Identifying Ad-hoc Synchronization for Enhanced Race Detection

IPDPS – 20 April, 2010
Ali Jannesari / Walter F. Tichy
Motivation

- Data races (unsynchronized accesses to share variables) are a common defect in parallel programs.
- They are difficult to find.
- Race detectors are impractical
  - They produce thousands to millions of false warnings.
  - Programmers are overwhelmed by false positives.

Why false positives?
- Ad-hoc, programmer-defined synchronizations
- Unknown synchronization libraries
- Detectors cannot reason about these, causing many false positives

Contribution: how to handle user-defined synchronization and unknown synchronization libraries, reducing false positives.
What is a Data Race?

- Two or more concurrent accesses to a shared location, at least one of them a write.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X = 0 )</td>
<td></td>
</tr>
<tr>
<td>( X++ )</td>
<td>( T = X )</td>
</tr>
</tbody>
</table>
Example – Data Race

- First Interleaving:
  1. Thread 1: X=0
  2. Thread 2: T=X
  3. Thread 1: X++

- Second Interleaving:
  1. Thread 1: X=0
  2. Thread 1: X++
  3. Thread 2: T=X

- T==0 or T==1?
How Can Data Races be Prevented?

- Explicit synchronization between threads:
  - Locks
  - Critical Sections
  - Barriers
  - Mutexes
  - Semaphores
  - Monitors
  - Events (signal/wait)
  - Etc.

```
Thread 1
Lock(m)
X=0
X++
Unlock(m)

Thread 2
Lock(m)
T=X
Unlock(m)
```
Detection Approaches

- **Static**: perform a compile-time analysis of the code, reporting potential races.
- **Dynamic**: use tracing mechanism to detect whether a particular execution of a program actually exhibits data-races.

  - The program must be instrumented with additional instructions to monitor shared variables and synchronization operations.
  - Every shared variable has a shadow cell in which the race detector stores additional information.
Dynamic Data Race Detection

- Dynamic Data Race Detection
  - Lockset analysis
  - Happens-before analysis
  - Hybrids (combining Lockset and Happens-before)
Lockset Analysis

- Observe all instances where a shared variable is accessed by a thread.
- Check whether the shared variable is always protected by the same lock.
- If variable isn’t protected, issue a warning.
- The lockset for a variable is initially set to all locks occurring in program.
- Whenever a variable is accessed, remove all locks from the variable’s lockset that are not currently protecting the variable.
- When the lockset is empty, issue a warning.
## Lockset Analysis

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Lockset(_v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock( m1 ); v = v + 1;</td>
<td>Lock( m1 ); v = v + 1;</td>
<td>{m1, m2, ...}</td>
</tr>
<tr>
<td>Unlock( m1 ); v = v + 1;</td>
<td>Unlock( m1 ); v = v + 1;</td>
<td>{m1}</td>
</tr>
</tbody>
</table>

v = v + 1;
Lockset - False Positives

- The lockset algorithm will produce a false alarm in the following simple case:
  - Not able to detect signal-wait operation

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=0</td>
<td></td>
</tr>
<tr>
<td>X++</td>
<td></td>
</tr>
<tr>
<td><strong>Signal(CV)</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Wait(CV)</strong></td>
</tr>
<tr>
<td></td>
<td>T=X</td>
</tr>
</tbody>
</table>
Happens-Before Relation

- Based on Lamport’s Clock
- Let event \( a \) be in thread A and event \( b \) be in thread B.
  - If event \( a \) and event \( b \) are paired synchronization operations, construct a happens-before edge between them:
    - E.g. If \( a = \text{unlock}(\mu) \) and \( b = \text{lock}(\mu) \) then \( a \xrightarrow{\text{hb}} b \) (\( a \) happens-before \( b \))
- Shared accesses \( i \) and \( j \) are concurrent
  - if neither \( i \xrightarrow{\text{hb}} j \) nor \( j \xrightarrow{\text{hb}} i \) holds.
- Data races between threads are possible if accesses to shared variables are not ordered by happens-before.
Happens-Before - Example 1

Happens-before analysis will **eliminate** the false alarm in the following simple case:

<table>
<thead>
<tr>
<th><strong>Thread 1</strong></th>
<th><strong>Thread 2</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>X=0</td>
<td></td>
</tr>
<tr>
<td>X++</td>
<td></td>
</tr>
<tr>
<td><strong>Signal(CV)</strong></td>
<td><strong>Wait(CV)</strong></td>
</tr>
<tr>
<td></td>
<td>T=X</td>
</tr>
</tbody>
</table>
Happens-Before - Example 2

Thread 1
lock(mu);
\[ v = v + 1; \]
unlock(mu);

Thread 2
lock(mu);
\[ v = v + 1; \]
unlock(mu);

The arrows represent happens-before. The events represent an actual execution of the two threads.
Helgrind\(^*\)

- Efficient hybrid dynamic race detector
  - Introduces a new hybrid algorithm based on lockset algorithm and happens-before analysis
  - Does runtime analysis and uses code and semantic information
- Different memory state machines for
  - short-running applications (during development - unit test)
    - More sensitive, but produces more false positives
  - long-running applications (integration testing)
    - Less sensitive, might miss a race on first iteration, but not on second
- Automatically handling of synchronization bug patterns related to condition variables without any source code annotation
  - Lost signal detector
  - Spurious wake-up detection
Ad-hoc (User-defined) Synchronization

- Synchronization constructs implemented by user for performance reasons
  - High level synchronizations (e.g. task queues)
  - Spinning read loop instead of a library wait operation

/*Initially \texttt{FLAG} is zero */

Thread 1

- \texttt{DATA++}
- \texttt{FLAG = 1}
- ...

Thread 2

- ...
- \texttt{while(FLAG == 0)}
- --do nothing
- \texttt{DATA--}

- Ad-hoc synchronizations are widely used
  - 12 - 31 in SPLASH-2 and 32 - 329 in PARSEC 2.0
Ad-hoc Synchronization

- Source of false positives
  - Apparent races (e.g. DATA)
  - Synchronization races (e.g. FLAG)
  - Detectors should identify and suppress them

- We developed a dynamic method to detect ad-hoc synchronization
  - Automatically without any user action
  - Capable of identifying synchronization primitives of unknown libraries
    - Eliminates false races (apparent and synchronization races) caused by unknown synchronization primitives of a library
    - No need to upgrade the detector for a new library
Common Pattern

- Spinning read loop (spin-lock) is a common pattern for ad-hoc synchronizations
  - Happens-before relation induced by spin-lock synchronization

Thread 1

do_before(X)

Set `CONDITION` to TRUE

... 

... 

Counterpart write

Thread 2

... 

while(!`CONDITION`){
    /* do_nothing() */
}

`do_after(X)`

Spinning read loop
Common Pattern

- Implementation of different synchronization primitives in libraries follows the same pattern as in spinning read loop
  - e.g. implementation of **Barrier()**:

  ```
  ... 
  Lock(L) 
  counter++ 
  Unlock(L) 
  while(!counter!=NUMBER_THREADS){
    /* do_nothing() */
  } 
  ... 
  ```
Detecting Ad-hoc Synchronizations

- General dynamic approach
  - **Instrumentation** phase and
  - **Runtime** phase

- **Instrumentation phase (code/semantic analysis)**
  - Search the binary code to find all loops
    - Control flow analysis on the fly
    - Consider small loops (3 to 7 basic blocks)
  - Detect the spinning read loop based on the following criteria:
    - The loop condition involves at least one load instruction from memory
    - The value of loop condition is **not changed** inside the loop
  - Instrument the loop and mark the variables that affect the value of the loop condition to be treated specially.
Detecting Ad-hoc Synchronizations

- **Runtime phase**
  - Data dependency analysis
    - Monitor all write/read accesses
    - Identify the write/read dependency
      - Between the variables of instrumented spinning loop condition and those in counterpart write
    - Establish a happens-before relation between corresponding parts

---

 Thread 1

`do_before(X)`

Set `CONDITION` to `TRUE`

...  

Counterpart write

 Thread 2

...  

`while(!CONDITION){`  

`/* do_nothing() */`

`}`

`do_after(X)`

Spinning read loop

---

Data dependency
Detecting Unknown Synchronization Primitives

- Synchronization operations are ultimately implemented by spinning read loops.
- Identify unknown synchronization operations if based on spinning read loops.
- **If this works, then we actually get a universal race detector**
  - Not limited to synchronization primitives of a particular library
  - General approach to identify synchronization operations
    - Information about libraries can be removed entirely from the detector
Implementation

- We implement the presented approach into our race detector **Helgrind**
- **Helgrind**
  - A hybrid dynamic race detector
  - Combines lockset algorithm and happens-before analysis
  - It is open source and built on top of Valgrind (a binary instrumentation tool)
Experiments & Evaluation

- The approach is evaluated on different benchmarks
  - data-race-test – a test suite framework for race detectors
  - PARSEC 2.0 Benchmarks

- All experiments were conducted on:
  - 2 * 1,86 GHz Xeon E5320 Quadcores, 8 GB RAM
  - OS: Linux (Ubuntu 8.10.2)

- New features in Helgrind+
  - Reduces the number of false positives due to ad-hoc synchronizations and unknown libraries dramatically
Test Suite – data-race-test

- 120 different test cases (2-16 Threads)
  - Test cases are racy or race-free programs (using Pthread)
    - Includes difficult cases
  - Spinning read loop detection of up to 7 basic blocks
    - 24 false positives and one false negative are removed
  - Removing information about Pthread library (unknown library)
    - Only one false positive more

<table>
<thead>
<tr>
<th>Tools</th>
<th>False alarms</th>
<th>Missed races</th>
<th>Failed cases</th>
<th>Correctly analyzed cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helgrind⁺ lib</td>
<td>32</td>
<td>8</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>Helgrind⁺ lib+spin(7)</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>105</td>
</tr>
<tr>
<td>Helgrind⁺ nolib+spin(7)</td>
<td>9</td>
<td>7</td>
<td>16</td>
<td>104</td>
</tr>
<tr>
<td>DRD</td>
<td>13</td>
<td>20</td>
<td>33</td>
<td>87</td>
</tr>
</tbody>
</table>
Test Suite – data-race-test

- Best result achieved with seven basic blocks using spinning read loop detection as a complementary method
- In most cases spinning read loops contain more than 3 basic blocks
  - Loop conditions use templates and complex function calls

<table>
<thead>
<tr>
<th>Tools</th>
<th>False alarms</th>
<th>Missed races</th>
<th>Failed cases</th>
<th>Correctly analysed cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helgrind+ lib+spin(3)</td>
<td>24</td>
<td>7</td>
<td>31</td>
<td>89</td>
</tr>
<tr>
<td>Helgrind+ lib+spin(6)</td>
<td>23</td>
<td>7</td>
<td>30</td>
<td>90</td>
</tr>
<tr>
<td>Helgrind+ lib+spin(7)</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>105</td>
</tr>
<tr>
<td>Helgrind+ lib+spin(8)</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>105</td>
</tr>
</tbody>
</table>
# PARSEC 2.0

<table>
<thead>
<tr>
<th>Program</th>
<th>Parallelization model</th>
<th>LOC</th>
<th>Synchronisation primitives</th>
<th>Ad-hoc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>CVs</td>
<td>Locks</td>
</tr>
<tr>
<td>blackscholes</td>
<td>POSIX</td>
<td>812</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>swaptions</td>
<td>POSIX</td>
<td>4,029</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>POSIX</td>
<td>3,689</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>canneal</td>
<td>POSIX</td>
<td>29,31</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>freqmine</td>
<td>OpenMP</td>
<td>10,279</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>vips</td>
<td>GLIB</td>
<td>1,255</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>bodytrack</td>
<td>POSIX</td>
<td>9,735</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>facesim</td>
<td>POSIX</td>
<td>1,391</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ferret</td>
<td>POSIX</td>
<td>2,706</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>x264</td>
<td>POSIX</td>
<td>1,494</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dedup</td>
<td>POSIX</td>
<td>3,228</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>streamcluster</td>
<td>POSIX</td>
<td>40,393</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>raytrace</td>
<td>POSIX</td>
<td>13,302</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Programs without Ad-hoc Synchronizations

- No false positives for first 4 programs
- In case of using the unknown library OpenMP only 2 false positives remain

<table>
<thead>
<tr>
<th>Program</th>
<th>Para. model</th>
<th>LOC</th>
<th>Racy Contexts</th>
<th></th>
<th></th>
<th></th>
<th>DRD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Helgrind⁺ lib</td>
<td>Helgrind⁺ lib+spin</td>
<td>Helgrind⁺ nolib+spin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blackscholes</td>
<td>POSIX</td>
<td>812</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>swaptions</td>
<td>POSIX</td>
<td>4,029</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>POSIX</td>
<td>3,689</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>canneal</td>
<td>POSIX</td>
<td>29,31</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>freqmine</td>
<td>OpenMP</td>
<td>10,279</td>
<td>153.4</td>
<td>2</td>
<td>2</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
In 5 out of 8 programs false positives are completely eliminated

<table>
<thead>
<tr>
<th>Program</th>
<th>Para. model</th>
<th>LOC</th>
<th>Racy Contexts</th>
<th>DRD</th>
</tr>
</thead>
<tbody>
<tr>
<td>vips</td>
<td>GLIB</td>
<td>1,255</td>
<td>Helgrind+lib</td>
<td>Helgrind+lib+spin</td>
</tr>
<tr>
<td>bodytrack</td>
<td>POSIX</td>
<td>9,735</td>
<td>50.8</td>
<td>0</td>
</tr>
<tr>
<td>facesim</td>
<td>POSIX</td>
<td>1,391</td>
<td>36.8</td>
<td>3.6</td>
</tr>
<tr>
<td>ferret</td>
<td>POSIX</td>
<td>2,706</td>
<td>113.8</td>
<td>0</td>
</tr>
<tr>
<td>x264</td>
<td>POSIX</td>
<td>1,494</td>
<td>111</td>
<td>2</td>
</tr>
<tr>
<td>dedup</td>
<td>POSIX</td>
<td>3,228</td>
<td>1100</td>
<td>19</td>
</tr>
<tr>
<td>streamcluster</td>
<td>POSIX</td>
<td>40,393</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>raytrace</td>
<td>POSIX</td>
<td>13,302</td>
<td>106.4</td>
<td>0</td>
</tr>
</tbody>
</table>
Programs with Ad-hoc Synchronizations

- 3 programs produce false positives (2 to 19 warnings)
  - Function pointers for condition evaluation and obscure implementation of task queue (do not match the spin patterns)

<table>
<thead>
<tr>
<th>Program</th>
<th>Para. model</th>
<th>LOC</th>
<th>Racy Contexts</th>
<th>DRD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Helgrind+lib</td>
<td>Helgrind+lib+spin</td>
</tr>
<tr>
<td>vips</td>
<td>GLIB</td>
<td>1,255</td>
<td>50.8</td>
<td>0</td>
</tr>
<tr>
<td>bodytrack</td>
<td>POSIX</td>
<td>9,735</td>
<td>36.8</td>
<td>3.6</td>
</tr>
<tr>
<td>facesim</td>
<td>POSIX</td>
<td>1,391</td>
<td>113.8</td>
<td>0</td>
</tr>
<tr>
<td>ferret</td>
<td>POSIX</td>
<td>2,706</td>
<td>111</td>
<td>2</td>
</tr>
<tr>
<td>x264</td>
<td>POSIX</td>
<td>1,494</td>
<td>1000</td>
<td>19</td>
</tr>
<tr>
<td>dedup</td>
<td>POSIX</td>
<td>3,228</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>streamcluster</td>
<td>POSIX</td>
<td>40,393</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>raytrace</td>
<td>POSIX</td>
<td>13,302</td>
<td>106,4</td>
<td>0</td>
</tr>
</tbody>
</table>
### Universal Race Detector

<table>
<thead>
<tr>
<th>Program</th>
<th>Para. model</th>
<th>LOC</th>
<th>Racy Contexts</th>
<th>DRD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Helgrind⁺ lib</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Helgrind⁺ lib+spin</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Helgrind⁺ nolib+spin</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>canneal</td>
<td>POSIX</td>
<td>29,31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>freqmine</td>
<td>OpenMP</td>
<td>10,279</td>
<td>153.4</td>
<td>2</td>
</tr>
<tr>
<td>vips</td>
<td>GLIB</td>
<td>1,255</td>
<td>50.8</td>
<td>0</td>
</tr>
<tr>
<td>bodytrack</td>
<td>POSIX</td>
<td>9,735</td>
<td>36.8</td>
<td>3.6</td>
</tr>
<tr>
<td>facesim</td>
<td>POSIX</td>
<td>1,391</td>
<td>113.8</td>
<td>0</td>
</tr>
<tr>
<td>ferret</td>
<td>POSIX</td>
<td>2,706</td>
<td>111</td>
<td>2</td>
</tr>
<tr>
<td>x264</td>
<td>POSIX</td>
<td>1,494</td>
<td>1000</td>
<td>19</td>
</tr>
<tr>
<td>dedup</td>
<td>POSIX</td>
<td>3,228</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>streamcluster</td>
<td>POSIX</td>
<td>40,393</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>raytrace</td>
<td>POSIX</td>
<td>13,302</td>
<td>106.4</td>
<td>0</td>
</tr>
</tbody>
</table>

**Happens-before detector**

- false positives are
  - Slightly increased in 4 cases

---

Identifying Ad-hoc Synchronization for Enhanced Race Detection
Performance

- Minor overhead due to the new feature for spinning read detection
- Memory consumption:
Performance

- Slight runtime overhead:
Summary

- Knowledge of all synchronization operations are crucial for accurate data race detection
  - Missing ad-hoc synchronizations causes a lot of false positives
- We present a dynamic method that is able to identify ad-hoc and unknown synchronizations in programs

- Universal race Detector
  - No need to upgrade the detector for unknown libraries
  - Best results achieved when using it as complementary method (applicable for every race detector)
- Future work: Improving the accuracy of the universal race detector by identifying the lock operations (enabling lockset analysis).
Thank you

Questions?


www.ipd.uka.de/Tichy/