EFFECTIVE RACE DETECTION FOR JAVA

Mayur Naik, Alex Aiken, and John Whaley @ Stanford University
Published in PLDI’06
Race Condition

“two threads...access the same memory location...at least one thread is write...”
Previous race detection approaches

→ Dynamic detector →
  |→ happens-before relations
  |→ the lockset algorithm (Eraser’s algorithm)
  |→ combination of the two above

→ Static detector →
  |→ flow insensitive type-base systems
  |→ flow-sensitive static versions of the lockset algorithm
  |→ path-sensitive model checkers
This paper uses *k-object sensitivity*

- The goal of *points-to analysis* for Java is to determine the set of objects pointed to by a reference variable or a reference object field.

- *Object sensitivity*, a new form of context sensitivity for flow-insensitive points-to analysis for Java.

- **Key idea**
  - to analyze a method separately for each of the object names that represent run-time objects on which this method may be invoked.
  - to ensure flexibility and practicality, a parameterization framework is proposed that allows analysis designers to control the tradeoffs between cost and precision in the object-sensitive analysis.
Design Objectives

- Precision

- Scalability

- Practicability
  - Synchronization idioms
  - Open programs (e.g., a library, a device driver)
  - Counterexamples
Design Overview

- Using 4 stages of pruning to make it precise
- Applying 4 static analyses
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}

public A() {
    f = 0; }

public int get() {
    return rd(); }

public sync int inc() {
    int t = rd() + (new A()).wr(1); 
    return wr(t); }

private int rd() {
    return f; }

private int wr(int x) {
    f = x;
    return x; }
Initial computation

• **Original-pairs computation**
  • The initial over-approximation of the set of unordered pairs of memory accesses potentially involved in a race

• **How:**
  • Both threads access the same instance field or the same static field or array elements
  • At least one is a write
Example

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}

private int rd() {
    return f;
}

private int wr(int x) {
    f = x;
    return x;
}

public A() {
    f = 0;
}

public int get() {
    return rd();
}

public sync int inc() {
    int t = rd() + (new A()).wr(1);
    return wr(t);
}

private int rd() {
    return f;
}

private int wr(int x) {
    f = x;
    return x;
}
```
The 4 stages (of pruning)

- *OriginalPairs* are just an over approximating
- 4 stages of pruning to increase precision
  - Reachable-pairs computation
  - Aliasing-pairs computation
  - Escaping-pairs computation
  - Unlocked-pairs computation
1st stage: reachable-pairs computation

- Purpose: to prune *OriginalPairs*
- How: using the fact that a pair of access involved in a race only if each access is reachable from a thread-spawning call site that is itself reachable from the synthesized *main*. 
Example: Reachable Pairs

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}

private int rd() {
    return f;
}
private int wr(int x) {
    f = x;
    return x;
}

public A() {
    f = 0;
}

public int get() {
    return rd();
}

public sync int inc() {
    int t = rd() + (new A()).wr(1);
    return wr(t);
}

private int rd() {
    return f;
}
private int wr(int x) {
    f = x;
    return x;
}
```
Example: Two Object-Sensitive Contexts

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}

private int rd() {
    return f;
}

private int wr(int x) {
    f = x;
    return x;
}
```

```java
public A() {
    f = 0; }

public int get() {
    return rd(); }

public sync int inc() {
    int t = rd() + (new A()).wr(1);
    return wr(t); }

private int rd() {
    return f; }

private int wr(int x) {
    f = x;
    return x; }
```

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}

private int rd() {
    return f;
}

private int wr(int x) {
    f = x;
    return x;
}
```
Example: 1st Context

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}

public A() {
    f = 0;
}

public int get() {
    return rd();
}

public sync int inc() {
    int t = rd() + (new A()).wr(1);
    return wr(t);
}

private int rd() {
    return f;
}

private int wr(int x) {
    f = x;
    return x;
}
```

Example: 2nd Context

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}
```

```java
public A() {
    f = 0;
}
```

```java
public int get() {
    return rd();
}
```

```java
public sync int inc() {
    int t = rd() + (new A()).wr(1);
    return wr(t);
}
```

```java
private int rd() {
    return f;
}
```

```java
private int rd() {
    return f;
}
```

```java
private int wr(int x) {
    f = x;
    return x;
}
```

```java
private int wr(int x) {
    f = x;
    return x;
}
```
Example: Reachable Pairs

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}
```

```java
public A() {
    f = 0;
}
```

```java
public int get() {
    return rd();
}
```

```java
public sync int inc() {
    int t = rd() + (new A()).wr(1);
    return wr(t);
}
```

```java
private int rd() {
    return f;
}
```

```java
private int wr(int x) {
    f = x;
    return x;
}
```
2\textsuperscript{nd} stage: aliasing-pairs computation

- Purpose: to prune \textit{ReachablePairs}
- How: using the fact that a pair of access involved in a race provided they access the same location
Example: Aliasing Pairs

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}

class A {
    int f = 0;
    public int get() {
        return rd();
    }
    public sync int inc() {
        int t = rd() + (new A()).wr(1);
        return wr(t);
    }
    private int rd() {
        return f;
    }
    private int wr(int x) {
        f = x;
        return x;
    }
}
```
3rd stage: escaping-pairs computation

- Purpose: to prune \textit{AliasingPairs}
- How: using the face that a pair of access involved in a race if they access thread-shared data
Example: Escaping Pairs

```java
class A {
    int f = 0;
    public A() {
        f = 0;
    }
    public int get() {
        return rd();
    }
    public sync int inc() {
        int t = rd() + (new A()).wr(1);
        return wr(t);
    }
}

class B {
    A a;
    public B() {
        a = new A();
    }
    public void main() {
        a.get();
        a.inc();
    }
}
```
4th stage: unlocked-pairs computation

- Purpose: to prune EscapingPairs
- How: 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
Example: Unlocked Pairs

```java
static public void main() {
    A a;
    a = new A();
    a.get();
    a.inc();
}

public A() {
    f = 0;
}

dpublic int get() {
    return rd();
}

dpublic sync int inc() {
    int t = rd() + (new A()).wr(1);
    return wr(t);
}

dprivate int rd() {
    return f;
}

dprivate int wr(int x) {
    f = x;
    return x;
}
```

# Experimental Results

<table>
<thead>
<tr>
<th>classes</th>
<th>LOC</th>
<th>brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsp</td>
<td>370</td>
<td>TSP implementation from ETH</td>
</tr>
<tr>
<td>hedi</td>
<td>422</td>
<td>web crawler from ETH</td>
</tr>
<tr>
<td>ftp</td>
<td>493</td>
<td>Apache FTP Server</td>
</tr>
<tr>
<td>vect 1.1</td>
<td>19</td>
<td>JDK 1.1 java.util.Vector</td>
</tr>
<tr>
<td>htbl 1.1</td>
<td>21</td>
<td>JDK 1.1 java.util.Hashtable</td>
</tr>
<tr>
<td>htbl 1.4</td>
<td>366</td>
<td>JDK 1.4 java.util.Hashtable</td>
</tr>
<tr>
<td>vect 1.4</td>
<td>370</td>
<td>JDK 1.4 java.util.Vector</td>
</tr>
<tr>
<td>jdbm</td>
<td>461</td>
<td>transactional persistence engine</td>
</tr>
<tr>
<td>jdbf</td>
<td>465</td>
<td>object-relational mapping system</td>
</tr>
<tr>
<td>pool</td>
<td>388</td>
<td>Apache Commons Pool</td>
</tr>
<tr>
<td>jtds</td>
<td>553</td>
<td>JDBC driver</td>
</tr>
<tr>
<td>derby</td>
<td>1,746</td>
<td>Apache Derby, an RDBMS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>annotations</th>
<th>pairs of accesses</th>
<th>races</th>
<th>bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsp</td>
<td>1m03s</td>
<td>1</td>
<td>6398784</td>
<td>1115</td>
</tr>
<tr>
<td>hedi</td>
<td>1m10s</td>
<td>0</td>
<td>6417868</td>
<td>3052</td>
</tr>
<tr>
<td>ftp</td>
<td>1m17s</td>
<td>1</td>
<td>6662532</td>
<td>712</td>
</tr>
<tr>
<td>vect 1.1</td>
<td>0m08s</td>
<td>1</td>
<td>32216</td>
<td>338</td>
</tr>
<tr>
<td>htbl 1.1</td>
<td>0m07s</td>
<td>1</td>
<td>32564</td>
<td>220</td>
</tr>
<tr>
<td>htbl 1.4</td>
<td>1m04s</td>
<td>1</td>
<td>6174952</td>
<td>214</td>
</tr>
<tr>
<td>vect 1.4</td>
<td>1m02s</td>
<td>1</td>
<td>6178650</td>
<td>1139</td>
</tr>
<tr>
<td>jdbm</td>
<td>1m33s</td>
<td>1</td>
<td>11189853</td>
<td>33443</td>
</tr>
<tr>
<td>jdbf</td>
<td>1m42s</td>
<td>1</td>
<td>12632041</td>
<td>95410</td>
</tr>
<tr>
<td>pool</td>
<td>5m29s</td>
<td>1</td>
<td>9741876</td>
<td>959</td>
</tr>
<tr>
<td>jtds</td>
<td>3m23s</td>
<td>1</td>
<td>28773219</td>
<td>386241</td>
</tr>
<tr>
<td>derby</td>
<td>26m03s</td>
<td>1</td>
<td>57681326</td>
<td>847236</td>
</tr>
</tbody>
</table>
Note on Atomocity Checking

• “The motivation behind atomocity 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.”
Conclusion

• A static race detector
  • Precision
  • Scalability

• 4 Stage of pruning and analysis
  • Each stage is to increase precision
Critique

- A well written paper
- Offers a good summary on the previous works done in race detections
- Some acknowledged unsoundness
  - May-alias analysis
  - Limited analysis on open programs
  - Some program features are not considered (reflection, dynamic class loading, JNI, etc.)
- Flow insensitivities contribute to all the false positives.
Reference

• The running example is taken from *Effective Static Race Detection for Java*, *PLID’06 presentation talk* by Mayur Naik