Static Object Race Detection

Ana Milanova and Wei Huang

Rensselaer Polytechnic Institute

Abstract. We present a novel static object race detection analysis. Our analysis is data-centric in the sense that dominance and ownership, as well as object-based reasoning about control, play a crucial role. Our empirical results show that the analysis scales well and has relatively low false-positive rate. In some cases, our analysis outperforms the leading static race detector Chord.

1 Introduction

A multithreaded program contains an object race when two threads invoke methods on the same object "simultaneously" (i.e., without ordering constraints between them). An object race is a generalization of a data race [15]. It may or may not lead to a data race; however, an object race is necessary in order for a data race to occur. Reasoning about object races is valuable in several ways. First, it entails reasoning about object structure, in particular dominance-based ownership structure [3], which may facilitate localization and correction of concurrency bugs. Second, it complements data race detection because object races may expose hidden data races (e.g., data races on internal objects of library classes, which typically are not reported by data race detectors).

In this paper we present a novel static analysis for object race detection. Dominance, as well as object-based reasoning about control, play a crucial role.

Dominance is defined in terms of the notion of *object graph*. Nodes in the object graph are objects, and edges capture references between those objects. An edge links object i to object j if i has a field that refers to j, or a variable in a method invoked on receiver i, refers to j. Object i dominates (or owns) j if all paths from the root of the object graph to j go through i. Dominance plays an important role in object race detection. Namely, synchronization on a dominator i protects all objects j internal to i's dominance boundary. Conversely, lack of synchronization on i may expose object races deep in the boundary of i. Fig. 1 shows a program and Fig. 2 shows an *abstract object graph* for this program. In this example, allocation sites j, m, b, t and w are executed many times, resulting in many concrete objects. A typical static abstraction scheme maps every concrete object to its allocation site, thus these concrete objects are mapped to the same abstract objects j, m, b, t, and w.

Additionally, we define the notion of the *call graph*. Nodes in the call graph are tuples $i.\mathsf{m}$ where i is an object and m is a method name; the tuple denotes that method m executes on receiver object i. The edges represent calls: there is an edge from $i.\mathsf{m}$ to $j.\mathsf{n}$ if method m executing on receiver i calls method

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 255–271, 2011.

n on receiver j. This object-based call graph is natural for object-oriented languages where objects and control are inherently intertwined and synchronization is naturally object-based. It facilitates object race detection. For example, when control descends into i.m, if m holds the lock on i, this lock protects not only i but all objects j dominated by i, that are accessed along the call chain from i.m. Fig. 2 shows the *abstract call graph* for the example program.

```
class J extends Thread {
  static C c; int wId;
  static void main(String[] arg) {
   c = new C();   [c]
   for (int num=1; num<=3; num++) {
    c.inc();
    for (int wID=0; wID<num; wID++) {
      J j = new J();   [j]
      j.wId = wID;
      j.start();
    }
   }
  }
  public void run() {
    M m = new M();   [m]
    m.init(c,whId); c.addThread(m);
    m.go();
  }
}

class C {
  W[] a = new W[10];   [a]
  int num = 0;
  synchronized void inc() {
    W w = new W();   [w]
    a[num++] = w;
   }
  synchronized void addThread(M m) { ... }
  synchronized W getW(int i) {
    W w = a[i]; return w;
  }
}

class W {
  int count = 0; S s = ...;
  void update(H h) {
    this.count++; s.put(h);
  }
  synchronized get() {
    return s;
  }
 }

class M {
   int wId; C c;
   T[] b = new T[10];   [b]
   void init(int wID, C c) {
     this.wId = wID; this.c = c;
   }
   void go() {
     T t = new T()   [t]
     b[0] = t; t.init(wId,c); t.process();
   }
}

class T {
  int wId; C c; W w;
  void init(int wID, C c) {
    wId = wID; this.c = c;
    w = c.getW(wId);
  }
  void process() {
    S s = w.get(); ... w.update(new ...);
  }
}
```

Fig. 1. Example program

Our analysis classifies objects as *distributed* or *owned*. A distributed object is dominated only by the root of the object graph. In contrast, an owned object is dominated by at least one object (owner). The analysis first identifies races on distributed objects, and then descends into the dominance boundary of each object to identify races on owned objects. The main intuition is that in order

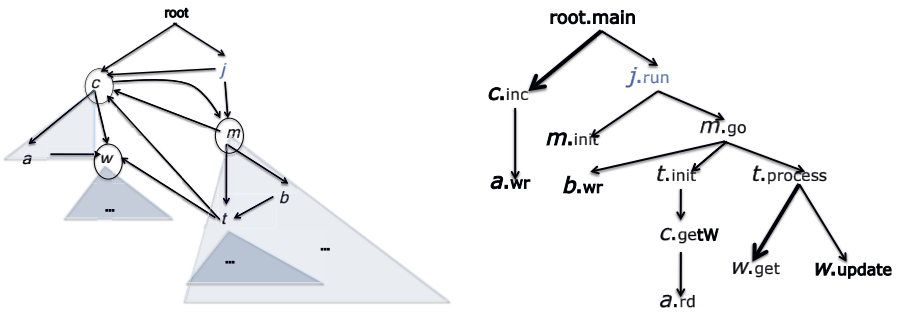

Fig. 2. Abstract object and call graphs for example program

to have a race on an owned object, we must first have a race on its dominator (owner). The analysis is best illustrated by an example. Consider Fig. 1 (modeled after benchmark SPECjbb). Method main forks multiple threads that act on the C (Company) object stored in static field c. Each thread creates a M (TransactionManager) object which in turn creates multiple T (Transaction) objects each accessing the C object and the W (Warehouse) objects. Our analysis identifies objects w, c and m as distributed (they are circled in the object graph in Fig. 2). There are two object races on w: $\langle w, \mathsf{get}, \mathsf{update}\rangle$ and $\langle w, \mathsf{update}, \mathsf{update}\rangle$. There are no races on c because all accesses to c are synchronized, and there are no races on m because each m is accessed only by its creating thread. The analysis proceeds to identify races in the boundary of w that are triggered by the two object races on w, $\langle w, \mathsf{get}, \mathsf{update}\rangle$ and $\langle w, \mathsf{update}, \mathsf{update}\rangle$. The lack of races on c entails that there are no races on a — a is owned by c and all accesses to a are protected by the lock on owner c. The lack of races on m entails that there are no races on owned t and b.

We have implemented the analysis and present results on several benchmarks. Our analysis presents relatively low false-positive rate and runs in less than 1 minute on all but one benchmark. On most benchmarks our analysis performs comparably to Chord [9], the leading static data race detector. On several benchmarks our analysis outperforms Chord, in some cases significantly.

The rest of the paper proceeds as follows. Section 2 formalizes the notions of object graph and call graph and presents a static analysis (abstract interpretation [4]) that infers safe abstract object and call graph. Section 3 describes the dominance inference analysis. Section 4 presents the object race detection analysis. Section 5 describes our implementation and experience with the analysis, Section 6 discusses related work and Section 7 concludes the paper.

2 Formal Account of Object Graphs

We explain our algorithm in terms of a core Java-like calculus. Throughout the paper we will use the following notation for graphs. An *object graph* G is a pair

cd	$::=$ class C extends D $\{\overline{fd}\ \overline{md}\}$	*class*	T	$::= tS$	*thread*
fd	$::= \tau$ f	*field*	H	$::= [] \mid H[i \mapsto o]$	*heap*
md	$::= \tau$ m$(\overline{\tau\ \mathsf{x}})\{\overline{\tau\ \mathsf{z}}$ s; return y$\}$	*method*	S	$::= \epsilon \mid \langle$m F s$\rangle S$	*stack*
s	$::=$ s; s $\mid$ x = new C() $\mid$ x = this.f	*statement*	F	$::= [] \mid F[$y $\mapsto i]$	*frame*
	$\mid$ this.f = y $\mid$ x = y.m$(\overline{\mathsf{z}})$		o	$::=$ C$(\overline{i})$	*object*
τ	$::=$ C	*type*			

Fig. 3. Syntax

(N, E) where N is a set of objects ranged over by variables i, j, k, l and E is a set of directed edges written $i \triangleright j$. We write $i \in G$ and $i \triangleright j \in G$ to test, respectively node and edge membership. A *call graph* C is a pair (N, E) where N is a set of tuples written $i.\mathsf{m}$, where i is an object and m is a method, and E is a set of directed edges written $i.\mathsf{m} \triangleright j.\mathsf{n}$. The meaning of a call graph edge is that method m invoked on receiver object i calls method n on receiver object j. Again, we write $i.\mathsf{m} \in C$ and $i.\mathsf{m} \triangleright j.\mathsf{n} \in C$ to test membership.

2.1 Concrete Semantics

For brevity, we restrict our formal attention to a core calculus in the style of [14] whose syntax appears in Fig. 3. The language models Java with a syntax in A-normal form. Fields are strongly private. Array accesses are modeled by special methods rd and wr — array read x=y[i] is treated as method call x = y.rd() and array write x[i]=y is treated as x.wr(y); index i is irrelevant for our purposes and is omitted. Throughout the paper, metavariables m and n range over all method names and rd and wr. Features not strictly necessary are omitted.

The concrete semantics operates over configurations of the form $H; \overline{T}; G; C; P$ where H is a single heap, $\overline{T}$ is a collection of threads, G is a summary object graph, C is a summary call graph and P holds auxiliary information necessary to construct C. A heap is a mapping from indices, ranged over by meta-variables i, j, k, l, to objects. Each thread T has its own stack S and a unique thread identifier t. A stack is a sequence of frames $\langle$m F s$\rangle$ consisting of a method name m, a mapping F from variables to locations and a statement s. An object $o = \mathsf{C}(\overline{i})$ consists of a class C and values $\overline{i}$ for the object fields. An object graph G summarizes all references between objects. A call graph C summarizes all method calls between objects.

We write $\overline{i}$ to denote a sequence of indices, $\overline{\tau\ \mathsf{z}}$ for a sequence of local variable declarations, etc. We write 0 to denote the null reference.

Following [14], a multi-threaded Java program is modeled as a fixed set of threads $\overline{T}$, each of which starts with a call to a run method, and terminates when the run method returns. The reduction relation $\xrightarrow{l}_t$ represents a step in the semantics; l is an action label and t is the identifier of the thread that executed that action. We use action labels on methods calls $\rightarrow i.\mathsf{m}$ (call), and on method returns $\leftarrow i.\mathsf{m}$ (return), as well as the empty label ϵ. Later in the

paper, labels are used to define traces and object races. Thread scheduling is modeled as a non-deterministic choice where each step picks one of the threads for reduction (see [14] for the rule). Due to space constraints, the rules of the concrete semantics are not shown here.

2.2 Abstract Semantics

We assume a *may* points-to analysis that computes a safe approximation of the heap $\widehat{H}$, collection of threads $\widehat{\overline{T}}$, and each stack $\widehat{S}$. The abstract semantics computes safe approximations of G and C, denoted $\widehat{G}$ and $\widehat{C}$ respectively. As $\widehat{H}$ and $\widehat{S}$ are conservative approximations, the semantics operates on sets of *abstract objects*. Thus, $\widehat{F}(\mathsf{x})$ evaluates to a set of abstract objects, not to a single object. Similarly, fields of an object in $\widehat{H}$ are sets of references (denoted I). We assume that all allocation sites are labelled with an unique identifier.

The abstraction function α is specific to our points-to analysis and is chosen so that $\alpha(i) = i'$ where i' is the index of the allocation site that created i. The abstraction applies to threads as well: $\alpha(t) = t'$ where t' is the index of the allocation site that created the $\mathsf{java.lang.Thread}$ object that started t's run. α acts on G in the obvious way: $\alpha(G) = (N, E)$, where $N = \{\alpha(i) \mid i \in G\}$ and $E = \{\alpha(i) \triangleright \alpha(j) \mid i \triangleright j \in G\}$. Similarly, α acts on C: $\alpha(C) = (N, E)$ where $N = \{\alpha(i).\mathsf{m} \mid i.\mathsf{m} \in C\}$ and $E = \{\alpha(i).\mathsf{m} \triangleright \alpha(j).\mathsf{n} \mid i.\mathsf{m} \triangleright j.\mathsf{n} \in C\}$.

As the points-to analysis is safe, the following two conditions hold at every step. The first condition ensures the safety of variables, and the second ensures the safety of fields.

$$\begin{array}{rcl} F(\mathsf{x}) = i & \Rightarrow & \alpha(i) \in \widehat{F}(\mathsf{x}) \\ H(i) = \mathsf{C}(...k_{\mathsf{f}}...) & \Rightarrow & \widehat{H}(\alpha(i)) = \mathsf{C}(...I_{\mathsf{f}}...) \ \wedge \ \alpha(k_{\mathsf{f}}) \in I_{\mathsf{f}} \end{array}$$

The rules of the abstract semantics use $\widehat{H}$ and $\widehat{F}$ and compute $\widehat{G}$ and $\widehat{C}$. We write $\widehat{G}$ += $i \triangleright j$ to denote the addition i and j to the nodes of $\widehat{G}$ and $i \triangleright j$ to the edges of $\widehat{G}$. Similarly, we write $\widehat{C}$ += $i.\mathsf{m} \triangleright j.\mathsf{n}$ to denote the addition of $i.\mathsf{m}$ and $j.\mathsf{n}$ to the nodes of $\widehat{C}$, and $i.\mathsf{m} \triangleright j.\mathsf{n}$ to the edges of $\widehat{C}$. Auxiliary function *dispatch* takes as argument the class of the receiver C and the call site id c and returns the run-time target n.

Fig. 4(a) (left column) shows the rules for constructing object graph $\widehat{G}$. (ANEW) adds new edges to $\widehat{G}$ from every abstract receiver i of current frame m, to the abstract object j created at allocation site j. Rule (ACALL) adds new edges to $\widehat{G}$ from every abstract object k in the points-to set of y, to every j in the points-to set of an actual argument z, and from each abstract receiver i of method m, to each j in the points-to set of the return variable of n, $\mathsf{ret_n}$. Note that calls through this, e.g., $\mathsf{this.n}(\overline{\mathsf{z}})$, do not add edges to $\widehat{G}$. This is correct because when the call is through this the relevant abstract edges are already in $\widehat{G}$ and there is no need to add them again.

Fig. 4(b) (right column) shows the rules for constructing call graph $\widehat{C}$. The first two rules compute sets $\widehat{P}_{k.\mathsf{n}}$, the set of caller tuples for $k.\mathsf{n}$. (ACALL) adds

(ANEW)
$\mathsf{x} = \mathsf{new}_j\ \mathsf{C}()$ in method $\mathsf{m} \Rightarrow$
$\quad \widehat{G}\ \mathrel{+}= \{k \triangleright j | k \in \widehat{F}(\mathsf{this_m})\}$

(ACALL)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$, $\mathsf{y} \neq \mathsf{this}$, in method $\mathsf{m} \Rightarrow$
$\quad$ *foreach* $k \in \widehat{F}(\mathsf{y})$
$\quad\quad \widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
$\quad\quad \widehat{G}\ \mathrel{+}= \{k \triangleright j \mid \overline{j} \in \widehat{F}(\overline{\mathsf{z}}) \wedge j \in \overline{j}\}$
$\quad\quad \widehat{G}\ \mathrel{+}= \{i \triangleright j \mid i \in \widehat{F}(\mathsf{this_m}) \wedge j \in \widehat{F}(\mathsf{ret_n})\}$

(a) Object graph $\widehat{G}$.

(ACALL)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$, $\mathsf{y} \neq \mathsf{this}$, in method $\mathsf{m} \Rightarrow$
$\quad$ *foreach* $k \in \widehat{F}(\mathsf{y})$
$\quad\quad \widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
$\quad\quad \widehat{P}_{k.\mathsf{n}}\ \mathrel{+}= \{i.\mathsf{m} \mid i \in \widehat{F}(\mathsf{this_m})\}$

(ACALLTHIS)
c: $\mathsf{x} = \mathsf{this.n}(\overline{\mathsf{z}})$ in method $\mathsf{m} \Rightarrow$
$\quad$ *foreach* $k \in \widehat{F}(\mathsf{y})$
$\quad\quad \widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
$\quad\quad \widehat{P}_{k.\mathsf{n}}\ \mathrel{+}= \widehat{P}_{k.\mathsf{m}}$

(ACALL) and (ACALLTHIS)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$ in method $\mathsf{m} \Rightarrow$
$\quad$ *foreach* $k \in \widehat{F}(\mathsf{y})$
$\quad\quad \widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
$\quad\quad \widehat{C}\ \mathrel{+}= \{i.\mathsf{m} \triangleright k.\mathsf{n} \mid i.\mathsf{m} \in \widehat{P}_{k.\mathsf{n}}\}$

(b) Call graph $\widehat{C}$.

Fig. 4. $\widehat{G}$, $\widehat{C}$, and each $\widehat{P}_{k.\mathsf{n}}$ are initialized to $\emptyset$. The rules (i.e., transfer functions) are applied iteratively until they reach fixpoint.

$i.\mathsf{m}$, where i is an abstract receiver of m, to $\widehat{P}_{k.\mathsf{n}}$. Rule (ACALLTHIS) adds $\widehat{P}_{k.\mathsf{m}}$, the set of caller tuples for $k.\mathsf{m}$ to $\widehat{P}_{k.\mathsf{n}}$. The last rule, applied to both (ACALL) and (ACALLTHIS), adds edges to $\widehat{C}$ from each tuple $i.\mathsf{m}$ in $\widehat{P}_{k.\mathsf{n}}$ to $k.\mathsf{n}$. Edges $i.\mathsf{m} \triangleright k.\mathsf{n}$ reflect that m, called on receiver i, calls n on receiver k (the edges "bypass" chains of calls on k through this). As it is customary with abstract interpretations, the rules (i.e., transfer functions) are applied repeatedly until $\widehat{P}_{k.\mathsf{n}}$ and $\widehat{C}$ reach fixpoint.

Note the explicit distinction of (ACALL) and (ACALLTHIS). A naive analysis will treat them identicaly, i.e., $\mathsf{y.n}()$ in method m would lead to edges from every tuple $i.\mathsf{m}$ where $i \in \widehat{F}(\mathsf{this_m})$ to every tuple $j.\mathsf{n}$, where $j \in \widehat{F}(\mathsf{y})$ regardless of whether y is this or not. $\widehat{F}(\mathsf{this_m})$ typically refers to a *set* of abstract objects. For example, if $\widehat{F}(\mathsf{this_m})$ is $\{i, j\}$ and the call is $\mathsf{this.n}()$, the naive analysis would lead to edges $i.\mathsf{m} \triangleright i.\mathsf{n}$, $i.\mathsf{m} \triangleright j.\mathsf{n}$, $j.\mathsf{m} \triangleright i.\mathsf{n}$ and $j.\mathsf{m} \triangleright j.\mathsf{n}$, when clearly, only the first and the last are feasible. A less naive analysis may make the distinction between (ACALL) and (ACALLTHIS), and at (ACALLTHIS) only create edges $i.\mathsf{m} \triangleright i.\mathsf{n}$, where $i \in \widehat{F}(\mathsf{this_m})$. This is not sufficient, because abstract edge $i.\mathsf{m} \triangleright i.\mathsf{n}$ may represent a call through this on the same concrete object, or a call from one concrete object i' to a different concrete object i'' where both i' and i'' are mapped to the same abstract i. Our analysis must capture transfer of control between distinct objects (i.e., inter-object transfer of control); by propagating the tuple that starts the chain of calls through this, it captures all inter-object transfer of control.

3 Dominance Inference Analysis

This section outlines dominance inference analysis and states its correctness results. This analysis is at the heart of object race detection (Section 4), but its details are beyond the scope of this paper. More details are available in [7].

We begin the description with several definitions. Let G be any directed graph. A *path* is written as $p = n_0 \triangleright n_1 \triangleright n_2 \triangleright \ldots n_{m-1} \triangleright n_m$; the trivial path is written as n_0 and a self-loop is written as $n_0 \triangleright n_0$. A root for G is a node $r \in G$ such that for all nodes $n \in G$ there is a (possibly trivial) path from r to n. A *boundary* for a node $n \in G$ is any graph $B_n \subseteq G$ such that n is a root of B_n. We assume that G has root root. A node $n \in G$ *dominates* node $n' \in G$ if all paths from root to n' go through n. The *dominance boundary* for a node $n \in G$ is the maximal boundary B_n such that for all nodes $n' \in B_n$, n *dominates* n' in G. We denote the dominance boundary of $n \in G$ as D_n.

3.1 Dominance Boundary

Dominance boundary analysis takes as input the abstract object graph $\widehat{G}$ and abstract object i, and computes $\widehat{B}_i \subseteq \widehat{G}$, the *abstract dominance boundary* of i. The following theorem holds for $\widehat{B}_i$:

Theorem 1. *Let G be any object graph and i be any object in G. Let B'_i be any boundary of i in G. If $\alpha(B'_i) \subseteq \widehat{B}_{\alpha(i)}$ then $B'_i \subseteq D_i$.*

The theorem states that the computed $\widehat{B}_{\alpha(i)}$ safely approximates the dominance boundary of i. That is, for any concrete boundary B'_i whose abstract representative is included in $\widehat{B}_{\alpha(i)}$, B'_i is included in D_i, or in other words, i dominates in G all of B'_i's nodes. Consider our running example. The abstract dominance boundary of object m, $\widehat{B}_m$, includes edges $m \triangleright b$, $m \triangleright t$ and $b \triangleright t$. The theorem states that every concrete m dominates the b and t objects it refers to.

3.2 Minimal Boundaries

Minimal boundary analysis takes as input an abstract object graph $\widehat{G}$ and an edge $i \triangleright j \in \widehat{G}$, and returns a set of objects, which we denote by $\widehat{minB}_{i \triangleright j}$. Each node $k \in \widehat{minB}_{i \triangleright j}$ is a root of a dominance boundary $\widehat{B}_k$ containing $i \triangleright j$. In our running example $\widehat{minB}_{b \triangleright t}$ equals $\{m\}$. Edge $b \triangleright t$ is contained in the boundary of root as well; however, the boundary of m is the minimal boundary. As another example, $\widehat{minB}_{t \triangleright w}$ equals $\{\mathsf{root}\}$.

Let G be any concrete object graph. We say that $k \in G$ *covers* $j \in G$ if for every path p from k to j, $p = k \triangleright \cdots \triangleright j$, $\alpha(p) \in \widehat{B}_{\alpha(k)}$. The following theorem ensures the safety of $\widehat{minB}$:

Theorem 2. *Let G be any concrete object graph and let $i \triangleright j \in G$ be any edge. There exists $k \in G$, $k \neq j$, such that (1) $\alpha(k) \in \widehat{minB}_{\alpha(i \triangleright j)}$ and (2) k covers j.*

The theorem guarantees the safety of the minimal boundary analysis. It states that $\widehat{minB}_{i \triangleright j}$ "covers" every concrete edge represented by $i \triangleright j$. In other words, for every concrete edge, we consider at least one root (and its boundary) that abstracts a *dominator* of that concrete edge (although not necessarily the immediate dominator). The theorem below ensures the minimality of $\widehat{minB}$:

Theorem 3. *Let $i \triangleright j \in G$ be any edge. Let $k \in G$, $k \neq j$ be such that (1) $\alpha(k) \in \widehat{minB}_{\alpha(i \triangleright j)}$ and (2) k covers j. For every k' if k dominates k' and k' covers j, then $\alpha(k') \in \widehat{minB}_{\alpha(i \triangleright j)}$.*

Informally, the theorem states that if there is a dominator k' which is closer than k, and k' covers j, then $\alpha(k')$ will be contained in $\widehat{minB}$.

4 Object Race Detection

We begin with the definition of an object race. In the style of [14], the execution of the program is viewed as a trace Tr of events $Tr = e_1, e_2 \cdots e_n$ performed by different threads. As in [14], an event is a tuple $e = (H, \overline{T}, l, t)$ which consists of a partial configuration $H; \overline{T}$, an action label l and a thread id t. An object race occurs when an event with a method call $\rightarrow j.\mathsf{n}$ occurs, and there is an outstanding call $j.\mathsf{n}'$ on the same receiver j made by a different thread (essentially, this is the complement of atomic set serializability as defined in [14]).

Definition 1. *There is an object race, denoted by $\langle j, \mathsf{n}, \mathsf{n}' \rangle$, when trace Tr contains event $e = (H, \overline{T}, j.\mathsf{n}, t)$, such that $\exists t' S \in \overline{T}$ where $t' \neq t$ and $\langle \mathsf{n}'\ F\ \mathsf{s} \rangle \in S$ and $F(\mathsf{this}) = j$.*

For convenience, we extend the above notation for events with calls to include the caller tuple. Namely, let event e correspond to step $H; \overline{T}; G; C; P \xrightarrow{\rightarrow j.\mathsf{n}}_t H'; \overline{T'}; G'; C'; P'$. Instead of $e = (H, \overline{T}, \rightarrow j.\mathsf{n}, t)$ we write $e = (H, \overline{T}, i.\mathsf{m} \triangleright j.\mathsf{n}, t)$ where $i.\mathsf{m} = P'$ (i.e., $i.\mathsf{m}$ started the chain of calls through this on receiver j).

An object race $\langle j, \mathsf{n}, \mathsf{n}' \rangle$ entails that there are paths $p = t.\mathsf{run} \triangleright \cdots \triangleright j.\mathsf{n} \in C$ and $p' = t'.\mathsf{run} \triangleright \cdots \triangleright j.\mathsf{n}' \in C$, where $t' \neq t$. Our object race detection analysis (Section 4.3) traverses pairs of abstract paths $p{=}t.\mathsf{run} \triangleright \cdots \triangleright j.\mathsf{n} \in \widehat{C}$ and $p'{=}t'.\mathsf{run} \triangleright \cdots \triangleright j.\mathsf{n}' \in \widehat{C}$, where abstract t and t' are not necessarily different, and discovers object races. The analysis uses reentrancy analysis (Section 4.1), and lock analysis (Section 4.2) to avoid infeasible races. Non-reentrancy of edge $i \triangleright j$ guarantees (informally) that no two threads can execute events on $i \triangleright j$; thus, $i.\mathsf{m} \triangleright j.\mathsf{n} \in p$ and $i.\mathsf{m}' \triangleright j.\mathsf{n}' \in p'$ does not contribute an object race on j. Lock analysis associates locksets with events; non-empty intersection of two locksets guarantees (again informally) that events are executed serially.

4.1 Reentrancy Analysis

Reentrancy analysis computes predicate *reentrant*: $i \triangleright j \in \widehat{G} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ with the following properties. Let $i \triangleright j$ be any concrete edge, in any G. If *reentrant*($\alpha(i \triangleright$

j)) equals **false**, then (a) i creates j due to (DNEW) and (b) trace Tr does not contain a pair of events $e = (H, \overline{T}, i.\mathsf{m} \triangleright j.\mathsf{n}, t)$ and $e' = (H', \overline{T'}, i.\mathsf{m}' \triangleright j.\mathsf{n}', t')$ such that $t' \neq t$. Informally, if an edge is not reentrant, no two distinct threads can execute events on it.

We compute *reentrant* by first computing two sets of edges, *Fields* and *Flows*. Set *Fields* contains all abstract field edges:

$$\widehat{H}(i) = \mathsf{C}(...I_\mathsf{f}...) \ \wedge \ j \in I_\mathsf{f} \quad \Rightarrow \quad \mathit{Fields} \mathrel{+}= i \triangleright j$$

Set *Flows* contains all edges that capture object flow (i.e., object transfer from one object to another). Set *Flows* is computed during the construction of $\widehat{G}$; specifically, rule (ACALL) in Fig. 4(a) (left column) is augmented with the following two lines after the last line $\widehat{G}$ += ...:

$$\mathit{Flows} \mathrel{+}= \{k \triangleright j \mid \overline{j} \in \widehat{F}(\overline{\mathsf{z}}) \wedge j \in \overline{j}\}$$
$$\mathit{Flows} \mathrel{+}= \{i \triangleright j \mid i \in \widehat{F}(\mathsf{this}_\mathsf{m}) \wedge j \in \widehat{F}(\mathsf{ret}_\mathsf{n})\}$$

Objects j passed as arguments or returned at calls $\mathsf{x} = \mathsf{y}.\mathsf{m}(\overline{\mathsf{z}})$, $\mathsf{y} \neq \mathsf{this}$, are transferred, and the resulting edges are added to *Flows*. We now define *reentrant*:

$$\mathit{reentrant}(i \triangleright j) = \begin{cases} \textbf{true} & \text{if } i \triangleright j \in \mathit{Fields} \vee i \triangleright j \in \mathit{Flows} \\ \textbf{false} & \text{otherwise} \end{cases}$$

In our running example, edge $j \triangleright m$ is not reenetrant. In every call to run, thread j creates a new m object; m is not stored as a field of j and m does not flow back to j. It is impossible for one thread j to access another thread's m. Note that the m objects are not thread-local, because they escape to a field of static object c.

One can show that $\mathit{reentrant}(\alpha(i \triangleright j)) = \textbf{false}$ implies properties (a) and (b) stated at the beginning of this section. To show (b), suppose that there exists an edge $i \triangleright j$ in some G, such that $\mathit{reentrant}(\alpha(i \triangleright j)) = \textbf{false}$, and the trace contains events on $i \triangleright j$ $e = (H, \overline{T}, i.\mathsf{m} \triangleright j.\mathsf{n}, t)$ and $e' = (H', \overline{T'}, i.\mathsf{m}' \triangleright j.\mathsf{n}', t')$ such that $t' \neq t$. Let thread t create j, and consider thread t' and method m'. Roughly, m' can obtain a reference to j through object creation (rule (DNEW)), flow (rules (DCALL) and (DRET)) or field read ($\mathsf{x} = \mathsf{this.f}$); object creation is impossible because t created j, flow is impossible as well because then we would have had $\alpha(i \triangleright j) \in \mathit{Flows}$ and therefore $\mathit{reentrant}(\alpha(i \triangleright j))$ would have been **true**, and field read is impossible, because then we would have had $\alpha(i \triangleright j) \in \mathit{Fields}$ and again, $\mathit{reentrant}(\alpha(i \triangleright j))$ would have been **true**.

4.2 Lock Analysis

The lock analysis computes a map $\widehat{L}$ from abstract call graph edges to locksets. $\widehat{L}$ has the following property. Consider a pair of events with calls on object j: $e = (H, \overline{T}, i.\mathsf{m} \triangleright j.\mathsf{n}, t)$ and $e' = (H', \overline{T'}, i'.\mathsf{m}' \triangleright j.\mathsf{n}', t')$ such that $t' \neq t$. $\widehat{L}(\alpha(i.\mathsf{m} \triangleright j.\mathsf{n})) \cap \widehat{L}(\alpha(i'.\mathsf{m}' \triangleright j.\mathsf{n}')) \neq \emptyset$ implies that $j.\mathsf{n}$ and $j.\mathsf{n}'$ must be executed serially, or in other words, that events e and e' do not contribute an object race $\langle j, \mathsf{n}, \mathsf{n}' \rangle$. Informally, a non-empty intersection of two locksets guarantees serializability of the represented concrete events.

Source and Target Locksets. The first part of the lock analysis associates two sets, Sl_c (*source lockset*), and Tl_c (*target lockset*) to every call site c. Source lockset refers to the source tuple $i.\mathsf{m}$, and target lockset refers to the target tuple $k.\mathsf{n}$ in the call graph edge due to c. For example, call c in synch (this) { ...c: y.n($\overline{\mathsf{z}}$)...} in method m, leads to this being in Sl_c. this refers to the receiver of m; thus, for every edge $i.\mathsf{m} \triangleright k.\mathsf{n}$ due to c, the executing thread holds the lock on i before descending into the execution of $k.\mathsf{n}$.

All sets Sl_c and Tl_c are initialized to $\emptyset$ and updated as follows. If c occurs in a method m declared synchronized, then this is added to Sl_c. If the target at c is declared synchronized, then this is added to Tl_c. Additionally, we consider four patterns of usage of synchronized blocks. We note, however, that even without considering synchronized blocks, the lock analysis will be reasonably powerful, because synchronization in Java is naturally object-based (i.e., it is achieved by declaring methods as synchronized).

Self locking occurs when the lock variable is this. We add this to Sl_c when the receiver variable $\mathsf{y} \neq \mathsf{this}$; we add this to Tl_c when the receiver variable is this.

$$\begin{aligned} \mathsf{synch}\ (\mathsf{this})\ \{...c\colon \mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})...\} \wedge \mathsf{y} \neq \mathsf{this} \quad &\Rightarrow \quad Sl_c \mathrel{+}= \mathsf{this} \\ \mathsf{synch}\ (\mathsf{this})\ \{...c\colon \mathsf{x} = \mathsf{this.n}(\overline{\mathsf{z}})...\} \quad &\Rightarrow \quad Tl_c \mathrel{+}= \mathsf{this} \end{aligned}$$

Global locking occurs when the lock variable lock is a static field, which is initialized exactly once during class initialization: static C lock = new C(); $\boxed{l}$.

$$\mathsf{synch}\ (\mathsf{lock})\ \{...c\colon \mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})...\} \quad \Rightarrow \quad Tl_c \mathrel{+}= l$$

Local-object locking occurs when the lock variable lock is an instance field, which is initialized exactly once during object initialization: there is a field C lock; and lock = new C(); $\boxed{l}$ occurs in the constructor. We have:

$$\mathsf{synch}\ (\mathsf{lock})\ \{...c\colon \mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})...\} \wedge \mathsf{y} \neq \mathsf{this} \quad \Rightarrow \quad Sl_c \mathrel{+}= l$$

Client-side locking occurs when the lock variable y is the receiver at the call.

$$\mathsf{synch}\ (\mathsf{y})\ \{...c\colon \mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})...\} \quad \Rightarrow \quad Tl_c \mathrel{+}= \mathsf{this}$$

When y is a local variable, the addition of this is safe because stack locations in Java do not have aliases. When y is an instance field, however, the addition of this is not necessarily safe. In bytecode synch (f) {...x = f.n($\overline{\mathsf{z}}$)...} becomes z = this.f; synch (z) {...w = this.f; w.n(...)...} and there could have been a write to f between the two reads. We add this to Tl_c only when our analysis proves that it is safe (e.g., f is initialized once in the constructor and is readonly afterwards).

Fig. 5 illustrates the above patterns of usage of synchronized blocks.

Map $\widehat{L}$. The second part of the lock analysis computes $\widehat{L}$, a map from call graph edges to locksets. The elements of a lockset are abstract objects plus this. For example, suppose that $\widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}) = \{l, \mathsf{this}\}$ where l is a global lock. The analysis guarantees that for every concrete edge $i'.\mathsf{m} \triangleright j'.\mathsf{n}$ represented by

```
class Account
  AccountImpl acc = ...;
  void update(int amt) {
    synch (this) {
   c:  x = acc.get();
       acc.put(amt+x);
    }
  }
```

(a) Self locking:
$Sl_c = \{\mathsf{this}\}$, $Tl_c = \emptyset$

```
class Account
  AccountImpl acc = ...
  Object lock = ... [l]
  void update(int amt) {
    synch (lock) {
   c:  x = acc.get();
       acc.put(amt+x);
    }
  }
```

(b) Local-object locking:
$Sl_c = \{l\}$, $Tl_c = \emptyset$

```
class Account
  AccountImpl acc = ...
  void update(int amt) {
    synch (acc) {
   c:  x = acc.get();
       acc.put(amt+x);
    }
  }
```

(c) Client-side locking:
$Sl_c = \emptyset$, $Tl_c = \{\mathsf{this}\}$

Fig. 5. Patterns of usage of synchronized blocks

(ACALL)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$, $\mathsf{y} \neq \mathsf{this}$, in method $\mathsf{m} \Rightarrow$
  foreach $k \in \widehat{F}(\mathsf{y})$
    $\widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
    $\widehat{P}_{k.\mathsf{n}}$ += $\{(i.\mathsf{m}, Sl_c, Tl_c) \mid i \in \widehat{F}(\mathsf{this}_\mathsf{m})\}$

(ACALLTHIS)
c: $\mathsf{x} = \mathsf{this.n}(\overline{\mathsf{z}})$ in method $\mathsf{m} \Rightarrow$
  foreach $k \in \widehat{F}(\mathsf{y})$
    $\widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
    $\widehat{P}_{k.\mathsf{n}}$ += $\{(i.\mathsf{m}', Sl \cup Sl_c, Tl \cup Tl_c) \mid$
      $(i.\mathsf{m}', Sl, Tl) \in \widehat{P}_{k.\mathsf{m}}\}$

(ACALL) and (ACALLTHIS)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$ in method $\mathsf{m} \Rightarrow$
  foreach $k \in \widehat{F}(\mathsf{y})$
    $\widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
    foreach $(i.\mathsf{m}, Sl, Tl) \in \widehat{P}_{k.n}$
      $lockset = Tl \cup Tl_c$
      if $i \triangleright k \notin \mathit{Flow} \vee i \triangleright k \in \widehat{B}_i$
        if $\mathsf{this} \in Sl \cup Sl_c$
          $lockset$ += i
        $lockset$ += $(Sl \cup Sl_c) \backslash \{\mathsf{this}\}$
      $\widehat{C}$ += $\{i.\mathsf{m} \triangleright k.\mathsf{n} \mid (i.\mathsf{m}, Sl, Tl) \in \widehat{P}_{k.\mathsf{n}}\}$
      $\widehat{L}' = \widehat{L}[i.\mathsf{m} \triangleright k.\mathsf{n} \mapsto S]$, where
        $S = \widehat{L}(i.\mathsf{m} \triangleright k.\mathsf{n}) \cap lockset$

Fig. 6. Lock analysis. $\widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n})$ are initialized to the maximal set of locks. The rules are applied iteratively until they reach fixpoint.

$i.\mathsf{m} \triangleright j.\mathsf{n}$, the thread executing the edge holds the lock of l and the lock of j' before descending into the execution of $j'.\mathsf{n}$.

The analysis (Fig. 6) extends $\widehat{P}_{k.\mathsf{n}}$ to hold caller tuples $i.\mathsf{m}$ and the source and target locksets associated with $i.\mathsf{m}$. (ACALL) records Sl_c and Tl_c with $i.\mathsf{m}$. (ACALLTHIS) propagates the locksets of caller tuples down the this-call chain. For example, consider a method m which contains synch (lock) {... c: y.n ()...}, where lock is a global lock that points to l, and, in turn, n contains a call c': this.n′(). The analysis creates call graph edges $i.\mathsf{m} \triangleright j.\mathsf{n}'$ where i is an abstract receiver of m, and j' is a receiver at call site c'. The lock analysis propagates $Sl_c = \emptyset$ and $Tl_c = \{l\}$ to c' with $i.\mathsf{m}$; clearly, the call to $j.\mathsf{n}'$ from $i.\mathsf{m}$ is protected by the global lock.

The last rule (right column in Fig. 6) associates locksets to call graph edges. When adding an edge to $\widehat{C}$, it also computes a lockset, *lockset*, for that edge. The most interesting aspect of this rule is that Sl is propagated to *lockset* only if $i \triangleright k$ is not in *Flows* or it is in the boundary of i. This is necessary to ensure safety

of the analysis. If $i \triangleright k$ is in *Flows* and $i \triangleright k$ is not in the boundary of i, abstract object i may refer to different concrete objects, say i' and i'', with concrete edges $i' \triangleright k$ and $i'' \triangleright k$ both represented by the same abstract edge; thus, at runtime, this would refer to i' along edge $i' \triangleright k$, and to i'' along edge $i'' \triangleright k$; if one thread executes events along the first edge, and another thread executes events along the second edge, k remains unprotected. Analogous reasoning applies when *Sl* contains a local-object lock l. For example, consider Fig. 5(b). Let update be called with abstract receivers i and j, and let k be the abstract AccountImpl object. Assuming $i \triangleright k \in \widehat{B}_i$, we have $\widehat{L}(i.\mathsf{update} \triangleright k.\mathsf{get}) = \{l\}$.

The lock analysis is safe but not complete. It exploits the fact that synchronization in Java is naturally object-based (i.e., through this), and focuses on the "specialness" this. The special handling of this is what sets our analysis apart from other lock analyses such as [8] and [10] which aim at generality and appear to treat this as a regular reference variable. Our analysis is able to handle the vast majority of cases handled by the more complex analyses such as [8]. The empirical results confirm this conjecture. Furthermore, our analysis is extensible, as one can easily add new patterns of synchronized blocks.

4.3 Object Race Detection Analysis

The object race detection analysis is shown in Fig. 7. For ease of presentation, we make the following simplifying assumptions. First, there is a single static thread-fork site y.start, and it is located in main, second, the points-to set of y contains a single abstract object t, and three, run contains no calls through this. The analysis can be extended to handle arbitrarily many and arbitrarily located thread-fork sites, arbitrary points-to sets of y, and run methods that contain calls through this. Our implementation handles all cases.

The analysis computes a set of abstract races R. The following holds for R.

Theorem 4. *For every object race* $\langle j, \mathsf{n}, \mathsf{n}' \rangle$ *in trace Tr,* $\langle \alpha(j), \mathsf{n}, \mathsf{n}' \rangle \in R$.

Procedure *AllRaces* is the main driver. First, it identifies races on distributed objects (line 1); second, it descends into the boundary of each i to identify races on owned objects (lines 3-6). Procedure *DistributedRaces* maintains sets S_t and $S_{t'}$, which represent the accesses made by an arbitrary pair of threads t and t'. Note that each access is recorded with its lockset. Lines 3-12 traverse $\widehat{C}$ starting at t.run (i.e, the single thread-fork site y.start). The interesting part of this traversal is that $j.\mathsf{n}$ is added to $S_{t'}$ only if $i \triangleright j$ is reentrant; if $i \triangleright j$ is not reentrant, then threads cannot race on j along this edge (recall Section 4.1). In addition, *DistributedRaces* traverses $\widehat{C}$ starting at root.main and discovers accesses due to the main thread (lines 13-20). Finally, it computes and returns the set of races on distributed objects (line 21-22).

Procedure *Reach* takes as input a tuple $k.\mathsf{m}'$ and records all accesses $j.\mathsf{n}$ reachable from $k.\mathsf{m}'$ within the dominance boundary of k, $\widehat{B}_k$. $j.\mathsf{n}$ is recorded in S only if k is a minimal boundary of $i \triangleright j$ (recall minimal boundaries form Section 3.2). If k is not a minimal boundary, then there exists a smaller boundary, say of k'. The analysis will first discover races on the "closer dominator" k';

procedure *DistributedRaces*$(\widehat{G}, \widehat{C})$
output R
[1] $R = \emptyset, W = \{t.\mathsf{run}\}$
[2] $S_t = \emptyset, S_{t'} = \emptyset$
[3] while $W \neq \emptyset$
[4] remove $i.\mathsf{m}$ from W, mark $i.\mathsf{m}$ visited
[5] foreach $i.\mathsf{m} \triangleright j.\mathsf{n} \in \widehat{C}$
[6] if $\mathsf{root} \in \widehat{minB}_{i \triangleright j}$
[7] if *reentrant*$(i \triangleright j)$
[8] S_t += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[9] $S_{t'}$ += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[10] else
[11] S_t += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[12] if $j.\mathsf{n}$ not visited then W += $j.\mathsf{n}$

[13] $W = \{\mathsf{root.main}\}$
[14] while $W \neq \emptyset$
[15] remove $i.\mathsf{m}$ from W, mark $i.\mathsf{m}$ visited
[16] if $i.\mathsf{m} = t.\mathsf{run}$ continue
[17] foreach $i.\mathsf{m} \triangleright j.\mathsf{n} \in \widehat{C}$
[18] if $\mathsf{root} \in \widehat{minB}_{i \triangleright j}$
[19] $S_{t'}$ = += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[20] if $j.\mathsf{n}$ not visited then W += $j.\mathsf{n}$

[21] R += $\{\langle j, \mathsf{n}, \mathsf{n}' \rangle \mid (j.\mathsf{n}, ls) \in S_t \wedge (j.\mathsf{n}', ls') \in S_{t'} \wedge ls \cap ls' = \emptyset\}$
[22] return R

procedure *Reach*$(k.\mathsf{m}', \widehat{G}, \widehat{C})$
output S
[1] $S = \emptyset, W = \{k.\mathsf{m}'\}$
[2] while $W \neq \emptyset$
[3] remove $i.\mathsf{m}$ from W, mark $i.\mathsf{m}$ visited
[4] foreach $i.\mathsf{m} \triangleright j.\mathsf{m} \in \widehat{C}$ s.t. $i \triangleright j \in \widehat{B}_k$
[5] if $k \in \widehat{minB}_{i \triangleright j}$
[6] S += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[7] if $j.\mathsf{n}$ not visited then W += $j.\mathsf{n}$
[8] return S

procedure *AllRaces*$(\widehat{G}, \widehat{C})$
output R
[1] R = *DistributedRaces*$(\widehat{G}, \widehat{C})$
[2] while R changes
[3] foreach new race $\langle i, \mathsf{m}, \mathsf{m}' \rangle \in R$
[4] S_t = *Reach*$(i.\mathsf{m}, \widehat{G}, \widehat{C})$
[5] $S_{t'}$ = *Reach*$(i.\mathsf{m}', \widehat{G}, \widehat{C})$
[6] R += $\{\langle j, \mathsf{n}, \mathsf{n}' \rangle \mid (j.\mathsf{n}, ls) \in S_t \wedge (j.\mathsf{n}', ls') \in S_{t'} \wedge ls \cap ls' = \emptyset\}$
[7] return R

Fig. 7. Object race detection

eventual races with $j.\mathsf{n}$ will be discovered when the analysis descends into the boundary of k'.

5 Implementation

The object and call graph analyses, dominance analysis, and object race detection analysis are implemented in Java using Soot 2.2.3 [13] and Spark [5]. We performed whole-program analysis with the Sun JDK 1.4.1 libraries. All experiments were done on a MacBook Pro laptop with a 2GHz Intel Core i7 processor and 4GB of RAM. The implementation, which includes Soot and Spark, was run with a max heap size of 1400MB; however, all benchmarks ran within a memory footprint of 800MB. Native methods are handled by utilizing the models provided by Soot. Reflection is handled by manually specifying the dynamically loaded classes. Our underlying points-to analysis analyzes constructors object-sensitively in the style of [6]. As a result, the running times reported for points-to analysis are approximately twice the running times of Spark.

Table 1. Results

Program	#Meth	ObjRace						Chord		Time[sec]	
		Distributed			Owned						
		Race-free	Racy		Race-free	Racy		False	Real	Points-to	Race
			False	Real		False	Real				
tsp	3414	5	**2**	0	2	0	0	**2**	0	35	1
hedc	3749	8	**2**	14	1	0	3	**2**	2	40	6
sor	3403	4	**2**	0	0	0	0	0	0	35	1
SPECjbb	4640	19	0	19	69	0	8	**30**	16	50	4
weblech	4461	3	0	8	3	0	0	0	2	52	3
jdbm	4331	1	0	4	25	0	0	0	2	45	3
jdbf	3994	90	0	20	0	0	2	**2**	4	55	5
commons	3551	0	**1**	8	13	0	0	0	7	42	2
jtds	5044	64	0	87?	10	0	63?	oom	oom	61	86

Our suite consists of benchmarks used in previous work on concurrency [15,9,14]. Column **#Meth** in Table 1 gives the size of the benchmarks in terms of the number of methods (user and library) reachable by Spark. Benchmarks tsp through weblech are whole-programs, and jdbm through jtds are libraries. For the libraries, we converted the single-threaded harnesses from [9] to the multithreaded model described in Section 4.3.

We compared our analysis with Chord [9], the leading static race detector. Chord's data race report includes a field-based view and an object-based view of data races. The object-based view groups data races per abstract object (distinguished by allocation site as in our analysis) and for each abstract object, the view provides a set of read/write access pairs. We counted each abstract object reported in the object-based view as a racy object. We used reports available at http://berkeley.intel-research.net/mnaik/research/pldi06_results.html for tsp and hedc. We ran Chord 2.0 and generated reports for the rest of the benchmarks, except for jtds, for which Chord ran out of memory with a max heap size 2GB. Both Chord and our analysis suppress race reports due to constructors and methods called within constructors.

Column **ObjRace** in Table 1 shows the number of non-thread-local (according to the escape analysis from [11]) objects, reported as race-free or racy by our analysis. Column **Chord** shows the number of racy objects reported by Chord. Our analysis classifies objects as Distributed or Owned. An object j is classified as Distributed when the test at line 6 in *DistributedRaces* fires **true**; it is classified as Owned otherwise. Note that, in general, an abstract object may be classified as both Distributed and Owned. When race-free, a distributed object is typically protected by its own lock (e.g., all methods called on that object are declared synchronized). In general, although there are race-free distributed objects, distributed objects tend to be racy. For owned objects (column Owned), when race free, an owned object is most often protected by synchronization on its owner. We observed many cases when synchronized methods access internal owned objects and the owned objects stay protected by their owner's lock. Although there are racy owned objects, owned objects tend to be race-free.

The authors examined the objects reported as racy by both our analysis and Chord, and classified those object races as false-positive (columns False), and feasible (columns Real). All feasible races reported by Chord were reported by our analysis as well. Our analysis reports more feasible races than Chord. One reason why feasible object races are not reported by Chord, is that although there is an object race, there is no immediate data race on the object's instance fields. In the majority of cases, an object race leads to calls that change state on internal unsynchronized library objects. The object races are symptoms not only of potential data races deeper in the boundary of the object, but of higher-level concurrency bugs such as atomicity errors and atomic serializability errors. We believe that it is valuable to report all object races. Another reason why feasible object races are not reported by Chord may be that Chord's lock analysis is unsafe [9], while our analysis is safe. There is a large number of racy objects reported on `jtds`, and we were unable to confirm with certainty whether those races were false or feasible; however, `jtds` is undersynchronized and it appears that the majority of the reported races are feasible.

6 Related Work

Concurrency is a large and active area of research and we cannot include a complete listing of related works. Below, we focus on the work closest to ours.

Von Pruan and Gross introduce the concept of the object race [15]. Their object race detection is dynamic. In fact, a primary goal is to optimize dynamic lockset- based race detection [12]; the higher-level concept of object race entails fewer dynamic checks and therefore lower overhead. In later work, von Praun and Gross introduce the concept of the Object Use Graph (OUG) [16] which allows reasoning about the temporal relation of object accesses, and further reduces the amount of dynamic checks in the lockset-based detector. Despite its name, the OUG is unrelated to the object graph from ownership types [3] that we infer. Our analysis reasons about object races as well. However, our analysis is entirely static. Furthermore, although [15] and [16] make use of "ownership", their notion of ownership is very different from ours. They refer to thread ownership, not dominance-based object ownership as we do. Similarly to [16], Choi et al. use static analysis as well as dynamic happens-before analysis, to optimize a dynamic lockset-based data race detector [2].

Chord [9] is the most advanced static race detector. Our work is similar in its goal: we wanted to build an effective static object race detector for Java. It is different from Chord in several ways. First, it focuses on object races while Chord focuses on data races. Second, our analysis uses a different algorithmic approach: it relies on *dominance analysis* at its heart, while Chord relies on *context-sensitive points-to analysis*. Dominance analysis entails a cheaper object abstraction — objects are represented by allocation site — which may lead to better scalability of our analysis. Our experiments indicate that our analysis is effective and that it complements Chord.

Work by Vaziri et al. [14] is most closely related to ours. It explores a type system for data-centric synchronization, and as in our work, dominance-based

ownership plays an important role. An object is viewed as an atomic set of fields, and the lock of that object protects its fields as well as internal (owned) objects. Work by Boyapati et al. [1] explores dominance-based ownership for safe multithreaded programming as well. In [14] and [1] ownership is specified by the programmer and checked by the type system. In contrast, we infer ownership and object races automatically.

7 Conclusions

We have presented a novel static object race detection analysis. We have shown its effectiveness by implementing a prototype, applying it to several large multithreaded Java benchmarks and comparing its results to the results of the leading static race detector Chord.

References

1. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 211–230 (2002)
2. Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: Conference on Programming Language Design and Implementation, pp. 252–269 (2002)
3. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 48–64 (1998)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In: Symposium on Principles of Programming Languages, pp. 238–252 (1977)
5. Lhoták, O., Hendren, L.: Scaling Java Points-to Analysis Using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
6. Milanova, A.: Light context-sensitive points-to analysis for java. In: Workshop on Program Analysis for Software Tools and Engineering, pp. 25–30 (2007)
7. Milanova, A., Vitek, J.: Static Dominance Inference. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 211–227. Springer, Heidelberg (2011)
8. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Symposium on Principles of Programming Languages, pp. 327–338 (2007)
9. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Conference on Programming Language Design and Implementation, pp. 308–319 (2006)
10. Pratikakis, P., Foster, J., Hicks, M.: LOCKSMITH: context-sensitive correlation analysis for race detection. In: Conference on Programming Language Design and Implementation, pp. 320–331 (2006)
11. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 43–55 (October 2001)
12. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. In: Symposium on Operating Systems Principles, pp. 27–37 (1997)

13. Vallée-Rai, R., Gagnon, E.M., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
14. Vaziri, M., Tip, F., Dolby, J., Hammer, C., Vitek, J.: A Type System for Data-Centric Synchronization. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 304–328. Springer, Heidelberg (2010)
15. von Praun, C., Gross, T.: Object race detection. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 70–82 (2001)
16. von Praun, C., Gross, T.: Static conflict analysis for multithreaded object-oriented programs. In: Conference on Programming Language Design and Implementation, pp. 115–128 (2003)