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

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

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
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 call it?
- Doesn’t sound so bad?
- 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

Intra-procedural Analyses

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

Intra-procedural 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 object-oriented programs

Object Sensitivity

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

```java
class X { ... }
class Y {
  X f;
  void set(X x) {
    this.f = 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.set(x1);
    y2.set(x2);
  }
}
```

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

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 bddbddb based on Binary Decision Diagrams to achieve scalability
Part II
Four Stages of the Race Detection Algorithm

Overview

Input - OriginalPairs

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?

Running Example

public class A {
  int f;
  public A() { this.f=0; //f }
  private int rd() { return this.f; //f }
  private int wr(int x) { this.f=x; //f 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
• 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
  \[
  \begin{aligned}
  (\text{instance field}) & \quad f \in F \\
  (\text{static field}) & \quad g \in G \\
  (\text{memory access}) & \quad e \in E = E_f \cup E_g \cup E_a \\
  (\text{get/set instance field}) & \quad F_f \subset F \\
  (\text{get/set static field}) & \quad G_g \subset G \\
  (\text{get/set array element}) & \quad A_a \subset E_a
  \end{aligned}
  \]
OriginalPairs

```java
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; }
  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);
  }
}
```

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
  - A function from call sites and contexts to pairs of callee methods and contexts
  - A set of call sites that spawn threads

ReachablePairs

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

ReachablePairs

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

public class ReachablePairs {
  int f;
  static public void main(String[] args) {
    A a;
    if (*) a.rd(); //fr
    if (*) a.wr();
    if (*) a.loc();
  }
}

• ReachablePairs=((fr,fw),(fw,fw),(fr,fw'),(fw',fw'),(fw, fw'))
• ReachablePairs'=
  ((fr,ca!,fw,ca!), (fr,ca!,fw,cba), (fw,ca!,fw,ca!), (fw, cba, fw,cba), (fw, cba, fw,cba))

AliasingPairs

public class AliasingPairs {
  int f;
  static public void main(String[] args) {
    A a;
    if (*) a.rd(); //fr
    if (*) a.wr();
    if (*) a.loc();
  }
}

• AliasingPairs=((fr,fw),(fw,fw))
• AliasingPairs'=
  ((fr,ca!,fw,ca!), (fr,ca!,fw,cba)(fw,ca!,fw,ca!), (fw, cba, fw,cba))

EscapingPairs

public class EscapingPairs {
  int f;
  static public void main(String[] args) {
    A a;
    if (*) a.rd(); //fr
    if (*) a.wr();
    if (*) a.loc();
  }
}

• EscapingPairs=((fr,fw),(fw,fw))
• EscapingPairs'=
  ((fr,ca!,fw,ca!), (fw,ca!,fw,ca!), (fw,cba,fw,cba))

AliasingPairs

• Intuitively: remove all pairs that alias analysis shows cannot access the same abstract object
• Can be thought of as a may-alias analysis
• Authors reported that no false negatives resulted from this simplification during their testing

EscapingPairs

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

Unlockeds

• 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

```java
public class A {
  int f;
  static public void main(String[] args) {
    A a;
    if (*) a=new A(); // c
    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); // c ba
    return this.wr(t);
  }
}
```

---

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

---

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

---

Part III
Evaluation

- 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

---

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