Effective Static Race Detection for Java

Mayer Naik, Alex Aiken, John Whaley

presented by Matt McGill
Part I
Introduction to Chord and Preliminaries
Chord

- Chord is a **static** (data) race detection tool for Java
Chord

- Problems addressed
  - Precision - minimizing false positives
  - Scalability - handling large programs
  - Synchronization Idioms
  - Open Programs
  - Counterexamples - making debugging less painful
Chord

- Synchronization Idioms supported
  - *lexically-scoped* lock-based synchronization
  - *fork/join* synchronization
    - relying partially on user-supplied annotations
  - *wait/notify* synchronization
    - reduces to lock-based synchronization in Java
Chord - Overview

Input - OriginalPairs

reachable-pairs computation → aliasing-pairs computation → escaping-pairs computation → unlocked-pairs computation

stages analyses

alias analysis → thread-escape analysis

call-graph construction → lock analysis
Chord - Overview

What are alias analysis/call-graph construction?
Preliminaries: Data Flow Analyses

- Call-graph construction
  - For each method, which other methods call it?
  - Doesn’t sound so bad? Undecidable.

- Alias analysis
  - For each abstract memory location, determine the set of L-value expressions that alias it at each program point. Also undecidable.

- We calculate approximations
Preliminaries: Data Flow Analyses

- Call-graph construction
  - For each method, which other methods may call it?
- Alias analysis
  - For each abstract memory location, determine the set of L-value expressions that may alias it at each program point.
- There are many techniques for solving these problems with varying degrees of precision
Intraprocedural Analyses

- Intraprocedural analyses examine individual procedures using control flow graphs (CFG)

```java
List map(F f, List lst) {
    List nlst = new ArrayList();
    for (int i=0; i<lst.size(); i++) {
        Object arg = lst.get(i);
        Object result = f.apply(arg);
        nlst.add(result);
    }
    return nlst;
}
```
Intraprocedural Analysis

- Data flow analysis represents the flow of data between program points as a system of equations.
- Provided the equations meet certain criteria, the equations can be solved using an iterative method.
- For call-graph construction/alias analysis to be remotely useful, we must identify and utilize relationships among procedures.
Interprocedural Analyses

- Interprocedural analysis techniques can be categorized according to:
  - flow sensitivity - does the analysis make use of control flow within procedures?
  - context sensitivity - does the analysis compute different results for methods based on context information (typically call sites)?
Chord

• Chord is *flow-insensitive*

• Chord is *context-sensitive*
  
  • Not in the traditional way!

  • Chord instead uses *k-object sensitivity*, a relatively recent notion of context tailored to the needs of object-oriented programs
Object Sensitivity

class X { ... }
class Y {
    X f;
    void set(X x) {
        this.f = x;
    }
    public static void
    main(String[] args) {
        X x1 = new X(); //s_1
        X x2 = new X(); //s_2
        Y y1 = new Y(); //s_3
        Y y2 = new Y(); //s_4
        y1.set(x1);
        y2.set(x2);
    }
}

- Flow-insensitive, context-insensitive alias analysis might infer:
Object Sensitivity

- For an allocation site $s_i$ in a method $m$, an object name $o_{ij}$ is created for every allocation site $s_j$ that creates an object that $m$ could be called on.

- Each object name is equated with a context.

- Each method $m$ is analyzed separately for each context in which it could be called.
Object Sensitivity

- Object names are used as contexts in which to analyze methods

```java
class W { ... }
class X { ... }
class Y {
    void foo(X x) {
        W w = new W(); //s
        w.bar(x);
    }
    public static void main(String[] args) {
        X x1 = new X(); //s
        X x2 = new X(); //s
        Y y1 = new Y(); //s
        Y y2 = new Y(); //s
        y1.foo(x1);
        y2.foo(x2);
    }
}
```

Contexts

- bar contexts
- foo contexts

- O₁
- O₂
- O₃
- O₄
- O₅
- O₆
k-Object Sensitivity

- We just described k-object sensitivity for $k=1$
- For higher $k$, increase the maximum length of the allocation-site chain (i.e. the number of subscripts on your object names, or the number of levels of indirection you’re following)
- Common OO design patterns such as Visitor harm precision for $k=1$
- Authors found $k=3$ to be suitable balance for detecting data races with useful precision
Efficient k-Object Sensitivity

- Publicly available k-object sensitivity tools ran out of memory for \( k = 1 \)
- Authors expressed k-object sensitive alias analysis (and other analyses, and race detection algorithm) in Datalog, a logic programming language similar to Prolog
- Used Datalog implementation bddbddbb based on Binary Decision Diagrams to achieve scalability
Part II

Four Stages of the Race Detection Algorithm
Overview

Input - OriginalPairs

reachable-pairs computation → aliasing-pairs computation → escaping-pairs computation → unlocked-pairs computation

stages analyses

alias analysis → thread-escape analysis

call-graph construction → lock analysis
Closing Open Programs

1. Declare a local variable of each type allowed as argument or return value of any method in the interfaces to be checked

2. Assign an object to each reference variable

3. Invoke each method in external interface on each legal combination of local reference variables
Closing Open Programs

- Problems with closing open programs:
  - missing *callees*
  - missing *callers*

- The method described is unsound (but in practice that’s OK!)
  - Question: how does this affect usefulness the authors’ stated goal of adding flow sensitivity at a later point?
public class A {
    int f;
    public A() { this.f=0; //f_w'}
    private int rd() { return this.f; //f_r}
    private int wr(int x) { this.f=x; //f_w
        return x; } 
    public int get() { return this.rd(); }
    public synchronized int inc() {
        int t = this.rd() + (new A()).wr(1);
        return this.wr(t);
    }
}

public class A {
    int f;
    static public void main(String[] args) {
        A a;
        if (*) a=new A();
        if (*) a.get();
        if (*) a.inc();
    }
}
OriginalPairs

• Intuitively, we start with every pair of memory accesses to the same field/array element that aren’t immediately ruled out by Java’s typing rules

\[
\begin{align*}
\text{(instance field)} & \quad f \in F \\
\text{(static field)} & \quad g \in G \\
\text{(memory access)} & \quad e \in E = E_f \cup E_g \cup E_a \\
\text{(get/set instance field)} & \quad F_r, F_w \subseteq E_f \times F \\
\text{(get/set static field)} & \quad G_r, G_w \subseteq E_g \times G \\
\text{(get/set array element)} & \quad A_r, A_w \subseteq E_a
\end{align*}
\]
public class A {
    int f;
    static public void
    main(String[] args) {
        A a;
        if (*) a=new A();
        if (*) a.get();
        if (*) a.inc();
    }
    public A() { this.f=0; //f_w’ }
    private int rd() { return this.f; //f_r }
    private int wr(int x) { this.f=x; //f_w
        return x; }
    public int get() { return this.rd(); }  
    public synchronized int inc() {
        int t = this.rd() + (new A()).wr(1);
        return this.wr(t);
    }
}

• OriginalPairs=((f_r,f_w),(f_w,f_r),(f_r’),(f_w’),(f_r,f_w’),(f_w,f_r’))
ReachablePairs

- Prune all pairs of memory accesses \((e_1, e_2)\) that cannot take place in different threads
- We’ll cover this stage in detail, look at examples for the others
ReachablePairs

- We start with:
  - a function from call sites (and accesses) to containing methods
    
    \[ M : (I \cup E) \rightarrow M \]
  
  - A function from call sites and contexts to pairs of callee methods and contexts
    
    \[ \mathcal{T} : (I \times C) \rightarrow \mathcal{P}(M \times C) \]
  
  - A set of call sites that spawn threads
    
    \[ I_{fork} = I_{ext} \cup I_{int} \]
ReachablePairs

- Next we define thread insensitive and sensitive relations from call sites/contexts to methods/contexts directly...

  \[(i, c') \implies (m, c) \triangleq (i, c') \rightarrow (m, c) \land i \notin I_{fork}\]

  \[(i, c') \rightarrow (m, c) \triangleq (m, c) \in T(i, c')\]

- And do something similar to transitive closure to obtain relations from call sites/contexts to methods/contexts *indirectly*:

  \[(i, c'') \implies^{n+1} (m, c) \triangleq\]
  \[
  \exists m', i', c' : (i, c'') \implies^n (m', c') \\
  \land \mathcal{M}(i') = m' \land (i', c') \implies (m, c)\]

  \[(i, c'') \rightarrow^{n+1} (m, c) \triangleq\]
  \[
  \exists m', i', c' : (i, c'') \rightarrow^n (m', c') \\
  \land \mathcal{M}(i') = m' \land (i', c') \rightarrow (m, c)\]
**ReachablePairs**

- Next, compute the “roots,” thread-spawning call sites reachable from main:

\[
\mathcal{R} \subseteq (I \times C) \\
\mathcal{R} = \{ (i, c) \mid i \in I_{fork} \land \exists i' : M(i') = main \land (i', c) \rightarrow^* (M(i), c) \}
\]
ReachablePairs

- Finally, retain pairs only if each access is reachable, without switching threads, from a root:

\[
\text{ReachablePairs}' = \{ (e_1, c_1, e_2, c_2) | \\
(e_1, e_2) \in \text{OriginalPairs} \\
\land \ \exists (i_1, c'_1, i_2, c'_2) \in R^2 : \\
\quad (i_1, c'_1) \implies^* (\mathcal{M}(e_1), c_1) \\
\quad \land \ (i_2, c'_2) \implies^* (\mathcal{M}(e_2), c_2) \} \\
\text{ReachablePairs} = \{ (e_1, e_2) | \\
\exists c_1, c_2 : (e_1, c_1, e_2, c_2) \in \text{ReachablePairs}' \} \]
ReachablePairs

public class A {
    int f;
    static public void
    main(String[] args) {
        A a;
        if (*) a=new A(); //caε
        if (*) a.get();
        if (*) a.inc();
    }
    public A() { this.f=0; //fw’}
    private int rd() { return this.f; //fr}
    private int wr(int x) { this.f=x; //fw
        return x; }
    public int get() { return this.rd(); }
    public synchronized int inc() {
        int t = this.rd()
            + (new A()).wr(1); // cba
        return this.wr(t);
    }
}

- **ReachablePairs**=((fr,fw),(fw,fw),(fr,fw’),(fw’,fw’),(fw,fw’)) Why?

- **ReachablePairs’**=
  ((fr,caε,fw,caε), (fr,caε,fw,cbα) (fw,caε,fr,caε) (fw,caε,fw,caε) (fw,cbα,fw,caε)(fw,cbα,fw,cbα) (fw,cbα,fw,cbα))
AliasingPairs

• Intuitively: remove all pairs that alias analysis shows cannot access the same abstract object
  • as is commonly done, all references to static field references are treated as if performed on a single dummy object
  • individual elements in arrays are merged
AliasingPairs

public class A {
    int f;
    static public void
    main(String[] args) {
        A a;
        if (*) a=new A(); //caε
        if (*) a.get();
        if (*) a.inc();
    }
    public A() { this.f=0; //fw’}
    private int rd() { return this.f; //fr}
    private int wr(int x) { this.f=x; //fw
        return x; }
    public int get() { return this.rd(); }
    public synchronized int inc() {
        int t = this.rd() + (new A()).wr(1); // cba
        return this.wr(t);
    }
}

• AliasingPairs=((fr,fw),(fw,fw))

• AliasingPairs’=
  ((fr,cac,fw,cac), (fr,cac,fw,cba) (fw,cac,fw,cac) (fw,cac,fw,cba) (fw,cba,fw,cac)(fw,cba,fw,cba))
EscapingPairs

- Intuitively: prune pairs that do not access thread-shared data
- Implemented in 3(!!) lines of Datalog
**EscapingPairs**

public class A {
    int f;
    static public void main(String[] args) {
        A a;
        if (*) a=new A(); //caε
        if (*) a.get();
        if (*) a.inc();
    }
    public A() { this.f=0; //fw’}
    private int rd() { return this.f; //fr}
    private int wr(int x) { this.f=x; //fw
        return x; }
    public int get() { return this.rd(); }
    public synchronized int inc() {
        int t = this.rd()
        + (new A()).wr(1); //cba
        return this.wr(t);
    }
}

• EscapingPairs=((fr,fw),(fw,fw))

• EscapingPairs’=
  ((fr,Cae,fw,Cae),(fw,Cae,fw,Cae),(fw,Cba,fw,Cba))
UnlockedPairs

- Intuitively: prune pairs of accesses that are always protected by a common lock
- Can be thought of as a static approximation of the Lockset technique
- To do this soundly requires a must-alias analysis, but they substitute their may-alias analysis
- Authors reported that no false negatives resulted from this simplification during their testing
UnlockedPairs

public class A {
    int f;
    static public void main(String[] args) {
        A a;
        if (*) a=new A(); //caε
        if (*) a.get();
        if (*) a.inc();
    }
    public A() { this.f=0; //fw’}
    private int rd() { return this.f; //fr}
    private int wr(int x) { this.f=x; //fw
        return x; }
    public int get() { return this.rd(); }
    public synchronized int inc() {
        int t = this.rd()
        + (new A()).wr(1); //cba
        return this.wr(t);
    }
}

• UnlockedPairs=((fr,fw),(fw,fw))

• UnlockedPairs’=
    ((fr,caε,fw,caε),(fw,caε,fw,caε))
Reporting Races

- Paths through context-sensitive call graph are used to provide a “stack trace” for each potential data race
- Races are classified by field and by (abstract) object
- Identification of the allocation site for objects involved in races should aid debugging
Part III
Evaluation
Sources of Unsoundness

- may-analysis instead of must-analysis in fourth stage
- method of closing open programs does not model all possibilities
- potential races in constructors/initializers are ignored
- no attempt to deal with reflection or dynamic class loading
Performance

- runtime was typically around one minute
- Apache Derby (~640KLOC) took 26 minutes
- memory consumption was unreported; test machine had 4GB
  - was Derby pushing the complexity limits of the BDD solver, or did that analysis suffer from disk paging performance hits?
Observations

• Number of reported problems was very reasonable given program sizes

• escape analysis had negligible impact in about half of the analyses

• the ability to check parts of programs at an arbitrary level of granularity makes Chord a very scalable tool