Detecting Race Conditions in Large Programs

Cormac Flanagan Stephen N. Freund
Compaq Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94301
{cormac.flanagan, stephen.freund}@compaq.com

ABSTRACT

The race condition checker `rccjava` statically identifies potential races in concurrent Java programs. This paper describes improvements to `rccjava` that enable it to be used on large, realistic programs. These improvements include not only extensions to the underlying analysis, but also an annotation inference algorithm and a user interface to help programmers understand warnings generated by the tool. Experience with programs containing up to 500,000 lines of code indicate that it is an effective tool for identifying races in large-scale software systems.

1. INTRODUCTION

A race condition occurs when two threads manipulate a shared data structure simultaneously, without synchronization. Race conditions are common errors in multi-threaded programs, and since they are timing-dependent, they are notoriously hard to catch using testing.

In a previous paper [10], we described `rccjava`, a static analysis tool that has successfully caught race conditions in a variety of small to medium-sized Java programs. This paper describes our work and experience in scaling `rccjava` to significantly larger programs. In particular, we address improvements to `rccjava` in the following three areas:

Annotation inference: `rccjava` is an annotation based tool, relying on the programmer to supply annotations that describe the locking discipline, such as which lock protects a particular field. In practice, this limitation has restricted the application of `rccjava` to small to medium-sized (about 20 KLOC) programs for which the task of writing annotations is tolerable. To achieve practical analysis of large programs, we developed an annotation inference system for `rccjava` based on the Houdini framework [11].

Reducing spurious warnings: Since statically detecting race conditions is undecidable in general, `rccjava` is incomplete by design and may produce false alarms for certain programming idioms. To keep the number of false alarms manageable when analyzing larger programs, we have extended `rccjava` to accurately reason about additional programming idioms that often occurred in larger programs.

User interface: The task of processing and understanding `rccjava`'s output is a labor-intensive process, particularly for large programs with many potential race conditions. To assist in this process, we have developed a simple but effective user interface that describes the potential races. It also clusters race conditions together according to their probable cause so that related race conditions can be dealt with as a single unit. In addition, the user interface describes the analysis performed by the annotation inference system in such a way that the programmer can easily understand the cause of warnings.

The combination of these improvements have allowed us to successfully analyze programs containing up to half a million lines of code.

The presentation of our results proceeds as follows. Section 2 starts with a review of `rccjava`. Section 3 introduces `Houdini/rcc`, our annotation inference system for `rccjava`, and Section 4 describes extensions to the system necessary for handling large programs. Section 5 describes the user interface. Section 6 describes our experience using `Houdini/rcc` to catch race conditions in several large test programs. Section 7 describes related work, and we conclude in Section 8.

2. THE RACE CONDITION CHECKER

This section reviews `rccjava`. As described in an earlier paper [10], `rccjava` is an extension of Java's type checker that identifies race conditions in multi-threaded Java programs. It supports the lock-based synchronization discipline [**?**], which is the dominant synchronization discipline in Java programs.

The `rccjava` checker relies on some additional type annotations providing information about the locking discipline, such as which lock guards a particular field. It checks that these annotations are respected by the program and are sufficient to ensure the absence of race conditions.

To illustrate this technique, consider the simple bank account class shown in Figure 1. This class contains a small number of `rccjava` type annotations. The annotation `guarded_by lock` on the field `balance` indicates that the lock `lock` must be held whenever **balance** is accessed or assigned. Similarly, the annotation `requires lock` on the method `update` indicates that `lock` must also be held whenever `update` is invoked.

PASTE'01, June 18-19, 2001, Snowbird, Utah, USA.

```
class Account {
  final Object lock = new Object();

  /*# guarded_by lock */
  int balance = 0;

  /*# requires lock */
  void update(int n) { balance = n; }

  void deposit(int x) {
    synchronized(lock) {
      update(balance + x);
    }
  }
}
```

Figure 1: An annotated `Account` class.

Thus, using `rccjava` to check the code fragment:

```
Account a = new Account();
a.balance = 100;
a.update(100);
```

would yield warnings that `lock` is not held either on the assignment to `balance` or on the invocation of `update`.

On the other hand, the following (correctly synchronized) code is accepted by `rccjava`:

```
Account a = new Account();
synchronized(a.lock) {
  a.balance = 100;
  a.update(100);
}
```

In general, lock names may be any constant values, such as `this`, any final variable, or any final field of a constant reference. To check that these annotations are respected by the program, `rccjava` computes a conservative approximation of the locks held at each program point and verifies that the necessary locks are held on field accesses and method calls.

Multi-threaded programs often contain a significant amount of data local to individual threads. For example, consider a Web server that contains a number of worker threads to handle http requests. Each of the worker threads may create and use data structures that are not shared among the other threads, and hence do not synchronization.

To handle this situation, `rccjava` allows a class to be annotated as `thread_local`; such a class does not require locks guarding its fields. All potentially shared fields (including static fields and instance fields of objects that are not local to a single thread) must be annotated with a guarding lock.

Although `rccjava` supports a number of additional features, such as classes parameterized by locks, this paper focuses primarily on `guarded_by`, `requires`, and `thread_local` annotations. For more details, we refer the interested reader to an earlier paper [10].

3. ANNOTATION INFERENCE

The `rccjava` checker has proven capable of detecting race conditions in a variety of programs. Unfortunately, before using `rccjava` to detect these race conditions in a program, it is necessary to first add appropriate `rccjava` type annotations to that program. Our experience to date indicates that a programmer can annotate an existing, unannotated program at the rate of 1000 lines per hour. While this annotation overhead is tolerable for small to medium-sized programs, it becomes quite costly for larger programs.

To make `rccjava` a more cost-effective tool for catching race conditions in large programs, we have developed an *annotation assistant* that infers suitable `rccjava` annotations for an unannotated program. This annotation assistant, called `Houdini/rcc`, is based on the Houdini annotation inference architecture [11]. `Houdini/rcc` infers annotations using the following algorithm.

```
generate candidate annotation set;
repeat
  invoke rccjava to refute annotations;
  remove the refuted annotations
until quiescence
```

The simplicity of this algorithm is due to its reuse of `rccjava` to reason about the correctness of particular annotations.

The first step in the algorithm is to generate a finite set of *candidates annotations*. Each candidate annotation is a conjectured property of the locking discipline used by the program. For each class, the candidate annotation set includes an annotation `thread_local` conjecturing that all instances of that class are local to a particular thread.

In addition, `Houdini/rcc` conjectures that each non-final field is guarded by a number of different candidate locks. The candidate locks include `this` and any final field declared in the same class or a superclass. Similarly, for each routine, `Houdini/rcc` conjectures that each of these candidate locks must be held on entry to that routine. `Houdini/rcc` does not conjecture requires clauses for methods that are called by the Java run-time systems without any locks being held: such methods include `main()`, the entry point of the program, and `run()`, the entry point of a particular thread.

Many candidate annotations will of course be incorrect. To identify incorrect annotations, the Houdini algorithm invokes `rccjava` on the annotated program. Like any invocation of `rccjava`, this invocation produces warnings about violations of the given annotations. `Houdini/rcc` interprets such warnings as identifying incorrect annotation guesses in the candidate set. In this sense, each invocation of `rccjava` has the effect of refuting some number of candidate annotations, and these annotations are then removed from the program.

Since removing one annotation may cause other annotations to become invalid, this check-and-refute cycle iterates until a fixed point is reached. At that point, all incorrect annotations have been removed from the program. The set of remaining annotations is a correct subset of the candidate set, and is in fact, the unique maximal such set [11].

We illustrate this annotation inference process for a small example program that includes the `Account` class from Figure 1. This program, together with the candidate annotations conjectured by `Houdini/rcc`, is shown in Figure 2. The new class `Add100` is a subclass of `Thread`. As such, invoking the start method of an `Add100` object causes the object's `run` method to be executed in the object's thread. Therefore, the `Add100.main` method spawns two new threads, both of which will add 100 to the balance of an `Account` object.

On the first iteration of the `Houdini/rcc` loop, the call to `rccjava` refutes three annotations. The annotation `thread_local` on the class `Add100` is refuted since each instance of this class

is also an instance of its superclass `java.lang.Thread`. Any object of this class can be accessed by two threads: both itself and the parent thread that started it. The `run` method of `Add100` is called by the Java run-time system without any locks being held, and hence `rccjava` also refutes the two requires annotations on this method.

On the next iteration of the `Houdini/rcc` loop, `rccjava` sees that the `run` method calls `deposit` without any locks being held, and hence refutes the two requires annotations on `deposit`. In addition, `rccjava` sees that the (now thread-shared) class `Add100` contains a reference to `Account`, and `rccjava` refutes the annotation `thread_local` on `Account`.

On the third iteration, `rccjava` refutes the annotation `requires this` on `update`, and on the fourth and final iteration, `rccjava` refutes the annotation `guarded_by this` on the field `balance`. The remaining annotations are all correct.

Once quiescence is reached, the `rccjava` checker is invoked last time on the now-annotated program to determine the set of warnings to be reported to the user. In the example, the final run produces no warnings, indicating that the program is indeed free of race conditions.

If, on the other hand, there were a race condition on a particular field, `Houdini/rcc` would refute all of the conjectured guarding locks. Thus, the final run of `rccjava` would produce a warning is that there is no lock guarding that field.

4. REDUCING FALSE ALARMS

While the basic `Houdini/rcc` algorithm can detect races in unannotated programs, experience has shown that it produces many false alarms. This section describes four extensions to `rccjava` and `Houdini/rcc` that help eliminate false alarms. We refer to each extension by the flag which configures `Houdini/rcc` to use that extension.

no_override: As mentioned in our previous work, `rccjava` produces a warning when a thread-local class overrides a method from a thread-shared superclass [10]. This warning is generated because of the potential to cast an object of the thread-local subclass to the superclass's type, allowing the object to be passed to multiple threads. These threads may then simultaneously access unguarded data in the thread-local portion of the object through the overridden method. Our experience has shown that, while many such overrides exist in large programs, they are typically not sources of races. The `no_override` option stops warnings due to method overrides from being reported. Clearly, this is an unsound extension, but it eliminates a significant number of false alarms.

cons_lock: A shared object is typically initialized in the object's constructor without acquiring any locks. This initialization pattern is sound provided that no references to the object being initialized escape from the creating thread until after the constructor exits. To allow such code to be checked without producing warnings, the `cons_lock` makes `Houdini/rcc` assume the lock `this` is held inside constructors. As in the previous case, this assumption eliminates many false alarms and is not considered to be a significant source of unsoundness.

read_only: Constant fields can be read safely by multiple threads without synchronization. Ideally, such fields would be declared as `final`, in which case `rccjava` would not warn about unsynchronized accesses. However, our experiments with `Houdini/rcc` indicated that a large number of shared constant fields are not declared as `final`.

To avoid false alarms in these cases, we extended `rccjava` to allow a field to be annotated with the annotation `readonly`. This annotation behaves much like Java's `final` annotation; in particular, a `readonly` field does not require a protecting lock. We modified `Houdini/rcc` to infer readonly annotations for constant fields that were inadvertently not declared `final`.

As an added benefit, since `readonly` reference fields are constant values, they can be included in the set of candidate locks for a class, thus increasing the set of candidate annotations conjectured by Houdini.

main_lock: Since static fields are accessible by all threads, `rccjava` requires that every static field is protected by a lock. However, a number of the programs we examined exhibited a common pattern whereby a static field would be

```
/*# thread_local */
class Account {
  final Object lock = new Object();

  /*# guarded_by lock */
  /*# guarded_by this */
  int balance = 0;

  /*# requires lock */
  /*# requires this */
  void update(int n) { balance = n; }

  /*# requires lock */
  /*# requires this */
  void deposit(int x) {
    synchronized(lock) {
      update(balance + x);
    }
  }
}

/*# thread_local */
class Add100 extends java.lang.Thread {
  final Account a;

  Add100(Account a) { this.a = a; }

  /*# requires this */
  /*# requires a */
  public void run() { a.deposit(100); }

  static public void main(String st[]) {
    Account a = new Account();
    (new Add100(a)).start();
    (new Add100(a)).start();
  }
}
```

Figure 2: The `Houdini/rcc` candidate annotations for `Account`. The shaded annotations are refuted by the `Houdini/rcc` algorithm.

```
class BadAccount {
  final Object lock = new Object();

  int balance = 0;

  void update(int n) { balance = n; }

  void deposit(int x) {
      update(balance + x);
  }
}

class Add100 extends java.lang.Thread {
  final BadAccount a;

  Add100(BadAccount a) { this.a = a; }

  public void run() { a.deposit(100); }

  static public void main(String st[]) {
    BadAccount a = new BadAccount();
    (new Add100(a)).start();
    (new Add100(a)).start();
  }
}
```

Figure 3: A version of `Account` that contains a data race on `balance`.

accessed exclusively by the main thread, without synchronization. To accommodate this programming pattern, we extended `rccjava` with the notion of a *main lock*, that is, a lock that is implicitly held by the main thread.

To infer annotations regarding the main lock, we extended `Houdini/rcc` so that it guesses the annotation `requires MainLock` for each method and the annotation `guarded_by MainLock` for each field. The refutation loop of `Houdini/rcc` then determines which fields are accessed and which methods are invoked only by the main thread. Thus, this extension avoids producing false alarms on fields accessed exclusively by the main thread.

5. USER INTERFACE

We now turn our attention to how a programmer can identify defects using the feedback from `Houdini/rcc`. The `Houdini/rcc` interface, based on the interface of Houdini for ESC/Java [12], generates the following output for an input program:

- a collection of HTML pages containing the *source code view* for each Java file analyzed. The source code view, described below, contains information about both the valid and invalid candidate annotations guessed by `Houdini/rcc`.
- a root HTML page listing the warnings produced by the final call to `rccjava`. Each warning message contains a hyper-link to the source view of the code at the location of the offending program line.

To illustrate the process of funding errors with this tool, consider `BadAccount`, a broken version of the `Account` class shown in Figure 3. Note that the synchronization code from `deposit` is missing, meaning that there is a potential race on field `balance`. Analyzing this program with `Houdini/rcc` generates a warnings file containing the following warning:

```
BadAccount.java:7: field 'BadAccount.balance'
must be guarded in a thread shared class
```

Clicking on this warning would open up the source code view for line 7 of the Account class (see Figure 4). The source code view displays all of the candidate annotations guessed by `Houdini/rcc`. A refuted annotation is grayed out, whereas a valid annotation is darkened (in this case, all annotations were refuted, but Figure 5, as described below, contains several valid annotations). `Houdini/rcc` also inserts the warning messages into in the source code view.

In a situation like this, a programmer who intended the field `balance` to be guarded by the lock `lock` would want to know why `Houdini/rcc` did not infer the annotation `guarded_by lock` for the field `balance`. Identifying the cause of most `rccjava` warnings often boils down to being able to answer such a question. To aid in this process, the `Houdini/rcc` interface makes each refuted annotation a hyper-link to the line of the program that refuted it. The figure shows that the set of candidate annotations for the field `balance` does in fact include a grayed-out annotation `guarded_by lock` . Clicking on this refuted annotation brings the user to an access of field `balance` where, as far as `rccjava` could tell, the lock `lock` is not held (line 10 of the program).

This may lead the user to understand whether the warning is caused by a real race condition in the program or is just a spurious warning, or this may just be one of a number of similar steps required to identify the source of the problem. For example, if the programmer had intended that the lock `lock` be held on entry to `update`, the link from the refuted annotation `requires lock` could be followed to line 15, where the required synchronization statement is missing. Surprisingly, our experience indicates that presenting the *refuted* annotations and the causes thereof is the most important aspect of the user interface.

Running `Houdini/rcc` on the correct version of `Account` produces no warnings and the source code view shown in Figure 5. This time the expected annotations were inferred and appear in bold.

5.1 Clustering Warnings

During our experiments, we noticed several cases where `Houdini/rcc` would incorrectly characterize a thread-local class `C` as thread-shared, due to conservative approximations introduced by the analysis. Unfortunately, in these cases, `Houdini/rcc` subsequently characterizes as thread-shared all classes reachable (transitively) from `C` and produces spurious warnings regarding race conditions on accesses to these fields.

To reduce this problem, we extended the user interface to group into a single cluster all of the warnings that were caused, either directly or indirectly, by `C` being characterized as thread-shared. The programmer can often deal with all the warnings in a cluster as a single unit. For example, a programmer who verifies that `C` is actually thread-local can easily ignore the entire cluster of warnings.

6. EVALUATION

We have evaluated `Houdini/rcc` on a number of test programs ranging in size from several thousand lines to a half

Program	LOC	Warnings per KLOC					
		rccjava	Basic Houdini/rcc	+no_override	+cons_lock	+read_only	+main_lock
Ambit	4,500	13.6	37.3	14.2	13.3	7.1	7.1
WebL	20,000	11.8	12.2	5.1	4.8	1.9	1.9
jbb2000	30,800	18.8	4.9	3.4	3.3	1.3	0.6
tlc	53,500	11.0	14.7	4.7	4.5	2.2	1.0
jigsaw	128,900	21.1	13.6	8.2	7.7	2.9	2.9
orange	28,000	17.7	33.3	14.0	13.6	6.0	3.6
red	445,000	16.6	9.0	5.2	4.8	2.2	2.2
Average		15.5	15.3	6.3	6.4	2.9	2.6

Table 1: Number of warnings produced by rccjava and various versions of Houdini/rcc.

Program	LOC	Annotations per KLOC		Time	Warnings		Number	Races
		Candidate	Valid	(min)	per KLOC	Total	Clusters	(found/clusters examined)
Ambit	4,500	433	78	4	7.1	32	6	0/6
WebL	20,000	262	43	9	1.9	37	10	6/10
jbb2000	30,800	282	74	9	0.6	17	17	0/17
tlc	53,500	758	124	31	1.0	52	30	4/30
jigsaw	128,900	375	49	62	2.9	367	78	0/30
orange	28,000	863	135	74	3.6	100	84	1/84
red	445,000	358	64	286	2.2	957	340	5/70

Table 2: Statistics for Houdini/rcc with all modifications.

million lines of code. This section describes our experiences with the following programs:

- Ambit: An interpreter for the Ambient calculus [6].
- WebL: An interpreter and run time for a scripting language for web applications [15].
- jbb2000: A Java SPEC benchmark modeling a server application [19].
- tlc: A multi-threaded model checker for the TLA specification language [26].
- jigsaw: A web server written in Java [25].
- orange and red: Internal Compaq systems.

Table 1 shows the results of running rccjava and various versions of Houdini/rcc on these programs. The table shows the size of each unannotated program and the number of warnings reported by running rccjava on it. The column labeled Basic Houdini/rcc corresponds to running Houdini/rcc in its original form from Section 3. The four remaining columns show the number of warnings reported by Houdini/rcc with the four extensions described in Section 4. Note that the additions to Houdini/rcc are cumulative across the columns so that the +cons_lock column reflects Basic Houdini/rcc run with both the no_override and cons_lock extensions. All of the columns are normalized to show the number of warnings reported per thousand lines of code.

Excluding method override warnings reduced the number of warnings by roughly a factor of two across the test programs. The read-only field inference also decreased the number of warnings by another factor of two. Although the other two extensions were not as consistent in their effectiveness, there were some programs in which they also significantly reduced the number of warnings produced by Houdini/rcc.

Table 2 shows more detailed statistics for running Houdini/rcc with all of the described extensions. From this table, it is clear that Houdini/rcc is able to successfully infer a nontrivial number of annotations. In general, it guesses roughly 350 candidate annotations per 1,000 lines of code, with roughly 1/4 of these annotations being valid. Our system ran in time proportional to the size of the program, processing approximately 2,000 lines per minute on a 667 MHz Alpha workstation.

In most examples, the clustering algorithm was successful at grouping related warnings. A representative situation of this appears in jigsaw. In that program, there is a DebugThread class which gathers and prints statistics about the program as it runs. This class accesses fields of a number of different objects both directly and through multiple levels of accessor methods without acquiring the necessary locks for those fields. The clustering algorithm identified the DebugThread as the common source of 92 such potential races. All of these races were deemed benign because the debugging code was intentionally designed to read the data in this way.

The last column in Table 2 reflects how many non-benign races we identified while studying the warnings reported by Houdini/rcc. Since we have not examined every warning or cluster for the larger example, this column shows both the number of real races found and the number of warning clusters examined. For example, six races were found in WebL while examining all 10 clusters.

7. RELATED WORK

A number of tools have been developed for race detection. Warlock [20] is a static race detection system for ANSI C programs. It is similar to rccjava in that it requires annotations, but it does not infer them. The extended static checker for Java (ESC/Java) is a tool for static detection of software defects [16]. It supports multi-threaded programming via annotations similar to the guarded_by and requires annotations, but it may still permit race conditions on unguarded fields, since it does not verify that unguarded fields only occur in thread-local classes. Other static and dynamic approaches for race detection are discussed in an earlier paper [9].

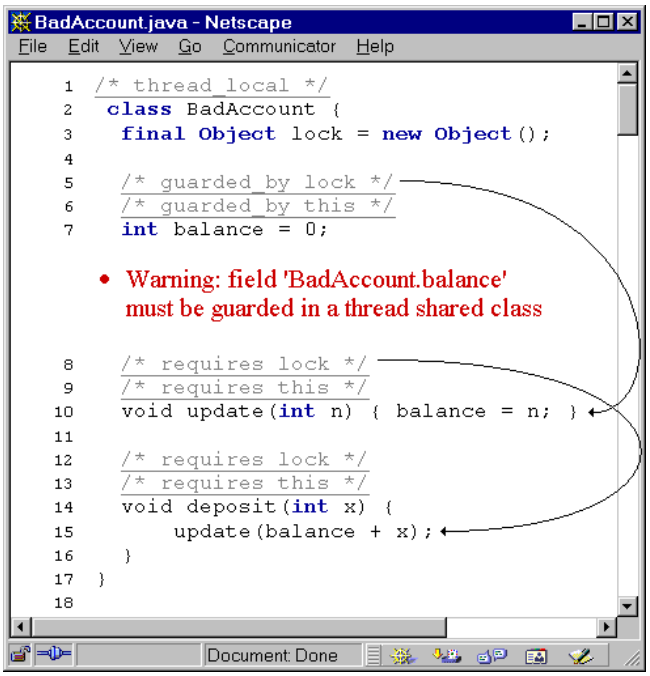

Figure 4: Screen shot showing the `Houdini/rcc` user interface for the broken `BadAccount` class. The overlayed arrows indicate where two of the hyper-links are pointing.

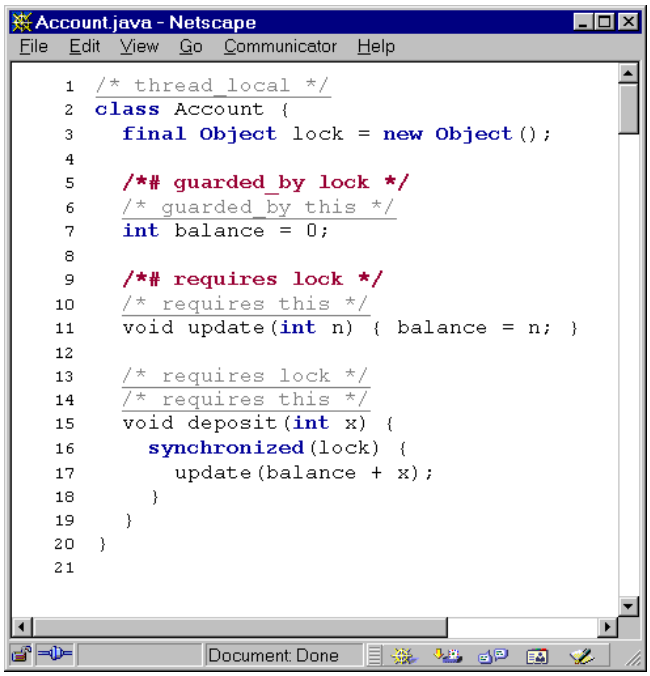

Figure 5: Screen shot showing the `Houdini/rcc` user interface for the `Account` class.

The `Houdini/rcc` algorithm can be viewed as an abstract interpretation [8], where the abstract state space is the power set lattice over the candidate annotations and `rccjava` is used to compute the abstract transition relation. `Houdini/rcc` may also be seen as a variant of predicate abstraction [13] in which each candidate annotation corresponds to a predicate. The Houdini algorithm finds the largest conjunction of these predicates that holds in all reachable states.

The `requires` annotations used by `rccjava` are similar to *effects* [14, 17, 18], in that the locks held on entry to a method constrain the effects that it may produce. Thus, the analysis performed by `Houdini/rcc` includes a basic form of effect reconstruction [22, 23, 2, 21], and the `Houdini/rcc` interface provides an explanation of why certain effects were inferred.

`Houdini/rcc` infers thread-local annotations for classes whose instances are never shared between threads. Other work on this escape analysis problem [7, 4, 5, 24, 1] has primarily focused on optimizing synchronization operations. A novelty of our work is that the `Houdini/rcc` interface provides an explanation of the reasoning performed by the analysis.

8. CONCLUSIONS AND FUTURE WORK

The techniques described in this paper are intended to make `rccjava` a practical tool for catching race conditions in large, realistic programs. These techniques have focused on reducing the annotation burden of running the checker by automatically inferring annotations, reducing the number of spurious warnings produced by the system, and reducing the effort required to understand the remaining warnings.

We believe that these improvements are sufficient to make `rccjava` a cost-effective tool in a development setting. We are currently working with the developers of some of the programs we tested in an attempt to validate this hypothesis.

9. REFERENCES

[1] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium*, September 1999.

[2] T. Amtoft, F. Nielson, and H. R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, 1997.

[3] A. D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.

[4] B. Blanchet. Escape analysis for object-oriented languages. Application to Java. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

[5] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

[6] L. Cardelli. Mobile ambient synchronization. Technical Report 1997-013, Digital Systems Research Center, Palo Alto, CA, July 1997.

[7] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[9] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of European Symposium on Programming*, March 1999.

[10] C. Flanagan and S. N. Freund. Type-based Race Detection for Java. In *Proceedings of the Symposium on the Programming Language Design and Implementation*, 2000.

[11] C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2001. To appear.

[12] C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Formal Methods Europe '01*, 2001.

[13] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.

[14] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[15] T. Kistler and J. Marais. WebL – a programming language for the web. *Computer Networks and ISDN Systems*, 30:259–270, April 1998.

[16] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. Technical Report 1999-002, Compaq Systems Research Center, Palo Alto, CA, May 1999. Also appeared in Formal Techniques for Java Programs, workshop proceedings. Bart Jacobs, Gary T. Leavens, Peter Muller, and Arnd Poetzsch-Heffter, editors. Technical Report 251, Fernuniversitat Hagen, 1999.

[17] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 47–57, 1988.

[18] F. Nielson. Annotated type and effect systems. *ACM Computing Surveys*, 28(2):344–345, 1996. Invited position statement for the Symposium on Models of Programming Languages and Computation.

[19] Standard Performance Evaluation Corporation. SPEC JBB2000. available from `http://www.spec.org/osg/jbb2000/`, June 2000.

[20] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[21] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[22] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[23] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[24] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

[25] World Wide Web Consortium. Jigsaw. available from `http://www.w3c.org`, January 2001.

[26] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME '99)*, number 1703 in Lecture Notes In Computer Science, pages 54–66. Springer-Verlag, September 1999.