Dynamic Inference of Polymorphic Lock Types

James Rose, Nikhil Swamy, and Michael Hicks [1]

Computer Science Department
University of Maryland
College Park, Maryland 20742 USA

Abstract

We present an FindLocks, an approach for automatically proving the absence of data races in multi-threaded Java programs, using a combination of dynamic and static analysis. The program in question is instrumented so that when executed it will gather information about locking relationships. This information is then used to automatically generate annotations needed to type check the program using the Race-Free Java type system. Programs that type check are sure to be free from races. We call this technique *dynamic annotation inference.* We describe the design and implementation of our approach, and our experience applying the tool to a variety of Java programs. We have found that when using a reasonably comprehensive test suite, which is easy for small programs but harder for larger ones, the approach generates useful annotations.

Email addresses: rosejr@cs.umd.edu (James Rose), nswamy@cs.umd.edu (Nikhil Swamy), mwh@cs.umd.edu (Michael Hicks).

[1] Hicks is also affiliated with the University of Maryland Institute for Advanced Computer Studies (UMIACS)

1 Introduction

Writing correct multi-threaded programs is much more difficult than writing correct sequential programs. As Java's language and library support has brought multi-threaded programming into the mainstream, there has been widespread interest in developing tools for detecting and/or preventing concurrency errors in multi-threaded programs, such as data races and deadlocks. There are two main approaches. *Static* approaches, such as those based on type systems, take a program annotated with locking information and prove that the program is free from certain multi-threaded programming errors. A canonical example is the Race-Free Java type system [1]. Dynamic approaches monitor the execution of the program to discover violations of locking protocols based on observed execution history. A canonical example is Eraser [2].

On the face of it, these two techniques that address the same problems seem very far apart.

- The static approach is appealing because static analysis can conservatively model all paths through a program. When a *sound* static analysis can show a fact, that fact must hold in all executions. [2] Thus static analysis can prove the absence of errors such as data races and deadlocks without ever running the program, and without requiring the overhead of run-time code monitoring. The downside is that because static analysis must be conservative, it will incorrectly signal errors in correct programs. Such false alarms can be reduced, but not eliminated, by employing sophisticated techniques—e.g., context-, flow-, or path-sensitivity—but at the cost of scalability and implementation complexity.
- The dynamic approach is appealing because run-time code monitors are relatively easy to implement and are less conservative than static analyses because they have precise information about the current execution state [5]. The downside of the dynamic approach is that dynamic systems see only certain executions of the program, and so in general they can only conclude facts based on those cases. This means that either the code monitor must be packaged with the code permanently, or else run the risk of post-deployment failures.

Because static analysis can reach sound conclusions and impose no runtime overhead, we believe it to be the preferred approach whenever possible. However, as we have just discussed, the limitations of static analysis sometimes make it "too hard." Indeed, many static analyses require users to provide additional program annotations to guide the static analyzer. Experience has shown that programmers are reluctant to provide any but the most minimal annota-

[2] Not all static analyses are sound. Indeed, unsound "pattern detectors" have proven to be quite useful for finding bugs [3,4].

tions. For example, the designers of ESC/Java [6] state that such reluctance "has been a major obstacle to the adoption of ESC/Java. This annotation burden appears particularly pronounced when faced with the daunting task of applying ESC/Java to an existing (unannotated) code base." [7].

An *annotation inference* tool can reduce or eliminate the need for annotations. A typical approach is to use a whole-program, constraint-based analysis [8]. Unfortunately, a sufficiently expressive inference algorithm may not scale: indeed, for our setting, context-sensitive annotation inference is known to be NP-complete [9]. In contrast, a dynamic analysis has no trouble scaling. Therefore, we propose to use a dynamic analysis to generate *candidate annotations* [7] which can be checked for soundness by the static system. The intuitive idea here is that, just like for problems in NP, it may be difficult to (statically) generate correct statements about a program, but it is easy to check them. We call this combination of dynamic and static analysis *dynamic annotation inference*.

1.1 Contributions

In this paper, we describe a prototype system that employs an Eraser-like dynamic analysis to generate candidate annotations for the original Race-Free Java (RFJ) type system [1]. This paper makes the following contributions:

- We present a new algorithm (Section 2) for inferring race-checking annotations. Our dynamic algorithm improves on prior static [7] and dynamic [10] algorithms in being fully context-sensitive (polymorphic), and thus is able to properly check more programs. Our approach provides a possibly lighter-weight alternative (or complement) to traditional static inference.
- We describe our experience applying our tool FindLocks to a number some small (less than 2000 lines) and one medium-sized (55,000 lines) Java programs (Section 3). In our experience, dynamic inference imposes reasonably little runtime overhead, and infers the annotations needed for typechecking, assuming reasonably complete test suites.
- After comparing to related work (Section 4) we present two key lessons learned, laying out a path for continuing work (Section 5). First, dynamic analysis can frequently discover properties that typical type-based static analyses cannot check. We must consider new, path-sensitive static analyses that can take advantage of dynamically-discovered information. Second, the larger the program being checked, the more difficult it is to write test cases that cover all its code. In the end, we believe the most effective approach will be to combine static, whole-program analysis [9] with dynamic traces to improve the quality of the inferred types.

```
class C<ghost Object lp> {
  int count guarded_by lp;
  int value guarded_by this;
  synchronized void set(int x) requires lp {
    count++;
    value = x;
} }
class D {
  public void entry() {
    C<this> o = new C<this>(); // D.entry.L1
    synchronized(this) {
      o.set(1);
      o.count++;
} } }
```

Fig. 1. Example RFJ program (annotations are in italics)

2 Dynamic Lock Type Inference

To check for data races, most static and dynamic tools seek to infer or check the *guarded-by* relation. This relation describes which memory locations are guarded by which locks. Assuming that this relation is consistent, we can be sure that a particular data object or operation will only be accessed when the same lock is held, thus ensuring mutual exclusion. In a dynamic system like Eraser [2], the guarded-by relationship is inferred at run-time. In static type checking systems, types expressing what locks guard which locations must be specified by the programmer as annotations, though well-chosen defaults can make the task easier.

2.1 Race-Free Java

The RFJ type system requires that each field `f` of class `C` be guarded by either an *internal* or an *external* lock for `f`. To prevent data races, this lock must be held whenever `f` is accessed. An internal lock is any "field expression" in scope within class `C`, i.e., an expression of the form `this.f1.f2...fn` or `D.f1.f2...fn`, where `f1` is a static field of `D`. An external lock is one that is acquired outside the scope of `C` by users of `C`'s methods or fields. In RFJ, an external lock is expressed as a class parameter called a *ghost variable* which can be referred to in guarded-by annotations.

The class `C` in Figure 1 uses both internal and external locking: `C`'s field `value` is guarded by `this`, an internal lock, while `count` is guarded by `lp`, an external lock. The method `set` ensures these locks are always held when the fields are accessed. In particular, the fact that `set` is synchronized means that the lock

`this` is held when it executes, and thus accessing `value` is legal. In addition, the `requires` clause ensures that `lp` is held whenever `set` is called, and thus accessing `count` is legal. The modification of `count` within `entry` is legal since `lock` is held.

Whenever a parameterized class is mentioned in the program, its ghost variables must be *instantiated* with legal field expressions. This is shown in the `entry` method in the figure. The code fragment `new C<this>()` creates a `C` object, instantiating its external lock `lp` with the parameter `this`. Given this instantiation, the call to `x.set(1)` is legal, because the external lock (`this`) is held before the call.

During execution, the expression f in the annotation `guarded_by` f and in the instantiation `C<`f`>` must always refer to the same object. Otherwise, even if we always acquire the lock stored in f, it may not be the same lock every time f is acquired, leading to possible data races. RFJ enforces consistent synchronization by requiring f to be a field expression of the form mentioned above, in which each field within the expression is declared `final`.

RFJ permits a class whose objects are never shared between threads to be declared *thread local*, and thus no lock need be held to access its fields. As this mechanism applies to entire classes, it is not possible for some objects of a particular class to be checked as thread local but not others.

Recently, Flanagan and Freund have developed RFJ2, a type system that improves upon RFJ [9]. RFJ2 can check thread-local fields, rather than just thread-local classes. As with fields guarded by a lock, a thread-local field is guarded by an *implicit lock* which is conceptually held by each thread that "protects" thread-local data. The type system is augmented with an escape analysis to ensure that fields declared as thread-local never become shared. In addition, RFJ2 does not require field expressions to be `final` as long as they can proven to be immutable (i.e., programmers often forget to add the `final` annotation, and so RFJ2 infers it). As we show in Section 3, these two improvements can eliminate a significant number of false alarms, and would be easily exploited by our approach. Unfortunately, the implementation of the RFJ2 checker was not available as of this writing.

2.2 Dynamic Annotation Inference

The goal of FindLocks is to automatically infer `guarded_by` and `requires` annotations for unannotated Java programs. It proceeds in two steps, as shown in Figure 2. First, the unannotated program is instrumented and executed to collect a *trace*. Second, this trace is analyzed to infer lock types on program variables, and the source of the program is annotated with the inferred types.

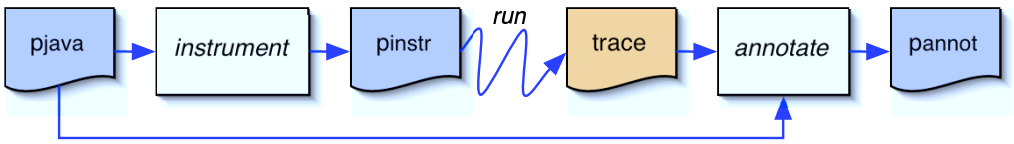

Fig. 2. Overview of FindLocks

The resulting annotated program is then checked using RccJava, which is a type checker for RFJ. We describe instrumented execution first, and then automatic annotation.

In the following, class names are denoted as C, D, E. Run time objects are denoted o (and are either actual addresses or $NULL$). Field names are denoted f, g, h; we sometimes treat `this` as a field for uniformity of notation. We use L to refer to a path of fields, e.g. $C.f.g.h$. This is a *static path* when $C.f$ is a static field, and an *instance path* otherwise. Instantiation sites are denoted I; these are program locations at which ghost variables of a class `C` may need to be instantiated; in what follows we consider *allocation sites* of the form $I =$ `new C(...)`, and *field declaration sites*) of the form $I =$ `C f`. When I is a field declaration, $I.f$ is used to refer to the field, and when I is an allocation, $I.C$ is used to refer to the class of the allocated object. In general, ghost variables may also need to be instantiated in the types of local variable declarations, in method parameters, in casts, or as the supertype in a class declaration. We ignore these cases in the following as the parameters used by a subclass to instantiate ghost variables in its superclass can be inferred much like allocation site parameters, method parameters can be treated much like fields, and local variables can be inferred through a simple intraprocedural static analysis.

Instrumented Execution A program trace consists of three maps gathered during execution: *lockset*, *alloc*, and *storedin*. These maps are generated with the help of two auxiliary data structures, *names* and *locksheld*; all are illustrated in Figure 3. To generate a trace, we instrument the program to perform some additional actions following certain instructions. In particular, every field access (read or write), object allocation, and lock acquire or release results in a call to our instrumentation code, whose actions are summarized in Figure 4.

The `acquire(`o`)` instruction attempts to enter the monitor associated with the object o. Once this instruction completes, we add o and its valid paths to the *locksheld* set. Valid paths L are calculated by taking the *closure* of the simple names of o, up to a fixed length k (since circular data structures may admit an infinite number of paths)[3]. The closure is defined as follows (where

[3] In our experiments we set this bound to 3. In our estimation, programmers rarely acquire locks using paths much larger than this.

Trace Output:

$lockset(o, C.f) : \{(o_1, \{L_{11}, ..., L_{1m}\}), ..., (o_n, \{L_{n1}, ..., L_{nm'}\})\}$

This is the set of locks—and their associated paths—that were held consistently when o's field f (defined in class C) was accessed.

$alloc(o) : (o', I)$

This indicates that o was allocated at site I while `this` $= o'$ (which will be recorded as $NULL$ if I is in a static method).

$storedin(o, C.f) : \{o_1, ..., o_n\}$

This set records all objects o_i that were ever stored in o's field f (defined in class C).

Auxiliary Structures:

$names(o) = \{(o_1, C_1.f_1), \ldots, (o_n, C_n.f_n)\}$

names maps each object o to all currently valid *simple names* for o, where simple name $(o_i, C_i.f_i) \in names(o)$ is a consequence of o being referenced by $o_i.f_i$ with the type of o_i given by C_i (o_i is $NULL$ if f_i is static).

$locksheld = \{(o_1, \{L_{11}, \ldots, L_{1n}\}), \ldots, (o_m, \{L_{m1}, \ldots, L_{mn'}\})\}$

This set records all locks that are currently held by the program; for each lock we also record paths for the lock in scope at the time the lock was acquired.

Fig. 3. Data structures used during instrumented execution

Instruction	**Action**
`acquire(`o`)`	$locksheld := locksheld \cup \{o, closure_k(o)\}$
`release(`o`)`	$locksheld := locksheld \setminus (o, \_)$
`read(`$o.f_C$`)`	$lockset(o, C.f) := lockset(o, C.f) \hat{\cap} locksheld$
`write(`$o_1.f_C := o_2$`)`	$lockset(o_1, C.f) := lockset(o_1, C.f) \hat{\cap} locksheld$ $names(o_2) := names(o_2) \cup \{(o_1, C.f)\}$ $names(o_1.f_C) := names(o_1.f_C) \setminus \{(o_1, C.f)\}$ $storedin(o_1, C.f) := storedin(o_1, C.f) \cup \{o_2\}$
`alloc`$_{(o',I)}$`(`o`)`	$alloc(o) := (o', I)$ $names(o) := \{(NULL, D.\texttt{this}) \mid D :> I.C\}$

Fig. 4. Actions of Execution Monitor

$names(NULL)$ is the empty set):

$$closure_0(o) = \{C.f \mid (o', C.f) \in names(o)\}$$
$$closure_i(o) = \{C.f \mid (o', C.f) \in names(o)\} \cup$$
$$\{L.f \mid (o', C.f) \in names(o) \wedge L \in closure_{i-1}(o')\}$$

As locks in Java are reentrant, the `release(`o`)` instruction decrements a counter on the lock, releasing the lock when the counter reaches 0; at this point we remove $(o, \_)$ from the *locksheld* set.

The `read(`$o.f_C$`)` instruction denotes a read from object o's field f, where f is defined by o's class (or superclass) C; this information is present in the bytecode. For a read we refine $lockset(o, C.f)$ by intersecting it with the current set of locks held. Intersection between an uninitialized lockset and the *locksheld* set is simply *locksheld*; intersection with a pre-existing lockset l is defined as follows:

$$l \hat{\cap} locksheld \equiv \{(o, z) \mid (o, x) \in l \wedge (o, y) \in locksheld \wedge z = x \cap y\}$$

When $o.f_C$ is written, we update its *lockset*, modify the *names* map for both the old and new contents of $o.f_C$, and update $storedin(o, C.f)$ to record the new object stored there.

Finally, we record for each allocated object o the site I at which it is instantiated, and the current `this` object o' (which is $NULL$ if I is within a static method). We also update the *names* to add $D.this$ as a valid name for o for all classes D that are superclasses of o's class $I.C$ (which includes $I.C$ itself).

Note that we do not instrument reads and writes of local variables: their names are not generally important, because they cannot be shared, either as a lock or a value.

Annotation Inference Given the output trace (Figure 3), the `resolve_locks` procedure shown in Figure 5 infers the `guarded by` clause for each field in the program, one at a time. It first attempts to find an internal lock for $C.f$ by calculating $resolve(R(C.f), C.f)$. Here, $R(C.f)$ is the *result set* for field $C.f$, which is simply an aggregation of all lockset information about that field:

$$R(C.f) \equiv \{(o, ls) \mid lockset(o, C.f) = ls\}$$

Given this information, we wish to find a consistent path for those locks held whenever f was accessed in some instance of C. Since we are looking

```
proc resolve_locks
  for each field C.f
    N  = resolve(R(C.f), I) where I is the declaration of C.f
    if N is not empty then
      annotate C.f with choose(N)
    else
      add fresh ghost parameter α to C, annotate C.f with α
      resolve_polymorphic(R(C.f), C, α)

proc resolve_polymorphic(R_old, C, α)
  for each instantiation site I of C
    N_I = resolve(R_I(R_old), I)
    if N_I is empty then
      if I is static then fail
      add fresh ghost parameter β to Class(I)
      instantiate α at I with β
      resolve_polymorphic(R_I(R_old), Class(I), β)
    else
      instantiate α at I with choose(N_I)
```

Fig. 5. Lock Resolution Algorithm

for an internal lock, this path must be in scope at the declaration of $C.f$ (`resolve_locks` sets I to be $C.f$):

$$resolve(R, I) \equiv \{L \mid L \in \bigcap_{(o,ls)\in R} ls \ \wedge \ L \text{ in scope at } \texttt{I}\}$$

If the resulting set is defined and non-empty, we choose a path with which to annotate the field declaration. This is typically `this` or a field of `C` (i.e., an internal lock), but it may be a static field of another class.

If the set was empty, `resolve_locks` parameterizes `C` with a ghost variable α and declares that `f` is guarded by α. Every time a ghost variable is added to a class `C`, it must be instantiated wherever the type `C` occurs. This is done by the procedure `resolve_polymorphic`. Instantiation site I of C are allocation sites `new C<`α`>()` or a field declaration sites `C<`α`> f`; `resolve_polymorphic` aims to find legal paths with which to instantiate α. To do this, it refines the result set to consider the locksets of the objects in R_{old} allocated (or stored) at site I:

$$R_I(R_{old}) \equiv \{(o', ls) \mid (o, ls) \in R_{old} \wedge o' \in ext(o, I)\}$$

where we have

$$ext(o, I) \equiv \begin{cases} \{o' \mid (o', I) = alloc(o)\} & I \text{ is an allocation site} \\ \{o' \mid o \in storedin(o', I.f)\} & I \text{ is a field declaration} \end{cases}$$

Then it applies the resolution procedure as before on this set for instantiation site I. If a valid path is found, then α is instantiated with that path. Otherwise, a parameter β is added to the class in which site I is defined, α is instantiated with β, and the process is repeated: result sets are created and solved at the allocation sites and field declarations of type $\texttt{Class}(I)$. Notice that o is replaced by o' in the definition of $ext(o, I)$—this is what permits the recursion. If the instantiation site I is an allocation occurring in a static method or a field declaration, then no further parameterization is possible, and this part of the program will not be typable.

Example We describe the execution of FindLocks on the sample program in Figure 1. Consider an execution of a single threaded program on a single instance `od` of the class `D`, with the public method `entry` as the only entry point to the class. Let `0x1` and `0x2` denote the addresses of `od` and `o`, respectively. An execution of this program would result in program trace structures as follows.

$$
\begin{aligned}
&\mathit{alloc}(\texttt{0x2}) &&= (\texttt{0x1}, \texttt{D.entry.L1})\\
&\mathit{lockset}(\texttt{0x2}, \texttt{C.value}) &&= \{(\texttt{0x1}, \{\texttt{D.this}\}), (\texttt{0x2}, \{\texttt{C.this}\})\}\\
&\mathit{lockset}(\texttt{0x2}, \texttt{C.count}) &&= \{(\texttt{0x1}, \{\texttt{D.this}\}), (\texttt{0x2}, \{\texttt{C.this}\})\} \hat{\bigcap} \{(\texttt{0x1}, \{\texttt{D.this}\})\}\\
& &&= \{(\texttt{0x1}, \{\texttt{D.this}\})\}
\end{aligned}
$$

From this trace, the `resolve_locks` considers each field of the class `C` separately, beginning with `C.value`:

$$
\begin{aligned}
&\mathit{resolve}(R(\texttt{C.value}), \texttt{C.value}) =\\
&\quad \mathit{resolve}(\{(\texttt{0x2}, \{(\texttt{0x1}, \{\texttt{D.this}\}), (\texttt{0x2}, \{\texttt{C.this}\})\})\}, \texttt{C.value}) = \{\texttt{C.this}\}
\end{aligned}
$$

Candidate path `D.this` is eliminated since it is not in scope at the declaration of `C.value`, so we indicate that `C.value` is `guarded by this`. By the same reasoning, `D.this` cannot guard `C.count` either:

$$
\begin{aligned}
&\mathit{resolve}(R(\texttt{C.count}), \texttt{C.count}) =\\
&\quad \mathit{resolve}(\{(\texttt{0x2}, \{(\texttt{0x1}, \{\texttt{D.this}\})\})\}, \texttt{C.count}) = \{\}
\end{aligned}
$$

Therefore, we and indicate that `count` is `guarded by lp` (where `lp` is a fresh name), and invoke `resolve_polymorphic`$(\{(\texttt{0x2}, \{(\texttt{0x1}, \{\texttt{D.this}\})\})\}, \texttt{C}, \texttt{lp})$. The only instantiation site of `C` in our program is the allocation site at `D.entry.L1`, and we have that $\mathit{ext}(\texttt{0x2}, \texttt{D.entry.L1}) = \{\texttt{0x1}\}$. Resolution discovers `D.this` as a candidate lock name since it is in scope at the instantiation

site `D.entry.L1`:[4]

$$resolve(R_{\texttt{D.entry.L1}}(\{(\texttt{0x2}, \{(\texttt{0x1}, \{\texttt{D.this}\})\})\}), \texttt{D.entry.L1})$$
$$= resolve(\{(\texttt{0x1}, \{(\texttt{0x1}, \{\texttt{D.this}\})\})\}, \texttt{D.entry.L1}) = \{\texttt{D.this}\}$$

Thus, the parameter `lp` at site `D.entry.L1` is instantiated with the name `this`. Finally, a simple static analysis adds the `requires` clause to `C.set` based on the added annotations.

Refinements Various refinements of the above algorithm were found to be helpful in the implementation of FindLocks. On the instrumentation side, we use the same approach as Eraser [2] to track whether a field is thread-local, read-only, or handed off between threads; this requires a simple modification to the *lockset*(,) map and its maintenance, and is useful for diagnosing warnings, as described in the next section.

On the inference side, we make two refinements. First, because the annotator resolves the lock for each field separately, a class could end up with as many parameters as it has externally-locked fields. FindLocks merges two parameters if, at every instantiation site, the parameters are the same.

Second, we do not infer polymorphically-recursive external lock parameters, as such inference is known to be undecidable [11]. Rather, we require every instantiation site of class `C` within `C`'s instance methods and instance field declarations to be instantiated with `C`'s ghost variables. For example, within the instance methods of `public class C<a,b> { ... }`, allocations of `C` will be expressed `new C<a,b>(...)`.

To do this, we change the way that we resolve external locks to consider the class of an external object. In particular, within `resolve_polymorphic`, we ignore all *internal* instantiation sites—those for class C that occur within instance methods or field initializers of C itself. Instead, we just instantiate them with α.

Consider the example in Figure 6. Executing this program will cause 10 `List` elements to be created, nine at site `List.add.L1`, and one at `A.go.L1`. When performing polymorphic resolution, we instantiate internal site `List.add.L1` with ghost variable `lp`. External site `A.go.L1` is instantiated as described in Figure 5.

[4] Note how the object address `0x1` associated in the result set refers to the *allocator* of the original object `0x2`. If it was necessary to extend resolution to another level, this would cause the algorithm to examine the scope of the allocators of `0x1`.

```
class List<ghost Object lp> {
  Object val guarded_by lp;
  List<lp> tail guarded_by lp; // List.tail
  void add(Object o) requires lp {
    tail = new List<lp>(); // List.add.L1
    tail.val = o;
} }
class A {
  void go(Object o) {
    List<this> head = new List<this>(); // A.go.L1
    synchronized (this) {
      head.val = o;
      for (int i = 0; i<9; i++) head.add(o);
} } }
```

Fig. 6. External Locking on a Recursive List Class

Note that this approach will rule out some programs that would otherwise be type correct in RFJ. For example, it will not allow you to have a list in which alternating elements are protected by different locks. We expect these cases to be rare in practice.

Finally, we note that FindLocks currently generates annotations from a single trace, but that it would be useful to generate annotations from multiple traces, for better coverage. This can be implemented by simply merging the traces prior to annotation inference, resolving any clashes of object references through α-conversion. More precisely, given two traces t_1 and t_2, if the set O defines those object addresses mentioned in both traces, then we define a set of fresh object addresses O' of size $|O|$ where $\forall o \in O'.o \notin t_1, o \notin t_2$. Let ς define a *substitution* that maps each $o \in O$ to a distinct $o' \in O'$, and is the identity otherwise. Then the merged trace is simply $\varsigma(t_1) \cup t_2$, where union is the intuitive merging of the maps and sets in the traces.

Implementation Dynamic instrumentation is performed using the BCEL [12] bytecode manipulation library, and our annotator is written in Java. BCEL uses a special class loader to add instrumentation as classes are loaded, thus making it easier to instrument existing programs. While convenient, on-demand instrumentation is challenging for standard library classes. For example, much of the interesting synchronization performed by programs revolved around the data structures in the `java.util` library, which includes classes used by the virtual machine (VM) itself. Therefore, we created an uninstrumented version of the classes under the package `umd.util` for use by the instrumentation code. Then, we created an instrumented version of `java.util` and prepended it to the `bootclasspath`, for use by the target code. Because the VM initialization process is sensitive, we use a layer of indirection between the hooks inserted

into the bytecode and the actual instrumentation library; this allows the bootstrap classloader to load the instrumented `java.util` without requiring all of the dependencies of the instrumentation library. This strategy should work for any subset of library classes, barring those that are implemented inside the VM or in native code.

3 Experimental Results

In this section, we describe our experience using FindLocks on a number of small Java programs. We present the resources required over four stages of execution: program execution (with and without instrumentation), annotation resolution, source code annotation, and type-checking with RccJava. Next we address the accuracy, expressiveness and completeness of the annotations emitted by FindLocks. Finally, we describe our experience using our tool on a larger program.

3.1 Sample Programs

Table 1 lists our benchmark programs, with relevant statistics in the first two columns. *Class* refers to the number of classes that were instrumented (the 152 classes in the `java.util` library are instrumented in all cases). *LOC* is the number of non-comment non-blank lines of code in the application alone. The `java.util` package contains 23741 non-comment non-blank lines of code.

The programs Elevators 1, Elevators 2, and Java-Server were written for a course in object-oriented programming at the University of Maryland; the former two simulate the scheduling of elevators in a building, and the latter is a small, dynamically-configurable HTTP server. Each was packaged with test cases, the former were simply command-line arguments to drive the simulation, the latter had a script that placed about 50 requests to the server. Proxy-Cache is an HTTP proxy that provides content caching, adapted from a program developed at the Technion, Israel Institute of Technology. Our test cases consisted of stressing Proxy-Cache with concurrent requests using httperf [13]. Weblech is a small web-crawler that was adapted from a program developed at MIT. To test Weblech we had it perform a depth 1 crawl from the University of Maryland home page.

Table 1
Memory Overhead

Program	**Class**	**LOC**	**Memory (MB)**		
	(+152)	*(+23741)*	**Orig**	**Inst**	**Ann**
Elevators1	4	567	8.8	21.1	110
Elevators2	4	408	8.7	13.4	112
Proxy-Cache	7	1218	9.7	30.8	112
Weblech	12	1306	14.1	52.1	127
Java-Server	32	1768	10.3	26.5	126

Table 2
Time Overhead

Program	**Time (sec)**				
	Orig	**Inst**	**Res**	**Annot**	**Rcc**
Elevators1	8.8	10.2	1.4	23.0	3.8
Elevators2	9.5	9.7	0.7	22.6	4.1
Proxy-Cache	12.5	19.9	1.9	14.9	4.3
Weblech	6.7	30.3	3.1	20.3	4.2
Java-Server	12.8	20.6	2.1	15.2	3.8

Table 3
Trace Sizes

Program	**Tot**	**Max**	**Ave**	**%-Empty**	**Name**	**Store**	**Alloc**
Elevators1	2742	14	6.35	93.16	2253	3008	356
Elevators2	2362	8	6.42	93.17	2050	2821	316
Proxy-Cache	9456	11	6.12	99.54	19387	21288	4954
Weblech	25500	38	17.8	97.51	24800	31009	4470
Java-Server	10340	1	1	99.94	25363	17801	3008

3.2 Peformance

Tables 1 and 2 quantify the cost of FindLocks, in terms of instrumentation overhead and annotation inference. For the former, the columns **Orig** and **Inst** refer to the resources consumed during the execution of the program without and with instrumentation, respectively. **Inst** includes both the time to instrument and execute the program.

For annotation inference, the memory usage is given in column **Ann** while the time taken is broken down in columns **Res** and **Annot**. Here, **Res** is the time to run `resolve_locks` (Figure 5), not including the time to annotate the original source code; this time is broken out in column **Annot**. The annotation phase is not intrinsically expensive; an inefficient (but convenient) parse-tree construction tool is used which may create many objects for a single nonterminal in the parse tree. We report these numbers only for completeness; they could be dramatically reduced by integrating annotation into the RccJava framework or by writing the parsing code more efficiently.

The time taken by RccJava to typecheck the annotated program is reported in the column labeled *Rcc*.

Table 3 shows statistics related to the trace data structures used by the instrumented programs. Unless otherwise stated, each figure is the largest value ever attained over the entire execution of the program. **Tot** refers to the size of the *lockset*(,) map; **Max** refers to the largest number of names in a single lockset; **Ave** refers to the average number of names in a single non-empty lockset at the end of program execution; **%-Empty** refers to the proportion of fields that had empty locksets at the end of program execution; **Name** refers to the *names*() map; **Store** refers to the *storedin*(,) map; and **Alloc** refers to the *alloc*() map. The *lockheld* set was typically an order of magnitude smaller than the *alloc*() map.

In each case, the numbers represent the median value from ten trials. The variation is not appreciable. These measurements were performed on a 2 GHz Pentium 4 with 750MB of RAM, running linux 2.4.21, and the Sun JRE 1.4.2.

3.3 Quality of Inferred Annotations

Table 4 describes the results of running RccJava on the annotated programs generated by FindLocks. The column *Classes* shows the number of classes that were actually annotated including the libraries. Classes contain no annotations if either the test cases did not cover the class, or sometimes if the class contains no fields. The column *Num* refers to the number of annotations that

Table 4
Checking Annotated Programs

Program	Annotations		Rcc Warnings			
	Classes	Num	Thl/RO	Final	Race	Oth
Elevators1	3	26	5	0	1	1
Elevators2	6	27	1	0	0	0
Proxy-Cache	7	69	15	0	0	4
Weblech	11	52	30	10	0	5
Java-Server	18	59	5	0	0	2

were added automatically by FindLocks. The section of the table labeled *Rcc Warnings* classifies the type of warnings issued by RccJava when run on the annotated programs. *Thl/RO* represents spurious data race warnings about fields that are in fact thread local, or are read-only. The column *Final* records RccJava warnings about the use of locks that are not final expressions. These are spurious warnings too since, in each case, the lock expressions though not final expressions are actually constants. The column *Race* records warnings about real data races.

Of all our test programs, only Elevators1 fails to typecheck under RccJava because it contains a real data race. This data race is also dynamically detected by FindLocks which adds a comment to the field noting the problem. In short, up to the limits of the RFJ type system, FindLocks *correctly inferred all needed annotations.*

In all, three of our test programs used external locking, for which FindLocks inferred correct polymorphic lock types: Elevators 2, Java-Server, and Weblech. In the latter two cases, the programmer defined externally-locked classes, while Elevators 2 uses external synchronization to guard instances of `java.util.HashSet`. In particular, there is a field of type `HashSet` and each access to this field is protected by obtaining a lock external to the scope of the `HashSet`. FindLocks infers that `HashSet` has a type that is polymorphic in the type of the lock and correctly annotates the `java.util.HashMap` field of the `HashSet` class as being guarded by the lock parameter. Furthermore, an inner class of `HashMap`, `HashMap.Entry` is also parameterized by the same lock parameter.

Many of the warnings issued by RccJava refer to fields determined to be thread-local or read-only by the dynamic analysis in the observed execution. In general, this information cannot be used by RFJ to prove accesses to these fields are safe; therefore FindLocks annotates the source with comments (invisible to RccJava) that can help the user classify RccJava warnings

as spurious or genuine. For example, RccJava warns about non-`final` expressions used as locks; the read-only comments added by FindLocks help the user to confirm that lock expressions are constant. As mentioned in Section 2.1, RFJ2 [9] and other type systems [14,15] would be able to use this information to check more idioms.

Various other warnings were classified as spurious by hand. For instance, RccJava (optionally) assumes that the `this` lock is held during object construction in order to allow for common initialization patterns; this is sound if the constructor does not allow `this` to escape. In the constructor of an object that synchronizes on a mutex different from `this`, RccJava issues warnings about the mutex not being held, since it only assumes that constructors hold only the `this` lock.

3.4 Annotation Quality and Test Coverage

We also ran FindLocks on HSQL, an open-source, JDBC-compliant database. [5] HSQL consists of 260 classes and about 55,000 lines of code. We did not have access to a test suite for the application, so we devised a simple test program that spawned a large number of threads and repeatedly performed simple queries on the database.

We found both the runtime overhead as well as the annotation overhead to be similar to that incurred by the smaller programs. The uninstrumented program ran our test case in 31 seconds, using 48MB of memory; running it under FindLocks took 117 seconds and consumed 160MB of memory. RccJava issued 208 warnings when checking the annotated program. These warnings arose from two sources. First, our extremely simple test case only managed to cover some 90 out of the 260 classes, so many fields lacking annotations and were flagged. Second, a large number of the warnings issued mentioned fields marked as thread local by FindLocks. On the one hand, while some or all of these warnings might be spurious due to the limitations of RFJ, it may also be that they just happened to be thread local for our particular trace; we cannot know for sure without a sound checking system that can handle them (like RFJ2). Our suspicion after inspecting the source is that for some of these fields, thread locality was a function of the particular execution and not of the program in general.

In light of this experience, it becomes clear that dynamic annotation inference is most likely to succeed when it complements a thorough testing regime. Achieving a good degree of code coverage is essential to inferring correct lock

[5] `http://hsqldb.sourceforge.net/`

relationships from program traces. We do not view this as a significant limitation, as testing for correctness is commonplace—existing tests can be used for annotation inference with little trouble.

4 Related Work

There are two ways that static and dynamic analysis are traditionally combined. Most common is what we refer to as *static-dynamic* analysis: Dynamic analysis (in the form of run-time checks) is used to enforce properties that are too hard to check statically. A canonical example is the property that a pointer will not be *NULL* when it is dereferenced; when this fact cannot be proven statically, it is checked dynamically with an inserted check. This technique has been used to check for data races in Java programs [16,17]. Its drawback is that it imposes run time overhead in all but the most simple cases, and if a data race is discovered dynamically, the only recourse may be to terminate the program.

The converse is *dynamic-static* analysis, in which runtime profiling data informs a static analysis; FindLocks is an example of a dynamic-static analysis. Ernst's Daikon tool [18] infers simple invariants between variables in a program through run-time profiling. Nimmer and Ernst [19,20] showed that many of the inferred invariants could be proven sound using a theorem prover. In their approach, dynamically-determined invariants are part of a *candidate set*, and the theorem prover removes those invariants that it cannot prove. Specification mining [21] is a technique for automatically discovering sequencing and data constraints on API calls. The information may be useful for a static verification tool. The most common example of dynamic-static analysis is profile-directed compilation [22–25]. In this case, generated code is improved by considering run-time profiles. This is a matter of performance, not correctness, so poor profiling information will not cause the program to produce the wrong answers.

A wide variety of type-checking systems have been developed for preventing possible data races. However, we know of only three approaches that infer types for such systems, to relieve the annotation burden on the programmer: Houdini [7], Agarwal and Stoller [10], and RFJ2 [9].

Houdini [7] is a self-described *annotation assistant* that statically generates sets of candidate annotations based on domain knowledge. Houdini was applied to a simplified version of race-free Java [26] that does not support external (polymorphic) locking. Recall that three of our test programs and the standard libraries use external locks, so Houdini could not infer annotations in these cases.

Concurrently with us, Agarwal and Stoller [10] developed a dynamic annotation inference algorithm for Parameterized Race-Free Java (PRFJ) [14]. Their algorithm is similar to ours in many respects. One difference is that it is less context-sensitive: it handles polymorphic instantiation, but not polymorphic generalization. In particular, it assumes that either a class has a single lock parameter, or if the class has multiple parameters then the user has annotated it as such. In contrast, our approach infers the need for lock parameters (ghost variables) automatically. Agarwal and Stoller's algorithm handles some advanced features of PRFJ not present in RFJ, such as *unique* and *read-only* objects.

RFJ2 is a recent refinement to the Race-Free Java system that includes support for static annotation inference [9]. RFJ2 supports thread-local fields and parameterized method declarations, and infers when non-final fields are in fact constant. These improvements would eliminate many of the false alarms we had with RFJ, as mentioned above. The problem of annotation inference in RFJ2 is proved to be NP-complete by a reduction to propositional satisfiability. This reduction naturally led to the idea of using a SAT-solver to perform inference; the approach is called Rcc/Sat. Program annotations are obtained from the model of a boolean formula that represents the locking requirements of the program. The search space of the SAT-solver for this problem is precisely the space of abstract program locations; a space that is complicated by the aliasing properties of the program. As SAT-solving takes exponential time in the worst case, this approach could have trouble scaling. As one data point, inference of a 11K line program took roughly 2 minutes, while inference of 30K line program took roughly 45 minutes. However, it is likely that our dynamic approach can assist in the reduction of this search space, as we propose below.

5 Conclusions and Future Work

Our experience thus far leads us to believe that dynamic analysis can usefully perform annotation inference. Since programmers typically write tests for their programs, dynamic annotation inference imposes only a small burden, and adds value by proving sound properties, in our case the absence of data races, based on collected traces. Indeed, our tool inferred all the annotations needed for idioms that RFJ could check. Moreover, a number of applications made use of external locking, and our approach correctly parameterized classes to express this fact.

However, our experience has exposed two limiting factors in the technique:

(1) A large program may only execute a portion of its code during common usage, and thus a dynamic tool may not generate annotations for the

entire program. This was a problem for our (very simple) testing code for HSQL.

(2) In general, a given static analysis may not be able to verify properties easily detected by dynamic analysis. For example, RFJ does not support treating classes as thread-local on a per-field basis. It also cannot check temporal shifts in protection, such as an object that is thread-local at first, but later becomes read-only or shared and locked. Our dynamic instrumentation could discover these situations easily, but the chosen static analysis could not check them.

To reduce the need for good test coverage, we could have the dynamic analysis "add value" to a static inference system. In particular, candidate annotations could be generated both statically and dynamically, and checked for soundness in the style of Houdini [26,7]. Indeed, the RFJ2 inference system Rcc/Sat seems like a prime candidate for this approach.

To address the second problem requires developing a stronger complementary static checking system. Indeed, both RFJ2 and PRFJ are more expressive than RFJ. Dynamic annotation inference could make practical a more sophisticated type system which requires many annotations, since it can infer at least some of those annotations automatically. For example, dynamic analysis can easily and efficiently capture the program execution *paths* for which a safety property holds. To make best use of this information, our static checking system could be path sensitive. Type systems with intersection-, union-, and dependent types can describe path-sensitive properties. Since (static) type inference in such a system is generally undecidable, dynamic path information could supply needed annotations.

More generally, an interesting avenue of future work is to evaluate, under a variety of metrics, when the technique of applying dynamic analysis to aid sound static analysis makes sense. In general, the fact that a property is satisfied by some set of executions does not imply that the property holds for the entire program. However, in our experience the guarded-by relation discovered by the dynamic instrumentation can frequently be proved sound for the whole program. The interesting question is when and why this is the case. While work has been done to characterize the computability classes of runtime analysis as compared to static analysis [5,27], little has been done to explore the two at the level of actual programs. For example, Ernst [18] has found that dynamically-inferred properties sometimes hold statically, but does little to explain why. One possibility is to consider program traces and programs that induce them in a single formalism such as abstract interpretation [28].

References

[1] C. Flanagan, S. N. Freund, Type-Based Race Detection for Java, in: Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver B.C., Canada, 2000, pp. 219–232.

[2] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs, in: Proceedings of the 16th ACM Symposium on Operating Systems Principles, St. Malo, France, 1997, pp. 27–37.

[3] D. Engler, K. Ashcraft, Racerx: effective, static detection of race conditions and deadlocks, in: Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, New York, 2003, pp. 237–252.

[4] D. Hovemeyer, W. Pugh, Finding bugs is easy, in: J. M. Vlissides, D. C. Schmidt (Eds.), OOPSLA Companion, ACM, 2004, pp. 132–136.

[5] K. W. Hamlen, G. Morrisett, F. B. Schneider, Computability classes for enforcement mechanisms, ACM Transactions on Programming Languages and Systems 27, to appear.

[6] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended Static Checking for Java, in: PLDI'02 [29], pp. 234–245.

[7] C. Flanagan, K. R. M. Leino, Houdini, an Annotation Assitant for ESC/Java, in: J. N. Oliverira, P. Zave (Eds.), FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods, no. 2021 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 2001, pp. 500–517.

[8] A. Aiken, M. Fähndrich, J. S. Foster, Z. Su, A Toolkit for Constructing Type- and Constraint-Based Program Analyses, in: X. Leroy, A. Ohori (Eds.), Proceedings of the Second International Workshop on Types in Compilation, Vol. 1473 of Lecture Notes in Computer Science, Springer-Verlag, Kyoto, Japan, 1998, pp. 78–96.

[9] C. Flanagan, S. N. Freund, Type Inference Against Races, in: R. Giacobazzi (Ed.), Static Analysis, 11th International Symposium, Vol. 3148 of Lecture Notes in Computer Science, Springer-Verlag, Verona, Italy, 2004.

[10] R. Agarwal, S. D. Stoller, Type Inference for Parameterized Race-Free Java, in: Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation, Vol. 2937 of Lecture Notes in Computer Science, Springer-Verlag, Venice, Italy, 2004.

[11] F. Henglein, Type Inference with Polymorphic Recursion, ACM Transactions on Programming Languages and Systems 15 (2) (1993) 253–289.

[12] Bytecode engineering library, `http://jakarta.apache.org/bcel/`.

[13] D. Mosberger, T. Jin, httperf: A tool for measuring web server performance, in: First Workshop on Internet Server Performance, ACM, 1998, pp. 59—67.

[14] C. Boyapati, M. Rinard, A Parameterized Type System for Race-Free Java Programs, in: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, 2001, pp. 56–69.

[15] D. Grossman, Type-Safe Multithreading in Cyclone, in: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation, New Orleans, Louisiana, USA, 2003, pp. 13–25.

[16] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridharan, Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs, in: PLDI'02 [29], pp. 258–269.

[17] R. O'Callahan, J.-D. Choi, Hybrid dynamic data race detection, in: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, 2003, pp. 167–178.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, Dynamically Discovering Likely Program Invariants to Support Program Evolution, IEEE Transactions on Software Engineering 27 (2) (2001) 99–123.

[19] J. W. Nimmer, M. D. Ernst, Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java, in: Proceedings of the First Workshop on Runtime Verification (RV '01), 2001.

[20] J. W. Nimmer, M. D. Ernst, Invariant Inference for Static Checking: An Empirical Evaluation, in: Tenth Symposium on the Foundations of Software Engineering, Charleston, South Carolina, USA, 2002, pp. 11–20.

[21] G. Ammons, R. Bodik, J. R. Larus, Mining specifications, in: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, 2002, pp. 4–16.

[22] G. Ammons, J. R. Larus, Improving data-flow analysis with path profiles, in: Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, 1998, pp. 72–84.

[23] T. Ball, J. R. Larus, Optimally profiling and tracing programs, in: Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, 1992, pp. 59–70.

[24] K. Pettis, R. C. Hansen, Profile guided code positioning, in: Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation, White Plains, New York, 1990, pp. 16–27.

[25] Y. Wu, J. R. Larus, Static branch frequency and program profile analysis, in: Proceedings of the 27th International Symposium on Microarchitecture, San Jose, CA, 1994, pp. 1–11.

[26] C. Flanagan, S. N. Freund, Detecting race conditions in large programs, in: Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, Utah, 2001, pp. 90–96.

[27] F. B. Schneider, Enforceable security policies, ACM Transactions on Information and Systems Security 3 (1) (2000) 30–50.

[28] P. Cousot, R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977, pp. 238–252.

[29] Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation.