Effective Static Race Detection for Java

Mayur Naik
Alex Aiken
Stanford University
What is a Race?

A condition in a shared-memory, multi-threaded program in which:

- same memory location accessed by different threads simultaneously (without holding a common lock)
- at least one of the accesses is a write
Why is Race Detection Important?

- Particularly insidious concurrency bug
  - Triggered non-deterministically
  - No fail-fast behavior even in safe languages like Java

- Fundamental in concurrency theory and practice
  - Lies at heart of many concurrency problems
    - atomicity checking, deadlock detection, ...
  - Today’s concurrent programs riddled with races
    - “… most Java programs are so rife with concurrency bugs that they work only ‘by accident’.”
    - *Java Concurrency in Practice*, Addison-Wesley, 2006
Our Result

- 412 bugs in mature Java programs comprising 1.5 MLOC
  - Many fixed within a week by developers
Our Race Detection Approach

all pairs

racing pairs
Challenges

Same location accessed by different threads simultaneously without common lock held

• Handle multiple aspects
  – Same location accessed
  – … by different threads
  – … simultaneously

• Correlate locks with locations they guard
  – … without common lock held

• Precision
  – Showed precise *may alias analysis* is central (PLDI’06)
  – low false-positive rate (20%)

• Soundness
  – Devised *conditional must not alias analysis* (POPL’07)
  – Circumvents *must alias analysis*
Our Race Detection Approach

- All pairs
- Aliasing pairs
- Shared pairs
- Parallel pairs
- Unlocked pairs
- Racing pairs

Same location accessed by different threads simultaneously without common lock held

False Pos. Rate: 20%
Alias Analysis for Race Detection

// Thread 1:  // Thread 2:
sync (l1) {
  ...
e1.f ...
}  sync (l2) {
  ...
e2.f ...
}

- Field f is race-free if:

  e1 and e2 never refer to the same value

MUST NOT ALIAS(e1, e2)
k-Object-Sensitive May Alias Analysis

• Range of work of may analysis [Milanova et al. ISSTA’03]

• Idea #1: Context-insensitive analysis
  – Abstract value = set of allocation sites

    foo() { bar() {
    ... e1.f ...    ... e2.f ... 
    }
  }

  \neg MAY-ALIAS(e1, e2) if Sites(e1) \cap Sites(e2) = \emptyset

• Idea #2: Context-sensitive analysis (k-CFA)
  – Context (k=1) = call site

    foo() { bar() {
    e1.baz();    e2.baz();
    }
  }

  Analyze function baz in two contexts
k-Object-Sensitive Analysis: Our Contributions

- No scalable implementations for even $k = 1$

- Insights:
  - Symbolic representation of relations
    - BDDs [Whaley-Lam PLDI’04, Lhotak-Hendren PLDI’04]
  - Demand-driven race detection algorithm
    - Begin with $k = 1$ for all allocation sites
    - Increment $k$ only for those involved in races

- Allow scalability to $k = 7$
Our Race Detection Approach

- All pairs
- Aliasing pairs
- Shared pairs
- Parallel pairs
- Unlocked pairs
- Racing pairs

Same location accessed by different threads simultaneously without common lock held
Alias Analysis for Race Detection

- Field f is race-free if:

  e1 and e2 never refer to the same value

  ¬ MAY-ALIAS(e1, e2)

  OR

  l1 and l2 always refer to the same value

  MUST-ALIAS(l1, l2)
Must Alias Analysis

• Small body of work
  – Much harder problem than may alias analysis

• Impediment to many previous race detection approaches
  – Folk wisdom: Static race detection is intractable

*Insight: Must alias analysis not necessary for race detection!*
New Idea: Conditional Must Not Alias Analysis

// Thread 1: sync (l1) { ... e1.f ... }

// Thread 2: sync (l2) { ... e2.f ... }

• Field f is race-free if:

Whenever l1 and l2 refer to different values, e1 and e2 also refer to different values

MUST-NOT-ALIAS(l1, l2) => MUST-NOT-ALIAS(e1, e2)
Example

```plaintext
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
x1 = a[*];
x2 = a[*];
sync (?) {
    x1.g.f = ...;
}
sync (?) {
    x2.g.f = ...;
}
```
Easy Case: Coarse-grained Locking

```java
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
x1 = a[0];
x2 = a[0];
sync (a) {
    x1.g.f = …;
    x2.g.f = …;
}
```

Field $f$ is race-free if:

$$\text{MUST-NOT-ALIAS}(a_1.a_2) \neq \text{MUST-NOT-ALIAS}(x_1.e.g, x_2.g)$$
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
x1 = a[ * ];
x2 = a[ * ];
sync (?) {
sync (?) {
    x1.g.f = ...;
    x2.g.f = ...;
}
}
Easy Case: Fine-grained Locking

Field \( f \) is race-free if:

\[
\text{MUST-NOT-ALIAS}(x_1.g, x_2.g) \land \text{MUST-NOT-ALIAS}(e_1.g, e_2.g)
\]
Example

\[
a = \text{new } h0[N];
\]
\[
\text{for } (i = 0; i < N; i++) \{
    a[i] = \text{new } h1;
    a[i].g = \text{new } h2;
\}
\]
\[
x1 = a[*]; \quad x2 = a[*];
\]
\[
\text{sync (?) \{} \quad \text{sync (?) \{} \\
    x1.g.f = ...; \quad x2.g.f = ...;
\}\}
\]
Hard Case: Medium-grained Locking

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}

x1 = a[*];
x2 = a[*];
sync (x1) {
    sync (x2) {
        x1.g.f = ...;
        x2.g.f = ...
    }
}
```

Field f is race-free if:

\[
\text{MUST-NOT-ALIAS}(x1, x2) = \text{MUST-NOT-ALIAS}(x1.g2, x2.g)
\]
Disjoint Reachability

In every execution, if:

- from distinct $h1$ values
- we can reach (via 1 or more fields)
- only distinct $h2$ values

then $\{h2\} \subseteq DR(\{h1\})$

Note: Values abstracted by sets of allocation sites
Conditional Must Not Alias Analysis using Disjoint Reachability

Field $f$ is race-free if:

\[
\text{MUST-NOT-ALIAS}(e_1, e_2) \subseteq \text{DR}(\text{MUST-NOT-ALIAS}(l_1, l_2))
\]

- $e_1$ reachable from $l_1$ and $e_2$ reachable from $l_2$
Hard Case: Medium-grained Locking

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
x1 = a[*];
x2 = a[*];
sync (x1) {
    sync (x2) {
        x1.g.f = ...;
        x2.g.f = ...;
    }
}
```

Field \( f \) is race-free if:

- \( \text{true} \geq \text{true} \)
- \( \text{true} \geq \text{reachables from } a \)
- \( \text{true} \geq \text{reachables from } x1 \)
- \( \text{true} \geq \text{reachables from } x2 \)
Experience with Chord

• Experimented with 12 multi-threaded Java programs
  – smaller programs used in previous work
  – larger, mature and widely-used open-source programs
  – whole programs and libraries

• Tool output and developer discussions available at:

• Programs being used by other researchers in race detection
## Benchmarks

<table>
<thead>
<tr>
<th>classes</th>
<th>KLOC</th>
<th>description</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>vect1.1</td>
<td>19</td>
<td>JDK 1.1 java.util.Vector</td>
<td>0m28s</td>
</tr>
<tr>
<td>htbl1.1</td>
<td>21</td>
<td>JDK 1.1 java.util.Hashtable</td>
<td>0m27s</td>
</tr>
<tr>
<td>htbl1.4</td>
<td>366</td>
<td>JDK 1.4 java.util.Hashtable</td>
<td>2m04s</td>
</tr>
<tr>
<td>vect1.4</td>
<td>370</td>
<td>JDK 1.4 java.util.Vector</td>
<td>2m02s</td>
</tr>
<tr>
<td>tsp</td>
<td>370</td>
<td>Traveling Salesman Problem</td>
<td>3m03s</td>
</tr>
<tr>
<td>hedc</td>
<td>422</td>
<td>Web crawler</td>
<td>9m10s</td>
</tr>
<tr>
<td>ftp</td>
<td>493</td>
<td>Apache FTP server</td>
<td>11m17s</td>
</tr>
<tr>
<td>pool</td>
<td>388</td>
<td>Apache object pooling library</td>
<td>10m29s</td>
</tr>
<tr>
<td>jdbm</td>
<td>461</td>
<td>Transaction manager</td>
<td>9m33s</td>
</tr>
<tr>
<td>jdbf</td>
<td>465</td>
<td>O/R mapping system</td>
<td>9m42s</td>
</tr>
<tr>
<td>jtds</td>
<td>553</td>
<td>JDBC driver</td>
<td>10m23s</td>
</tr>
<tr>
<td>derby</td>
<td>1746</td>
<td>Apache RDBMS</td>
<td>36m03s</td>
</tr>
</tbody>
</table>
Pairs Retained After Each Stage (Log scale)
## Classification of Unlocked Pairs

<table>
<thead>
<tr>
<th></th>
<th>harmful</th>
<th>benign</th>
<th>false</th>
<th># bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>vect1.1</td>
<td>5</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>htbl1.1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>htbl1.4</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vect1.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tsp</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>hedc</td>
<td>170</td>
<td>0</td>
<td>41</td>
<td>6</td>
</tr>
<tr>
<td>ftp</td>
<td>212</td>
<td>3</td>
<td>43</td>
<td>32</td>
</tr>
<tr>
<td>pool</td>
<td>105</td>
<td>10</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>jdbm</td>
<td>91</td>
<td>0</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>jdbf</td>
<td>130</td>
<td>0</td>
<td>34</td>
<td>18</td>
</tr>
<tr>
<td>jtds</td>
<td>34</td>
<td>14</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>derby</td>
<td>1018</td>
<td>0</td>
<td>78</td>
<td>319</td>
</tr>
</tbody>
</table>
Developer Feedback

• 16 bugs in jTDS
  – Before: “As far as we know, there are no concurrency issues in jTDS …”
  – After: “It is probably the case that the whole synchronization approach in jTDS should be revised from scratch …”

• 17 bugs in Apache Commons Pool
  – “Thanks to an audit by Mayur Naik many potential synchronization issues have been fixed” -- *Release notes for Commons Pool 1.3*

• 319 bugs in Apache Derby
  – “This looks like *very* valuable information and I for one appreciate you using Derby … Could this tool be run on a regular basis? It is likely that new races could get introduced as new code is submitted …”
Related Work

• Static (compile-time) race detection
  – Need to approximate multiple aspects
  – Need to perform must alias analysis
  – Sacrifice precision, soundness, scalability

• Dynamic (run-time) race detection
  – Current state of the art
  – Inherently unsound
  – Cannot analyze libraries

• Shape Analysis
  – much more expensive than disjoint reachability
Summary of Contributions

• Precise race detection (PLDI’06)
  – Key idea: *k*-object-sensitive may alias analysis
  – Important client for may alias analyses

• Sound race detection (POPL’07)
  – Key idea: Conditional must not alias analysis
  – Has applications besides race detection

• Effective race detection
  – 412 bugs in mature Java programs comprising 1.5 MLOC
  – Many fixed within a week by developers
The End

Chord: A Static Race Detector for Java