# Effective Static Race Detection for Java

Mayur Naik Alex Aiken John Whaley

Computer Science Department
Stanford University
{mhn,aiken,jwhaley}@cs.stanford.edu

## Abstract

We present a novel technique for static race detection in Java programs, comprised of a series of stages that employ a combination of static analyses to successively reduce the pairs of memory accesses potentially involved in a race. We have implemented our technique and applied it to a suite of multi-threaded Java programs. Our experiments show that it is precise, scalable, and useful, reporting tens to hundreds of serious and previously unknown concurrency bugs in large, widely-used programs with few false alarms.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification — Reliability; D.2.5 [*Software Engineering*]: Testing and Debugging — Debugging aids

***General Terms*** Experimentation, Reliability

***Keywords*** static race detection, Java, synchronization, concurrency, multi-threading

## 1. Introduction

A multi-threaded program contains a *race* if two threads can access the same memory location without ordering constraints enforced between them and at least one access is a write. A race often implies a violation of a program invariant. Due to the non-deterministic nature of the thread schedules under which races occur, however, races are notoriously difficult to reproduce and fix. As a result, race detection tools are valuable for improving the reliability of multi-threaded programs.

The large body of work on race detection may be broadly classified as either dynamic or static. Dynamic race detectors are based on either the *happens-before relation* [1, 13–15, 35, 43, 46], the lockset algorithm [2, 10, 11, 38, 45, 49, 50], or a combination of the two [16, 28, 39, 40, 54]. Static race detectors are either primarily flow insensitive type-based systems [7, 8, 19, 20, 41, 44], flow sensitive static versions of the lockset algorithm [12, 17, 47], or path sensitive model checkers [29, 42].

Despite significant advances in static race detection, state-of-the-art race detection tools are still predominantly dynamic. We set out to develop a static race detection tool for Java and identified five key problems that we felt such a tool must address to be useful:

1. *Precision* — Does the tool have a tolerable false-positive rate?
2. *Scalability* — Does the tool handle large programs?
3. *Synchronization Idioms* — Does the tool handle the synchronization idioms used in real-world programs?
4. *Open Programs* — Does the tool handle open programs (e.g., a library, a device driver, etc.)?
5. *Counterexamples* — Does the tool provide sufficient information to identify and fix the bug, if any, manifested in a race?

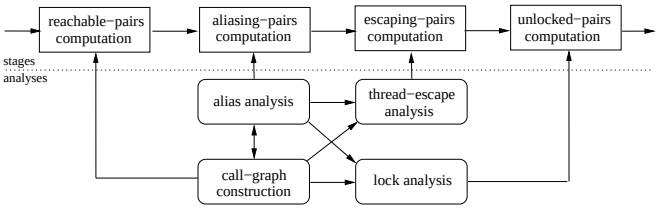


**Figure 1.** Overview of race detection algorithm.

In this paper, we present a novel technique for static race detection that satisfies the above criteria. We have implemented our method in a tool *Chord* and applied it to a suite of open-source multi-threaded Java programs, most of which are mature and widely used, and found 387 distinct bugs. In our largest benchmark (Apache Derby, a popular relational database engine), *Chord* analyzed 646K lines of Java source code and reported races revealing 319 distinct bugs. Tens of bug reports in two other programs, JdbF and jTDS, led developers to overhaul the synchronization in those systems. In a popular generic object-pooling library (Apache Commons Pool) that enables optimizing usage of resources like threads, sockets, and database connections, *Chord* exposed 17 bugs, all of which were fixed in 5 immediate dedicated patches. The output of our tool for all benchmarks along with detailed descriptions of the bugs and developers' responses is available at `http://www.cs.stanford.edu/~mhn/chord.html`.

Our race detection algorithm is illustrated in Figure 1. It consists of four stages and four static analyses. The arrows in the figure denote the order in which the stages and analyses are performed. The stages, *reachable pairs*, *aliasing pairs*, *escaping pairs*, and *unlocked pairs*, successively refine an initial over-approximation of the set of pairs of memory accesses potentially involved in a race. The stages depend upon four static analyses: call-graph construction, alias analysis, thread-escape analysis, and lock analysis. The bi-directional arrow between the call-graph and alias analyses denotes that they are mutually dependent and are computed simultaneously. Also, the thread-escape and lock analyses depend upon both the call-graph and alias analyses.

We next explain in Section 1.1 how we solve problems (1)–(3) outlined above, followed by an example in Section 1.2 illustrating how we solve problems (4) and (5).



*PLDI'06* June 11–14, 2006, Ottawa, Ontario, Canada.

### 1.1 Rationale

Central to our approach is a recent form of context sensitivity called *$k$-object sensitivity* [36, 37] that treats abstract contexts and abstract objects uniformly by defining the abstract contexts of an instance method as the abstract objects to which its distinguished $0^{th}$ argument, `this`, may be bound at run-time. (Static methods, which lack the `this` argument, have a lone abstract context denoted $\epsilon$.) We have found $k$-object sensitivity indispensable to the precision of alias analysis, and a precise alias analysis is vital to our approach since all other static analyses we use depend upon it (see Figure 1). Indeed, we chose this form of context sensitivity after experimenting with a variety of alias analyses, including inexpensive ones like CHA-based and context insensitive alias analyses that have been used in previous approaches to static race detection as well as expensive ones such as Whaley and Lam's $k$-CFA-based context sensitive alias analysis [53] where $k$ is the depth of the program's call graph after reducing strongly connected components to single nodes. In our experience, $k$-object sensitive alias analysis for even relatively small $k$ is much more precise than both context insensitive alias analysis and Whaley and Lam's $k$-CFA-based alias analysis: $k = 3$ sufficed in our experiments.

The increased precision from $k$-object sensitivity for even $k = 3$ incurs a significant scalability cost. All publicly available implementations of $k$-object sensitive alias analysis ran out of memory on most of our benchmarks for $k = 1$. Recent work has demonstrated the effectiveness of Binary Decision Diagrams (BDDs) [9] in scaling whole-program context sensitive analyses [6, 32, 53, 55]. We built *Chord* upon this work, expressing each of the four stages and analyses in our race detection algorithm in Datalog, a logic programming language, and solving them using `bddbddb`, a BDD-based implementation of Datalog [30].

Our race detection algorithm is context sensitive but flow insensitive. In general, lack of flow sensitivity helps scalability and hurts precision. In our race detection algorithm, however, lack of flow sensitivity primarily affects the kinds of synchronization idioms we can handle. We handle three idioms: lexically-scoped, lock-based synchronization, fork/join synchronization, and wait/notify synchronization. Our lack of flow sensitivity interacts with these idioms in the following ways:

1. Java encourages programmers to acquire and release locks in a lexically-scoped manner using the `synchronized (l) {...}` construct, which is naturally suited to flow insensitive analysis. (In contrast, a language like C does not have a lexically-scoped synchronization construct, thereby encouraging programmers to acquire and release locks in an interprocedural or path sensitive manner, which necessitates flow sensitive or path sensitive analysis.)
2. While lexically-scoped, lock-based synchronization is the prevalent synchronization idiom, Java programs also use fork/join synchronization, which necessitates flow sensitive analysis. Due to the lack of flow sensitivity in our approach, however, we rely on annotations specifying which threads cannot execute in parallel (see Section 2.2.1) and which fields/objects, although thread-shared, cannot be accessed simultaneously by different threads (see Section 2.2.3). The annotation burden is very small, partly because of the nature of the annotations and partly because our algorithm infers most of them automatically: we provided just 42 annotations in our benchmark suite containing 1.5 million lines of Java source code.
3. The `wait`, `notify`, and `notifyAll` constructs have little impact on race detection. A thread must acquire a lock on an object prior to calling any of these methods on that object. A call to `wait` on an object causes the calling thread to release the lock it acquired on that object prior to the call and block, but the thread resumes executing only after re-acquiring the lock. A call to `notify` or `notifyAll` does not cause the calling thread to release any lock. Thus, from the perspective of race detection, handling Java's wait/notify synchronization reduces to handling lock-based synchronization (item (1) above).

```
public class Database {
    private ConnectionManager cm;
    public int insert(...) throws MappingEx {
        Connection c = cm.getConnection(...);
        ...
    }
    public int delete(...) throws MappingEx {
        Connection c = cm.getConnection(...);
        ...
    }
}

public class ConnectionManager {
    private Map conns =
        Collections.synchronizedMap(new HashMap());
    public Connection getConnection(String s)
        throws MappingEx {
        try {
            ConnectionSource c = conns.get(s);
            if (c != null) return c.getConnection();
            throw new MappingEx(...);
        } catch (SQLEx e) { ... }
    }
}

public class ConnectionSource {
    private Connection conn;
    private boolean used;
    public Connection getConnection() throws SQLEx {
        if (!used) {
            used = true;
            return conn;
        }
        throw new SQLEx(...);
    }
}
```

**Figure 2.** An example from JdbF.

```
public class Harness {
    static public void main() {
        Database v;
        if (*) v = new Database(...);
        if (*) v.insert(...);
        if (*) v.delete(...);
        ...
    }
}
```

**Figure 3.** Harness synthesized for JdbF.

Finally, the lack of flow sensitivity causes our race detection approach to sacrifice soundness: checking whether a pair of accesses is guarded by a common lock requires a must-alias analysis, which is necessarily flow sensitive, whereas our lock analysis uses our (may) alias analysis, which is an unsound approximation (see Section 2.2.4). While the lack of soundness has not affected the usefulness of *Chord*, which found tens to hundreds of serious and previously unknown concurrency bugs in large, widely-used programs, we hope to incorporate flow sensitivity in our approach to perform sound race detection in the future.

### 1.2 Example

In this section, we illustrate our race detector by means of a real-world example. Figure 2 presents a code fragment from one of our benchmarks, JdbF, an object-relational mapping system that simplifies the work of retrieving, saving, and deleting objects from a relational database. The `Database` class provides an interface to clients of JdbF for performing various operations on the database. We only show the relevant parts of the `insert` and `delete` operations. Each operation acquires a connection, performs its task on the database, and releases the connection (not shown). The `ConnectionManager` class maintains a map of all available connections. Each `Connection` object is encapsulated in a `ConnectionSource` object intended to ensure that the connection is used by at most one database operation at any instant. JdbF is an open program — it is a library that cannot be run by itself.

Our race detection tool, *Chord*, automatically generates a harness, whose relevant parts are shown in Figure 3, and reports a race on instance field `used` declared in class `ConnectionSource`. Specifically, it reports in the form of a graph each pair of paths in the program's call graph leading from a pair of call sites in the harness to a pair of accesses of the field, at least one of which is a write. For efficiency of the analysis (see Section 2.1), the generated harness is single-threaded. Our race detection algorithm simulates the effect of a multi-threaded harness, so each pair of paths must be viewed as being executed in different threads. One such reported pair of paths is as follows, where `[...]` abbreviates `org.jdbf.engine.sql.connection`:

```
field reference ([...].ConnectionSource.used) [Rd]
[...].ConnectionSource.getConnection(ConnectionSource.java:104)
[...].ConnectionManager.getConnection(ConnectionManager.java:186)
org.jdbf.engine.database.Database.insert(Database.java:230)
Harness.main(Harness.java:25)

field reference ([...].ConnectionSource.used) [Wr]
[...].ConnectionSource.getConnection(ConnectionSource.java:105)
[...].ConnectionManager.getConnection(ConnectionManager.java:186)
org.jdbf.engine.database.Database.delete(Database.java:406)
Harness.main(Harness.java:26)
```

The race illustrates a serious violation of the invariant that each connection must be used by at most one database operation at a time. The JdbF developers incorrectly believed that making the map `conns` of available connections in class `ConnectionManager` a synchronized map suffices to ensure the invariant: the synchronized map serializes queries on the map of available connections but it does not serialize the processing of the query results.

In the above example, our tool reports six pairs of paths, each originating from a pair of calls to either the `insert` and `delete` methods, the `insert` method itself, or the `delete` method itself in the `main` method of class `Harness` and terminating in a pair of read/write or write/write accesses of the `used` field in the `getConnection` method of class `ConnectionSource`. Our tool also reports additional pairs of paths because the `Database` class defines methods besides `insert` and `delete`, and also because each such method not only acquires a connection but also releases it and the method that implements releasing a connection in class `ConnectionSource` writes to the `used` field (setting it to `false`). The JdbF developers fixed the bug by synchronizing the relevant parts of all methods in the `Database` class on the `this` object.

The above example highlights two hallmarks of our approach: the ability to analyze open programs and the ability to report counterexamples. We have found both features essential in practice. Many multi-threaded applications are written as open programs that provide an interface to interact with their environment. For instance, JdbF provides the `Database` class as an interface to clients. Developers of such programs would like to detect races before deploying the programs in specific environments.

Explaining a race involves various aspects such as the object on which the race occurs, the pair of threads accessing the object, and the sets of locks held by the threads. Merely reporting a race on the `used` field in an internal class like `ConnectionSource` makes it cumbersome to determine whether the race is a bug, a *benign race* (that is, a race that does not violate any program invariant), or a false positive. Thus, it is very useful to additionally report a pair of paths which, albeit at the call-graph level, shows that the race occurs if the `insert` and `delete` methods are invoked in different threads on the same `Database` object.

The rest of the paper is organized as follows. In Section 2, we present our race detection algorithm and its implementation in *Chord*. Section 3 presents our experimental results. In Section 4, we survey related work and, finally, Section 5 concludes.

## 2. Race Detection Algorithm

In this section, we present our race detection algorithm and its implementation in our tool *Chord*. We describe harness synthesis in Section 2.1, the four stages of our algorithm along with the four static analyses it uses in Section 2.2, a post-processing phase in Section 2.3, and soundness aspects in Section 2.4.

### 2.1 Harness Synthesis

Our race detection algorithm performs a whole-program analysis. As argued in Section 1.2, however, the ability to detect races in open programs is important. There are two problems with analyzing open programs: missing callees and missing callers.

We model missing callees using "stubs" in place of commonly used native methods in the JDK standard library and treat all other missing and native methods unsoundly as no-ops. We model missing callers by automatically synthesizing a harness that simulates many scenarios in which the program's environment might exercise its interface. Due to the complexity of the Java language, our current harness synthesis algorithm is not sound in that it does not simulate all possible scenarios. Nevertheless, it is much easier in principle to construct a sound harness for our technique than for a model checker that also performs whole-program analysis, because model checkers are typically path sensitive whereas none of the static analyses used in our technique is path sensitive and so the harness required by our technique need not be as elaborate.

Our harness synthesis algorithm takes as input an open program and a set $\mathcal{I}$ of interfaces whose implementations must be checked, and builds a main method that, for each interface $I \in \mathcal{I}$:

1. declares a local variable of each type allowed as an argument type or result type of any method declared in $I$,
2. non-deterministically assigns to each local variable of reference type $T$, an object of each concrete class of type $T$, and
3. non-deterministically invokes each method declared in $I$ on each combination of local variables respecting the argument types of the method and assigns the return value if any to each local variable respecting the result type of the method.

The non-determinism is needed to prevent the Soot framework [48] used in *Chord* from performing flow sensitive peep-hole optimizations (our race detection algorithm, being flow insensitive, does not need the non-determinism). We denote the set of call sites generated in step (3) by $I_{ext}$. Notice that the synthesized harness is single-threaded: our race detection algorithm simulates executing each pair of calls in $I_{ext}$ in separate threads on shared data. An example harness synthesized for the open program in Figure 2 is shown in Figure 3.

For convenience, we assume that a main method is synthesized for closed programs as well, containing a single call site that ex-

plicitly starts a thread that invokes the main method of the original program. We define $I_{ext}$ as the set containing this lone call site.

Henceforth, we assume that irrespective of whether we are given an open or a closed program, we are looking at a complete program with a main method *main* generated by our harness synthesis algorithm, and a set $I_{ext}$ of call sites that spawn threads in *main*. We use the program in Figure 4 as our running example. The set $I_{ext}$ for this program consists of call sites `a.get` and `a.inc` in the `main` method generated by our harness synthesis algorithm.

### 2.2 Algorithm

Our race detection algorithm is illustrated in Figure 1. It consists of the following four stages that successively refine an initial over-approximation of the set of pairs of memory accesses potentially involved in a race:

1. *reachable-pairs* computation,
2. *aliasing-pairs* computation,
3. *escaping-pairs* computation, and
4. *unlocked-pairs* computation.

These stages employ four static analyses: call-graph construction, alias analysis, thread-escape analysis, and lock analysis.

Given a complete program with a main method generated by our harness synthesis algorithm, we use the Soot framework [48] to compute *OriginalPairs*, our initial over-approximation of the set of unordered pairs of memory accesses potentially involved in a race. Java's strong typing dictates that a pair of accesses may be involved in a race only if both access the same instance or static field or both access array elements (and at least one access is a write). We use this fact to compute *OriginalPairs* (see Figure 5). It requires the following program information, where $x$, $y$, and $i$ are local variables:

- The set $\mathbb{F}_r$ (resp. $\mathbb{F}_w$) characterizing statements $y = x.f$ (resp. $x.f = y$) that read (resp. write) instance field $f$.
- The set $\mathbb{G}_r$ (resp. $\mathbb{G}_w$) characterizing statements $y = C.g$ (resp. $C.g = y$) that read (resp. write) static field $g$ of class $C$.
- The set $\mathbb{A}_r$ (resp. $\mathbb{A}_w$) characterizing statements $y = x[i]$ (resp. $x[i] = y$) that read (resp. write) an array element.

It is easy to transform a Java program such that each statement that accesses a field or an array element is of one of the above forms.

In our running example in Figure 4, the set *OriginalPairs* contains pairs $(\mathtt{f}_r, \mathtt{f}_w)$, $(\mathtt{f}_w, \mathtt{f}_w)$, $(\mathtt{f}_r, \mathtt{f}'_w)$, $(\mathtt{f}'_w, \mathtt{f}'_w)$, and $(\mathtt{f}_w, \mathtt{f}'_w)$, where $\mathtt{f}_r$ denotes the read access referencing field `f` in method `rd`, and $\mathtt{f}_w$ and $\mathtt{f}'_w$ denote the write accesses referencing field `f` in method `wr` and the constructor of class `A`, respectively.

We also use Soot to obtain program information needed as input to the four static analyses used by our race detection algorithm. These analyses, expressed in Datalog, are solved using bddbddb, in an order respecting the dependencies between them. We postpone describing each analysis until the presentation of the stage that uses it. Once all four analyses have been computed, we perform the four stages of our algorithm in order. Like the analyses, the stages are also specified in Datalog and solved using bddbddb. We next describe these stages in Sections 2.2.1–2.2.4.

#### 2.2.1 Reachable-Pairs Computation

The first stage of our algorithm prunes *OriginalPairs* using the fact that a pair of accesses may be involved in a race only if each access is reachable from a thread-spawning call site that is itself reachable from the synthesized *main*.

```
public class A {
    int f;
    static public void main() {
        A a;
        if (*) a = new A();
        if (*) a.get();
        if (*) a.inc();
    }
    public A() { this.f = 0; }
    private int rd() { return this.f; }
    private int wr(int x) { this.f = x; return x; }
    public int get() { return this.rd(); }
    public synchronized int inc() {
        int t = this.rd() + (new A()).wr(1);
        return this.wr(t);
    }
}
```

**Figure 4.** Running example.

$$
\begin{array}{rrcl}
(\mathit{instance\ field}) & f & \in & \mathbb{F} \\
(\mathit{static\ field}) & g & \in & \mathbb{G} \\
(\mathit{memory\ access}) & e & \in & \mathbb{E} = \mathbb{E}_f \cup \mathbb{E}_g \cup \mathbb{E}_a \\
(\mathit{get/set\ instance\ field}) & \mathbb{F}_r, \mathbb{F}_w & \subseteq & \mathbb{E}_f \times \mathbb{F} \\
(\mathit{get/set\ static\ field}) & \mathbb{G}_r, \mathbb{G}_w & \subseteq & \mathbb{E}_g \times \mathbb{G} \\
(\mathit{get/set\ array\ element}) & \mathbb{A}_r, \mathbb{A}_w & \subseteq & \mathbb{E}_a
\end{array}
$$

$$
\begin{array}{rcl}
F & = & \bigcup_{f \in \mathbb{F}} \{\, (e_1, e_2) \mid (e_1, f) \in \mathbb{F}_r \cup \mathbb{F}_w \ \wedge\ (e_2, f) \in \mathbb{F}_w \,\} \\
G & = & \bigcup_{g \in \mathbb{G}} \{\, (e_1, e_2) \mid (e_1, g) \in \mathbb{G}_r \cup \mathbb{G}_w \ \wedge\ (e_2, g) \in \mathbb{G}_w \,\} \\
A & = & \{\, (e_1, e_2) \mid e_1 \in \mathbb{A}_r \cup \mathbb{A}_w \ \wedge\ e_2 \in \mathbb{A}_w \,\}
\end{array}
$$

$$
\mathit{OriginalPairs} \ = \ F \cup G \cup A
$$

**Figure 5.** Computation of *OriginalPairs*.

The set *ReachablePairs*, a subset of *OriginalPairs*, is computed as shown in Figure 6. It uses a call-graph analysis that provides a context sensitive call graph of the program via:

- The function $\mathcal{M}$ that, given a call site $i \in \mathbb{I}$, returns the method containing $i$. For convenience, we extend $\mathcal{M}$ to provide the containing method of each memory access $e \in \mathbb{E}$ as well.
- The function $\mathcal{T}$ that, given a call site $i \in \mathbb{I}$ and a caller context $c \in \mathbb{C}$, returns all pairs of callee methods and callee contexts.
- The set $I_{fork}$ of all call sites that spawn threads, namely, those occurring in the synthesized *main* ($I_{ext}$) and those occurring in the original program ($I_{int}$).

We use $(i, c') \longrightarrow^* (m, c)$ to denote that method $m$ is reachable (can be called directly or indirectly) in context $c$ from call site $i$ in context $c'$. We use $(i, c') \Longrightarrow^* (m, c)$ to denote that method $m$ is reachable in context $c$ from call site $i$ in context $c'$ without switching threads, that is, without executing a call site $i' \in I_{fork}$. Intuitively, $\longrightarrow^*$ and $\Longrightarrow^*$ capture thread insensitive and thread sensitive paths, respectively, in the context sensitive call graph of the program.

We employ a recent notion of context sensitivity called *k-object sensitivity* [36, 37]. The set of abstract contexts $\mathbb{C}$ in this form of context sensitivity includes, for each instance method, each string of at most $k$ object allocation sites representing an abstract object to which its distinguished $0^{th}$ argument `this` may be bound at run-time. (For brevity, we henceforth use the terms context and object without qualifying them as abstract.) Static methods like *main* that lack the `this` argument possess a single context denoted

$\epsilon$. We construct the call graph *on the fly*, that is, we perform call-graph construction and alias analysis simultaneously (see Figure 1) thereby yielding more precise versions of both than in a setting in which the call graph is constructed separately.

We next compute the set of *roots* $\mathcal{R}$ containing each $(i, c)$ such that thread-spawning call site $i$ in context $c$ is reachable from *main* in context $\epsilon$. Then, a pair of accesses in *OriginalPairs* is retained in *ReachablePairs* only if each access is reachable, without switching threads, from some thread spawned at a root.

Due to lack of flow sensitivity in our approach, *ReachablePairs* may contain spurious elements in the presence of the flow sensitive fork/join synchronization idiom. The definition of *ReachablePairs* assumes $\mathcal{R}^2 = \mathcal{R} \times \mathcal{R}$ by default, that is, every pair $i_1, i_2 \in I_{fork}$ may spawn a pair of threads that may execute in parallel. But a pair $(i_1, i_2)$ may be excluded if no pair of threads spawned at $i_1$ and $i_2$ can execute in parallel due to fork/join synchronization. Hence, we allow the user to provide annotations for pruning *ReachablePairs*. We also piggyback other cases in these annotations, such as allowing the user to exclude:

- A pair $(i_1, i_2)$ where $i_1, i_2 \in I_{ext}$ invoke methods in the interface of an open program that are not intended to be invoked in parallel on shared data by a client.
- A pair $(i, i)$ where $i \in I_{int}$ spawns a single thread (or spawns multiple threads but only one is alive at any instant).

The annotation burden is small because call sites in $I_{ext}$, which is often large, may be specified at the interface-level (e.g., merely specifying that any pair of methods in interfaces $I_1$ and $I_2$ may not be called in parallel on shared data) while call sites in $I_{int}$, which is small even for large programs, may be specified individually. The total number of such annotations needed for each of our benchmark programs is shown in the "roots" column in Figure 11.

In our running example in Figure 4, pairs $(\mathtt{f}_r, \mathtt{f}'_w)$, $(\mathtt{f}'_w, \mathtt{f}'_w)$ and $(\mathtt{f}_w, \mathtt{f}'_w)$ in *OriginalPairs* are not retained in *ReachablePairs* because each of these pairs involves an access $\mathtt{f}'_w$ occurring in a constructor and we elide checking races in constructors (see item (3) of Section 2.4). Using 1-object sensitivity (which suffices for this example), method `rd` is reachable in a single context $\mathtt{c}_a$ from both thread-spawning call sites `a.get` and `a.inc` in context $\epsilon$, while method `wr` is reachable in two contexts $\mathtt{c}_a$ and $\mathtt{c}_b$ from a single thread-spawning call site `a.inc` in context $\epsilon$, where $\mathtt{c}_a$ and $\mathtt{c}_b$ denote the `new A()` allocation sites in methods `main` and `inc`, respectively. As a result, *ReachablePairs*$'$ contains elements $(\mathtt{f}_r,\mathtt{c}_a,\mathtt{f}_w,\mathtt{c}_a)$, $(\mathtt{f}_r,\mathtt{c}_a,\mathtt{f}_w,\mathtt{c}_b)$, $(\mathtt{f}_w,\mathtt{c}_a,\mathtt{f}_w,\mathtt{c}_a)$, $(\mathtt{f}_w,\mathtt{c}_a,\mathtt{f}_w,\mathtt{c}_b)$, $(\mathtt{f}_w,\mathtt{c}_b,\mathtt{f}_w,\mathtt{c}_a)$, and $(\mathtt{f}_w,\mathtt{c}_b,\mathtt{f}_w,\mathtt{c}_b)$.

### 2.2.2 Aliasing-Pairs Computation

The second stage of our algorithm prunes *ReachablePairs* using the fact that a pair of accesses may be involved in a race provided they access the same location. A pair of instance field or array element accesses, that is, a pair of accesses referencing either $x.f$ and $y.f$ or $x[i]$ and $y[j]$, may access the same location only if $x$ and $y$ are aliases.[1] A pair of accesses to the same static field, on the other hand, always alias and so this stage cannot eliminate such pairs.

The set *AliasingPairs* is computed as shown in Figure 7. It uses an alias analysis that provides a function $\mathcal{P}$ that, given a statement in $\mathbb{E}_f$ or $\mathbb{E}_a$ referencing instance field $x.f$ or array element $x[i]$, and a context of the containing method, returns the set of objects to which variable $x$ may point in that context. It is convenient to assume that, given a statement in $\mathbb{E}_g$ and a context of the containing method, this function returns $\{ h_0 \}$ where $h_0$ is a distinguished dummy object. It is conventional for alias analyses to define such an object and treat *every* static field defined in the program as an instance field of this object. We exploit this property in our thread-escape analysis (see Section 2.2.3).

The alias analysis we use is *k-object sensitive alias analysis* [36, 37] which is a flow insensitive, context sensitive, object sensitive, field sensitive, and inclusion-based alias analysis. In this form of alias analysis, objects and contexts are treated uniformly, that is, domains $\mathbb{H}$ and $\mathbb{C}$ are one and the same. We distinguish between them in our presentation, however, since $\mathbb{H}$ and $\mathbb{C}$ are disjoint for most alias analyses.

Returning to the computation of the set *AliasingPairs*, a pair of accesses in *ReachablePairs* referencing either $x.f$ and $y.f$ or $x[i]$ and $y[j]$ is retained in *AliasingPairs* if there exist contexts $c_1$ and $c_2$ of their containing methods, respectively, and there exists an object $h$ such that both $x$ in context $c_1$ and $y$ in context $c_2$ may point to $h$.

In our running example in Figure 4, elements $(\mathtt{f}_r,\mathtt{c}_a,\mathtt{f}_w,\mathtt{c}_b)$, $(\mathtt{f}_w,\mathtt{c}_a,\mathtt{f}_w,\mathtt{c}_b)$, and $(\mathtt{f}_w,\mathtt{c}_b,\mathtt{f}_w,\mathtt{c}_a)$ in *ReachablePairs*$'$ are not retained in *AliasingPairs*$'$ because accesses $\mathtt{f}_r$ and $\mathtt{f}_w$ are of the form `this.f` and variable `this` in contexts $\mathtt{c}_a$ and $\mathtt{c}_b$ has disjoint points-to sets $\{\mathtt{c}_a\}$ and $\{\mathtt{c}_b\}$, respectively, using 1-object sensitive alias analysis. Thus, *AliasingPairs*$'$ contains only $(\mathtt{f}_r,\mathtt{c}_a,\mathtt{f}_w,\mathtt{c}_a)$, $(\mathtt{f}_w,\mathtt{c}_a,\mathtt{f}_w,\mathtt{c}_a)$, and $(\mathtt{f}_w,\mathtt{c}_b,\mathtt{f}_w,\mathtt{c}_b)$.

### 2.2.3 Escaping-Pairs Computation

The third stage of our algorithm prunes *AliasingPairs* using the fact that a pair of accesses may be involved in a race only if they access thread-shared data.

The set *EscapingPairs*, a subset of *AliasingPairs*, is computed as shown in Figure 8. It uses a thread-escape analysis that specifies a set $\mathcal{E}$ of objects that may be thread-shared. The thread-escape analysis we use is specified in 3 lines of Datalog. It depends on the call-graph and alias analyses for the set of thread-spawning call sites, $I_{fork}$, and:

- The set $\mathcal{A}$ containing each triple $(i, n, h)$ such that argument $n$ of call site $i$ may point to object $h$ in some context.
- The set $\mathcal{F}$ containing each triple $(h', f, h)$ such that the instance field $f$ of object $h'$ may point to object $h$.

It states that an object may be thread-shared if (i) it is reachable from some argument of a thread-spawning call site or (ii) it is reachable from some static field. Specifically, $h \in \mathcal{E}$, that is, object $h$ may be thread-shared, if any of the following holds:

1. Some argument of a thread-spawning call site $i \in I_{fork}$ may point to $h$. This rule uniformly captures explicit and implicit thread-spawning call sites: explicit sites, which invoke the `start()` method, have a single argument (the `this` object of type `java.lang.Runnable`) whereas implicit sites, generated by our harness synthesis algorithm for open programs (see Section 2.1), may have arbitrary arguments since they invoke user-defined methods. This rule in combination with the following one captures situation (i) above.
2. There exists some thread-shared object $h'$ and instance field $f$ of $h'$ that may point to $h$.
3. Object $h$ is the distinguished dummy object $h_0$. Recall from Section 2.2.2 that all static fields defined in the program are instance fields of $h_0$. Thus, this rule in combination with the previous one captures situation (ii) above.

Returning to the computation of the set *EscapingPairs*, a pair of accesses in *AliasingPairs* referencing either $x.f$ and $y.f$ or $x[i]$ and $y[j]$ is retained in *EscapingPairs* provided there exist contexts $c_1$ and $c_2$ of the methods containing those accesses, respectively, and an object $h$ such that $x$ in context $c_1$ and $y$ in context $c_2$ may

[1] More precisely, a pair of accesses referencing array elements $x[i]$ and $y[j]$ access the same location if (i) $x$ and $y$ are aliases and (ii) $i = j$. However, we conservatively elide checking condition (ii).

$$
\begin{array}{rcl}
(\mathit{method}) & m \in \mathbb{M} & = \{\ \mathit{main}, \ldots\ \} \\
(\mathit{call\ site}) & i \in \mathbb{I} & = \{\ \mathit{all\ call\ sites}\ \} \\
(\mathit{context}) & c \in \mathbb{C} & = \{\ \epsilon, \ldots\ \}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{M} & : & (\mathbb{I} \cup \mathbb{E}) \to \mathbb{M} \\
\mathcal{T} & : & (\mathbb{I} \times \mathbb{C}) \to \mathscr{P}(\mathbb{M} \times \mathbb{C}) \\
I_{fork} & = & I_{ext} \cup I_{int} \\
\mathcal{R} & \subseteq & (\mathbb{I} \times \mathbb{C})
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ReachablePairs} & = & \{\ (e_1, e_2) \mid \exists c_1, c_2 :\ (e_1, c_1, e_2, c_2) \in \mathit{ReachablePairs}'\ \} \\
\mathit{ReachablePairs}' & = & \{\ (e_1, c_1, e_2, c_2) \mid (e_1, e_2) \in \mathit{OriginalPairs} \wedge \exists (i_1, c_1', i_2, c_2') \in \mathcal{R}^2 : \\
 & & \qquad (i_1, c_1') \Longrightarrow^* (\mathcal{M}(e_1), c_1) \ \wedge\ (i_2, c_2') \Longrightarrow^* (\mathcal{M}(e_2), c_2)\ \}
\end{array}
$$

$$
\mathcal{R} \ =\ \{\ (i, c) \mid i \in I_{fork} \wedge \exists i' :\ \mathcal{M}(i') = \mathit{main} \wedge (i', \epsilon) \longrightarrow^* (\mathcal{M}(i), c)\ \}
$$

$$
\begin{array}{rcl}
(i, c') \Longrightarrow (m, c) & \triangleq & (i, c') \longrightarrow (m, c) \ \wedge\ i \notin I_{fork} \\
(i, c') \longrightarrow (m, c) & \triangleq & (m, c) \in \mathcal{T}(i, c')
\end{array}
$$

$$
\begin{array}{rcl}
(i, c'') \Longrightarrow^{n+1} (m, c) & \triangleq & \exists m', i', c' :\ (i, c'') \Longrightarrow^{n} (m', c') \ \wedge\ \mathcal{M}(i') = m' \ \wedge\ (i', c') \Longrightarrow (m, c) \\
(i, c'') \longrightarrow^{n+1} (m, c) & \triangleq & \exists m', i', c' :\ (i, c'') \longrightarrow^{n} (m', c') \ \wedge\ \mathcal{M}(i') = m' \ \wedge\ (i', c') \longrightarrow (m, c)
\end{array}
$$

**Figure 6.** Call graph and computation of *ReachablePairs*.

$$
\begin{array}{rcl}
(\mathit{object}) & h \in \mathbb{H} & = \{\ h_0, \ldots\ \} \\
\mathcal{P} & : & (\mathbb{E} \times \mathbb{C}) \to \mathscr{P}(\mathbb{H})
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{AliasingPairs} & = & \{\ (e_1, e_2) \mid \exists c_1, c_2 :\ (e_1, c_1, e_2, c_2) \in \mathit{AliasingPairs}'\ \} \\
\mathit{AliasingPairs}' & = & \{\ (e_1, c_1, e_2, c_2) \mid (e_1, c_1, e_2, c_2) \in \mathit{ReachablePairs}' \ \wedge\ (\textstyle\bigcap_{j \in \{1,2\}} \mathcal{P}(e_j, c_j)) \neq \emptyset\ \}
\end{array}
$$

**Figure 7.** Alias analysis and computation of *AliasingPairs*.

$$
\begin{array}{rcl}
\mathcal{E} & \subseteq & \mathbb{H} \\
\mathcal{A} & \subseteq & \mathbb{I} \times \mathbb{N} \times \mathbb{H} \\
\mathcal{F} & \subseteq & \mathbb{H} \times \mathbb{F} \times \mathbb{H}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E}(h) & :- & \mathcal{A}(i, n, h),\ I_{fork}(i). \\
\mathcal{E}(h) & :- & \mathcal{E}(h'),\ \mathcal{F}(h', f, h). \\
\mathcal{E}(h) & :- & h = h_0.
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{EscapingPairs} & = & \{\ (e_1, e_2) \mid \exists c_1, c_2 :\ (e_1, c_1, e_2, c_2) \in \mathit{EscapingPairs}'\ \} \\
\mathit{EscapingPairs}' & = & \{\ (e_1, c_1, e_2, c_2) \mid (e_1, c_1, e_2, c_2) \in \mathit{AliasingPairs}' \wedge \\
 & & \qquad (\textstyle\bigcap_{j \in \{1,2\}} \mathcal{P}(e_j, c_j)) \cap \mathcal{E} \neq \emptyset\ \}
\end{array}
$$

**Figure 8.** Escape analysis and computation of *EscapingPairs*.

$$
\begin{array}{rcl}
(\mathit{sync\ stmt}) & s \in \mathbb{S} \\
(\mathit{path}) & w \in \mathbb{W}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S} & : & (\mathbb{I} \cup \mathbb{E}) \to \mathscr{P}(\mathbb{S}) \\
\mathcal{P} & : & (\mathbb{S} \times \mathbb{C}) \to \mathscr{P}(\mathbb{H}) \\
\mathcal{W} & : & (\mathbb{I} \times \mathbb{C} \times \mathbb{M} \times \mathbb{C}) \to \mathscr{P}(\mathbb{W}) \\
\mathcal{Q} & : & (\mathbb{M} \times \mathbb{C}) \to \mathscr{P}(\mathbb{W}) \\
\mathcal{L} & : & (\mathbb{W} \times \mathbb{E} \times \mathbb{C}) \to \mathscr{P}(\mathbb{H})
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{UnlockedPairs} & = & \{\ (e_1, e_2) \mid \exists c_1, c_2 :\ (e_1, c_1, e_2, c_2) \in \mathit{UnlockedPairs}'\ \} \\
\mathit{UnlockedPairs}'' & = & \{\ (e_1, c_1, e_2, c_2) \mid (e_1, c_1, e_2, c_2) \in \mathit{EscapingPairs}' \wedge \\
 & & \quad \exists w_1 \in \mathcal{Q}(\mathcal{M}(e_1), c_1) :\ \exists w_2 \in \mathcal{Q}(\mathcal{M}(e_2), c_2) :\ \mathcal{L}(w_1, e_1, c_1) \cap \mathcal{L}(w_2, e_2, c_2) = \emptyset\ \} \\
\mathit{UnlockedPairs}' & = & \{\ (e_1, c_1, e_2, c_2) \mid (e_1, c_1, e_2, c_2) \in \mathit{EscapingPairs}' \wedge \\
 & & \quad \bigcap \{\ \mathcal{L}(w_1, e_1, c_1) \mid w_1 \in \mathcal{Q}(\mathcal{M}(e_1), c_1)\ \} \cap \bigcap \{\ \mathcal{L}(w_2, e_2, c_2) \mid w_2 \in \mathcal{Q}(\mathcal{M}(e_2), c_2)\ \} = \emptyset\ \}
\end{array}
$$

$$
\mathcal{W}(i, c', m, c) \ =\ \{\ w \mid w \text{ is of the form } (i, c') \Longrightarrow^* (m, c)\ \}
$$

$$
\mathcal{Q}(m, c) \ =\ \bigcup \{\ w \mid \exists i, c' :\ (i, c') \in \mathcal{R} \ \wedge\ w \in \mathcal{W}(i, c', m, c)\ \}
$$

$$
\mathcal{L}(w, e, c) \ =\ \{\ h \mid \exists s \in \mathcal{S}(e) :\ \mathscr{P}(s, c) = \{h\}\ \} \ \cup\ \{\ h \mid \exists i'', c'' :\ w \text{ contains } (i'', c'') \ \wedge\ \exists s \in \mathcal{S}(i'') :\ \mathscr{P}(s, c'') = \{h\}\ \}
$$

**Figure 9.** Lock analysis and computation of *UnlockedPairs*.

point to $h$ and $h$ may be thread-shared. This stage, like the previous one, does not eliminate any pairs referencing static fields.

The set *EscapingPairs* may contain spurious elements in the presence of the flow sensitive fork/join synchronization idiom, namely, even if an object is thread-shared, multiple threads may not be able to access it simultaneously because of fork/join synchronization. Hence, we allow the user to provide an annotation per field (resp. class) to exclude from *EscapingPairs* all pairs referencing that field (resp. any field of that class). The annotation burden is minor because the field-based view (resp. object-based view) of races in the output of *Chord* (see Section 2.3) helps identify such fields (resp. classes) readily. The total number of such annotations needed for each of our benchmark programs is shown in the "local" column in Figure 11.

In our running example in Figure 4, the element ($\mathtt{f}_w$,$\mathtt{c}_b$,$\mathtt{f}_w$,$\mathtt{c}_b$) from *AliasingPairs*′ is not retained in *EscapingPairs*′ whereas elements ($\mathtt{f}_r$,$\mathtt{c}_a$,$\mathtt{f}_w$,$\mathtt{c}_a$) and ($\mathtt{f}_w$,$\mathtt{c}_a$,$\mathtt{f}_w$,$\mathtt{c}_a$) are retained because accesses $\mathtt{f}_r$ and $\mathtt{f}_w$ are of the form `this.f`, variable `this` in contexts $\mathtt{c}_a$ and $\mathtt{c}_b$ has points-to sets $\{\mathtt{c}_a\}$ and $\{\mathtt{c}_b\}$, respectively, and our thread-escape analysis states that $c_b$ is thread-local but $\mathtt{c}_a$ may be thread-shared since argument `a` of thread-spawning call sites `a.inc` and `a.get` in method `main` may point to it.

#### 2.2.4 Unlocked-Pairs Computation

The final stage of our algorithm prunes *EscapingPairs* using the fact that a pair of accesses may be involved in a race only if they are executed by a pair of threads without holding a common lock.

The set *UnlockedPairs*, a subset of *EscapingPairs*, is computed as shown in Figure 9. It uses a lock analysis that depends on the call-graph and alias analyses. We first characterize the set of locks held by a given thread while executing a given access along a given path in the context sensitive call graph. It is important to note, however, that we use paths only to simplify our presentation: computing *UnlockedPairs* involves performing a traversal of the entire graph once and for all instead of enumerating paths.

Let $\mathbb{W}$ denote the set of all paths in the context sensitive call graph and let $\mathcal{W}$ denote the function that, given a call site $i$ in context $c'$ and a method $m$ in context $c$, returns the set of all paths $(i, c') \Longrightarrow^* (m, c)$. Given a thread spawned at call site $i$ in context $c'$, that is, $(i, c') \in \mathcal{R}$, an access $e$ in context $c$, and a path $w$ in the context sensitive call graph that the thread may execute from $(i, c')$ to $(\mathcal{M}(e), c)$, that is, $w \in \mathcal{W}(i, c', \mathcal{M}(e), c)$, we say that lock $h \in \mathcal{L}(w, e, c)$ is held by that thread during that access along that path if either of the following holds:

- The access $e$ is lexically enclosed in a `synchronized (l)` $\{\ldots\}$ statement and the points-to set of `l` in context $c$ is $\{h\}$.
- The path $w$ executes some call site $i''$ in context $c''$ such that call site $i''$ is lexically enclosed in a `synchronized (l)` $\{\ldots\}$ statement and the points-to set of `l` in context $c''$ is $\{h\}$.

Notice that we approximate the lock held on `l` by its singleton points-to set $\{h\}$, which is unsound, as we show next.

The set *UnlockedPairs*$''$ is computed as shown in Figure 9. It retains an $(e_1, c_1, e_2, c_2) \in$ *EscapingPairs*$'$ provided accesses $e_1$ and $e_2$ may be executed in contexts $c_1$ and $c_2$ by a pair of threads along a pair of paths $w_1$ and $w_2$ without holding a common lock, that is, $\bigcap_{j\in\{1,2\}} \mathcal{L}(w_j, e_j, c_j) = \emptyset$. Determining whether a common lock is held requires a must-alias analysis whereas we use our may-alias analysis in the computation of $\mathcal{L}$ above, which is an unsound approximation. In particular, if our algorithm declares a pair of accesses race-free, then it guarantees that each pair of threads executing it will hold a lock on a pair of concrete objects $o_1$ and $o_2$ represented by the same abstract object $h$ in our alias analysis, but it cannot guarantee that $o_1 = o_2$. In our experiments, however, we did not find any case in which locks were held on different concrete objects that were represented by the same abstract object in our $k$-object sensitive alias analysis. Henceforth, we use the phrase common lock to mean the same abstract object.

The set *UnlockedPairs*$''$ is precise in that a pair of accesses is excluded from it (that is, declared race-free) even if different pairs of paths leading to it hold different sets of locks, as long as the sets of locks held along each pair of paths contain some common lock. This precision, however, comes at the exponential cost of enumerating every pair of paths in the context sensitive call graph leading from every pair of roots in $\mathcal{R}$ to every pair of accesses in *EscapingPairs*$'$. We therefore compute an approximation *UnlockedPairs*$'$ to *UnlockedPairs*$''$.

The set *UnlockedPairs*$'$ is computed as shown in Figure 9. It excludes an $(e_1, c_1, e_2, c_2) \in$ *EscapingPairs*$'$ only if a common lock is held along *all* paths in the set of paths that includes every path originating at some root $(i, c) \in \mathcal{R}$ and terminating in $(e_1, c_1)$ or $(e_2, c_2)$ without switching threads. It is immediate that our approximation is sound:

**Fact 1.** *UnlockedPairs*$'' \subseteq$ *UnlockedPairs*$'$.

Besides being cheap to compute — computing it involves performing a single traversal of the entire context sensitive call graph as opposed to enumerating each pair of paths from each pair of roots leading to each pair of accesses — it is nearly complete, in a sense made precise by the following lemma:

**Lemma 2.** $(e_1, c_1, e_2, c_2) \in$ *UnlockedPairs*$'$ *and* $(e_1, c_1, e_2, c_2) \notin$ *UnlockedPairs*$''$ *implies* $\exists j \in \{1, 2\} : \forall w \in \mathcal{Q}(\mathcal{M}(e_j), c_j) : |\mathcal{L}(w, e_j, c_j)| > 1$.

*Proof.* By contradiction. Let (1) $(e_1, c_1, e_2, c_2) \in$ *UnlockedPairs*$'$ and (2) $(e_1, c_1, e_2, c_2) \notin$ *UnlockedPairs*$''$. From (1) and the defn. of *UnlockedPairs*$'$, we have (3) $(e_1, c_1, e_2, c_2) \in$ *EscapingPairs*$'$ and (4) $\bigcap \{\, \mathcal{L}(w_1, e_1, c_1) \mid w_1 \in \mathcal{Q}(\mathcal{M}(e_1), c_1) \,\} \cap$
$\bigcap \{\, \mathcal{L}(w_2, e_2, c_2) \mid w_2 \in \mathcal{Q}(\mathcal{M}(e_2), c_2) \,\} = \emptyset$
From (2) and the defn. of *UnlockedPairs*$''$ and (3), we have:
(5) $\forall w_1 \in \mathcal{Q}(\mathcal{M}(e_1), c_1) : \forall w_2 \in \mathcal{Q}(\mathcal{M}(e_2), c_2) :$
$\mathcal{L}(w_1, e_1, c_1) \cap \mathcal{L}(w_2, e_2, c_2) \neq \emptyset$
To prove $\exists j \in \{1, 2\} : \forall w \in \mathcal{Q}(\mathcal{M}(e_j), c_j) : |\mathcal{L}(w, e_j, c_j)| > 1$. Suppose for the sake of contradiction that:
(6) $\exists w_3 \in \mathcal{Q}(\mathcal{M}(e_1), c_1) : |\mathcal{L}(w_3, e_1, c_1)| \leq 1$
$\exists w_4 \in \mathcal{Q}(\mathcal{M}(e_2), c_2) : |\mathcal{L}(w_4, e_2, c_2)| \leq 1$
From (5) and (6), we have:
(7) $\mathcal{L}(w_3, e_1, c_1) = \mathcal{L}(w_4, e_2, c_2) = \{h\}$ (say)
From (5) and (7), we have:
(8) $\forall w_1 \in \mathcal{Q}(\mathcal{M}(e_1), c_1) : h \in \mathcal{L}(w_1, e_1, c_1)$
$\forall w_2 \in \mathcal{Q}(\mathcal{M}(e_2), c_2) : h \in \mathcal{L}(w_2, e_2, c_2)$
which contradicts (4). □

Lemma (2) states that if a pair of accesses is declared race-free using *UnlockedPairs*$''$ but not using *UnlockedPairs*$'$, then it must be the case that at least one of those accesses is guarded by multiple locks along every path reaching it in some context. An access that *happens* to be guarded by multiple locks is conceivable but one that *relies* on it for race-freedom appears at most very rare.

Returning to our running example in Figure 4, the element $(\mathtt{f}_w, \mathtt{c}_a, \mathtt{f}_w, \mathtt{c}_a)$ in *EscapingPairs*$'$ is excluded from *UnlockedPairs*$'$ because the only path in the context sensitive call graph leading from thread-spawning call site `a.inc` in context $\epsilon$ to method `wr` in context $\mathtt{c}_a$ holds a lock on `this`, whose points-to set in context $\mathtt{c}_a$ is $\{c_a\}$. However, the element $(\mathtt{f}_r, \mathtt{c}_a, \mathtt{f}_w, \mathtt{c}_a)$ in *EscapingPairs*$'$ is contained in *UnlockedPairs*$'$ since there exists a path leading from thread-spawning call site `a.get` in context $\epsilon$ to method `rd` in context $\mathtt{c}_a$ along which no lock is held.

### 2.3 Post-processing

We report each $(e_1, c_1, e_2, c_2) \in$ *UnlockedPairs*$'$ as a possible race. Reporting races found by a static race detection tool in a useful manner poses several challenges:

- Since a race involves a pair of accesses, a potentially quadratic blowup is inherent in the output of the tool.
- Races are symptoms as opposed to causes of bugs. Thus, a single race may indicate multiple bugs and, conversely, multiple races may indicate the same bug.
- Determining whether a reported race is real or a false alarm involves manually inspecting various aspects such as the object on which the race occurs, the pair of threads accessing the object, and the sets of locks held by the threads.
- Even if a reported race is real, manual inspection is needed to determine whether it is a symptom of a bug or a benign race. The problem is exacerbated by the subtleties of the Java memory model [34].

We address the above issues by reporting counterexamples for each race and by categorizing races. For each $(e_1, c_1, e_2, c_2) \in$ *UnlockedPairs*$'$, we provide a graph representing each pair of paths in the context sensitive call graph leading to the containing methods of $e_1$ and $e_2$ in contexts $c_1$ and $c_2$. We also provide two views, a *field-based view* and an *object-based view*, categorizing races in *UnlockedPairs*$'$ by the field and the set of abstract objects, respectively, on which the races occur. The field-based view is useful in quickly discarding all races reported on a field that is intentionally accessed without synchronization (e.g., an integer-valued field that tracks statistics approximately, a boolean field that one thread periodically polls and another writes to notify it, etc. [39]). The object-based view is useful in:

- quickly identifying all false alarms arising from a single source of imprecision in our alias or thread-escape analysis (such races are distributed over multiple categories in the field-based view)
- grouping races on elements of different arrays separately (such races are clumped into a single category in the field-based view)
- grouping races triggered on fields of classes used by different clients separately, for instance, if unrelated classes `A` and `B` have a field of type `C`, then categorizing races triggered on a field of `C` through objects of `A` and `B` separately.

In our running example in Figure 4, we report not only that there is a race between the pair of accesses $\mathtt{f}_r$ and $\mathtt{f}_w$, but also that the

race occurs on the field `f` of an object allocated at the `new A()` allocation site in the `main` method. Furthermore, we provide the pair of paths in the call graph illustrating that the race can occur when the `get` and `inc` methods are executed in separate threads on the same object of class `A`.

### 2.4 Soundness

Our race detection approach has four sources of unsoundness:

1. Lacking must-alias analysis, the fourth stage of our algorithm uses may-alias analysis to check whether a pair of accesses is guarded by a common lock, which is unsound.
2. We do not analyze open programs soundly: missing callee methods are treated as no-ops while missing caller methods are modeled by an automatically synthesized harness that simulates many but not all usage scenarios.
3. Like most static race detectors, we elide checking races on accesses in class initializers, constructors, and finalizers which typically lack synchronization and seldom contain races but cause many false alarms without a method-escape analysis.
4. We ignore the effects of reflection and dynamic class loading.

| | classes | LOC | brief description |
|---|---|---|---|
| `tsp` | 370 | 76,026 | TSP implementation from ETH |
| `hedc` | 422 | 82,992 | web crawler from ETH |
| `ftp` | 493 | 103,183 | Apache FTP Server |
| `vect` 1.1 | 19 | 2,632 | JDK 1.1 java.util.Vector |
| `htbl` 1.1 | 21 | 2,688 | JDK 1.1 java.util.Hashtable |
| `htbl` 1.4 | 366 | 75,342 | JDK 1.4 java.util.Hashtable |
| `vect` 1.4 | 370 | 75,675 | JDK 1.4 java.util.Vector |
| `jdbm` | 461 | 115,364 | transactional persistence engine |
| `jdbf` | 465 | 121,569 | object-relational mapping system |
| `pool` | 388 | 123,878 | Apache Commons Pool |
| `jtds` | 553 | 164,820 | JDBC driver |
| `derby` | 1,746 | 646,447 | Apache Derby, an RDBMS |

**Figure 10.** Benchmarks.

## 3. Experiments

We evaluate our race detection algorithm on a suite of open-source multi-threaded Java programs. Figure 10 shows, for each of these programs, the number of classes and lines of Java source code in the initial call graph computed by Soot. We use the best call graph computable using Soot, namely, one constructed *on-the-fly* using context insensitive alias analysis. The benchmark suite contains three closed programs: `tsp`, `hedc`, and `ftp`. The rest are open programs that provide interfaces to clients. Programs `tsp`, `hedc`, `vect` 1.1, and `htbl` 1.1 have been analyzed in previous work on race detection; the rest are mature and widely-used programs, except for `jdbf`, which is in its developmental stages.

The results of our experiments are shown in Figure 11. All experiments were done on a 2.4GHz machine with 4GB memory. The "time" column shows the total running time of our tool for each benchmark. The next two columns show the number of annotations we provided: those in the "roots" column specifying which threads cannot execute in parallel (see Section 2.2.1) and those in the "local" column specifying which fields/objects, although thread-shared, cannot be accessed simultaneously by different threads (see Section 2.2.3). The next five columns give the sizes of *OriginalPairs*, *ReachablePairs*, *AliasingPairs*, *EscapingPairs*, and *UnlockedPairs*, namely, the number of pairs of accesses in the initial over-approximation computed by Soot followed by the number of pairs of accesses retained after each of the four successive stages in our race detection algorithm. The next three columns partition *UnlockedPairs* into harmful races, benign races, and false alarms. Finally, the "bugs" column reports the number of distinct fixes that were needed in the source code to eliminate all harmful races. Most fixes involved one of the following: (1) adding synchronization to a piece of code where none existed, (2) extending an existing synchronized block, (3) changing the object on which the lock was held by a synchronized block, (4) declaring a field volatile, or (5) removing synchronization because one of the above rendered it redundant (letting it remain could degrade performance or introduce deadlocks). In many cases, a harmful race was triggered in code far from the code where synchronization was needed to eliminate the race, for instance, we found many harmful races in library code that were eliminated by adding synchronization to application code. We next describe each experiment briefly.

The `tsp` program contains a main thread that creates an array of worker threads which are objects of class `TspSolver`. The instance fields of `TspSolver` are local to each worker thread but its static fields are shared by all worker threads. We provided 2 annotations: one to restrict reporting to races between worker threads (and not between the main thread and a worker thread since the main thread is idle from the point it forks the worker threads to the point it joins them), and another to suppress reporting races on instance fields of `TspSolver`, since these fields are local to each worker thread but our thread-escape analysis cannot infer that because the alias analysis does not distinguish between different elements in the array of worker threads. *Chord* reported 7 harmful races, 0 benign races, and 12 false positives due to flow insensitivity. The harmful races, all on static field `MinTourLen` of `TspSolver`, were grouped in a single category, making it easy to identify the underlying bug. This bug has been reported by previous race detectors. The false positives were grouped in just 2 categories in the object-based view of the race reports and were therefore easy to identify and ignore.

*Chord* reported 4 harmful races, 2 benign races, and 13 false positives for `hedc`. The harmful races indicating 1 bug as well as the benign races have been diagnosed by previous race detectors. All the 13 false positives are due to flow insensitivity. We provided 9 annotations specifying thread-local objects/fields.

*Chord* reported 45 harmful races, 3 benign races, and 23 false positives for `ftp`. The harmful races revealed 12 distinct bugs: 11 bugs were fixed within a day of reporting and developers acknowledged the remaining bug but it remains open because fixing it involves making widespread changes. Indeed, 21 of the 45 harmful races are attributed to this bug. The 23 false positives may be attributed to 2 causes: once we identified those causes, it was easy to determine whether a race report was a real race or a false positive. We provided 11 annotations: 7 specifying which threads may execute in parallel and 4 specifying certain objects/fields thread-local.

We provided 1 annotation for each of the `vect` and `htbl` programs instructing our tool to report races between every pair of methods in the `java.util.Vector` and `java.util.Hashtable` interfaces, respectively. *Chord* reported 5 harmful races in `vect` 1.1, all due to a single bug first reported in [20]. All benign races are due to lack of synchronization in tiny methods such as `size` and `isEmpty` (client code is expected to explicitly synchronize calls to these methods) except for the 9 benign races in `htbl` 1.4, which are due to unsynchronized methods `values`, `keySet`, and `entrySet`. One of these methods is:

```
private volatile Set keySet = null;

public Set keySet() {
    if (keySet == null)
        keySet = Collections.synchronizedSet(new KeySet(), this);
    return keySet;
}
```

| | time | annotations | | pairs of accesses | | | | | races | | | bugs |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | roots | local | original | reachable | aliasing | escaping | unlocked | harmful | benign | false | |
| `tsp` | 1m03s | 1 | 1 | 6398784 | 1115 | 475 | 363 | 19 | 7 | 0 | 12 | 1 |
| `hedc` | 1m10s | 0 | 9 | 6417868 | 3052 | 1492 | 958 | 19 | 4 | 2 | 13 | 1 |
| `ftp` | 1m17s | 7 | 4 | 6662532 | 712 | 349 | 268 | 71 | 45 | 3 | 23 | 12 |
| `vect` 1.1 | 0m08s | 1 | 0 | 32216 | 338 | 292 | 292 | 17 | 5 | 12 | 0 | 1 |
| `htbl` 1.1 | 0m07s | 1 | 0 | 32564 | 220 | 154 | 150 | 6 | 0 | 6 | 0 | 0 |
| `htbl` 1.4 | 1m04s | 1 | 0 | 6174952 | 214 | 161 | 161 | 9 | 0 | 9 | 0 | 0 |
| `vect` 1.4 | 1m02s | 1 | 0 | 6178650 | 1139 | 1046 | 1022 | 0 | 0 | 0 | 0 | 0 |
| `jdbm` | 1m33s | 1 | 0 | 11189853 | 33443 | 7511 | 2756 | 91 | 91 | 0 | 0 | 2 |
| `jdbf` | 1m42s | 1 | 0 | 12632041 | 95410 | 6849 | 307 | 130 | 130 | 0 | 0 | 18 |
| `pool` | 5m29s | 5 | 0 | 9741876 | 959 | 776 | 705 | 115 | 105 | 10 | 0 | 17 |
| `jtds` | 3m23s | 2 | 0 | 28773219 | 386241 | 26506 | 8430 | 48 | 34 | 14 | 0 | 16 |
| `derby` | 26m03s | 7 | 0 | 57681326 | 847236 | 83817 | 80662 | 1018 | 1018 | 0 | 0 | 319 |

**Figure 11.** Experimental results.

There are 3 races in the above method because it contains one write access and two read accesses of the `keySet` instance field. Hence, there are 9 races in all in the 3 methods. If two threads simultaneously invoke the `keySet` method on the same `Hashtable` object, then one of the updates to `keySet` may be lost. However, this does not affect correctness, and hence we regarded these races as benign.

*Chord* reported 91 harmful races in `jdbm` indicating 2 distinct bugs that have not yet been confirmed by developers. It also found 130 harmful races in `jdbf` indicating 18 distinct bugs, prompting developers to overhaul the synchronization in the project. In both programs, we provided 1 annotation to restrict reporting to races between methods of their only exported interface.

*Chord* reported 105 harmful races in `pool`, a generic object-pooling library that provides two APIs and five reference implementations (three of one plus two of the other). The races exposed bugs in each of the five implementations, for a total of 17 distinct bugs. All bugs were fixed and five patches were released in less than a week from reporting the bugs. The 5 annotations were needed because, although the five implementations are logically distinct, we generated a single harness exercising all of them. The annotations instructed *Chord* to report races only within each implementation as opposed to between different implementations.

*Chord* reported 34 harmful races revealing 16 distinct bugs in `jtds`. The developers initially expressed concerns about degrading performance and introducing deadlocks in the process of fixing the bugs in what they said was fairly mature code that seems to work well enough for most people. However, the seriousness of the bugs manifested in the counterexamples reported by our tool led them to conclude that it was dangerous to let the races lurk, and they fixed all of them and released a patch.

Finally, in our largest benchmark, `derby`, *Chord* reported 1018 races revealing 319 distinct bugs. The developers acknowledged the bugs, requested us to file bug reports, and promised to look at them in detail in the near future. They also inquired about the possibility of running our tool regularly on the Apache Derby source code to prevent new races from being introduced over time.

## 4. Related Work

In this section, we discuss related work, including dynamic race detection techniques (Section 4.1), static race detection techniques (Section 4.2), and recent work on atomicity (Section 4.3).

### 4.1 Dynamic Race Detection

State-of-the-art race detection tools are predominantly dynamic. Dynamic race detectors enjoy both precision and, more recently, scalability. However, besides being inherently unsound, they are difficult to apply to a program early in development either because the program is an open one that lacks sufficient client code or because the program is closed but lacks sufficient input data. Dynamic race detectors may be broadly classified into *happens-before-based*, *lockset-based*, and *hybrid*.

Happens-before-based dynamic race detectors [1, 13–15, 35, 43, 46] are based on Lamport's happens-before relation [31] which is a partial order on all events of all threads in a concurrent execution such that if a pair of accesses performed by a pair of threads on a memory location are not ordered by this relation, then they are deemed to be involved in a race because there exists a concurrent execution in which they can occur simultaneously. The key problems with happens-before-based race detection are that it is difficult to implement efficiently and, although it produces no false positives, it produces many false negatives.

The lockset algorithm is tailored to the common lock-based synchronization discipline: a pair of accesses performed by a pair of threads on a memory location are deemed to be involved in a race if the threads do not hold a common lock. The original implementation in the Eraser tool [45] incurred a slow-down of 10–30X but several static and dynamic optimization techniques, e.g., [2, 38, 49, 50], have reduced it significantly, with a recent implementation having a run-time overhead of only 13–42% [11]. The primary problems with lockset-based race detection are that it produces many false positives when synchronization idioms other than lock-based synchronization are used, as well as having the usual potential for false negatives of any dynamic analysis.

Dinning and Schonberg [16] first proposed combining happens-before-based and lockset-based race detection (in fact, they originated the lockset-based approach in order to improve the happens-before-based approach). Since then, several hybrid techniques have been proposed that gain the benefits of both approaches without suffering the disadvantages of either [28, 39, 40, 54].

### 4.2 Static Race Detection

Static race detectors are either primarily flow insensitive type-based systems [7, 8, 19, 20, 41, 44], flow sensitive static versions of the lockset algorithm [12, 17, 47], or path sensitive model checkers [29, 42].

The most closely related work is that of Choi et al. [12]. Their approach has the same basic inspiration as ours: using a series of stages, the pairs of memory accesses potentially involved in a race can be gradually filtered to a very small, high quality set. Also, at a high level, the static analyses they use are similar to ours (alias analysis, escape analysis, etc.). However, their implementation apparently was never applied beyond small Java programs, most likely

due to several key differences. Their algorithm is context insensitive, whereas we have found context sensitivity central to producing useful results; indeed, they conclude that even for small Java programs, more precise analysis is needed. The ordering of the phases is different; we have found the choice of phase ordering affects scalability, as it is desirable to place the relatively cheaper stages that filter out the most pairs early in the pipeline. Finally, they do not address the other aspects of usability, such as counterexamples and handling open programs.

Two static versions of the lockset algorithm we are aware of for C are Warlock [47] and RacerX [17]. Warlock does not trace paths through loops or recursive functions while RacerX targets operating systems code and uses heuristics specific to such code to determine which locks guard which accesses, which code is multi-threaded, and which unguarded accesses are benign.

Type-based and model-checking-based approaches to race detection are appealing in part because of their ability to check open programs and to produce counterexamples, respectively. Our approach possesses both these abilities. The key difference between our approach and the type-based ones is that the latter focus on specifying the synchronization discipline by means of types. Inferring this information automatically has proven difficult and, although significant advances have been made, it remains an active area of research [3, 21, 23, 44]. The key difference between our approach and the model-checking-based ones is that the latter are typically path sensitive. As a result, they can handle various synchronization idioms whereas we handle only lexically-scoped, lock-based synchronization, fork/join synchronization, and wait/notify synchronization. However, path sensitivity, besides affecting scalability, tends to limit model checkers to closed programs since they require an elaborate harness for open programs.

Finally, our technique finds more bugs than all previous static race detection techniques. RacerX [17] found 16 bugs in all in two operating systems (Linux comprising 1.8 MLOC and "System X" comprising 500 KLOC) which, to the best of our knowledge, is the largest number of bugs reported in any previous work on static race detection. Tools like RacerX handle large programs but apparently imprecisely so that not many bugs are found. The other class of static tools, including those based on type systems and model checking, perform a relatively precise analysis but have only been applied to small programs. However, there are just not many bugs to find in such programs, as the results of these tools, many of which are sound, indicate. Our tool, despite being unsound, detected all bugs that those tools found in such programs (specifically, the `tsp`, `hedc`, `vect` 1.1, and `htbl` 1.1 programs in our benchmark suite in Section 3).

Other static approaches to race detection include language-based ones such as nesC [27] for C and Guava [5] for Java. Finally, Aiken and Gay [4] present an effect inference system that checks whether SPMD programs use barrier synchronization correctly.

### 4.3 Atomicity Checking

Recent work on verification of concurrent programs has focused on checking *atomicity* [2, 18, 22, 24–26, 44, 51, 52]. The motivation behind atomicity is that race freedom is neither sound nor complete: the presence of races does not necessarily indicate the presence of concurrency errors (so-called *benign races*) and the absence of races does not necessarily indicate the absence of concurrency errors. However, we believe race freedom is important because of the following reasons:

1. Race freedom is a natural property for existing languages. Atomicity requires programmers to specify which code fragments are intended to be atomic. In the absence of such specifications, atomicity checkers check whether every method in the program is atomic. But this can lead to "benign atomicity violations", one of the very shortcomings of race freedom that atomicity was intended to remedy.
2. Many concurrency errors manifested as atomicity violations are also manifested as races. As a result, from the perspective of finding bugs in existing programs that lack atomicity specifications, checking for races is a viable alternative.
3. Race freedom is a first step in many atomicity checkers. In particular, atomicity checkers based on Lipton's theory of reduction [33] must show that each statement accessing a memory location is both a so-called *left mover* and a *right mover*, which is done by proving the absence of races on that location.

## 5. Conclusions

We have presented a novel technique for static race detection. We have shown its effectiveness by implementing it in a tool, applying it to several large, widely-used multi-threaded Java programs, and finding tens to hundreds of concurrency bugs in these systems.

## Acknowledgments

We thank Suhabe Bugrara, Jong-Deok Choi, Brian Hackett, John Kodumal, Ondrej Lhotak, Paul Twohey, Yichen Xie, and the anonymous PLDI reviewers for useful comments. We also thank Dawson Engler and Manu Sridharan for helpful discussions. Christoph von Praun kindly provided us the benchmarks used in previous work on race detection. This research was supported in part by NSF grants CCF-0430378 and CNS-0509558 and a Microsoft fellowship.

## References

[1] S. Adve, M. Hill, B. Miller, and R. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA'91)*, pages 234–243, 1991.

[2] R. Agarwal, A. Sasturkar, Wang L, and S. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 233–242, 2005.

[3] R. Agarwal and S. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, pages 149–160, 2004.

[4] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 342–354, 1998.

[5] D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 382–400, 2000.

[6] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 103–114, 2003.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, 2002.

[8] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the ACM*

*SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 56–69, 2001.

[9] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[10] G. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'98)*, pages 298–309, 1998.

[11] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 258–269, 2002.

[12] J. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research, 2001.

[13] J. Choi, B. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, 1991.

[14] M. Christiaens and K. Brosschere. TRaDe: A topological approach to on-the-fly race detection in Java programs. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 105–116, 2001.

[15] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'90)*, pages 1–10, 1990.

[16] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging (PADD'91)*, pages 85–96, 1991.

[17] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 237–252, 2003.

[18] C. Flanagan. Verifying commit-atomicity using model-checking. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software (SPIN'04)*, pages 252–266, 2004.

[19] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of the 8th European Symposium on Programming (ESOP'99)*, pages 91–108, 1999.

[20] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 219–232, 2000.

[21] C. Flanagan and S. Freund. Detecting race conditions in large programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 90–96, 2001.

[22] C. Flanagan and S. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 256–267, 2004.

[23] C. Flanagan and S. Freund. Type inference against races. In *Proceedings of the 11th International Static Analysis Symposium (SAS'04)*, pages 116–132, 2004.

[24] C. Flanagan, S. Freund, and M. Lifshin. Type inference for atomicity. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'05)*, pages 47–58, 2005.

[25] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 338–349, 2003.

[26] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 1–12, 2003.

[27] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The *nesC* language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 1–11, 2003.

[28] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of the 7th International SPIN Workshop on Model Checking Software (SPIN'00)*, pages 331–342, 2000.

[29] T. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 1–13, 2004.

[30] M. Lam, J. Whaley, B. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'05)*, pages 1–12, 2005.

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[32] O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 158–169, 2004.

[33] R. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[34] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 378–391, 2005.

[35] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 4th Annual Conference on Supercomputing (SC'91)*, pages 24–35, 1991.

[36] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 1–11, 2002.

[37] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering Methodology*, 14(1):1–41, 2005.

[38] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Proceedings of the 3rd Vir-*

*tual Machine Research and Technology Symposium (VM'04)*, pages 127–138, 2004.

[39] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 167–178, 2003.

[40] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 179–190, 2003.

[41] P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, 2006.

[42] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 14–24, 2004.

[43] M. Ronsse and K. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[44] A. Sasturkar, R. Agarwal, L. Wang, and S. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 83–94, 2005.

[45] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97)*, pages 27–37, 1997.

[46] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'89)*, pages 285–297, 1989.

[47] N. Sterling. WARLOCK - a static data race analysis tool. In *Proceedings of the Usenix Winter 1993 Technical Conference*, pages 97–106, 1993.

[48] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pages 125–135, 1999.

[49] C. von Praun and T. Gross. Object race detection. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 70–82, 2001.

[50] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 115–128, 2003.

[51] L. Wang and S. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 61–71, 2005.

[52] L. Wang and S. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, 2006.

[53] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 131–144, 2004.

[54] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 221–234, 2005.

[55] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 145–157, 2004.