Introduction

● Problem
  – Multi-threaded programming introduces concurrency errors:
    • Deadlocks
    • Starvation
    • *Data Races*
  – Two types of Data Races
    • Low-level
    • *High-level*
  – These errors are hard to find
Introduction

• Solution
  – An algorithm for the automated detection of *High-level Data Races*
    • Totally automated (similar to Eraser)
    • Raises warnings during execution
    • Neither sound, nor complete
Low-level Data Races

• The traditional data race
  – Occurs on a single variable
• Can be avoided by protecting shared variables with a lock
• There are several techniques for their detection
  – Dynamic analysis
    • Lock set-based (Eraser)
    • Happens before-based
  – Static analysis (Chord)
High-level Data Races

• Basic definition
  – “A high-level data race can occur when two concurrent threads access a set V of shared variables, which should be accessed atomically, but at least one of the threads does not access the variables in V atomically”
High-level Data Race Example

class Coord {
    double x, y;  // X and Y should be accessed atomically

    public Coord(double px, double py) { x = px; y = py; }

    synchronized double getX() { return x; }
    synchronized double getY() { return y; }
    synchronized Coord getXY() { return new Coord(x, y); }

    synchronized void setX(double px) { x = px; }
    synchronized void setY(double py) { y = py; }
    synchronized void setXY(Coord c) { x = c.x; y = c.y; }
}
High-level Data Race Example

Assume we have two threads tr and tw, a shared Coord object c.

If tw executes

```java
void writeOperation(){
    c.setX(0);
    c.setY(0);
}
```

and tr executes

```java
void readOperation(){
    c.getXY(0);
}
```

concurrently.
High-level Data Race Example

Assuming $x=1$ and $y=1$:

- **tw** sets $x=0$
- **tw** sets $y=0$
- **tr** reads $x=0$ and $y=0$

Safe Situation
High-level Data Race Example

Assuming $x=1$ and $y=1$:

- **tw** sets $x=0$
- **tw** sets $y=0$
- **c**: Coord
  - `setX(0)`
  - `setY(0)`
- **tr**: Thread
  - `getXY()`
  - `tr reads x=0 and y=1`

**High Level Data Race!**
High-level Data Races

- If only getXY, setXY, and the constructor are used by any thread, the pair is treated atomically by all accessing threads.
- The versatility offered by the other accessor (get/set) methods is dangerous.
- The difficulty in analyzing inconsistencies lies in the wish to still allow such partial accesses to sets of fields.
Refined Definition

• The previous definition yields false positives

• Refinement of Basic Definition
  – “A high-level data race can occur when two concurrent threads access a set $V$ of shared variables, which should be accessed atomically, but at least one of the threads accesses $V$ partially several times such that those partial accesses diverge.”
    • Partial accesses diverge when they do not form a chain.
### Refined Definition Examples

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
<th>Thread t3</th>
<th>Thread t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1 = new Coord(1,2);</td>
<td>x2 = c.getX();</td>
<td>x3 = c.getX();</td>
<td>x4 = c.getX();</td>
</tr>
<tr>
<td>c.setXY(d1);</td>
<td>use(x2);</td>
<td>y3 = c.getY();</td>
<td>use(x4);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>use(x3, y3);</td>
<td></td>
</tr>
</tbody>
</table>

- Consider four threads t1, t2, t3 and t4 sharing a Coord c.
- The threads use local variables $x_i$ and $y_i$ of type double and $d_i$ of type Coord
**Refined Definition Examples**

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
<th>Thread t3</th>
<th>Thread t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1 = new Coord(1,2);</td>
<td>x2 = c.getX();</td>
<td>x3 = c.getX();</td>
<td>x4 = c.getX();</td>
</tr>
<tr>
<td>c.setXY(d1);</td>
<td>use(x2);</td>
<td>y3 = c.getY();</td>
<td>use(x4);</td>
</tr>
<tr>
<td>use(x2);</td>
<td></td>
<td>use(x3,y3);</td>
<td>d4 = c.getXY();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x4 = d4.getX();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y4 = d4.getY();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>use(x4,y4);</td>
</tr>
</tbody>
</table>

- The situation between t1 and t2 is completely safe
- As long as t2 does not use y as well
  - The read of x in t2 can be seen as a partial read
  - May be interpreted as reading \{x, y\} and discarding y.
Refined Definition Examples

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
<th>Thread t3</th>
<th>Thread t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1 = new Coord(1,2); c.setXY(d1);</td>
<td>x2 = c.getX(); use(x2);</td>
<td>x3 = c.getX(); y3 = c.getY(); use(x3, y3);</td>
<td>x4 = c.getX(); d4 = c.getXY(); x4 = d4.getX(); y4 = d4.getY(); use(x4, y4);</td>
</tr>
</tbody>
</table>

- The situation between t1 and t3 is similar to the previous example
### Refined Definition Examples

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
<th>Thread t3</th>
<th>Thread t4</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d1 = new Coord(1,2);</code></td>
<td><code>x2 = c.getX();</code></td>
<td><code>x3 = c.getX();</code></td>
<td><code>x4 = c.getX();</code></td>
</tr>
<tr>
<td><code>c.setXY(d1);</code></td>
<td><code>use(x2);</code></td>
<td><code>y3 = c.getY();</code></td>
<td><code>d4 = c.getXY();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>use(x3, y3);</code></td>
<td><code>x4 = d4.getX();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>y4 = d4.getY();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>use(x4, y4);</code></td>
</tr>
</tbody>
</table>

- The situation between t1 and t4 should also be regarded as safe.
- The variable sets accessed by t4 form a chain
  - The set of variables in the first operation of t4 is a subset of the set of variables in its second operation
High-level Data Races: Formal Definition

• View consistency
  – Lifts the common notion of a data race on a single shared variable to a higher level
  – Covers sets of shared variables and their uses
  – Assumes that the specification of what fields have to be treated atomically is extracted by program analysis
Views

- Views express what fields are guarded by a lock
- A lock guards a shared field if it is held during an access to that field.
  - The same lock may guard several shared fields
Views

- Let $I$ be the set of object instances generated by a particular run of a Java program.
- Then $F$ is the set of all fields of all instances in $I$.
- A view $v \in P(F)$ is a subset of $F$. 
Views

- Let $l$ be a lock, $t$ a thread, and $B(t, l)$ the set of all synchronized blocks using lock $l$ executed by thread $t$.
- For $b \in B(t, l)$, a view generated by $t$ with respect to $l$, is defined as the set of fields accessed in $b$ by $t$.
- The set of generated views $V(t) \subseteq P(F)$ of a thread $t$ is the set of all views $v$ generated by $t$. 
# Views: Examples

### Thread t1
```
d1 = new Coord(1,2);
c.setXY(d1);
```

### Thread t2
```
x2 = c.getX();
use(x2);
```

### Thread t3
```
x3 = c.getX();
y3 = c.getY();
use(x3,y3);
```

### Thread t4
```
x4 = c.getX();
y4 = d4.getY();
use(x4,y4);
```

- t1 generates view \( v_1 = \{x, y\} \)
- t2 generates view \( v_2 = \{x\} \)
- t3 generates two views \( V(t_3) = \{ \{x\}, \{y\} \} \)
- t4 also generates two views \( V(t_4) = \{ \{x\}, \{x, y\} \} \)
Maximal Views

- A view $v_m$ generated by a thread $t$ which is maximal with respect to set inclusion in $V(t)$.
- $M(t)$ is the set of maximal views in $t$.
- Example:
  - If $V(t)=\{ \{x,y\},\{y\},\{x\} \}$
    - Partial order of $V(t)=\{ \{x\},\{y\},\{x,y\} \}$
    - $M(t)=\{ \{x,y\} \}$
Overlapping Views

- Only two views which have fields in common can be responsible of a conflict

\[
\text{overlap}(t, v') \equiv \{ v' \cap v \mid (v \in V(t)) \land (v \cap v' = \emptyset) \}\]

- Example:
  - If \( V(t1) = \{ \{y\}, \{x\} \} \) and \( M(t2) = \{ \{x,y\} \} \)
  - \( \text{overlap}(t1, \{x,y\}) = \{ \{x\}, \{y\} \} \) since
    - \( \{x, y\} \cap \{x\} = \{x\} \) and
    - \( \{x, y\} \cap \{y\} = \{y\} \)
Compatible Views

- Check if all overlapping views of $t$ with $v_m$ form a chain.

$$\text{compatible}(t, v_m) \text{ iff }$$
\[ \forall v_1, v_2 \in \text{overlap}(t, v_m) \left[ v_1 \subseteq v_2 \lor v_2 \subseteq v_1 \right] \]

- Example:
  - If $\text{overlap}(t1, M(t2)) = \text{overlap}(t1, \{x,y\}) = \{ \{x\}, \{y\} \}$
    - $t1$ is not compatible with $t2$ since
      - $\{x\} \not\subseteq \{y\}$ nor
      - $\{y\} \not\subseteq \{x\}$
View Consistency

- Mutual compatibility between all threads

\[ \forall t_1 \neq t_2, \, v_m \in M(t_1) \, [\text{compatible}(t_2, v_m)] \]
View Consistency vs Eraser

• The Eraser algorithm
  – Considers the set of locks protecting a single variable

• This idea is turned upside down
  – The variable set associated to a lock is now of interest
Formal Definition: Examples

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
<th>Thread t3</th>
<th>Thread t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1 = new Coord(1,2); c.setXY(d1);</td>
<td>x2 = c.getX(); use(x2);</td>
<td>x3 = c.getX(); y3 = c.getY(); use(x3,y3);</td>
<td>x4 = c.getX(); use(x4); d4 = c.getXY(); x4 = d4.getX(); y4 = d4.getY(); use(x4,y4);</td>
</tr>
</tbody>
</table>

V(t1)={ {x,y} }  
M(t1)={ {x,y} }  

V(t2)= { {x} }  
M(t2)= { {x} }  

- overlap(t2, M(t1)) = { {x} }  
- compatible(t2, M(t1))?  
  - Yes
### Formal Definition: Examples

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
<th>Thread t3</th>
<th>Thread t4</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d1 = new Coord(1,2);</code></td>
<td><code>x2 = c.getX();</code></td>
<td><code>x3 = c.getX();</code></td>
<td><code>x4 = c.getX();</code></td>
</tr>
<tr>
<td><code>c.setXY(d1);</code></td>
<td><code>use(x2);</code></td>
<td><code>y3 = c.getY();</code></td>
<td><code>use(x4);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>d4 = c.getXY();</code></td>
<td><code>d4 = c.getXY();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x4 = d4.getX();</code></td>
<td><code>x4 = d4.getX();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>y4 = d4.getY();</code></td>
<td><code>y4 = d4.getY();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>use(x4,y4);</code></td>
<td></td>
</tr>
</tbody>
</table>

V(t1)=\{ \{x,y\} \}  
M(t1)=\{ \{x,y\} \}  

V(t3)=\{ \{x\}, \{y\} \}  
M(t3)=\{ \{x\}, \{y\} \}  

- `\text{overlap}(t3, M(t1)) = \{ \{x\}, \{y\} \} `  
- `\text{compatible}(t3, M(t1))?`  
  - No
### Formal Definition: Examples

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
<th>Thread t3</th>
<th>Thread t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1 = new Coord(1,2); c.setXY(d1);</td>
<td>x2 = c.getX(); use(x2);</td>
<td>x3 = c.getX(); y3 = c.getY(); use(x3,y3);</td>
<td>x4 = c.getX(); use(x4); d4 = c.getXY(); x4 = d4.getX(); y4 = d4.getY(); use(x4,y4);</td>
</tr>
</tbody>
</table>

V(t1) = \{ \{x,y\} \}  
M(t1) = \{ \{x,y\} \}

V(t4) = \{ \{x\}, \{x,y\} \}  
M(t4) = \{ \{x,y\} \}

- overlap(t4, M(t1)) = \{ \{x\}, \{x,y\} \}
- compatible(t4, M(t1))?  
  - Yes
Detecting High-level Data Races using View Consistency

- This approach tries to infer shared fields which must be accessed atomically
  - An atomic access to a set of shared fields is an indication of atomicity
  - Views are used to detect the most likely candidates, or maximal views

- View consistency is used to detect violations of accesses to these sets
  - A violation indicates a high-level data race
Soundness and Completeness

- This approach is not sound.
  - An inconsistency may not automatically imply a fault in the software
  - Sets of fields may be used atomically even if there is no requirement for atomic access

- This approach is not complete
  - This is possible if all views are consistent, but locking is still insufficient
  - A particular run through the program may not reveal the inconsistent views

- The increased probability of detecting errors strongly outbalances this
Implementation

● An algorithm for extracting the views generated by all threads
  – Dynamic analysis
  – Carried out using the JPaX run-time verification tool
    • Produces an instrumented version of the program
    • Observes the execution of the program to determine the correctness of examined properties
Experiments

- Experiments indicate that
  - Experienced programmers intuitively adhere to the principle of view consistency
  - Violations can be found, but are not very common
  - Some optimizations produce warnings that constitute no error
  - The definition of view consistency still needs some refinement

<table>
<thead>
<tr>
<th>Application</th>
<th>Size [LOC]</th>
<th>Number of classes</th>
<th>Run time [s], uninstrumented</th>
<th>Run time [s], instrumented</th>
<th>Log size [MB]</th>
<th>Warnings issued</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator</td>
<td>500</td>
<td>5</td>
<td>16.7</td>
<td>17.5</td>
<td>1.9</td>
<td>2</td>
</tr>
<tr>
<td>SOR</td>
<td>250</td>
<td>3</td>
<td>0.8</td>
<td>343.2</td>
<td>123.5</td>
<td>0</td>
</tr>
<tr>
<td>TSP, very small run (4 cities)</td>
<td>700</td>
<td>4</td>
<td>0.6</td>
<td>1.8</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>TSP, larger run (10 cities)</td>
<td>7000</td>
<td>82</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1</td>
</tr>
<tr>
<td>NASA’s K9 Rover controller</td>
<td>7000</td>
<td>82</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1</td>
</tr>
</tbody>
</table>