis. We also explain the automatic extraction of other types of bugs in the TR.

**Infeasible Executions.** Some executions are infeasible as they violate the constraints imposed by the original locks. Given an execution context where lock aliasing is fully resolved, Grafl applies the lockset algorithm [26] to find the infeasible execution. The pair of statements in it are guarded by aliased locks. For example, the concurrent execution $[AD]$ in Figure 1a is infeasible in the current context because the this locks guarding $A$ and $D$ are aliased.

**Potential Buggy Executions.** After the buggy executions are eliminated due to the bug fixing, some safe executions may become buggy. We refer to such executions as potential buggy executions. In our running example (Figure 1a), after the buggy $[BC]$ execution is eliminated, the concurrent execution $[AC]$ may result in a deadlock. Given the buggy/infeasible executions, we automatically discover the potential buggy executions through fixpoint computation over the state transition graph (STG) [25]. Specifically, the STG — as shown in Figure 2a — captures the transitions (edges) among all states (vertices). Each step in the computation finds new potential buggy states by applying Lemma 3.1.

**Lemma 3.1.** A state is a potential buggy state iff it does not have any safe child state in the state transition graph.

**Proof Sketch.** For a state $s$ with no safe child states, the child states are eventually eliminated by the fix and become infeasible, leaving $s$ unreachable to any state. This means the state $s$ corresponds to a deadlock in the Petri net [25].

**Fixpoint Computation.** The STG in Lemma 3.1 is constructed by recursively exploring the child states. Assume the input set $S_1$ of buggy states in the STG. The first iteration would yield a new set $S_2$ of buggy states. The next iteration would then treat $S_1 \cup S_2$ as the input set, etc. Starting from the initial state (empty state), a state transitions to another state (denoted by $\rightarrow$) in two cases: (1) A new token/thread enters the Petri net, e.g. $[A] \rightarrow [AC]$; or (2) an existing token/thread moves to the next place, e.g. $[AC] \rightarrow [BC]$. The input buggy/infeasible states are used to avert state explosion: We stop exploring the children of a buggy/infeasible state. As an example, the buggy/infeasible state in Figure 2b does not have children. This optimization does not affect the correctness or performance of our fixes as the states reachable only through the buggy/infeasible state will be rendered unreachable by fixes that we eventually add. The fixpoint computation always terminates with a solution (See our TR).

The fixpoint computation is complicated by branch and loop semantics. Given a branch structure, depending on the branch condition, a state $s$ may follow the left branch to transition to child state $s_{\text{left}}$ or the right branch to transition to another child state $s_{\text{right}}$. We refer to them as conditional transitions. Determining which branch is taken is undecidable statically, and so we conservatively assume the worst: A conditional transition leading to a buggy state always occurs. Formally, a state is discovered as buggy if (1) at least one conditional transition leads to a buggy/infeasible state, and (2) all transitions that are not conditional lead to buggy/infeasible states. In our example, the state $[AC]$ in Figure 2b is buggy because one of its conditional transitions (dotted line) leads to an infeasible state $[AD]$, and the transition that is not conditional leads to the buggy state $[BC]$.

**Last, we carry out the actual fix synthesis.** Having found the safe states $A$ and the buggy states $\mathcal{F}$, and the conditions associated with them, we automatically synthesize the fixes in two steps: (1) We find the linear constraints separating the buggy state vectors from the safe state vectors, i.e. the constraints satisfied by the safe states but violated by the buggy states. (2) We enforce the linear constraints by producing fixes.

We reuse existing frameworks: For (1), separating two sets of points (vectors) with hyperplanes (linear constraints) in high-dimensional space is a well-studied problem in geometry [27]. Given $\mathcal{F}$ and $A$, we apply Mixed Integer Programming (see [27], equations 36-42 at page 11) to separate $\mathcal{F}$ from $A$. This results in the minimum set of separating constraints. Constraints are in the form of linear inequality, $l s \leq b$, where $l$ is a column vector of non-negative integer weights and $b$ is a non-negative integer. For example, the constraint $s(A) + s(C) \leq 1$ essentially states that the total number of tokens/threads in the places/statements $A$ and $C$ should be at most 1. For (2), given the input linear constraints, we apply the Supervision Based on Place Invariants (SBPI) framework [35, 19] to compute the Petri net additives that realize the constraints in the net. The additives are new control places connecting with some existing Petri net transitions and blocking some token movement, which behave equivalently to new locks. We implement them as locks in the code, of which the locking operations are placed at the control flow edges modeled by the transitions to block certain thread executions. We insert the locking operations via instrumentation, similarly to existing approaches [35, 19]. More details appear in the TR.

The locks computed above are, however, oblivious to the manifesting conditions. They always prohibit the concurrent execution $[AC]$ in Figure 2a. We refine them so that they prohibit the concurrent execution $[AC]$ only when its manifesting conditions are satisfied at runtime. In the example
of Figure 2d. Grail achieves this fine-grained synchronization by creating a unique string object out of the arguments of
lock(...). We then use the built-in monitor of the string object for locking (Unsafe.monitorEnter(string)). A more natural transformation is to enclose the buggy code in a synchronized block. Grail offers this option when the transformation does not change the locking order, which is typically the case. We discuss other forms of optimizations in the TR.

A point of interest is that we do not encode the inequality checking in the fix, which is in fact safe. Informally, the inequality condition exposes the buggy executions that are infeasible in the equality condition. For example, the execution [BD] in Figure 2b is buggy in the inequality condition thisth1 ≠ thisth2, but is feasible in the equality condition. Without checking the inequality, the fix may exert the synchronization even in the equality condition, i.e., when the execution is infeasible. This does not affect the correctness.

4. FULL ALGORITHM

In this section, we address remaining challenges, including specifically (i) modeling of unbounded concurrency and (ii) missing may-alias information.

Bounded Thread Modeling. In general, only one token — representing a single thread — is created to execute a Petri net. To model concurrent execution of the same code by parallel threads, we create two tokens that move inside the same Petri net. Lemma 4.1 asserts that this form of bounded modeling suffices to address all the "minimal" buggy concurrent executions, and thus all concurrent executions in general.

**Lemma 4.1.** For the bugs considered in Section 2.7, all buggy concurrent executions are eliminated if all the minimal buggy concurrent executions are eliminated.

**Proof Sketch.** Given our declared scope, any buggy concurrent execution using three or more threads has a corresponding minimal buggy execution involving two threads. The former execution is automatically eliminated by the constraints constructed to eliminate the minimal buggy execution, e.g., $s(B) + s(C) \leq 1$, following the monotonicity of the lefthand functions (with non-negative coefficient) of the constraints.

**Lock Aliasing.** Computing the aliasing relation precisely is undecidable in general [31]. We rely on lock aliasing information to identify infeasible executions, and subsequently analyze potential buggy executions. As explained in Section 2.3, we place the assumption that the aliasing relationship between a pair of lock references does not change during the thread executions inside the guarded lock regions. We then iterate over all possible configurations of may-alias lock pairs while skipping invalid aliasing configurations, such as $l_1 = l_3$ & & $l_1 = l_3$.

Splitting and Merging STGs. As different execution contexts lead to different classification of buggy/safe states, we create a unique STG per execution context. We refer to this as the splitting process. Splitting happens in two phases: (1) Before STG construction, we fully resolve the may-alias lock pairs, each configuration corresponding to a unique set of infeasible executions. We create one STG per configuration and annotate it uniquely. (2) During STG construction, when different memory state conditions lead to different classifications of a state, we split the STG into two clones, one for the condition and one for its negation. The workflow in Section 2b is carried out within each STG, producing the buggy and safe states associated with the execution context. Figure 2d (upper part) summarizes the results for different STGs, one per line (buggy/safe states in left/right box), with the execution context shown above.

Prior to fix synthesis, to reduce the number of fixes as well as the number of conditions in each fix, merging is applied. Specifically, if a state is buggy in “twin contexts” — i.e., contexts that are exactly identical except at one condition — then we fix the state in the disjunction of the contexts. For example, the state [BC] is buggy in both in context addressesth1 = addressesth2 & thisth1 = thisth2 and in its twin context addressesth1 = addressesth2 & thisth1 ≠ thisth2. With merging, we merely need to fix context addressesth1 = addressesth2 and fix once. we copy the safe states from both of the twin contexts to their disjunction to ensure that fixes do not sacrifice any safe states. The merging process is iterative.

**Preprocessing.** We apply a known transformation [11] [10] to preprocess the input bugs to ensure the correct lock placement, i.e., the paired lock/unlock in each method.

5. FORMAL GUARANTEES

In this section, we formally state performance and correctness guarantees. We also discuss limitations of our approach.
Theorem 5.1 (Optimality). Assuming the fix can only introduce mutual-exclusion-based synchronization, Grail generates optimal fixes, in that mutual exclusion is applied only in the exact scenarios when the bug may manifest.

Proof Sketch. Our fixes prohibit only the buggy concurrent executions, as guaranteed by Grail’s underlying theoretical framework for precise vector-based fixpoint analysis, vector separation and constraint realization (Section 3). Further, buggy executions are only prohibited when their manifesting conditions are satisfied, as ensured by runtime context checking. The manifesting conditions are calculated precisely, and the merging step in Section 4 neither strengthens nor weakens the conditions associated with a buggy state. Note that vector separation always has a solution as the vectors are linearly separable [27]. We prove this in the technical report.

Practical Issues. In practice, there are several sources of conservativeness: (1) In the implementation of the context-aware synchronization, the hash API in the helper function lock(), which is essentially System.identityHashCode(arg), may return the same value for two different argument objects, which leads to unnecessary blocking of some safe executions. However, this API is normally able to enforce a uniform distribution. (2) Preprocessing of the input bugs essentially enlarges the scope of the atomic region, which could change certain safe states into buggy states. (3) In the technical report, we describe the “locking-map” optimization that reduces the number of locks. This could lead to unnecessary blocking. In terms of overall performance, neither (1) nor (2) lead to any observable overhead, as anticipated, while (3) has been introduced deliberately as an optimization. Indeed, our experimental data suggests that Grail fixes incur a slowdown of merely 2%, which is negligible.

Theorem 5.2 (Correctness). With the Grail fix applied, the input bug does not manifest at runtime, and equally importantly, the fix does not cause new deadlocks to arise between any pair of threads.

Proof Sketch. Deadlock freedom (over pairs of threads) is guaranteed by the fact that the Petri net models all relevant lock operations. Reasoning applied to the model detects, in addition to the buggy states pertaining to the input bugs, also the potential buggy (deadlock) states.

6. Evaluation

In this section, we describe the experimental results we obtained by comparing Grail, a tool realizing our approach, with the Axis and AFix algorithms on real-world bugs.

6.1 Prototype Implementation and Benchmarks

We implemented Grail atop the Soot compiler framework [4]. For Petri net model extraction, it is fully automatic. Similarly to Wang et al. [38], we implement it using the depth-first traversal of the control flow graph. For MIP calculation (see Section 3), Grail uses the popular Ip_solve mixed integer linear programming solver [8] which it references as a shared library invoked via the JNI interface. Our current Grail prototype is applicable to Java programs.

For our evaluation, we focused on bugs in popular applications. Most of the bugs, e.g., Derby-3260, were originally reported into the Apache JIRA bug tracker [7], whereas other bugs were detected by automated tools [9]. Our benchmark suite includes the large applications such as the Lucene text search engine, the Derby database engine, the Tomcat web container and the Jigsaw web server. The details of the bugs are characterized in our TR.

6.2 Experimental Setting and Methodology

To compare between the different fixing approaches, both in general and in reference to the original (buggy) code, we built a test harness for each of the benchmarks. The harnesses are all based directly on the test cases exposing the bugs, which are either provided as part of the application package or kept at the JIRA bug tracker. We applied only minor changes to the test cases for the purpose of our experiments, such as (i) changing the number of active threads or (ii) changing loop bounds for worker threads. We also make publicly available (https://sites.google.com/site/grailfixing/) the performance test drivers that contain our fix and fixes produced by other approaches, so that the readers can reproduce the reported data.

All measurements were conducted using an x86_64 Thinkpad W530 workstation with eight 2.30GHz Intel Core i7-3610QM processors, 16GB of RAM and 6MB caches. The workstation runs version 12.04 of the Ubuntu Linux distribution, and has the Sun 64-Bit 1.7 Java virtual machine (JVM) installed.

The performance numbers we report below represent the average across 20 independent runs of the test driver. We also validated, as part of our methodology, the correctness of our Grail implementation by confirming, for each of the fixes generated by Grail, that (i) the original bug is no longer present, and (ii) no deadlocks were introduced as a consequence of our patch. For (i), we applied the bug detection tool [20, 9] to the patched version. We also attempted replay of the bug on the patched version [10]. As expected, we could not reproduce any of the bugs. For (ii), we applied the deadlock detection tool [13] to the patched version. Again, we could not detect any deadlocks due to the patch.

6.3 Performance Results: Fixing and Parallel Execution

Our experience with the bugs in our suite indicates that the Grail fixing algorithm is efficient. For all the benchmarks in our suite, Grail requires less than 1.5 seconds to complete, and exhibits negligible memory and resource consumption. This is correlated with the small Petri net models that Grail builds internally, each of which contains less than 1000 places and leads to fewer than 4000 states.

We are further encouraged by the performance results shown in Figure 3. The graphs depict running time in milliseconds (the Y axis) as a function of the concurrency level (i.e., number of threads; X axis) for the original program as well as its versions patched by AFix, Axis and Grail. We point to two main trends that emerge from the graphs:

- Across the entire set of benchmarks and bugs that we studied, the Grail version of the code and the original code cluster together, while the Axis and AFix versions form a separate cluster. In certain cases (most noticeably, Lucene-651, but also Derby-5561), the fix

\[ \text{http://lpsolve.sourceforge.net/5.5/} \]

\[ \text{http://www.sable.mcgill.ca/soot} \]

\[ \text{http://lpsolve.sourceforge.net/5.5/} \]
due to Grail leads to better parallelism and scalability! As we discuss below in detail, this is because certain atomicity violations manifest as performance bugs.

- In 11 out of the 15 cases, (all except Derby-3260, Tomcat-50885, Jigsaw and Weblech), there is a visible performance gap and scalability gap between Grail and its competing fixing approaches, AFix and Axis. For some of the benchmarks, the synchronization synthesized by Axis and AFix causes linear performance degradation, e.g., in the application Cache4j.

In summary, the Grail fixes mostly outperform both Axis and AFix. The performance gap in these cases is greater than 40% on 8 threads. The overhead added by the Grail fix compared to the original version is typically negligible, ranging around 2%, and in some cases Grail is able to outperform the original problem by eliminating performance problems due to insufficient synchronization. These results are highly encouraging, suggesting that the fixes generated by Grail are not only optimal in theory but also efficient in practice, and in particular, they are significantly better than previous fixing approaches. This validates the main hypothesis of this paper, which is that a contextual and global analysis model is required for quality bug fixing.

6.4 Detailed Discussion

Beyond the high-level trends outlined above, we now provide in-depth analysis of each of the bugs and its corresponding fixes. We discuss each of the benchmarks in turn.

**Log4j-24159.** [Log4j-24159](a) is a deadlock. As Figure 3a shows, it occurs when one thread first acquires lock c and then tries to acquire a to invoke the synchronized method `getPersistedState`, and another thread acquires a when entering `setPersistedState` and then tries to acquire c. Axis fixes the deadlock internally by applying the Gadaleta approach [19]. AFix cannot fix deadlocks. We collected data only from runs where the deadlock did not manifest. For these runs, our fix incurs ≤ 2% overhead, and outperforms the Axis version by more than 60% when there are more than 4 threads. Under the hood, our fix coordinates the threads only when they use the same c and a locks, while the Axis fix always coordinates the threads, being oblivious to the locks used and making the execution almost sequential.

**Apache Mina-869.** The bug Apache Mina-869 is the one that we show in Section 1. The performance results in Figure 3b show that our fix incurs less than 5% overhead, outperforms the other fixes (AFix/Axis) by 40% when there are 8 threads or more. The improvement clearly demonstrates that our bug fixing, underpinned by the contextual and global reasoning framework as explained in Sections 3-5, produces the very lightweight fix.

**Cache4j Bug.** As shown in Figure 3c, our fix introduces negligible overhead, which outperforms the AFix/Axis fixes by at least 50% and by up to 93% when the number of threads increases to 16. Under the hood, the interleavings shown in Figure 3c cause the `InterruptedException` at line 6 (reported by Sen et al. [33]), which can be avoided by preventing the access of `_sleep` at line 1 from interleaving the accesses at line 4 and line 9. Our fix, which is aware of the runtime program states, coordinates two threads only when their accessed `_sleep` fields belong to the same object, only in which case the bug could manifest. Comparatively, the AFix or Axis fixes coordinate them whenever they reach the code region, even if they access different `_sleep` fields. As a consequence, the threads have to unnecessarily wait for the thread that sleeps while holding the fix lock.

**Derby-3260.** Similarly, to fix the atomicity violation in Figure 3d which leads to `NullPointerException` (NPE) as reported in bug Derby-3260, our fix coordinates the threads only when they invoke the APIs on the same `PreparedStatement` instance `ps`, as the invocations upon different `PreparedStatement` instances are commutative and never cause the incorrect composition of `rePrepare()` and `getActClass`. Here, the incorrect composition [20] stands for the composition intended as atomic but implemented incorrectly as non-atomic. It is a special kind of atomicity violation and manifests in the condition that the invocations are not commutative. The commutativity property [17] among invocations is explained in TR. Comparatively, the AFix/Axis fixes are oblivious to the condition and always coordinate the threads that execute the relevant code. The performance results in Figure 3e shows that our fix incurs only 1% overhead and it outperforms the AFix/Axis fixes by around 14%.

**Derby-5561.** Our approach adds the synchronizations to fix the atomic violation bug Derby-5561, of which the non-atomic interleavings are shown in Figure 3f. The added fix coordinates the threads only when they access the same field `physical`, which represents the shared physical network.
connection. Note that one access of the field is originally inside the invocation of the method checkNull, it is replaced as the invocation following the preprocessing (Section 4), and the field accessed is replaced as its equivalent this.physical at the invocation site accordingly. As shown in Figure 4e, our fix usually incurs 2% overhead, while the AFix/Axis fixes incur around 11% (when there are more than 8 threads). In addition, there are 15 other methods that may be interleaved non-atomically by the statement at line 1. Our approach produces the fixes that systematically encode sets of conditions to avoid all the atomicity violations.

**Ftpserver Bug.** Our fix of the Ftpserver bug in Figure 3e coordinates two threads only when they invoke upon the same instance props. In other words, our fix allows the multiple server threads to independently manage the information of the hundreds of incoming users, while the fixes produced by existing approaches disallow the concurrency unnecessarily. The performance results in Figure 4f show that, when the number of threads is 8 or exceeds 8, our fix incurs around 5% overhead, while the AFix/Axis fixes incur more than 73% overhead and they even double the execution time in the presence of 12 threads. Our fix outperforms the AFix/Axis fixes by around 40%.

**Lucene-481.** The Lucene-481 bug, shown in Figure 3f, leads to the FileNotFoundException when one thread first deletes the file to (line 2) and then creates it using the method renameTo, while the other thread interleaves to open the file to. To the best of our knowledge, existing automatic tools cannot find the bug as it involves the shared file in the disk, but we anticipate the emergence of tools that can find it, given its severity. The bug is an incorrect composition. We manually specify the commutativity property among the invocations, i.e., the invocation of the method open is commutative with the invocation of the delete or renameTo only when their involved parameters (including the implicit this parameter) are completely different. Leveraging the condition, our fix coordinates the threads only when the invocations share common parameter instances and does not coordinate them otherwise. The fixes produced by existing approaches always coordinate the threads. The performance results in Figure 4g show that our fix incurs only 2% overhead. Our version is faster by 40% than the version patched by existing approaches, which incurs around 70% overhead.

**Lucene-651.** Lucene-651 (Figure 3g) is reported in the JIRA tracker as the performance bug, rather than the functionality bug. Specifically, the code at line 1 and line 5 needs to be composed atomically, as proposed by the bug reporter, so that only the first thread, that fails to find in the cache the records specific to a Field instance f, needs to conduct the expensive computations at line 4. Otherwise, if the atomicity violation occurs, multiple concurrent threads may have to conduct the expensive computations. According to the commutativity specification [8, 17] of the cache, the atomicity violation manifests only when different threads involve the same Field instance f at the caching operations (line 1 and line 5). Accordingly, our fix coordinates the threads only in the condition. First, we evaluate the scenario in which different threads share the same cache: As shown in Figure 4h, our patched version, which fixes the performance bug, is faster than the original version by 40%. Meanwhile, the version patched by existing approaches also outperforms the original version, by around 30% when there are 4 or 8
threads. Our version outperforms their version by up to 36% (16 threads), due to that our fix does not impede the concurrency among the threads that retrieve the records that relate to different Field instances. Second, we evaluate the scenario in which different threads do not share the cache: As shown in Figure 11, the version patched by AFix/Axis is slower than our version by around 60% when there are more than 8 threads. The slowdown is unnecessarily caused by the unconditional coordination among threads.

Tomcat-37458. To fix the atomicity violation bug Tomcat-37458 that causes the NPE as shown in Figure 3i, our fix coordinates the threads only when they access the same field manifest. In other words, our fix allows multiple threads to load classes from different jars that contain different manifest files. Comparatively, existing approaches produce the fixes that serialize the executions by different class loader threads. As shown in Figure 16, our fix incurs 5% overhead on average and up to 9% when there are 16 threads. The version patched by existing approaches is slower than our version by 13%-153%. The slowdown grows dramatically when the number of threads increases.

Tomcat-50885. Figure 3j shows the atomicity violation bug Tomcat-50885 in Tomcat: When the two accesses of the JspServlet instance _servlet by a thread are interleaved non-atomically by a thread that destroys the instance, the exception occurs. Our fix coordinates the threads if they involve the same _servlet instance, while existing approaches coordinate them unconditionally. As shown in Figure 16, our fix incurs typically 4% overhead. As compared to our fix, the fixes produced by existing approaches are slower by 40% when there are 12 threads and slower by 14% when there are 16 threads. Given that the code (lines 1-3) lies in the core component of Tomcat and is frequently executed, the developers hesitate to fix those bugs in the component and obviously cannot afford the around 40% overhead caused by the conservative AFix/Axis fixes.

Tomcat-53498. The Tomcat-53498 bug, as shown in Figure 3k is due to the incorrect composition of the invocations of the methods contains() and remove(), both from the thread-safe class ConcurrentHashMap. The bug may cause the variable r to be null and cause the NPE in the later use. According to the commutativity specification 5 of the Map, the atomicity violation manifests only when the threads involve the same map and the same key at the Map operations. Comparatively, the fixes produced by existing approaches coordinate the threads once they reach the code. Correspondingly, when different threads do not share the map, as shown in Figure 11, our fix incurs up to 5% overhead and outperforms other fixes by 60% (when there are more than 8 threads); when different threads share the map, as shown in Figure 11, our fix incurs 10%-17% overhead when there are more than 8 threads, and outperforms other fixes by 50%. The AFix/Axis version takes almost 2X running time as ours. The overhead of our fix is caused by the coordination when different threads access the same key, and the improvement over other fixes is due to the concurrency allowed among the map operations upon different keys.

Webblech Bug. Figure 3k shows the incorrect composition in Webblech, reported by Lucia et al. 22. The invocations of q.size() and q.dequeue() are not composed atomically. As a result, the interleaving update may cause the invocation of q.dequeue() to throw the exception. Our fix encodes the commutativity condition needed by the bug to manifest. Figure 1n shows the evaluation results, where the original version and different patched version run similarly, which indicates the quality of the fixes does not matter.

Bugs detected in Jigsaw. We also apply our tool to all the atomicity violations automatically detected 9 in Jigsaw. Our tool successfully computes the fixes, given the large number (754) of input bugs, which actually correspond to the combinations of statements from very few methods. Figure 10 shows the performance comparison with the Axis version (The AFix version frequently deadlocks 19). Our fix incurs negligible overhead and outperforms the Axis version by 15% when there are more than 10 threads. To better understand the improvement over Axis, we break down the overhead of the fixes by grouping the bugs that share the same container method and fixing each group of bugs separately. It is interesting that fixing certain group of bugs using both approaches do not make any differences. We also identify the main source of improvement over the Axis version: Our approach differs from Axis in fixing the violation of the atomicity between store.lookupResource() and store.loadResource in the method NewStoreEntry.lock(). One should be able to observe obvious slowdown (close to 10%) by introducing a stationary object to unconditionally protect the two invocations in an atomic region, which explains why the Axis version incurs high overhead.

7. RELATED WORK

Many solutions have been proposed to fix concurrency bugs. There are dynamic methods 32, 33, 30, 11, 12 that integrate dynamic bug detection and fixing. On the other hand, static methods patch the program offline and avoid the runtime bug detection overhead. Our method belongs to this category. In addition, Weeratunge et al. 39 insert additional synchronizations to enforce the inferred atomicity. Besides, researchers also propose methods to address concurrency bugs that are not discussed within this paper, e.g., order violations 12 and type-state violations 23, 44.

Another related area of research is code synthesis, including lock synthesis 24, 46, specification-based control synthesis 6, 12, 47 and fix synthesis for semantic bugs 30, 29. Within this area, there are works that share our motivation of ensuring a maximally permissive solution. One example is Vechev et al.’s study on conditional critical regions (CCRs) 24, where the challenge is to find appropriate guard expressions for atomic sections. Vechev et al. present a greedy algorithm that — given program P satisfying specification S and language LG of guards — infers under certain conditions a program P’ that is maximally permissive w.r.t. LG and also satisfies S.

8. CONCLUSION

We have presented Grail, a context-aware fixing algorithm for concurrency bugs. Unlike previous techniques, Grail is able to guarantee both correctness and optimal performance. The insight is that Grail accounts for both the synchronization and execution contexts in which the bug arises. We have confirmed the advantages of Grail compared to existing approaches in a study over 12 real-world bugs.
References


