Understanding the Interleaving-Space Overlap across Inputs and Software Versions

Dongdong Deng, Wei Zhang, Borui Wang, Peisen Zhao, Shan Lu
University of Wisconsin, Madison
Concurrency bug detection is critical

• Concurrency bugs are a big problem
  – Widely exist and cause severe damage [Northeast blackout, NasDaq]

• They only get worse
  – Multi-core machines are prevalent
Concurrency bug detection is costly

- Data race detection incurs 10-200X slowdown
- Software many inputs to test
- Test all inputs again upon new version

Software companies cannot afford thorough testing
How to speed up?

Input Space

Interleaving space

Software evolving
Redundancy across inputs (example)

Thread 1

```javascript
js_InitAtomState(...){
  state->table = ...;
  if(state->table)
    iterate state->table;
}
```

Thread 2

```javascript
js_FreeAtomState(..){
  ...
  state->table = NULL;
}
```
Redundancy across inputs

How to find inputs' interleaving space overlap before heavy-weight detection?
We have an opportunity

Interleaving Space Overlap across Inputs

Race Detection
We have a problem

Interleaving Space Overlap across Inputs

Interleaving Space Analysis
A solution

Here are all my inputs ...
Contributions

• Characteristics study of bug detection redundancy
  – Among different inputs
  – Among different versions

• A good metric to approximate interleaving space
  Concurrent Function Pairs
  – Reasonably accurate in depicting interleaving space
  – Lightweight in measurement

• CFP-guided race detection

• Similar ideas can be applied to regression test, etc.
Outline

• Motivation
• Challenges
• Characteristics study
• Solution in details
• Experiment Evaluation
• Conclusion
Characteristics Study Methodology

• 5 benchmarks written in C/C++
  – Aget, Click, FFT, Mozilla and Pbzip2
• One input set for each benchmark
  – Developer designed: Click and Mozilla
  – We designed: Aget, FFT and Pbzip2
• Detect data races and atomicity violations
  – Exposed by every input
  – Exposed by every version
Characteristics study

Y-axis # of bugs
X-axis Input id
- Bug detected by 2-4 inputs
- Bug detected by 8 inputs

Harmful Data Race

Please refer to our paper for more data
Characteristics study

Data Race (Benign & Harmful)

Please refer to our paper for more data
Please refer to our paper for more data
### Characteristics study

<table>
<thead>
<tr>
<th>Y-axis</th>
<th># of bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-axis</td>
<td>Input id</td>
</tr>
<tr>
<td></td>
<td>Bug detected by 2-4 inputs</td>
</tr>
<tr>
<td></td>
<td>Bug detected by 8 inputs</td>
</tr>
<tr>
<td></td>
<td>Bug detected by 1 input</td>
</tr>
</tbody>
</table>

**Atomocity Violation**
(Benign & Harmful)

Please refer to our paper for more data.
### Summary of characteristics study

<table>
<thead>
<tr>
<th>Application</th>
<th># Inputs that expose the same bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget</td>
<td>1</td>
</tr>
<tr>
<td>Click</td>
<td>1</td>
</tr>
<tr>
<td>FFT</td>
<td>1</td>
</tr>
<tr>
<td>Mozilla</td>
<td>1</td>
</tr>
<tr>
<td>Pbzip2</td>
<td>1</td>
</tr>
</tbody>
</table>
Outline

- Motivation
- Challenges
- Characteristics study
- Solution in details
- Experiment Evaluation
- Conclusion
Here are all my inputs..

Step 1

Step 2

Step 3
**Concurrent Function Pair**

*CFP*, Concurrent *Function Pair*, set of function pairs that can be potentially executed with each other.
Step 1 How to calculate CFP?

Thread 1

Trace

<table>
<thead>
<tr>
<th>THD</th>
<th>Addr</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>entry</td>
<td>0xab</td>
</tr>
<tr>
<td>1</td>
<td>exit</td>
<td>0xab</td>
</tr>
<tr>
<td>2</td>
<td>entry</td>
<td>0xab</td>
</tr>
<tr>
<td>2</td>
<td>lock</td>
<td>0xab</td>
</tr>
<tr>
<td>2</td>
<td>unlock</td>
<td>0xab</td>
</tr>
<tr>
<td>2</td>
<td>exit</td>
<td>0xab</td>
</tr>
</tbody>
</table>

Thread 2
Step 2 Design testing plan

Goal: select inputs and functions to avoid redundant CFP coverage

CFP of input 1
- \{f2, f4\}
- \{f4, f5\}

CFP of input 2
- \{f1, f2\}
- \{f3, f5\}

CFP of input 3
- \{f2, f3\}
- \{f3, f4\}

CFP Selection Result

<table>
<thead>
<tr>
<th>Input</th>
<th>Selected Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{f1, f2, f3, f4, f5}</td>
</tr>
<tr>
<td>2</td>
<td>{f3, f4, f5}</td>
</tr>
</tbody>
</table>


Step 3 CFP-based Race Detection

• What do we instrument?
  – Only instrument *selected* functions for *selected* inputs.

• What data race detector do we use?
  – Hybrid of Happen-before & lock-set
CFP-based Bug Detection Methodology

Step 1
LLVM

Step 2
C++

Step 3
Pin

Here are all my inputs ..

CFP Detector
## Input and function selection results

<table>
<thead>
<tr>
<th>Application</th>
<th>Input Section</th>
<th>Log-size Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget</td>
<td>3 out of 8</td>
<td>94%</td>
</tr>
<tr>
<td>Click</td>
<td>6 out of 6</td>
<td>64%</td>
</tr>
<tr>
<td>FFT</td>
<td>1 out of 8</td>
<td>83%</td>
</tr>
<tr>
<td>Mozilla</td>
<td>7 out of 8</td>
<td>33%</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>2 out of 8</td>
<td>44%</td>
</tr>
</tbody>
</table>
Miss any failure inducing data race?
How much redundancy we reduced?

<table>
<thead>
<tr>
<th>Application</th>
<th># Inputs that expose the same bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget</td>
<td>6.67</td>
</tr>
<tr>
<td>Click</td>
<td>2.19</td>
</tr>
<tr>
<td>FFT</td>
<td>4.67</td>
</tr>
<tr>
<td>Mozilla</td>
<td>5.47</td>
</tr>
<tr>
<td>Pbzip2</td>
<td>4.17</td>
</tr>
</tbody>
</table>

VS

<table>
<thead>
<tr>
<th>Application</th>
<th># Inputs that expose the same bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget</td>
<td>1</td>
</tr>
<tr>
<td>Click</td>
<td>1.63</td>
</tr>
<tr>
<td>FFT</td>
<td>1</td>
</tr>
<tr>
<td>Mozilla</td>
<td>2.73</td>
</tr>
<tr>
<td>Pbzip2</td>
<td>1.16</td>
</tr>
</tbody>
</table>
How much time we saved?

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedup (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget</td>
<td>0.76</td>
</tr>
<tr>
<td>Click</td>
<td>2.14</td>
</tr>
<tr>
<td>FFT</td>
<td>7.13</td>
</tr>
<tr>
<td>Mozilla</td>
<td>1.27</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>1.72</td>
</tr>
</tbody>
</table>

*The time in our approach is calculated as the sum of three steps introduced before.*
Aget (Data Race)

```c
typedef struct hist_data {
    ...
    int bwritten;
    ...
} hist_data;

while(td->offset < foffset) {
    ...
    dw = pwrite(...);
    td->offset += dw;
    ...
    bwritten += dw;
}

void save_log() {
    ...
    h.bwritten += bwritten;
    ...
}
```
Conclusions

• Concurrency bug detection results are redundant across inputs and software versions

• Concurrent function pair (CFP) is good coverage metric for interleaving space
  – Lightweight
  – Reasonably accurate

• CFP can be used to guide concurrency-bug detection across inputs and software versions
Future Work

• Applying CFP to help detect other types of bugs

• Preprocessing to get more accurate CFP
  – Remove functions that do not access shared variables

• Trying different heuristics for different testing budgets

• Exploring other coverage metrics
Thanks!

Wei Zhang
http://pages.cs.wisc.edu/~wzh