Dynamic Inference of Polymorphic Lock Types

James Rose
rosejr@cs.umd.edu

Nikhil Swamy
nswamy@cs.umd.edu

Michael Hicks
mwh@cs.umd.edu

Computer Science Department
University of Maryland, College Park

ABSTRACT

We present an approach for automatically proving the absence of race conditions 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 fed to our tool, FindLocks, that generates annotations needed to type check the program using the Race-Free Java [12] type system. Our approach extends existing inference algorithms by being fully context-sensitive. We describe the design and implementation of our approach, and our experience applying the tool to a variety of Java programs. We have found the approach works well, but has trouble scaling to large programs, which require extensive testing for full coverage.

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, including race conditions, timing dependencies, 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 [12]. Dynamic approaches monitor the execution of the program to discover violations of locking protocols based on observed execution history. A canonical example is Eraser [23].

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. Thus static analysis can prove the absence of errors such as race conditions 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 [17], in part because they have complete, precise information about the current execution state. 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[1] 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 annotations. For example, the designers of ESC/Java [15] 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." [14].

1.1 Dynamic Annotation Inference

An *annotation inference* tool can reduce or eliminate the need for annotations. A typical approach is to use a whole-program, constraint-based analysis [2]. For detecting race conditions, such an analysis must consider the aliasing relationships between objects in the program. Unfortunately, the state-of-the-art in scalable points-to analysis can be imprecise when modeling common linked data structures, such as lists and trees. These analyses often model elements of a linked structure as a single abstract location, and thus will fail to distinguish between a lock that protects one list element versus another.

In contrast, a dynamic analysis has ready access to program objects and their aliasing relationships. Therefore, we could use a dynamic analysis to generate *candidate annotations* [14] based on its observations, and these can be checked by the static system. The intuitive idea here is that,

[1]Not all static analyses are sound. Indeed, unsound "pattern detectors" have proven to be quite useful for finding bugs [10, 18].

just like for problems in NP, it may be difficult to 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*.

1.2 Contributions

We are exploring the possible benefits of dynamic annotation inference. In this paper, we describe the progress we have made on a prototype system that employs an Eraser-like dynamic analysis to generate candidate annotations for the Race-Free Java (RFJ) type system [12]. This paper makes the following contributions:

- We present a new algorithm (Section 2) for dynamic annotation inference. Our algorithm improves on prior static [14] and dynamic [1] algorithms in being fully context-sensitive (polymorphic), and thus is able to infer types for more programs.
- We describe our experience applying our tool FindLocks to prove the absence of race conditions in a number of variously-sized Java programs (Section 3). This experience speaks to both the power of the static checking system (RFJ) and the efficacy of dynamic inference. We extend prior studies in both areas [13, 1]. In our experience, dynamic inference imposes reasonably little runtime overhead, and infers most needed annotations.
- After comparing to related work (Section 4) we present a number of lessons learned, and lay out a path for continuing work (Section 5). Two key lessons learned are as follows. 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 with dynamic traces to improve the quality of the inferred types.

2. DYNAMIC LOCK TYPE INFERENCE

To check for race conditions, most static and dynamic tools seek to infer or check the *guarded-by* relation. This relation describes which memory locations (and possibly what sequence of operations) 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 a lock is held, thus ensuring mutual exclusion. In a dynamic system like Eraser [23], 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 race conditions, 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*. The guarded-by annotation can refer to this variable as if it were a local field; however, it cannot be acquired within the class, because it is a type-level variable that exists only at compile-time. All locks must be `final`.

Here is a small example of a class that contains both an internal and external lock:

```
class C<ghost Object o> {
  int count guarded_by this;
  int value guarded_by o;
  synchronized void set(int x) requires o {
    count++;
    value = x;
  }
}
```

`C`'s field `count` is guarded by `this`, an internal lock, while `value` is guarded `o`, 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 `count` is legal. In addition, the `requires` clause ensures that `o` is held whenever `set` is called.

A ghost variable is instantiated when the object is created. For example:

```
C<this> x = new C<this>();
synchronized (this) {
  x.set(1);
}
```

This code fragment creates a `C` object, instantiating its external lock with the current object. Thus, the call to `x.set(1)` is legal, because the external lock is held before the call.

RFJ also supports the notion of a class whose objects are never shared between threads (they are *thread local*), and thus no lock need be held to access their fields. This mechanism for thread-local data is a weakness of the RFJ type system, as we elaborate further in Section 3.

2.2 Dynamic Annotation Inference

Our goal is to automatically infer `guarded_by` and `requires` annotations for unannotated Java programs. To do this, the target program is instrumented and executed, and we use an Eraser-like algorithm to infer the guarded-by relationship dynamically. The results are used to generate candidate annotations that can be checked statically.

During execution, for every object o and field `f` we maintain the set $lockset(o, \texttt{f})$. The lockset is the set of locks that are consistently held when accessing o.`f`. In Eraser, locks and objects o are merely machine addresses. We must additionally store statically expressible "names" for each lock. These names are of the form (`C`,L), such that within an instance of class `C`, L is a valid field expression for the object. Because some objects have an infinite number of potential names (e.g., `List.next.next.next` ... for a cyclic linked list) we set a constant maximum length for the names.

We also maintain a set containing all locks currently held, and their names. Every field access uses this set to refine the lockset for the involved object/field pair: the set of objects in the lockset is intersected with the set of locks currently held; additionally, the set of names for each object in the lockset is intersected with the set of names that is currently valid for that lock.

Once the execution has completed, we attempt to resolve each field's lockset into either an internal or external lock. Consider the *result set*:

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

If we can find a single field expression L within the scope of `C` such that, for all $(o, ls) \in R(\texttt{C.f})$, L names an object in ls, then we have found an internal lock for `C.f`. In the common case, this means simply looking for a field of `C`, preferably `this`, that guards `f` in every instance of `C`. We also consider static fields of another class if no internal field exists.

If no internal lock exists, we must parameterize `C` and look for names at `C`'s allocation sites. Let $allocsite(o)$ be the allocation site where the object o was allocated. Let $allocator(o)$ be the address of `this` when o was allocated. We can partition the result set based on allocation site I, as follows:

$$R(\texttt{C.f})[I] = \left\{ (a, ls) \;\middle|\; \begin{array}{c} (o, ls) \in R(\texttt{C.f}) \\ I = allocsite(o) \\ a = allocator(o) \end{array} \right\}$$

$R(\texttt{C.f})[I]$ can be used to compute the value of the instantiation parameter for `f` for objects created at allocation site I, just as $R(\texttt{C.f})$ contained the information that was used in the initial attempt to resolve an internal lock for `f`.

In other words, we consider separately the n locations where `C` is allocated, and attempt to provide a lock name for `f` that is valid at that location. Because the set associated with each location is smaller than the original set (when $n > 1$), it is possible that each subset is resolvable (i.e., will have a consistent name for the protecting lock) even when the original set is not. If a subset is not resolvable, we repartition the subset: because the partitions have the same structure as the original set, this procedure can be repeated recursively until we reach a static allocation site. Each repartitioning adds a class parameter to the class to be instantiated.

Because each field's lock is resolved separately, a class will initially have as many parameters as it has externally-locked fields. Because classes rarely use more than a single external lock, these parameters will generally be redundant. Thus, FindLocks will merge parameters if, at every instantiation site, the parameters are equivalent.

Consider the example of external locking shown in Figure 1 adapted from Java-Server, one of our case studies. Here the class `CircList` is used as a buffer for log messages. The class provides no synchronization constructs of its own. Instead, the client `LocalLog` protects accesses to the buffer using itself as the lock.

When attempting to infer the `guarded_by` clause of the `alist` field of `CircList`, FindLocks is unable to discover a candidate among the names within the scope of `CircList`. By examining the set of names available at the allocation site of `CircList` within `LocalLog`, FindLocks is able to add the appropriate instantiation parameter for `clist` and make the type of `CircList` polymorphic in the lock.

The above description omits one important special case: classes which allocate themselves. An example is an externally guarded linked list, where each element creates its successor. If we were to use the above algorithm on such a list, we would end up with as many parameters as there are elements in the list. The initial partitioning of the set would result in a set of size one (containing the head of the list), and a set of size $len - 1$, containing the rest of the elements; $len - 1$ subsequent repartitionings would be required to each the ancestral allocator of the tail of the list, outside of the list class.

```
class LocalLog {
  CircList<this> clist guarded_by this;
  LocalLog(int size) {
    clist = new CircList<this>(size);
  }
  synchronized void add(LogRecord lr){
    clist.add(lr);
  }
}

class CircList<ghost Object _l>{
  ArrayList alist guarded_by _l;
  CircList(int size) requires _l {
    alist = new ArrayList(size);
  }
  boolean add(LogRecord lr) requires _l {
    if (isFull()) {
      int oldestIdx = getOldestIndex();
      alist.set(oldestIdx, lr);
    }
    else alist.add(lr);
    return true;
  }
}
```

Figure 1: Example of Polymorphic Locking

To solve this case, we require that every instantiation site of class `C` within class `C` must supply the allocated class with the same parameters that it received; i.e., class `C<a,b>` will only allocate `C` as `C<a,b>`. In effect, we forbid polymorphic recursion of class parameters.

Now, the resolution on result sets is thus changed so that, instead of partitioning objects based on their allocation sites, they are partitioned based on their most recent *external* ancestor in the tree formed by the allocation relation $\{(a, o) \mid a = allocator(o)\}$. For example, assume there are three classes `C`, `D`, `E` which each create a linked list of ten elements of class `L`. Even though 27 of the elements were allocated within class `L`, a single recursive step will result in all 30 objects being associated with the allocation sites of the head elements within `C`, `D`, and `E`, skipping the sites within `L`.

2.3 Implementation

FindLocks executes in two phases. First, the target program is instrumented using the BCEL [7] bytecode manipulation library and executed. During execution, we track field writes to maintain the mapping between the objects and names; we track lock acquire/release to maintain the set of locks held and their names; and we track field reads/writes in order to determine the locks that are consistently held for every object/field pair. We also record the allocation site of every object, to be used in resolving external locks. We do not need to instrument bytecodes that read and write local variables: their names are not relevant, because they can't be locks; and, because the analysis is dynamic, we always have the identity of objects when they are used. Once execution terminates, we resolve the locksets as described above to produce annotations. These annotations are added to the source code by an external tool, which is then checked by RccJava, a type-checker for RFJ.

It should be noted that most JVMs make strong assumptions with regard to the layout of the certain core classes, particularly those in the the `java.lang` package. Our technique involves instrumenting all application code and, only where permissible, instrumenting library classes. We have found that in certain situations it is not straightforward to instrument packages such as `java.util` since there exist circular dependencies from this package to other packages such as `java.lang` which may not be modified. To avoid this problem, core library classes could be annotated manually; this would only need to be done once. In the description of our experiments that follow the exact set of instrumented classes will be noted.

3. EXPERIMENTAL RESULTS

In this section, we describe our experience using FINDLOCKS on a number of Java programs. We present the runtime overhead of running FINDLOCKS. Next we address the accuracy, expressiveness and completeness of the annotations emitted by FINDLOCKS. Finally, we describe our experience using our tool on a large program.

3.1 Sample Programs

Table 1 lists our benchmark programs, with relevant statistics in the first two columns. *Classes* refers to the number of classes that were instrumented. The numbers in parentheses indicate the number of library classes that were instrumented. *LOC* is the number of non-comment non-blank lines of code.

The programs ELEVATORS1 and ELEVATORS2 were written by students at the University of Maryland as part of CMSC433, a course in object-oriented programming. They simulate the scheduling of elevators in a building. Each program came with its own test cases that ran various simulations. For both ELEVATORS programs we were able to instrument the 152 classes that form the `java.util` library.

The program PROXY-CACHE was adapted from a program developed at the Technion, Israel Institute of Technology. It consists of an HTTP proxy that runs on a local server. It also provides content caching. Our test cases consisted of stressing PROXY-CACHE with concurrent requests using HTTPERF [19].

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.

JAVA-SERVER is small HTTP server that was developed at the University of Maryland. The program came with its own test cases that consisted of placing about 50 requests to the server.

3.2 Runtime Overhead

In Table 1 the column *Orig* refers to the maximum memory and elapsed time consumed by the program prior to instrumentation. The columns labeled *Instr* refers to the resources consumed by the instrumented program. The *Annot* columns refer to the resources consumed while annotating the source-code with the results of the inference. 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.

In each of these cases the overhead incurred by the instrumented code is within acceptable limits. While the resources used by the annotation phase may appear extravagant, it should be noted that this phase can easily be integrated into the RCCJAVA framework. The cost of annotation can thus be amortized against the cost of analysis performed by RCCJAVA. We report the expense of the annotation phase only for completeness.

3.3 FINDLOCKS and RCCJAVA

Table 2 describes the results of running RCCJAVA on the annotated programs. The column *Classes* shows the number of classes that were actually annotated. Classes contain no annotations if either the test cases did not cover the class, or sometimes if the class contains no fields. The column *Auto* refers to the number of annotations that were added automatically by FINDLOCKS. In a few cases we were required to add annotations manually. These are recorded in the column *Manual*. The section of the table labeled *Rcc Warnings* classifies the type of warnings issued by RCCJAVA when run on the annotated programs. *Thl* represents spurious race condition 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 race conditions.

It is clear from the table that the overwhelming majority of warning issued by RCCJAVA refer to thread local fields. Our analysis is able to easily discover when a field is accessed only by a single thread, or if the field, after initialization, is a read-only field. Furthermore, FINDLOCKS notes in particular the case where an object is constructed by one thread and is then handed off to another thread. In each of these cases FINDLOCKS annotates the source with comments (invisible to RCCJAVA) that assists the user in classifying RCCJAVA warnings as spurious or genuine. These results reflect the weakness of the RFJ type system's handling of thread-local data. A more advanced type system would be able to check these usages [8, 16].

The dynamic analysis also assists the user in ignoring RCCJAVA warnings about non-final expressions used as locks. The read-only annotations added by FINDLOCKS help the user to confirm that lock expressions are constant. This was particularly useful in the case of WEBLECH. Again, a stronger static analysis would be able to check these cases.

Manual annotations were added in some cases to suppress some warnings. For example, 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. However, ELEVATORS1 uses a dummy object as a mutex instead of synchronizing on `this`. Thus in the constructor of the object RCCJAVA issues warnings about the mutex not being held, since it only assumes that constructors hold the `this` lock. Despite adding the annotation to escape these warnings ELEVATORS1 fails to typecheck under RCCJAVA because it contains a real race condition. This race condition is also detected by FINDLOCKS. In this case, FINDLOCKS adds a comment to the field noting the problem.

ELEVATORS2 uses an external synchronization mechanism to guard instances of `java.util.HashSet`. That is, there is a field `elevators` of type `HashSet` and each access to this field is protected by obtaining a lock external to the

Program	Classes	LOC	Memory (MB)			Time (sec)		
			Orig	Instr	Annot	Orig	Instr	Annot
Elevators1	4(+152)	567	8.8	48.1	110	8.9	10.0	23.0
Elevators2	4(+152)	408	8.7	46.6	112	8.4	10.2	22.6
Proxy-Cache	7	1218	12.0	49.7	112	9.8	21.4	14.9
Weblech	12	1306	33.1	48.7	127	17.5	18.8	20.3
Java-Server	36	1768	10.3	37.4	126	7.0	7.9	15.2

Table 1: Runtime Overhead for FindLocks

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

Table 2: Checking Annotated Programs

scope of the `HashSet`. FindLocks infers that `HashSet` has a type that is polymorphic in the type of the lock. FindLocks correctly annotates the `java.util.HashMap` field of the `HashSet` field as being guarded by the lock parameter. Furthermore, an inner class of `HashMap`, `HashMap.Entry` is also parameterized by the same lock parameter. (This is why Elevators2 annotates six classes: three are from the program itself, and two are from the standard library.)

Java-Server also uses a similar external synchronization construct as described in Section 2.2. Two manual annotations were required to handle a class that was not executed by our test cases.

We were able get Proxy-Cache to type check without any further annotations. RccJava does, however, issue warnings regarding a two fields of array type. The contents of the array are read-only expressions and do not require any synchronization.

Attempting to type-check Weblech reveals another limitation of RccJava. The code in Figure 2 is illegal in RccJava: even though every access to the field `Spider.q` is guarded by `Spider.q`, it is not possible to instantiate the lock parameter of the `DownloadQueue` object with `Spider.q`. Using a separate mutex allows the lock parameter to be instantiated correctly.

Weblech also reveals a problem associated with subtyping of methods in the presence of `requires` annotations. The `DownloadQueue` object overrides the `Object.toString()` method in which it accesses all its fields. But annotating the overridden method with a requires clause that contains the lock parameter is illegal since required lock sets on function types are contravariant with regard to subtyping. Thus, an assertion that the lock was held was added to handle this case.

```
class Spider {
  DownloadQueue<q> q guarded_by q;
  Spider() {
    q = new DownloadQueue<q>();
  }
  URL nextURL(){
    synchronized(q) {
      return q.nextURL();
    }
  }
}
class DownloadQueue<ghost Object _l>{
  ArrayList urls guarded_by _l;
  DownloadQueue() requires _l {
    alist = new ArrayList();
  }
  URL nextURL() requires _l {
    return urls.remove(0);
  }
  String toString() requires _l {
    return urls.toString();
  }
}
```

Figure 2: Illegal Code snippet from Weblech

3.4 Scaling to Large Programs

We also ran FindLocks on HSQL[2], an open-source, JDBC-compliant database. HSQL consists of 260 classes and about 55000 lines of code. Unfortunately, we did not have access to a comprehensive test-suite for the application. Instead, we devised a simple test program that spawned a large number of threads and repeatedly performed simple queries on the database.

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

We found both the runtime overhead as well as the annotation overhead to be acceptable. However, the accuracy of the annotations that were inserted were greatly undermined by the extreme simplicity of the test case. Our test case only managed to cover some 90 out of the 260 classes. The annotations generated are thus skewed with respect to the particular execution trace that the instrumented program generated.

A large number of the 208 warnings issued by RccJava were with regard to fields that were marked thread local by FindLocks. While the inability to handle thread-local fields is an obvious limitation of RccJava, the accuracy of the annotations generated by FindLocks is also questionable. As with any dynamic analysis, FindLocks is limited to drawing unsound conclusions about the program based only on the executions that the analysis has witnessed. A stronger type system is needed in this case.

In light of this experience, it becomes clear that our approach 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 relationships from program traces.

4. RELATED WORK

Our approach is an example of what we call a *dynamic-static analysis*, in which dynamically-gathered information is used to support or improve a static analysis. Ernst's Daikon tool [11] infers simple invariants between variables in a program through run-time profiling. Nimmer and Ernst [20, 21] 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 [4] 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 [5, 6, 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 race conditions. However, we know of only two approaches that infer types for such systems, to relieve the annotation burden on the programmer. Houdini [14] is a self-described *annotation assistant* that statically generating sets of candidate annotations based on domain knowledge. Houdini was applied to a simplified version of race-free Java [13]. Unfortunately, Houdini does not support external (polymorphic) locks, restricting the set of programs for which it can infer types.

Concurrently with us, Agarwal and Stoller [1] developed a dynamic inference algorithm for the Parameterized Race-Free Java (PRFJ) [8] type system. Their algorithm is similar to ours in many respects. One difference is that it is not *fully* context-sensitive in that 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. With the knowledge of these lock parameters, their algorithm can infer how to most appropriately instantiate them. Our algorithm not only instantiates lock parameters, but can infer them as well by maintaining an allocation map between an object and the object that allocated it. This allows us to generalize (or "resolve" using our terminology) to arbitrary depth in the allocation chain, creating as many parameters as needed. This is particularly useful for library-like functions, like the `CircList` class we used as an example, which may wish to admit a variety of locking patterns. Agarwal and Stoller's algorithm handles some advanced features of PRFJ not present in RFJ, such as *unique* and *read-only* objects. It would be interesting to carefully consider how the two approaches could be combined, as we discuss below.

5. CONCLUSIONS

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 race conditions, based on collected traces. Indeed, our tool inferred the majority of annotations needed for idioms RFJ could check. Moreover, a number of applications made use of external locking, and our approach correctly parameterized classes to express this fact, an improvement over past work [1, 14].

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

1. 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 tool discovered these situations easily, but the static analysis could not check them.

   The solution is to develop a stronger complementary static checking system. Indeed, PRFJ fixes the first problem, and, to a limited degree, the second, by allowing uniquely referenced objects to be unguarded. This is sufficient to allow hand-offs between threads, which are supported in RFJ only by escapes from the type system. (Other idioms, such as barrier synchronization, remain uncheckable.) The type inference algorithm for PRFJ developed by Agarwal et al. [1] is able to indicate uniquely referenced objects (using a complementary static algorithm described in [3]). An immediate possibility for future work is to develop an inference algorithm for PRFJ that combines this ability with our algorithm's ability to infer multiple owner parameters, which are analogous to external locks in RFJ.

   A more ambitious approach would be to develop a more sophisticated type system which requires more annotations, since we have a tool to assist with annotation inference. 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 should 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 will supply needed annotations.

2. 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 HSQL.

   We believe that the right approach to this problem, beyond having more comprehensive tests, is to have the dynamic analysis "add value" to a more traditional static inference system. This is similar to the idea of profile-directed compilation [5, 22]. In this case, generated code is improved by considering run-time profiles. In our case, candidate annotations could be generated both statically and dynamically, and checked for soundness in the style of Houdini [13, 14]. One challenge would be the effective handling of class parameterization.

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 work 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 [17, 24], little has been done to explore the two at the level of actual programs. For example, Ernst [11] has found that dynamically-inferred properties sometimes hold statically, but does little to explain why. We intend to consider program traces and programs that induce them, following abstract interpretation [9].

6. REFERENCES

[1] Rahul Agarwal and Scott D. Stoller. Type Inference for Parameterized Race-Free Java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, Venice, Italy, January 2004. Springer-Verlag.

[2] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96, Kyoto, Japan, March 1998. Springer-Verlag.

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 311–330, October 2002.

[4] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 2002.

[5] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–84, Montreal, Canada, June 1998.

[6] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, New Mexico, January 1992.

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

[8] Chandrasekhar Boyapati and Martin 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*, pages 56–69, November 2001.

[9] Patrick Cousot and Radhia 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*, pages 238–252, 1977.

[10] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, Bolton Landing, New York, October 2003.

[11] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[12] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver B.C., Canada, June 2000.

[13] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 90–96, Snowbird, Utah, June 2001.

[14] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assitant for ESC/Java. In J. N. Oliverira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods*, number 2021 in Lecture Notes in Computer Science, pages 500–517, Berlin, Germany, March 2001. Springer-Verlag.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002.

[16] Dan Grossman. Type-Safe Multithreading in Cyclone. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 13–25, New Orleans, Louisiana, USA, January 2003.

[17] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. Technical Report 2003-1908, Cornell Unviversity Department of Computer Science, 2003.

[18] David Hovemeyer and William Pugh. Finding Bugs Is Easy. `http://www.cs.umd.edu/~pugh/java/bugs/docs/findbugsPaper.pdf`, 2003.

[19] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[20] Jeremy W. Nimmer and Michael 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)*, July 2001.

[21] Jeremy W. Nimmer and Michael D. Ernst. Invariant Inference for Static Checking: An Empirical Evaluation. In *Tenth Symposium on the Foundations of Software Engineering*, pages 11–20, Charleston, South Carolina, USA, November 2002.

[22] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, New York, June 1990.

[23] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, St. Malo, France, October 1997.

[24] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.

[25] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 1–11, San Jose, CA, 1994.