Towards Generating Thread-Safe Classes Automatically

Haichi Wang
wanghaichi@tju.edu.cn
College of Intelligence and Computing
Tianjin University, China

Zan Wang
wangzan@tju.edu.cn
College of Intelligence and Computing
Tianjin University, China

Jun Sun
junsun@smu.edu.sg
School of Information Systems
Singapore Management University, Singapore

Shuang Liu*
shuang.liu@tju.edu.cn
College of Intelligence and Computing
Tianjin University, China

Ayesha Sadiq
ayesha.sadiq@monash.edu
Monash University, Australia

Yuan-Fang Li
yuanfang.li@monash.edu
Monash University, Australia

ABSTRACT

The existing concurrency model for Java (or C) requires programmers to design and implement thread-safe classes by explicitly acquiring locks and releasing locks. Such a model is error-prone and is the reason for many concurrency bugs. While there are alternative models like transactional memory, manually writing locks remains prevalent in practice. In this work, we propose AutoLock, which aims to solve the problem by fully automatically generating thread-safe classes. Given a class which is assumed to be correct with sequential clients, AutoLock automatically generates a thread-safe class which is linearizable, and does it in a way without requiring a specification of the class. AutoLock takes three steps: (1) infer access annotations (i.e., abstract information on how variables are accessed and aliased), (2) synthesize a locking policy based on the access annotations, and (3) consistently implement the locking policy. AutoLock has been evaluated on a set of benchmark programs and the results show that AutoLock generates thread-safe classes effectively and could have prevented existing concurrency bugs.

1 INTRODUCTION

Concurrent programs are prevalent these days due to the pervasive availability of multi-core and many-core systems. Concurrency bugs are undesirable outcomes that arise when two programs execute concurrently but do not show up if execute sequentially. They are notoriously hard to detect and fix. Existing research mostly focuses on bug detection, e.g., for data races [1–5], deadlocks [6] and atomicity bugs [7]. Recently, there have been studies explore to automatically fix concurrency bugs [8–12]. While the studies have shown impressive performance in some cases, they do not address the fundamental problem, which is to develop techniques that are capable of *preventing* concurrency bugs systematically in the first place.

The existing concurrency model for Java requires programmers to design and implement thread-safe classes based on synchronization primitives like acquiring locks and releasing locks. Such a model is error-prone and is the reason for many concurrency bugs. In theory, a programmer should know precisely what specification a thread-safe class should satisfy; and above all make sure the program works correctly in a sequential environment. Afterwards, the programmer needs to derive a locking policy which systematically guarantees that the specification is satisfied when the program is used in a concurrent environment, and finally consistently implements the locking policy throughout the program. This process, however, is often flawed in practice. First of all, it is impractical to assume the availability of a full specification. There have been multiple attempts on getting programmers to write specifications [13–15], and yet specifications are scarce in practice. Second, systematically deriving a locking policy requires information on how variables are accessed, which could be complicated due to complications like aliasing and instance escaping. Lastly, consistently implementing a locking policy takes good discipline as well as systematic tracking of where and how variables are accessed. There have been proposals on alternative concurrency models which aim to solve the problem, e.g., transactional memories [16], and yet the Java model remains very much relevant in practice.

In this work, we aim to prevent concurrency bugs by fully automating the process of generating thread-safe classes from sequential programs. That is, given a class which is assumed to be correct in a sequential environment, we automatically generate a thread-safe class, and do it in a way without requiring a specification of the class. Our approach takes three main steps: (1) infer access annotations through static analysis, (2) synthesize a locking policy based on the access annotations, and (3) consistently and automatically implement the locking policy. Furthermore, we optimize the generated class safely (without breaking linearizability [17]) by reducing the scope of locking. For correctness, we prove that the generated class is linearizable (modulo some assumptions) to the sequential class and is always deadlock-free.

Our approach is implemented as a self-contained toolkit called AutoLock for Java programs, and empirically evaluated on a set of 45 benchmark programs (with a total of 64,447 lines). AutoLock is evaluated to show that it could have been applied to prevent known concurrency bugs. Furthermore, AutoLock is efficient, i.e., on average, it takes only 0.48 seconds to generate a thread-safe class.

*Shuang Liu is the corresponding author.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416625

In addition, for 8 data structures from the Java Development Kit, we compare the thread-safe versions generated by our approach with the ones crafted by experts of domain to show that our generated versions are reasonably efficient (and provably correct).

The rest of the paper is organized as follows. Section 2 defines the research problem. Section 3 presents an overview of our AutoLock method. Section 4 describes the access annotation inference approach. Section 5 and Section 6 present our approach on inferring and implementing a locking policy based on the inferred access annotations. Section 7 discusses the evaluation of the proposed technique for realistic Java programs. Section 8 reviews related work. Finally, We conclude the proposed technique in Section 9.

2 PROBLEM DEFINITION

We assume that the user-provided sequential class is in the form of a tuple $\mathcal{P} = (Var, PubM, PriM)$ where Var is a finite set of mutable variables; $PubM$ is a finite set of public methods; and $PriM$ is a finite set of private methods. Each method m takes an optional sequence of input parameters, possibly updates the variables in the class, and produces certain output. Note that we assume that variables are only accessed through methods. A method invocation by a thread t is written as $inv(x, m, a^*, t)$ where x is the name of an object of class $\mathcal{P}$; m is the method name; a^* is an optional list of parameter values; and t is a thread identifier. A response to a method invocation is written as $res(x, m, o, t)$ where o is the returned value. A method invocation and a response match if and only if their object names agree, their method signatures agree, and their thread identifiers agree. When it is clear from the context, we omit x to save space.

Without loss of generality, we assume that $\mathcal{P}$ is sequentially correct with respect to a specification $\mathcal{S}$, i.e., a single thread which calls any public method in the class through any object of $\mathcal{P}$ in arbitrary order always satisfies the specification $\mathcal{S}$. Ideally, given a sequential program $\mathcal{P}$ and its specification $\mathcal{S}$ (which is satisfied by $\mathcal{P}$), we aim to construct automatically a program $\mathcal{Q}$ such that $\mathcal{Q}$ is thread-safe with respect to $\mathcal{S}$. That is, multiple threads can call any public method in $\mathcal{Q}$ through an object of $\mathcal{Q}$ concurrently and $\mathcal{S}$ is always satisfied. As discussed above, it is often infeasible to obtain $\mathcal{S}$ in practice. In this work, we do not assume that $\mathcal{S}$ is known and instead aim to generate $\mathcal{Q}$ such that it satisfies three desirable properties: linearzability, deadlock-freeness and efficiency.

In the following, we formalize our assumptions and then define the correctness requirements. Without loss of generality, we focus on a single object obj of the class $\mathcal{P}$. A history of method invocations on obj is a finite sequence of events $\pi = \langle e_1, e_2, \cdots, e_n \rangle$ where e_i is either a method invocation or a response. π is sequential if it starts with an invocation, and every invocation is followed immediately with the matching response (or the invocation is the last event). Otherwise, we say it is concurrent. A sequential history has no real concurrency since the methods' execution never overlaps. We say that π is deadlock-free if every invocation in π can be followed by a matching response eventually. Our assumptions are as follows. For every sequential history π of $\mathcal{P}$, π satisfies $\mathcal{S}$ and π is deadlock-free.

Given an arbitrary (sequential or otherwise) history π, we write $\pi \restriction t$ where t is a thread identifier to be the projection of π on t, i.e., all those invocation and response concerning thread t are removed. The following defines a correctness criterion for concurrent programs which is widely adopted in the community [18].

Definition 2.1 (Linearizability). Given two programs $\mathcal{P}$ and $\mathcal{Q}$ which share the same set of public methods, we say $\mathcal{Q}$ is linearizable with respect to $\mathcal{P}$ if and only if, for all history $\pi = \langle e_1, e_2, \cdots, e_n \rangle$ of $\mathcal{Q}$, there exists a sequential history π' of $\mathcal{P}$ such that

- $\pi \restriction t = \pi' \restriction t$ for all threads t; and
- $res(m, o, t)$ precedes $inv(m', a^*, t')$ in π' only if $res(m, o, t)$ precedes $inv(m', a^*, t')$ in π.

Intuitively, a history is linearizable if, from each thread's point of view, each method invocation behaves as if it is executed immediately. We say that $\mathcal{Q}$ is linearizable to $\mathcal{P}$ iff every history of $\mathcal{Q}$ is linearizable to some sequential history of $\mathcal{P}$. Intuitively, since by assumption any sequential history of $\mathcal{P}$ satisfies the specification, $\mathcal{Q}$ always satisfies $\mathcal{S}$ (if $\mathcal{S}$ is preserved by linearizability).

Deadlock-freeness is another desirable property of concurrent programs, i.e., a history of $\mathcal{Q}$ is always deadlock-free. We thus aim to generate $\mathcal{Q}$ which is always deadlock-free. Furthermore, it is insufficient to require only thread-safety or deadlock-freeness, which can be achieved trivially by having a universal lock which is acquired at the beginning of every public method and released only after the method completes. Thus, we further require that $\mathcal{Q}$ ideally should be as efficient as possible so long as it is still thread-safe.

Our problem definition is thus as follows. Given a program $\mathcal{P}$ which satisfies the above-mentioned assumptions, synthesize a program $\mathcal{Q}$ such that (1) $\mathcal{Q}$ is linearizable with respect to $\mathcal{P}$; (2) $\mathcal{Q}$ is deadlock-free; (3) $\mathcal{Q}$ is reasonably efficient. Note that the last requirement on efficiency is kept vague for now and we shall formalize it properly in Section 5.

The primary means of synthesizing $\mathcal{Q}$ is to refactor $\mathcal{P}$ with an implementation of a consistent locking policy. Intuitively, a locking policy specifies how each instance variable in Var is protected.

Definition 2.2 (Locking policy). A locking policy is a function $Var \rightarrow Lck \cup \{\bot\}$ where Lck is a finite set of locks (i.e., objects) and $\bot$ is a special lock denoting no locking.

Intuitively, assuming lp is a locking policy, $lp(x) = l$ means that variable x is guarded by lock l. For efficiency, we assume that when a lock is acquired, we have the option of acquiring it for either reading or writing. Note that multiple threads can obtain the same lock for reading whereas writing is exclusive, i.e., a lock acquired for writing can succeed if and only if there are currently no threads which hold the lock for reading or writing. There are existing implementations for such locks, like *ReadWriteLock* in Java. We write $lock(l, R)$ to denote that lock l is acquired for reading and $lock(l, W)$ to denote that lock l is acquired for writing.

3 HIGH-LEVEL OVERVIEW

The high-level idea of AutoLock is to automatically synthesize a locking policy based on static analysis of $\mathcal{P}$, and then consistently implement the locking policy to produce $\mathcal{Q}$ as the result. The overall workflow is shown in Figure 1. First, we perform static analysis of the source code (i.e., based on data flow and alias flow analysis) to obtain the permission-based access annotations for methods in a class. Intuitively speaking, a permission-based access annotation abstracts how variables are accessed and how methods manipulate

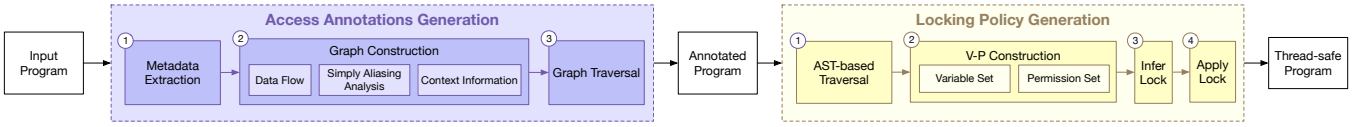

Figure 1: Overall algorithm.

```
1  class RatePath extends PathId{
2      // + @related(pathValue, pathDate, nAcceptedPathValue)
3      // + @shared @guardedBy(nAcceptedPathValue_pathDate_pathValueLock)
4      private double[] pathValue;
5      // + @shared @guardedBy(nAcceptedPathValue_pathDate_pathValueLock)
6      private int[] pathDate;
7      // + @shared @guardedBy(nAcceptedPathValue_pathDate_pathValueLock)
8      private int nAcceptedPathValue = 0;
9  +   // @write(pathValue) * @write(pathDate) * @write(nAcceptedPathValue)
10     private void readRatesFile(String dirName, String filename) throws DemoException {
11         File ratesFile = new File(dirName, filename);
12         BufferedReader in;
13         if (!ratesFile.canRead()) {throw new DemoException("Cannot read the file " + ratesFile.toString()); }
14         try {in = new BufferedReader(new FileReader(ratesFile));}
15         catch (FileNotFoundException fnfex) {throw new DemoException(fnfex.toString());}
16         int iLine = 0, initNlines = 100, nLines = 0;
17         String aLine;
18         Vector allLines = new Vector(initNlines);
19         try { while ((aLine = in.readLine()) != null) {
20                 iLine++;
21                 allLines.addElement(aLine);}
22         } catch (IOException ioex) {throw new DemoException("Problem from file "+ioex.toString());}
23         nLines = iLine;
24 +       nAcceptedPathValue_pathDate_pathValueLock.writeLock().lock();
25         this.pathValue = new double[nLines];
26         this.pathDate = new int[nLines];
27         nAcceptedPathValue = 0;
28         iLine = 0;
29         nAcceptedPathValue = iLine;
30         setname(ratesFile.getName());
31         setstartDate(pathDate[0]);
32         setendDate(pathDate[nAcceptedPathValue - 1]);
33 +       nAcceptedPathValue_pathDate_pathValueLock.writeLock().unlock();
34         setdTime((double) (1.0 / 365.0));}
35     //@read(pathValue)
36     public double getPathValue(int index) {
37 +       try {
38 +           nAcceptedPathValue_pathDate_pathValueLock.readLock().lock();
39             return (pathValue[index]);
40 +       } finally {nAcceptedPathValue_pathDate_pathValueLock.readLock().unlock();}}}
```

Figure 2: An illustrative example

them. Next, we synthesize a locking policy based on the access annotations. Afterwards, the locking policy is systematically implemented through program re-factoring. Note that the implementation is optimized for efficiency and deadlock-freeness.

In the following, we walk through each step using an illustrative example. Figure 2 shows partly a Java class named *RatePath* which is from the Grande benchmark [19]. The class contains multiple instance variables and it is meant to be thread-safe. Note that the comments and "+"-statements are introduced automatically by AutoLock. This example is interesting for the following reasons. First, it has 438 lines of code (excluding those in the super-classes or interfaces), 8 instance variables and 16 methods (all of which are public). Note that only a minimum part of the class is shown due to the space constraint. Second, there are file IO operations (e.g., at the beginning of the method in lines 11 – 23) which are time-consuming in multiple methods, which means that simply *synchronizing* every method would not be efficient.

Step 1: Generate access annotations. In this step, static analysis is performed to systematically identify the access annotations for each variable and each method. Access annotations are associated with each variable in V and each method in *PubM* and *PriM*. For instance, variable *pathValue* is associated with an annotation *shared* which means that the object can be (potentially) updated by multiple threads at the same time. For another instance, method *readRatesFile* is associated with the annotations shown at line 9, which intuitively means that the method writes variable *pathValue*, *pathDate* and *nAcceptedPathValue*. The annotation at line 2 states that the three variables *pathValue*, *pathDate* and *nAcceptedPathValue* are related, which intuitively means their values are constrained to satisfy certain conditions according to the specification $\mathcal{S}$. Note that access annotations characterize the way a shared resource can be accessed by multiple threads, and thus can be used to infer a safe (parallel) execution order of different program parts. For instance, a variable annotated *immutable* is thread-safe and thus allows maximum parallelism, and two methods can be executed in parallel if the variables they access do not overlap.

Step 2: Generate locking policy. In this step, we synthesize a locking policy based on the access annotations. In the illustrative example, 3 mutable instance variables are shown, i.e., *pathValue*, *pathDate* and *nAcceptedPathValue*. A locking policy assigns a lock for each and every mutable variable in the class. By default, different mutable variables are guarded by different locks for better efficiency, unless multiple variables are annotated *related* and thus must be guarded by the same lock. For instance, because *pathValue*, *pathDate* and *nAcceptedPathValue* are related, they are assigned with the same lock. The resultant locking policy is documented in the form of annotations at lines 3, 5 and 7, where *nAcceptedPathValue_pathDate_pathValueLock* is a freshly declared lock.
Step 3: implement locking policy. In this step, we implement the locking policy consistently throughout the class by automatic program re-factoring. For each method, we examine the access annotations and acquire the respective locks according to the locking policy.

We decide where exactly to acquire the locks based on two considerations. First, the scope of the locks should be minimized to improve efficiency. Second, there should be no lock-ordering deadlocks. For the latter, the locks are always acquired in a fixed global ordering throughout the class (in an ascending order alphabetically based on the names). For the former, a lock is acquired as late as possible except that it must be before the variable to guarded is accessed while not breaking the global ordering for acquiring locks. For instance, the first access to the three variables is a write access to *pathValue* at line 25 and thus the corresponding lock *nAcceptedPathValue_pathDate_pathValueLock* is acquired immediately. Note that this excludes the expensive file IO operations from the synchronized block and thus improves efficiency. The last access to the three variables is at line 32 and thus the lock is released as soon as possible at line 33. Note that we sometimes use *finally* (e.g., line 37 to 40) to make sure all locks are always released before the method ends in the presence of potential exceptions. We similarly implement the same locking in method *getPathValue*, where the lock is acquired for reading since this is a *getter* method.
The resultant *RatePath* class is guaranteed to be thread-safe. For instance, variable *pathValue* is systematically protected by lock *nAcceptedPathValue_pathDate_pathValueLock* in the class. In particular, as shown in Figure 2, the lock is acquired (for writing) at line 24 and released only at line 33 in function *add* and acquired for reading at line 38. We remark that multiple threads may execute method *getPathValue* at the same time whilst only one thread can execute line 25 to line 32 at any time.

4 INFERRING ACCESS ANNOTATIONS

In this section, we show what access annotations are used in our approach and how they are generated automatically.

4.1 Access Annotations

Our annotations are inspired by existing research on access permissions [20, 21]. Access permissions are abstract capabilities that model the mutability and aliasing of a referenced object at one place. There are five kinds of permissions, i.e., *unique*, *full*, *share*, *pure* and *immutable*, that encode whether or not an object is being aliased, whether a given reference can modify the referenced object, and whether there are other references (aliases) that point to the same object. In this work, we simplify the access permissions to a few access annotations which are friendly to programmers and yet are sufficient for generating locking policies.

In general, we keep the access annotations rather simple so that, once generated, they can be efficiently reviewed by programmers and amended if necessary. Programmers need to be kept in the loop since the generated annotations (and subsequently the implementation) must confirm to the specification $\mathcal{S}$ which we assume is only known by the programmers. In other words, the annotations serve as an abstract layer which explains why the implementation is generated in a certain way.

Each variable in *Var* is associated with three different annotations: *immutable*, *thread-safe*, and *shared*. The annotation *immutable* means that the variable is of an immutable type (e.g., *String* and *Integer* in Java) or that the variable is never updated. Note that an immutable variable is by definition thread-safe. The annotation *thread-safe* means that the variable is of a thread-safe type, e.g., *Vector*, *SynchronizedArrayList* and classes in package *java.util.concurrent.atomic*[1]. Note that such variables are by itself thread-safe except that there might be high-level data races if they are related with some other variables. The annotation *shared* specifies that the associated variable is mutable and it is not *thread-safe*.

Intuitively, an *immutable* variable does not require protection through locking; a *thread-safe* variable requires protection only if it is related with other mutable variables in certain ways; and *shared* variables require protection through a consistent locking policy.

Additionally, we adopt an annotation in the form of *related(x,y)* which intuitively means that variables x and y are related. In general, two variables x and y are considered 'related' if their values are constrained by some predicate $\phi(x, y)$ which is implied by the specification $\mathcal{S}$ and that cannot be captured by constraints on x and y separately. Note that this relation is transitive in nature, i.e., *related(x, y)* and *related(y, z)* implies *related(x, z)*.

Each method in *PubM* and *PriM* is annotated with a set of annotations, one for each *thread-safe* or *shared* variable in the class, which can be either *write* or *read*. That is, we annotate a method with *@write(x)* if x is modified in the method body; or otherwise it gets annotated with *@read(x)*. For instance, method *readRateFile()* is annotated with three annotations shown at line 9.

4.2 Generating Access Annotations

In the following, we present the approach to generate access annotations at the method level. To generate access annotations for a given method, following the access permission semantics [20, 21], we need to identify the way (i.e., read or write) an instance variable is accessed. Moreover, we need to identify and track aliases of the referenced objects, to maintain the integrity of data during analysis.

For this purpose, we perform a modular static analysis of the input Java program based on its abstract syntax tree (AST) and extracts the data-flow, alias-flow and context information for all the instance variables accessed at the method level. The extracted information is then consolidated as access annotations for each method. The objective here is to explicitly show the implicit dependencies that

[1] The term *thread-safe* is misleading as whether a class is thread-safe or not depends on its specification, e.g., thread-safety for *SynchronizedHashMap* and *ConcurrentHashMap* means something different. Here we have no choice but wishfully assume that the programmer has chosen the right 'thread-safe' class according to the specification.

exist between the code (method) and mutable states. In general, we systematically analyze each method in $PubM$ and $PriM$ with the following three steps.

In the first step, we parse a method's signature and its body to identify and track the variables' accesses as read, write and aliasing information. We recursively parse each expression in an expression statement in the AST to distinguish between *read-only* and *read-write* expressions. The analysis further depends on the type of the reference variable such as an object (class) field, a parameter or a method's local variable accessed in each expression. Note that we ignore the method's local variables unless they are aliases of the instance variables. It is because manipulating local references does not affect the access rights of the current methods.

We perform flow-insensitive analysis of the source code which ignores the order of execution of statements. However, our analysis preserves the semantics of assignment statements by determining the type of a reference variable on the left-hand side of an assignment statement based on its right-hand side expression type. We recursively parse the right and the left side of each assignment statement to identify the expression type and precisely extract the data-flow and alias-flow information of all the shared variables accessed at the method level. For example, in Figure 2 for method *readRatesFile()* at line 25 and 26, we analyze the expressions *this.pathValue = new double[nLines]* and *this.pathDate = new int[nLines]* and determine *write* access for variables *pathData* and *pathValue* is required. Note that the context analysis for the variables accessed in the current method depends on how the same variables are being accessed in other methods (e.g., aliasing).

For instance, the assignment expression *nAcceptedPathValue = 0* at Line 27 is considered as a <*value-flow*> statement as the right-hand side is a <*NumberLiteral*> expression. The approach maps this information as *write* access by the current method for the variable present on left-hand side of assignment expression i.e., *nAccepted-PathValue*.

The second step organizes the data flow and alias flow information extracted in step 1 in a graph model for each method, where nodes represent the variables accessed in the method, and the labeled edges represent the way (read/write/alias) the current method accesses the variables. In the last step, access annotations are generated by traversing the read, write and alias edges between the method and variable nodes in the generated graph.

Once annotations for each method have been generated, we generate annotations for each variable x in Var. If x is declared of a type which is known be *immutable* or *thread-safe*, x is annotated as *immutable* or *thread-safe* respectively. Note that a white-list of immutable classes and thread-safe classes in the Java development kit are embedded in our toolkit. If x is never written in any of the methods (which can be checked based on the annotations for the methods), x is also annotated as *immutable*.

Next, we generate annotations in the form of *related(x,y)*. Ideally, whether two variables are related should be inferred based on the specification $\mathcal{S}$. For instance, variable *lower* and *upper* in a *Range* class should be inferred *related* as the specification of Range implies that *lower* must be always no larger than *upper*. Since we do not assume that $\mathcal{S}$ is known, we rely on the following heuristics to infer whether two variables are related: two variables are related if there does not exist a public method such that one of them is updated without accessing (reading or writing) the other one. Intuitively, if two variables x and y are related (i.e., constrained to satisfy certain predicate $\phi(x, y)$), updating only one of those without reading or updating the other risks breaking the predicate $\phi(x, y)$ and consequently the specification $\mathcal{S}$. Note that our assumption is that any public method can be invoked for an arbitrary number of times. For instance, a method which updates *lower* should read the value of *upper* so as to make sure that the relation $lower \leq upper$ is not violated. We remark that this heuristic is based on our assumption that class $\mathcal{P}$ is correct if its methods are called sequentially.

5 GENERATING LOCKING POLICY

In this section, we describe how to synthesize a locking policy based on the inferred access annotations. Let Lck be a set of locks. For simplicity, let us assume that Lck is a set of fresh objects which are created to solely serve as the locks. Recall that a locking policy lp is a mapping from variables to locks. It is synthesized according to the following two general principles.

† A mutable variable is always guarded by the same lock.
‡ Related mutable variables are guarded by the same lock.

These principles are adopted from [22]. Guarding the same variable with the same lock makes it impossible to have a data race on the variable, whereas guarding related variables with the same block avoids high-level races, i.e., a race which breaks certain relation between two or more variables. The goal is to synthesize the most efficient locking policy which respects these two principles. To compare the efficiency of different locking policies, we formulate the following definition.

Definition 5.1 (Efficiency of locking policies). Given two locking policies lp_1 and lp_2 for program $\mathcal{P}$, we say that lp_1 is more efficient than lp_2 if and only if the following conditions are satisfied.

- For any pair of variables (x_1, x_2), $lp_1(x_1) = lp_1(x_2)$ only if $lp_2(x_1) = lp_2(x_2)$.
- There exists a pair of variables (x_1, x_2), $lp_1(x_1) \neq lp_1(x_2)$ and $lp_2(x_1) = lp_2(x_2)$.

A locking policy lp for $\mathcal{P}$ is the most efficient iff there does not exist a locking policy lp' such that lp' is more efficient than lp.

Our locking policy synthesis algorithm works as follows. First, we assign a $\perp$ lock to variables which are annotated with *immutable*, since they do not need protection. Second, we compute the transitive closure of the *related* relation between the remaining variables. The result is a partition of all variables such that each group of variables are related. Next, we assign a fresh lock object for each group if the group contains at least one *shared* variable or has at least two variables. The details are shown in Algorithm 1, where $related^*$ is the transitive closure of *related*.

For the example shown in Figure 2, the locking policy is documented in the form of annotations. That is, variable *domain* is to be guarded by no locks since it is *immutable*. Variables *data* and *timePeriodClass* are guarded by the same lock since they are *related*. The other variables are guarded with different locks.

PROPOSITION 5.2. *The locking policy synthesized by Algorithm 1 is the most efficient one with respect to the access annotations and principles † and ‡.* □

Algorithm 1: Synthesize locking policy

```
1  for each immutable variable x do
2  |  assign lp(x) to be ⊥
3  end
4  for each variable y which is not assigned do
5  |  if there exists z such that related*(y, z) then
6  |  |  assign lp(y) and all x such that related*(x, z) the
   |  |    same fresh lock l
7  |  end
8  |  else
9  |  |  if y is shared or there is z s.t. related(y,z) then
10 |  |  |  assign lp(y) to be a fresh lock object
11 |  |  end
12 |  |  else
13 |  |  |  assign lp(y) to be ⊥
14 |  |  end
15 |  end
16 end
```

The proposition can be proved straightforwardly through contradiction and thus we skip the proof. Note that here we assume the locking policy treats each method as a whole, i.e., it would be possible to have more efficient locking policy if we consider the internal of methods (as we do in Section 6). Note that the locking policy is synthesized based on the annotations, among which, the *related* annotation is 'guessed' based on heuristics. Therefore there is no guarantee on the correctness of the annotations and consequently the synthesized locking policy. This is inevitable as we do not have the specification. It is also why the access annotations must be presented to the programmers for validation.

6 IMPLEMENTING LOCKING POLICY

In this section, we present details on how the synthesized locking policy is implemented systematically and consistently. Recall that a locking policy lp is *consistently implemented* in $\mathcal{P}$ if and only if, for every variable x, every access of x (for either reading or writing) is guarded with the corresponding lock in every possible run of the program (i.e., the statement is executed after the lock has been acquired and before the lock is released).

First, static analysis including inter-procedural control flow analysis and aliasing analysis is systematically applied to build an abstraction of each method m. Let R be the set of variables read in m and W be the set of variables written in m; and $V = R \cup W$. The abstraction m_a is of the form of a labeled finite state automaton $m_a = (S, init, E, L, F)$ where S is a finite set of control locations; $init \in S$ is the unique initial location (i.e., the method head); $E \subseteq S \times S$ is a set of labeled transitions which capture the control flow; $L : S \rightarrow 2^V$ is a labeling function which labels each node with a set of variables, which are the set of variables read or written at the line; and F is a set of final nodes (i.e., control locations where the method terminates). We remark that over-approximation is applied whenever it is impossible to precisely determine the control flow or aliasing, i.e., E may contain infeasible control flow and, L may be a superset of the actual set.

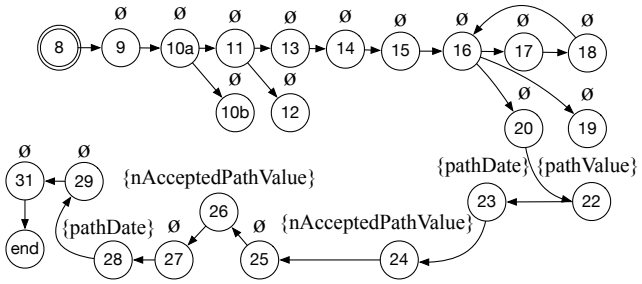

Figure 3: The finite-state automaton for method *readRateFile()* in the program in Figure 2.

The following are the requirements that must be satisfied by an implemention of the locking policy.

- First, for every variable x in R, along every path from $init$ to a transition which reads x, lock $lp(x)$ must be acquired for reading before the transition; for every variable x in W, along every path from $init$ to a transition which reads or writes x, lock $lp(x)$ must be acquired for writing before the transition.
- Second, the lock can only be released after the corresponding variable is accessed for the last time and all locks must be released by the end of the method.
- Third, for efficiency reasons, the lock should be acquired as late as possible and the lock release should be as early as possible (without breaking other constraints).
- Fourth, the locking policy must not introduce lock ordering deadlock.

We thus design the following approach for implementing the locking policies. First, we fix a global ordering on the locks, i.e., the ascending alphabetical order. Note that since all locks are freshly introduced, there are no aliasing problems and thus their names are sufficient to uniquely distinguish them. Let $<$ denote the ordering. The ordering dictates that if $o_1 < o_2$, lock o_1 is always acquired before lock o_2 along any path.

Algorithm 2 shows details on how to insert the lock acquire statement and release statement at the right place. That is, we systematically traverse through every path of m (i.e., unfolding each loop only once). Whenever there is a path such that a variable x in V is accessed without acquiring $lp(x)$ first, we insert a statement for locking $lp(x)$ for reading if x in R (or for writing if x in W) right before the node labeled with x. To release the lock, for each lock l acquired somewhere in the method, we identify $last(l)$ to be a set of nodes such that there is no path starting from any n in the set which accesses any variable x such that $lp(x) = l$ and there does not exist n' satisfying this condition and a path from n' to n (i.e., n is the first such node along any path). Note that $last(l)$ can be systematically identified using a backward breath-first-search algorithm from F. Lastly, line 11 and 12 reorder the lock acquire statements so that the locks are always acquired in a fixed global ordering. That is, we systematically traverse through every path of m (i.e., unfolding each loop only once). Whenever there is a path such that a lock l is acquired before another lock l' (before l is released) such that $l' < l$, we move the statement for locking l' before that for locking l.

Algorithm 2: Implement locking policy

Input: A m method $(S, init, E, L, F)$

Let V be $R \cup W$;
while *there is path which accesses a variable* $x \in R \cup W$ *and* $lp(x)$ *is not acquired before the access* **do**
  if x *is in* R **then**
    insert $lock(lp(x), R)$ before the first access;
  end
  else
    insert $lock(lp(x), W)$ before the access;
  end
end
for *each lock* l *such that* $\exists x : V.\ lp(x) = l$ **do**
  release l right after $last(l)$;
end
while $\exists$ *a path containing* $lock(l)$ *and subsequently* $lock(l')$ *and* l *is not unlocked in between and* $l' < l$ **do**
  move $lock(l')$ to right before $lock(l)$;
end

For instance, in our running example shown in Figure 2, method *readRateFile()* is abstracted as the finite-state automaton shown in Figure 3, where the nodes are labeled with the corresponding line numbers for readability. If there are multiple statements on the same line, we distinguish them by extending the line number with a letter, e.g., 10a and 10b. Note that the automaton captures control flow due to exceptions, e.g., the transition from node 16 to node 19 where *IOException* due to *in.readLine()* is caught. The first variable to be protected is *pathValue* which is accessed at line 22. Thus, the lock which protects *pathValue* (as well as *pathDate* and *nAcceptedPathValue*) is acquired at line 21. By traversing through the automaton, we determine $last(nAcceptedPathValue\_pathDate\_pathValueLock)$ is line 29. Thus, the lock is released at line 30. We remark method *getPathValue()* is similarly handled except that the access to the instance variable happens on a return statement at line 36. As a result, $last(dataTimePeriodClassLock)$ is calculated to be the exit of the method. Thus a *finally* block is introduced to release the lock. Note that this is a common practice. Lastly, we examine the lock ordering to check whether there is any path which holds multiple locks at the same time and the locks are locked in an order not following the fixed global ordering (i.e., the ascending alphabetical order based on the lock names). In this example, there is no such path as only one lock is acquired in each of the two methods.

In the following, we establish the soundness of our approach based on the notation of linearizability. The proof is shown below. Recall that we assume, without loss of generality, that variables are only accessible through methods in the class. We can always rewrite the direct variable reference with a `getter` function.

THEOREM 6.1. *The synthesized program is linearizable with respect to the given program.* □

The following establishes that the synthesized program is always deadlock-free. The proof is straightforward as locks are always acquired in a fixed global ordering along any program path in Algorithm 2.

THEOREM 6.2. *The synthesized program is deadlock-free.* □

PROOF. We sketch the proof below using a simple setting with two threads (say t_1 and t_2) and two methods (say m_1 and m_2). Let π be ab arbitrary trace of the synthesized program, which is composed of the following kinds of events $inv(m_i, a_i, t_i)$, $res(m_i, o_i, t_i)$, $lock_i(l)$, $release_i(l)$, $read_i(x)$, $write_i(x)$, and τ_i where $i \in \{1, 2\}$, l is any fresh clock introduced in the program, x is any shared variable, $read_i(x)$ is the event of thread i reading x's value to be v, $write_i(x)$ is the event of thread i writing x, and τ_i represents all the rest of the events (i.e., local transitions of thread i). By the soundness of the synthesized locking policy, if x is mutable, events $read(x)$ and $write(x)$ must be preceded with $lock(lp(x))$ and followed by $release(lp(x))$, and each $lock(lp(x))$ to $release(lp(x))$ block does not contain any variable related to y (including x) by the other thread. To prove the theorem, we show a procedure which constructs a trace π' which satisfies the two conditions for linearizability. The general idea is to expand the scope of each $lock_i(l)$ to $release_i(l)$ block in between a pair of $inv(m_i, a_i, t_i)$ and $res(m_i, o_i, t_i)$, $lock_i(l)$ such that the block encloses all events in between the invocation and the return, without changing the invocation and return events.

First, it is easy to see that a $lock_i(l)$ to $release_i(l)$ block can be 'expanded' to include all τ_i events immediately proceeding $lock_i(l)$ or immediately following the $release_i(l)$ event. Afterwards, we have a sequence of lock-release blocks which may overlap. By the correctness of Algorithm 1 and 2, the overlapping blocks must not update related variables, and thus we can re-order the events to have two non-overlapping blocks with the same sequence of events for each thread, without affecting the return events. The result is a sequence of back-to-back lock-release blocks which do no overlap. Afterwards, we delay the $inv(m_i, a_i, t_i)$ to the start with the first subsequent lock-release block of the same thread and obtain the resultant π'.

The above argument can be generalized to an arbitrary number of threads and methods as follows. If the theorem is invalid, there must be a concurrent history which is non-linearable, i.e., there exists a history composed of two threads and invocation of two methods which is not linearable. □

7 IMPLEMENTATION AND EVALUATION

In this section, we first briefly present how AutoLock is implemented and then evaluate its effectiveness through multiple experiments.

AutoLock has been implemented based on JDK 1.8 and it is open source [23]. It is built on top of the Java bytecode analysis and modification tool ASM for code instrumentation. It has total 2,327 lines of code. It relies on the Sip4J project [24, 25] to perform the data flow and aliasing flow analysis required for generating access annotations and that relies on *org.eclipse.jdt* for conducting the AST-based static analysis of the Java source code.

7.1 Evaluation

In the following, we conduct multiple experiments to answer the following research questions (RQ).

- *RQ1: How effectiveness is our proposed technique in avoiding the concurrency bugs?*
- *RQ2: What is the time overhead of our method?*

Table 1: Buggy programs.

program name	LOC	#vars	type	#locks	fixed	time(s)
log4j.Appender AttachableImpl	128	2	Race	1	✓	0.894
java.io.Buffered InputStream	314	17	Atom	5	✓	1.209
log4j.File Appender	156	9	Atom	1	✓	1.086
java.util. HashTable	4083	154	Race	24	✓	14.445
log4j.Null Appender	160	15	Atom	7	✓	0.997
dbcp.PerUser PoolDataSource	1101	80	Race	27	✓	1.792
dbcp.Shared PoolDataSource	1115	73	Atom	20	✓	1.786
java.util.Syn chronizedMap	4107	154	DL	24	✓	14.981
jfree. TimeSeries	303	9	Race	1	✓	1.181
java.util. Vector117	221	4	Atom	1	✓	1.124
java.util. Vector142	1940	103	Atom	27	✓	3.81
jfree. XYSeries	142	9	Race	2	✓	1.169

- *RQ3: How efficient are the generated thread-safe classes?*

RQ1 aims to study whether indeed AutoLock can be used to eliminate concurrency bugs systematically in the first place. RQ2 aims to evaluate whether AutoLock can be applied in practical scenarios, i.e., handing real-world sized programs and having an acceptable overhead. RQ3 aims to evaluate whether the thread-safe classes generated by AutoLock is reasonably efficient. In the following, we present details of the experiments for answering the RQs one by one. All experiments are conducted on a computer with Intel Xeon CPU E5-2640 of 40 cores and 128GB memory. All programs are running under ubuntu 18.04 and JDK 1.8.

To answer RQ1, we collect a set of concurrent programs which are known to be buggy. Table 1 shows details of the 12 classes. These classes are collected from CovCon [26], each of which contains one concurrency bug. In the table, the first four columns show the program name, the number of non-blank non-comment line of code in the program, the number of variables, and a broad categorization of the bug (DL indicates dead lock). These classes range from hundreds of lines of code to thousands, some of which have more than 100 instance variables.

Instead of patching the program by introducing additional synchronization (as is the case in [9, 12, 27]), we systematically remove all the synchronization (e.g., *synchronized* keyword and *lock* objects) and apply our approach to generate a thread-safe version of the classes. The last three columns show the number of locks used in the generated classes, whether the bug is fixed and how long it takes to generate the classes. To check whether the bug is fixed, we manually examine the generated classes one-by-one and check whether the buggy execution is eliminated successfully.

In all of the 12 cases, the bug is fixed successfully. In the following, we illustrate how bug fixing is achieved using the example shown in Figure 6. This class has a subtle concurrency bug in the method *add*. Specifically, a data race on variable *data* occurs when two threads execute method *add()* at the same time. In short, when two threads call function *add()* at the same time, line 16 might be executed concurrently by the two threads at the same time. Because variable *data* is of the type *ArrayList* (which is not a thread-safe class), a data race happens when *data.add()* is called by the two threads at the same time. A potential result is *IndexOutOfBoundException* as there may be only one element in *data* whereas its length has already been set to be 2 by the other thread. The exact details of the bug can be found at [26].

After applying our approach, the access annotations are generated at line 2 to 11, which leads to the locking policy shown at line 3 to 9. Note that because all of these four variables are accessed in method *add()* and some of them are updated, they are *related* and thus are guarded by the same lock. To protect these variables, the lock is acquired for writing at line 14 and released at line 26 in a *finally* block. As a result, no two threads are allowed to execute line 16 concurrently and thus the bug is fixed.

In terms of the number of locks used in the generated program, it ranges from 1 (guarding at most 9 variables) to 27 (guarding 80 variables). This statistic suggests that variables in the same class are not always related and thus we could use different locks to guarded different variables for better efficiency. We further summarize the time taken for generating the thread-safe classes, which is all within 15 seconds (and the average is 0.48 seconds). Compared to the time that people typically spend on fixing a single concurrency bug, e.g., 73 days [28], we believe that the time overhead is justified. In summary, we show that AutoLock could be applied to systematically prevent concurrency bugs.

To answer RQ2, we examine the performance of AutoLock in relation to the size of the given program, the number of variables, and the number of locks generated in the following experiment. The aim is to see whether AutoLock can be applied to large real-world programs. Although the time overhead in the above experiment seems reasonable, the experiment is limited to a set of buggy programs. We thus systematically collect a set of additional 33 programs from the Java Grande Benchmark [29], Æminium and Plaid Benchmark [30, 31], Pulse [32] and JDK 1.6 [33]. The details of the programs are shown in Table 2. Column 2 to 5 show the total lines of code, the number of classes, the total number of methods and variables respectively. In total, our benchmark has 64,447 lines of code, 724 classes and 7,544 methods. The largest one is *plural* with 246 classes and 2,189 methods. Column 6 shows the total number of locks generated to guard the variables. Note that each *related* annotation only concerns two variables. The last four columns show the time taken in seconds by each step and the total time.

On average, it takes total of 7.54 seconds to handle each program. The first step of generating access annotations takes the most time (i.e., 93.6%), whereas the rest two steps are efficient. This is expected as the first step requires non-trivial static analysis. For the biggest program *pural*, the total time is slightly less than 4 minutes. In summary, the time overhead of AutoLock is reasonable.

To answer RQ3, we conduct experiments to systematically compare the efficiency of programs generated using AutoLock with

```
1  public class TimeSeries extends Series implements Serializable {
2      //+ @related(timePeriodClass,data,maximumItemCounthistoryCount)
3      //@shared @guardedBy(ddhmrtLock)
4      private Class timePeriodClass;
5      //@shared @guardedBy(ddhmrtLock)
6      private List data;
7      //@shared @guardedBy(ddhmrtLock)
8      private int maximumItemCount;
9      //@shared @guardedBy(ddhmrtLock)
10     private int historyCount;
11     //@write(timePeriodClass) * @write(data) * @read(maximumItemCount) * @read(historyCount)
12     public void add(TimeSeriesDataItem pair) throws SeriesException {
13         try {
14 +           ddhmrtLock.writeLock().lock();
15             if (!pair.getPeriod().getClass().equals(timePeriodClass)) { throw new SeriesException(); }
16             int index = Collections.binarySearch(data, pair);
17             if (index < 0) {
18                 this.data.add(-index - 1, pair);
19                 if (getItemCount() > this.maximumItemCount) {this.data.remove(0);}
20                 if ((getItemCount() > 1) && (this.historyCount > 0)) {
21                     long latest = getTimePeriod(getItemCount() - 1).getSerialIndex();
22                     while ((latest - getTimePeriod(0).getSerialIndex()) >= historyCount) { this.data.remove(0); }}
23                 fireSeriesChanged();
24             } else {throw new SeriesException("");}
25         } finally {
26 +           ddhmrtLock.writeLock().unlock();}}}
```

Figure 6: The *TimeSeries* class as a fixed example.

Table 2: Efficiency evaluation.

program name	#line	#classes	#methods	#vars	#lock	generate anno. (s)	generate LP (s)	apply LP (s)	total (s)
arraylist	740	7	96	16	4	1.468	0.539	0.377	2.384
bitset	355	1	38	10	1	0.945	0.274	0.21	1.429
hashmap	765	12	116	37	9	1.162	0.314	0.231	1.707
hashtable	2729	47	506	151	24	5.101	0.475	0.408	5.984
identityhashmap	996	11	109	34	9	1.29	0.319	0.246	1.855
linkedhashmap	1237	18	143	52	17	1.335	0.331	0.246	1.912
linkedlist	1016	11	138	27	7	1.41	0.376	0.27	2.056
random	141	1	15	13	2	0.749	0.159	0.128	1.036
treemap	1892	25	285	70	22	3.178	0.492	0.326	3.996
vector	2634	45	552	136	20	6.201	0.413	0.339	6.953
aeminium.blackscholes	1062	5	72	37	9	1.098	0.303	0.272	1.673
aeminium.fft	96	3	7	7	3	0.714	0.125	0.112	0.951
aeminium.fibonacci	36	0	3	2	1	0.648	0.017	0.108	0.773
aeminium.gaknapsack	1154	5	24	49	15	0.885	0.277	0.221	1.383
aeminium.health	810	5	12	66	15	0.772	0.251	0.174	1.197
aeminium.integral	43	0	3	4	1	0.64	0.028	0.099	0.767
aeminium.lud	326	3	16	16	4	0.787	0.177	0.156	1.12
aeminium.quicksort	68	2	5	3	1	0.658	0.118	0.103	0.879
aeminium.raytracer	601	12	65	62	16	1.089	0.363	0.278	1.73
aeminium.shellsort	64	1	5	6	3	0.658	0.028	0.101	0.787
aeminium.webserver	149	2	8	6	2	0.722	0.115	0.11	0.947
jomp.crypt	489	4	39	27	8	0.853	0.316	0.255	1.424
jomp.euler	891	6	50	63	16	1.362	0.42	0.332	2.114
jomp.jgfutil	253	4	36	16	3	1.292	0.39	0.29	1.972
jomp.lufact	570	4	41	26	5	1.37	0.307	0.235	1.912
jomp.moldyn	612	6	42	56	9	0.963	0.342	0.261	1.566
jomp.montecarlo	1375	17	195	137	29	1.697	0.668	0.495	2.86
jomp.search	672	7	51	49	8	0.906	0.328	0.259	1.493
jomp.series	359	4	36	20	6	0.844	0.265	0.212	1.321
jomp.sor	323	4	33	24	5	0.818	0.269	0.217	1.304
jomp.sparsematmult	330	3	33	30	6	0.802	0.277	0.214	1.293
plural	20413	246	2189	594	196	228.56	1.882	1.187	231.629
pulse	7476	22	220	91	25	3.215	0.634	0.465	4.314

alternative approaches. We collect a set of 8 data structures (i.e., *arraylist*, *hashmap* and so on) from the standard JDK library. These are selected as there are different thread-safe versions of the programs in the library. For each program, we systematically compare the performance of four versions, i.e., the original sequential version, a baseline version in which a single lock is used to guard every method (i.e., every method is *synchronized* as would be the result of a naive programmer), a corresponding expert version (e.g., *CopyOnWriteList*, *ConcurrentHashMap* and so on) which is adopted from the JDK library and the version generated by AutoLock. Note that the expert version from the JDK library is highly optimized and it is highly non-trivial, if not impossible, to generate programs which

are more efficient. In fact, because the expert version often "relaxes" the specification in order to achieve better efficiency, e.g., *ConcurrentHashMap* allows multiple threads modifying different *buckets* in the map at the same time which is not allowed by *SynchronizedMAP*.

Performance testing of concurrent programs is a highly non-trivial task. Our performance test is set up in accordance to the performance test example shown in [22]. In particular, for each data structure, we set up an increasing number of threads which randomly invoke methods on the data structure through a shared object. A *timedbarrier* is used to start the timer and all the threads at the same time. The timer is stopped (again through the *timedbarrier*) as soon as all the threads finish. Afterwards, we divide the number of operations performed on the object by the total execution time to get the average throughput (the larger the better). The same random seed is used so that the same methods with the same parameters are executed. For the origin version, we just run it sequentially with previous settings because it's not thread-safe. Each experiment is executed for 20 times and we report the average as the result.

Our evaluation is conducted in two scenarios. In the first scenario, the methods called are mostly read-only. In particular, among a total of 512,000 method invocations, 90% of them are read-only (e.g., method *contains()*) and 10% of them write (e.g., method *remove()*). Note that whether a method reads only or writes as well can be easily checked based on the generated access annotations. To test the performance of the program under different workload, we run the experiments multiple times with the number of threads increasing from 2, 4, 8 to 256 and report the results separately. In the second scenario, only methods which write to the data structure are called (for the same total of 512, 000 times) and we compare the throughput of each version with different number of threads.

The result is shown in Figure 7 and Figure 8, where the y-axis is the throughput. For comparison, we additionally measure the time taken by the original version in the sequential setting, i.e., a thread is calling the same methods one after another. For the expert version, we use the highly optimized *java.util.concurrent.ConcurrentHashMap* for maps and *java.util.concurrent.CopyOnWriteArrayList* for lists and vectors. For jobs which are mostly read-only, the expert version, i.e., *ConcurrentHashMap* has the best performance in most of the cases (i.e., 6 out of 8).It is resonable because such classes have been carefully tuned for years and are widely used by millions of programs. However, in two cases, the program generated by AutoLock consistently performs the best. The reason is that, although the programs generated by AutoLock only has one lock (there are methods in the class which access all variables at the same time), we make use of *ReadWriteLock* which allows multiple threads executing read-only methods at the same time. Overall, AutoLock generates programs which are more efficient than the baseline version in 3 cases and as efficient as the baseline in the rest. Note that due to lock contention and the overhead of locking and unlocking, the sequential version is not always the slowest.

When only writing methods are invoked, the results are shown in Figure 8. First, we observe that while *ConcurrentHashMap* consistently performs best in multiple cases, *CopyOnWriteArrayList*'s performance is often on the bottom as expected (due to the overhead of copying the data structure every time it is written). AutoLock again wins in 3 cases but are slightly less efficient in 4 cases. The reason is that all locks are acquired for writing and thus the benefit of *ReadWriteLock* diminishes.

Limitations AutoLock aims to generate a thread-safe class by automatically synthesizing locking policies, and the protection is provided to the variables within the target class. There maybe cases that the target class accesses an object from an external library, which contains concurrency bugs. In this case, the bug is due to the the thread-safety issue of the external library, and not our target class. To avoid accidentally returning references of objects of the target class, we explicitly add a `getter` function, which conducts deep copy, for each object in the target class. This properly protect objects in the target class, but may introduce some performance overhead.

8 RELATED WORK

The access annotations in this work are closely related to work on access permission sharing models. Access permission sharing models [20, 34] have been used in many formal approaches such as Plural[35], Chalice [36, 37], Pulse[38], Verifast[39], Viper [40, 41] and VerCors[42–44], to provide thread-local reasoning for the shared-memory concurrent (multi-threaded) programs and to ensure race-free sharing of heap locations. In these approaches, the general idea is to explicitly associate access permission (read/write) information to program references (threads) to access memory locations and track the permission flow through the system to enforce mutual exclusion mechanisms. Moreover, permission-based programming paradigms [45–48] have been recently developed that parallelize execution of sequential programs based on access permission contracts. These approaches, however, require manually-specified access permission contracts.

This work is closely related to work on automatically generating safe concurrent programs. In [49–52], the authors proposed to use read-copy-update and read-log-update to adopt synchronization automatically. In particular, Herlihy [52, 53] use compare and swap (CAS), which has been widely used to implement wait-free or lock-free synchronizations. Dig *et al.* [54] proposed to use concurrency libraries to refactor sequential program. Based on read-copy-update and read-log-update, Zhang *et al.* [55] proposed to automatically convert sequential C++ data abstractions to concurrent lock-free implementations using compiler technology. It relies on software transactional memory (STM) which was proposed by Shavit and Touitou [16]. Such an approach however works without users acknowledging its underlying assumptions. In [56–59], authors provides approaches to make atomic sections automatically to guarantee programs' atomicity. Michael *et al.* [60] proposed regression testing for two versions of thread-safe classes. Different from them, our approach aims to compare the non-thread-safe class and its thread-safe version.

This work is also related to work on detecting and fixing concurrency bugs. Multiple approaches [8–11] tried to detect and fix concurrency bugs by detecting erroneous interleaving patterns. Huang *et al.* [8] attended to fix concurrency bugs via adding synchronization. There are a few approaches for fixing concurrent bugs of atomicity violation. AFix [9] adds a mutex lock to the program based on the CTrigger's [61] output to fix concurrency bugs. CFix [10] extends AFix to fix order violation concurrency bugs. CFix enforces all A-B or first A-B order relationships and mutual exclusion to fix

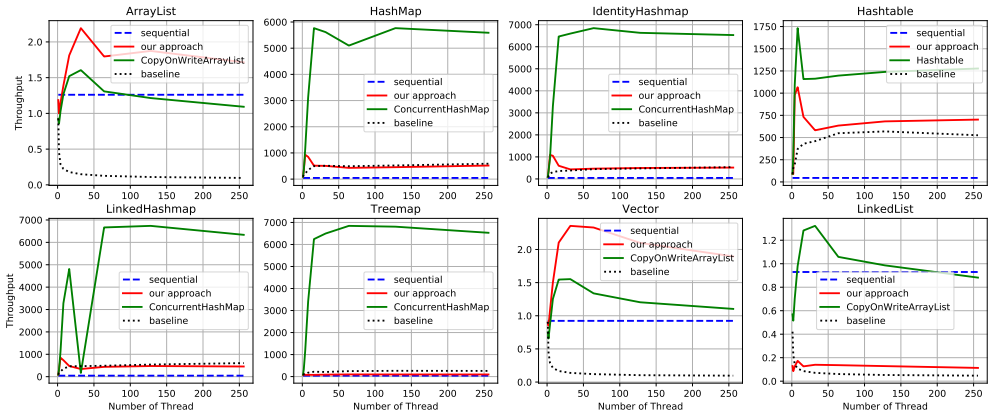

Figure 7: Throughput with mostly-read jobs.

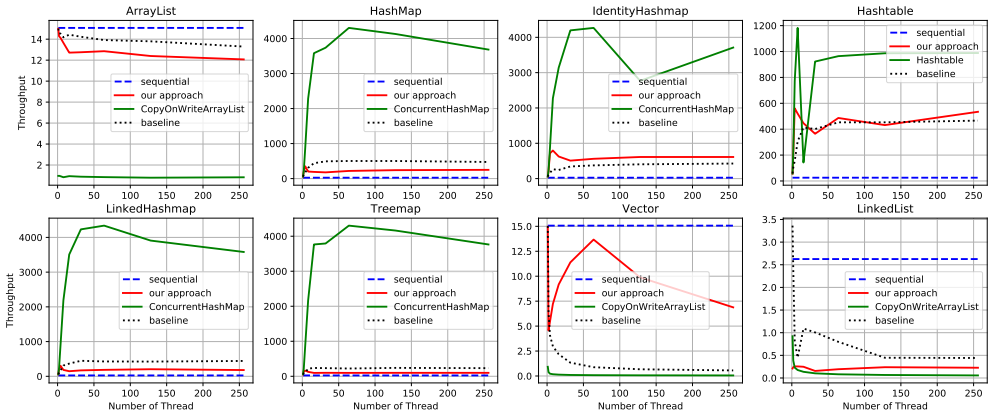

Figure 8: Throughput with write-only jobs.

order violation. Similar to AFix, Axis [11] fixes atomicity violations by adding mutual exclusion locks and synchronization measures. Besides, Axis works on reducing the possibility of introducing deadlocks. AlphaFixer [62] fixes atomicity violations by introducing locks. It fine-tunes the locking by analyzing the lock acquisitions and thus it is possible to reduce the introduction of deadlocks. Liu *et al.* [27] proposed HFix, which designs fix strategies based on a survey of 77 manual patches of real-word concurrency bugs. Besides using mutex locks, Hfix can also use the multi-thread operations, i.e., create and join, to achieve the purpose of fix while modifying the original locks. Grail [27] fixes concurrency bugs by adding locks in ways similar to AFix and Axis. Grail builds a Petri net analysis model which is context-aware and can consider lock alias. It allows Grail to take measures for deadlock-freedom. Grail can be time consuming due to the use of constraint solving and besides, it cannot fix multivariable bugs. PFix [12] proposes to fix concurrency bugs based on memory access patterns. Instead of fixing concurrency bugs, we aim to prevent concurrency bugs systematically.

9 CONCLUSION

In this work, we propose an approach of avoiding concurrency bugs by automatically generating thread-safe classes which are linearizable. The novel idea is to infer access annotations for each method automatically and synthesize as well as implement a locking policy based on the access annotations.

REFERENCES

[1] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[2] Christoph Von Praun and Thomas R Gross. Static conflict analysis for multi-threaded object-oriented programs. In *ACM Sigplan Notices*, volume 38, pages 115–128. ACM, 2003.

[3] Shuang Liu, Guangdong Bai, Jun Sun, and Jin Song Dong. Towards using concurrent java api correctly. In *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 219–222. IEEE, 2016.

[4] Kai Lu, Zhendong Wu, Xiaoping Wang, Chen Chen, and Xu Zhou. Racechecker: efficient identification of harmful data races. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 78–85. IEEE, 2015.

[5] Dejan Perkovic and Peter J Keleher. Online data-race detection via coherency guarantees. In *OSDI*, volume 96, pages 47–57, 1996.

[6] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM Sigplan Notices*, volume 44, pages 110–120. ACM, 2009.

[7] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145. ACM, 2008.

[8] Jeff Huang and Charles Zhang. Execution privatization for scheduler-oblivious concurrent programs. In *ACM SIGPLAN Notices*, volume 47, pages 737–752. ACM, 2012.

[9] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *ACM Sigplan Notices*, volume 46, pages 389–400. ACM, 2011.

[10] Guoliang Jin, Wei Zhang, and Dongdong Deng. Automated concurrency-bug fixing. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 221–236, 2012.

[11] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 299–309. IEEE, 2012.

[12] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. Pfix: fixing concurrency bugs based on memory access patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 589–600. ACM, 2018.

[13] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[14] Wladimir Araujo, Lionel C Briand, and Yvan Labiche. Enabling the runtime assertion checking of concurrent contracts for the java modeling language. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 786–795. ACM, 2011.

[15] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[16] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[17] Lu Zhang, Arijit Chattopadhyay, and Chao Wang. Round-up: Runtime checking quasi linearizability of concurrent data structures. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 4–14. IEEE, 2013.

[18] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[19] Grande. Grande benchmark. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite.

[20] Kevin Bierhoff and Jonathan Aldrich. *Modular typestate checking of aliased objects*, volume 42 of *OOPSLA '07*. ACM, 2007.

[21] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd ACM SIGPLAN Conference*, OOPSLA '08, pages 227–244, 2008.

[22] Brian Goetz, Tim Peierls, Doug Lea, Joshua Bloch, Joseph Bowbeer, and David Holmes. *Java concurrency in practice*. Pearson Education, 2006.

[23] Autolock. https://github.com/autolock-anonymous/AutoLock, 2020.

[24] Ayesha Sadiq, Yuan-Fang Li, Sea Ling, Li Li, and Ijaz Ahmed. Statically inferring permission-based specifications for sequential java programs. *arXiv preprint arXiv:1902.05311*, 2019.

[25] A. Sadiq, L. Li, Y. Li, I. Ahmed, and S. Ling. Sip4j: Statically inferring access permission contracts for parallelising sequential java programs. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1098–1101, 2019.

[26] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 266–277. IEEE, 2017.

[27] Haopeng Liu, Yuxi Chen, and Shan Lu. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 715–726. ACM, 2016.

[28] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 329–339. ACM, 2008.

[29] Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Polymorphic fractional permission inference. 2009.

[30] Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Practical api protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, pages 195–219. Springer, 2009.

[31] Stefan Heule, K Rustan M Leino, Peter Müller, and Alexander J Summers. Abstract read permissions: Fractional permissions without the fractions. In *VMCAI'13*, pages 315–334, 2013.

[32] Stefan Blom and Marieke Huisman. The vercors tool for verification of concurrent programs. In *International Symposium on Formal Methods*, pages 127–131. Springer, 2014.

[33] jdk16. https://github.com/zxiaofan/JDK.

[34] John Boyland. Checking Interference with Fractional Permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72. Springer-Verlag, 2003.

[35] Kevin Bierhoff and Jonathan Aldrich. Plural: Checking protocol compliance under aliasing. In *ICSE Companion '08*, pages 971–972, 2008.

[36] K. Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In *FOSAD 2007/2008/2009 Tutorial Lectures*, pages 195–222. Springer Berlin Heidelberg, 2009.

[37] K Rustan M Leino and Peter Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, pages 378–393. Springer, 2009.

[38] Radu I. Siminiceanu, Ijaz Ahmed, and Néstor Cataño. Automated verification of specifications with typestates and access permissions. *ECEASST*, pages 1–15, 2012.

[39] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.

[40] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[41] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*. 2017.

[42] Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The VerCors Project: Setting Up Basecamp. In *Programming Languages meets Program Verification (PLPV 2012)*, 2012.

[43] Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *Proceedings - IEEE 14th International Symposium on Parallel and Distributed Computing, ISPDC 2015*, 2015.

[44] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing.

[45] Jonathan Aldrich, Robert Bocchino, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. Plaid: a permission-based programming language. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 183–184. ACM, 2011.

[46] Jonathan Aldrich, Nels E Beckman, Robert Bocchino, Karl Naden, Darpan Saini, Sven Stork, and Joshua Sunshine. The Plaid language: Typed core specification. Technical report, DTIC Document, 2012.

[47] Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. Aeminium: A permission-based concurrent-by-default programming language approach. *TOPLAS,*, 36(1):1–42, 2014.

[48] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[49] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[50] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mv-rlu: Scaling read-log-update with multi-versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 779–792. ACM, 2019.

[51] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages

168–183. ACM, 2015.

[52] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[53] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[54] Danny Dig, John Marrero, and Michael D Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, pages 397–407. IEEE Computer Society, 2009.

[55] Jiange Zhang, Qing Yi, and Damian Dechev. Automating non-blocking synchronization in concurrent data abstractions.

[56] Michael Emmi, Jeffrey S Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. *ACM SIGPLAN Notices*, 42(1):291–296, 2007.

[57] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, 2006.

[58] Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[59] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. *ACM SIGPLAN Notices*, 43(6):304–315, 2008.

[60] Michael Pradel, Markus Huggler, and Thomas R Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 13–25, 2014.

[61] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 25–36. ACM, 2009.

[62] Yan Cai, Lingwei Cao, and Jing Zhao. Adaptively generating high quality fixes for atomicity violations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 303–314. ACM, 2017.