Data Races vs. Data Race Bugs

by Baris Kasikci, Cristian Zamfir, and George Candea
École Polytechnique Fédérale de Lausanne (EPFL)
ASPLOS 2012

Christian Klauser
Data Races vs. Data Race Bugs

Telling the difference with “Portend”

- **Data race** defined as
  - two or more concurrent memory accesses to the same location
  - at least one access is a write-access

- “fixing” all data races rarely
  - desired (performance)
  - necessary (76%-90% are harmless)
Data Races vs. Data Race Bugs

Telling the difference with “Portend”

Data race bugs are very hard to reproduce.

To fixing data race bugs proactively:

- find data races
- determine whether harmful
- develop a fix
Data Races vs. Data Race Bugs

Telling the difference with “Portend”

Data race bugs are very hard to reproduce

To fixing data race bugs proactively:

- find data races
- determine whether harmful
- develop a fix

- tools
- this paper
- by hand
Data Races vs. Data Race Bugs

Telling the difference with “Portend”

Key contributions

- Four-category taxonomy of data races
- Technique for predicting consequences of data races
- Implementation of the above: “Portend”
# Data Race Classification

<table>
<thead>
<tr>
<th><strong>specViol</strong></th>
<th>Program crashes or programmer specification violated</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>outDiff</strong></td>
<td>Output differs between executions</td>
</tr>
<tr>
<td><strong>k-witness</strong></td>
<td>Harmless for at least $k$ different paths and schedules</td>
</tr>
<tr>
<td><strong>singleOrd</strong></td>
<td>Only a single ordering possible$^1$</td>
</tr>
</tbody>
</table>

$^1$ Arguably not a data race
Data Race Classification

- specViol
- outDiff
- k-witness
- outSame
- singleOrd

- harmful
- harmless

- true positive
- false positive

- reported data race
Portend Pipeline

Dynamic data race detector

“Single-Path” Analysis

“Multi-Path” Analysis

Record&Replay Engine

Classification Debug Report

other race detection reports
Single-Pre/Single-Post Analysis

Detect singleOrd data races, deadlocks

- Run schedule from report/detector
- On first racing access of data race $d$
  - Take snapshot $pre$ of current program state
  - Continue
- On second racing access of data race $d$
  - Take snapshot $post$ of current program state

(cont)
Single-Pre/Single-Post Analysis

Detect `singleOrd` data races, deadlocks

(cont)

- **Reset** program state to `pre`
- **Run** different schedule
  - **Preempt** thread that won last time
  - **Capture** program output as *alternate*
- **Run** original schedule
  - **Reset** program state to `post`
  - **Capture** program output as *primary*
Single-Pre/Single-Post Analysis

Possible outcomes

- No alternate scheduling
  - detected deadlock, treat as specViol
  - program doesn’t terminate (timeout)
    - if infinite loop, treat as specViol
    - else treat as singleOrd

- Found alternate scheduling
  - check for semantic specification violations (specViol)
  - compare program outputs (outDiff, outSame)
    - important: compare program output, not state after race
Portend Pipeline

Dynamic data race detector

“Single-Path” Analysis

“Multi-Path” Analysis

Record&Replay Engine

other race detection reports

Classification Debug Report
Multi-Path Data Race Analysis

Multiple inputs over the same thread schedule

- Execute program using **symbolic** inputs
  - path explosion
  - Abandon paths that contradict thread schedule
- After 2\textsuperscript{nd} racing access
  - collect $M_p$ different paths
  - call these our “primaries”

(cont)
Multi-Path Data Race Analysis

Multiple inputs over the same thread schedule

(cont)

For each “primary”:

► record **symbolic output**
► let SMT solver find **concrete inputs**
► run **alternate scheduling** with concrete inputs
  ► as in single-path analysis
► **compare** recorded output with symbolic reference output from primary
  ► again via SMT solver
Multi-Schedule Data Race Analysis

Multiple schedules over multiple inputs

Idea: after data race, don’t follow existing schedule

- Generate $M_a$ different post-race schedules (randomized)
- Record output and compare with primary

If output matches, we have

$$k = M_p \times M_a$$

“witnesses” that race is harmless (k-witness)
Portend Implementation Details

- Consumes **LLVM bitcode** programs
- 8K lines of **C++**, excluding libraries
  - **KLEE**, symbolic virtual machine for LLVM bitcode
  - **Cloud9** (EPFL), parallel symbolic execution engine
  - Symbolic **POSIX** emulation (part of Cloud9)
- Preemption points for single processor scheduling
  - **POSIX** synchronization primitives
  - data racing memory accesses
Portend Accuracy
How many races does Portend classify correctly?

- Testsuite with 93 data races
  - SQLite 3.3.0, memcached 1.4.5, ...
  - hand-written micro-benchmarks
- Classify data races by hand
- Compare with Portend classification
  - 92 out of 93 races correctly classified
  - One k-witness race was actually harmful
  - $k = 5$ (same result with $k = 10$)
Portend Results
What did Portend report?

- Multi-path + multi-schedule were vital for accuracy
- **Single-path** often classified 80% or more data races as **singleOrd**
- “Ctrace” mainly resulted in **outDiff**

<table>
<thead>
<tr>
<th>Program</th>
<th>Total # of races</th>
<th># of “Spec violated” races</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Deadlock</td>
</tr>
<tr>
<td>SQLite</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pbzip2</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>ctrace</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>fmm</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>memcached</td>
<td>18</td>
<td>0</td>
</tr>
</tbody>
</table>

![Chart showing accuracy percentages for different programs]
Portend Performance

- k-witness races take very long
  - many executions to simulate
- memcached: **11 minutes** to classify races
- Classification compared to bitcode interpretation
  - factor $\times 1.1$ to $\times 49.9$ longer
- Extreme ($\times 3$) variance in some cases
Limitations of Portend

- Performance nowhere near “interactive”
  - but $M_p \times M_a$ executions could be run in parallel

- Only POSIX synchronization primitives
  - Ignores machine-specific mechanisms (x86)

- Only single-processor model
  - Assumes memory-consistency, serializable execution
Thank you

Questions?
Additional Slides
### Portend Performance Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Cloud9 running time (sec)</th>
<th>Portend classification time (sec)</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>3.10</td>
<td>4.20</td>
<td>4.09</td>
<td>4.25</td>
<td></td>
</tr>
<tr>
<td>ocean</td>
<td>19.64</td>
<td>60.02</td>
<td>19.90</td>
<td>207.14</td>
<td></td>
</tr>
<tr>
<td>fmm</td>
<td>24.87</td>
<td>64.45</td>
<td>65.29</td>
<td>72.83</td>
<td></td>
</tr>
<tr>
<td>memcached</td>
<td>73.87</td>
<td>645.99</td>
<td>619.32</td>
<td>730.37</td>
<td></td>
</tr>
<tr>
<td>pbzip2</td>
<td>15.30</td>
<td>360.72</td>
<td>61.36</td>
<td>763.43</td>
<td></td>
</tr>
<tr>
<td>ctrace</td>
<td>3.67</td>
<td>24.29</td>
<td>5.54</td>
<td>41.08</td>
<td></td>
</tr>
<tr>
<td>bbuf</td>
<td>1.81</td>
<td>4.47</td>
<td>4.77</td>
<td>5.82</td>
<td></td>
</tr>
<tr>
<td>AVV</td>
<td>0.72</td>
<td>0.83</td>
<td>0.78</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>DCL</td>
<td>0.74</td>
<td>0.85</td>
<td>0.83</td>
<td>0.89</td>
<td></td>
</tr>
<tr>
<td>DBM</td>
<td>0.72</td>
<td>0.81</td>
<td>0.79</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td>RW</td>
<td>0.74</td>
<td>0.81</td>
<td>0.81</td>
<td>0.82</td>
<td></td>
</tr>
</tbody>
</table>
References

- Illustrations and tables taken from “Data Races vs. Data Race Bugs: Telling the Difference with Portend, Baris Kasikci, Cristian Zamfir, and George Candeа, ASPLOS’12”