AtomRace: Data Race and Atomicity Violation Detector and Healer

Zdeněk Letko
Tomáš Vojnar
Bohuslav Křena

Brno University of Technology, Czech Republic (EU)
Agenda

- AtomRace approach
- Detection abilities
- Problem exhibition technique
- Experimental results
What is a data race?

- A data race occurs when two concurrent threads access a shared variable and:
  - at least one access is a write and
  - the threads use no explicit mechanism to prevent the accesses from being simultaneous.
Lockset-based approach

• Instructions of interest:
  – Memory access.
  – Lock operations.

• Principle:
  – For each variable maintains a set of candidate locks.

• Disadvantages:
  – Support only synchronization based on locks.
Happens-before approach

• Instructions of interest:
  – Memory access.
  – Synchronization (posting, receiving).

• Principle:
  – Sync events split execution into segments.
  – Segments that are not related by the happens-before relation are simultaneous.

• Disadvantage:
  – Support only predefined synchronization.
Self-healing requirements

- Support for any kind of synchronization.
- No false alarms.
- Dynamic $\rightarrow$ minimal performance degradation.
- Never mind if race which does not occur within the current execution is not reported.
- Report problem before it happens.
AtomRace approach

• Instructions of interest:
  – Memory access.

• Every access to a variable is enclosed by meta instructions.

```java
beforeAccessEvent(p, loc6);
p = null;
afterAccessEvent(p, loc6);
```

Primitive atomic section related to p
Data race detection

- A race is detected if two primitive atomic sections related to the same variable instance overlap and at least one access is for writing.

- Reports *simultaneous accesses*.
An example

Thread 1

if (p != null) {
    p.doSomeWork();
}

Thread 2

// set p to null
p = null;
An example

Thread 1

if (p != null) {
    p.doSomeWork();
}

Thread 2

p = null;

Problem detected
An example

Thread 1

if (p != null){
  p.doSomeWork();
}

Thread 2

p = null;
Atomicity Violation

Thread 1

if (p != null){
    p.doSomeWork();
}

More general atomic section related to p
More general atomic section

- Atomic section related to a variable:
  - If a thread $t$ starts executing within the atomic section over a variable $v$, no other thread should access $v$ in an unserializable way before the thread $t$ reaches the end point of the atomic section.

- Serializability:
  - Each atomic region is executed atomically.

Zdeněk Letko, PADTAD 2008
More general atomic section

- Serializability expressed by the set of access modes that are \textbf{only allowed}:

\[ A \subseteq \{ \text{read}, \text{write} \} \]

- Associated with each end point of each atomic section.
Atomicity violation detection

- Within an atomic section gather a set of suspected accesses to a variable:

```java
beforeAccessEvent(p, loc3)
if (p != null)
{
    p.doSomeWork();
    afterAccessEvent(p, loc4)  A = \{\textit{read}\}
}
atomExitEvent(p, loc5)     A = \{\textit{read, write}\}
```
Atomicity violation detection

- Within an atomic section gather a set of suspected accesses to a variable:

  \[
  \text{beforeAccessEvent}(p, \text{loc3}) \\
  \text{if (p} \neq \text{null)}\{ \\
  \quad p = \text{null}; \\
  \quad p.\text{doSomeWork}(); \\
  \quad \text{afterAccessEvent}\(p, \text{loc4}\) \quad A = \{\text{read}\} \\
  \}\] \\
  \text{atomExitEvent}(p, \text{loc5}) \quad A = \{\text{read, write}\}
Atomicity violation detection

- Within an atomic section gather a set of suspected accesses to a variable:

```java
beforeAccessEvent(p, loc3)
if (p != null){
    p = null;
    p.doSomeWork();
    (Thread 2, write, loc6)
    afterAccessEvent(p, loc4)
    A = {read}
}
atomExitEvent(p, loc5)
A = {read, write}
```
When not to detect - Java

- Safe to skip detection:
  - Variable declared as `final`
    - Cannot change the value.
  - Variable declared as `volatile`
    - Designed to tolerate simultaneous accesses.
When not to detect - Java

• Avoid false alarms:
  – Variable set within a `<clinit>` method
    • Initializes static variables – executed on demand.
    • Synchronization ensured by the JVM.
  – Example:
    ```java
    static int p = 10;
    ...
    beforeAccessEvent(var, loc9);
    System.out.println("Value of p is:" + p);
    afterAccessEvent(var, loc9);
    ...
    ```
Special situations

• Overlapping atomic sections
  - Multiple threads execute multiple atomic sections over a single variable.

Example:

Thread 1

\[
\text{local} = p;
\]

Thread 2

\[
\text{local2} = p + 10; \quad \text{(Corrected)}
\]
Special situations

- Circular atomic sections
  - Section starts and ends at the same location within a loop.

Example:

```java
for (int i=0; i<5; i++){
    local += p;
}
```
Special situations

• Recursive atomic sections
  − Atomic section is entered recursively.

Example:

```c
int method(int param){
    if (param < limit)
        p = p + method(param+1);
    return param;
}
```
Obtaining atomicity

- Pattern based static analysis
  - Load and store bug patterns
  - Test and use bug patterns

- Access Interleaving Invariants
Problem manifestation

- How to make data races and atomicity violations manifest – so they can be detected?
Problem manifestation

- How to make data races and atomicity violations manifest – so they can be detected?

- Get use of noise injection:
  - ConTest tool (IBM Haifa Research Labs)
    - Synchronization focused
    - Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Noise makers need to know where to be silent - producing schedules that find bugs. ISOLA. 2006.
Noise injection

- Noise injected inside atomic sections.
- The noise is put inside atomic sections using before/afterAccessEvents → makes them longer.

- Noise injection policy:
  - Random
  - Variable focused
  - Location focused
AtomRace with Noise Injection

- 4 processors, TIDorb Java (1400+ classes)

<table>
<thead>
<tr>
<th></th>
<th>Conflicts detected</th>
<th>Coverage of problematic variables</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only AtomRace</td>
<td>1.58</td>
<td>0.37</td>
<td>5.76 sec</td>
</tr>
<tr>
<td>Random</td>
<td>24.19</td>
<td>2.27</td>
<td>106.17 sec</td>
</tr>
<tr>
<td>Location based</td>
<td>11.38</td>
<td>1.71</td>
<td>7.40 sec</td>
</tr>
<tr>
<td>Variable based</td>
<td>261.38</td>
<td>3.63</td>
<td>34.44 sec</td>
</tr>
</tbody>
</table>

Average in 100 runs, noise 50ms, noise frequency 50%
Conclusions

- AtomRace
  - Supports any kind of synchronization.
  - Meets the requirements for dynamic self-healing.
- Problem manifestation technique
  - Increases the efficiency of dynamic detection technique – better if noise injected wisely.
- Future work
  - Performance, obtaining more general atomicity, healing the first occurrence of atomicity violation.
Thank you.
Proposed Architecture

Data race and atomicity violation detection and healing tool

Analysis Engine

Problem indications

External Repository with Atomicity Information

Event information

Execution Monitoring

Healing Logic

Managed Application

Obtaining Atomicity

Monitored event

Healing action

Bytecode

Atomicity

Zdeněk Letko, PADTAD 2008
## AI-invariants: unserializable interleavings

<table>
<thead>
<tr>
<th>Interleaving scenario</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read&lt;sub&gt;local&lt;/sub&gt;</td>
<td>The interleaving write makes the two reads have different views of the same memory locations.</td>
</tr>
<tr>
<td>write&lt;sub&gt;remote&lt;/sub&gt;</td>
<td>The local read does not get the local result it expectcts.</td>
</tr>
<tr>
<td>read&lt;sub&gt;local&lt;/sub&gt;</td>
<td>Intermediate result that assumed to be invisible to other threads is read by a remote access.</td>
</tr>
<tr>
<td>write&lt;sub&gt;local&lt;/sub&gt;</td>
<td>The local variable relies on a value from the preceding local read that is then overwritten by the remote write.</td>
</tr>
</tbody>
</table>