Detecting and Tolerating Asymmetric Races

1Paruj Ratanaworabhan, 2Martin Burtscher, 3Darko Kirovski
3Benjamin Zorn, 4Rahul Nagpal, and 5Karthik Pattabiraman

1Cornell University
2The University of Texas at Austin
3Microsoft Research Redmond
4Indian Institute of Science
5University of Illinois at Urbana-Champaign
Overview

• Introduction of multicore processors made parallel programming ubiquitous

• Parallel programming is hard
  – Suffers from all the problems of sequential programming
  – Introduces additional sources of errors
    • E.g., deadlock, atomicity violation, and data races
Scope of this work

• Data races
  – Asymmetric races

• Large code base of parallel applications
  – Lock-based programs
  – Written mostly in C/C++
  – Use add-on libraries for threading and synchronization

• ToleRace: detects and tolerates races at runtime
Talk outline

• Overview and scope
• Asymmetric races
• The oracle ToleRace
• Pin-ToleRace
• Evaluation of Pin-ToleRace
• Ideal software ToleRace
• Summary
Asymmetric races

• One thread correctly protects a shared variable

• Another thread accesses the same variable with improper synchronization

Thread 1:

```c
CSEnter(mutex_A)
if (gScript == NULL)
    baseScript = default;
else
    baseScript = gScript;
CSEnter(mutex_A)
```

Thread 2:

```c
gScript = NULL
baseScript = gScript;
```
Why focus on asymmetric races

- Prevalent in software development projects
  - Direct experience from Microsoft developers
  // K and flag are declared volatile
  - Possible reasons:

Thread 1:  Thread 2:
K = x;   while (flag != true);
flag = true;  y = K;

Symmetric races are often benign
- Correct local reasoning but lock convention broken
- Assumptions in legacy codes invalidated
Characterizing asymmetric races

$T_1 = \text{safe thread taking proper locks}$  $t_2 = \text{unsafe thread improperly synchronized}$

$T'_1 T''_1 t_2$

- Race
- True (non-repeatable read)

$WX^* wx^* WX^*$

False = $wx^* WX^* WX^*$

No race = $T_1$ and $t_2$ operations are serializable

Upper case for safe thread

Lower case for unsafe thread

Read (r, R)  Write (w, W)  Don’t care (x, X)

Read-dependent write (rw, RW)

+ denotes one or more of the preceding operation

* denotes zero or more of the preceding operation
### Characterizing asymmetric races

Possible interaction sequences: \( R+(r+), WX*(wx*), \) and \( R+WX*(r+wx*) \)

#### Table:

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( t_2 )</th>
<th>( T''_1 )</th>
<th>race</th>
<th>( T'_1 )</th>
<th>( t_2 )</th>
<th>( T''_1 )</th>
<th>race</th>
<th>( T'_1 )</th>
<th>( t_2 )</th>
<th>( T''_1 )</th>
<th>race</th>
</tr>
</thead>
<tbody>
<tr>
<td>R+</td>
<td>r+</td>
<td>R+</td>
<td>false</td>
<td>R+</td>
<td>wx*</td>
<td>R+</td>
<td>true</td>
<td>I</td>
<td>R+</td>
<td>r+wx*</td>
<td>R+</td>
</tr>
<tr>
<td>R+</td>
<td>r+</td>
<td>WX*</td>
<td>false</td>
<td>R+</td>
<td>wx*</td>
<td>WX*</td>
<td>true</td>
<td>III</td>
<td>R+</td>
<td>r+wx*</td>
<td>WX*</td>
</tr>
<tr>
<td>R+</td>
<td>r+</td>
<td>R+WX*</td>
<td>false</td>
<td>R+</td>
<td>wx*</td>
<td>R+WX*</td>
<td>true</td>
<td>I</td>
<td>R+</td>
<td>r+wx*</td>
<td>R+WX*</td>
</tr>
<tr>
<td>WX*</td>
<td>r+</td>
<td>R+</td>
<td>false</td>
<td>WX*</td>
<td>wx*</td>
<td>R+</td>
<td>true</td>
<td>I</td>
<td>WX*</td>
<td>r+wx*</td>
<td>R+</td>
</tr>
<tr>
<td>WX*</td>
<td>r+</td>
<td>WX*</td>
<td>true</td>
<td>II</td>
<td>WX*</td>
<td>wx*</td>
<td>WX*</td>
<td>false</td>
<td>WX*</td>
<td>r+wx*</td>
<td>WX*</td>
</tr>
<tr>
<td>WX*</td>
<td>r+</td>
<td>R+WX*</td>
<td>true</td>
<td>II</td>
<td>WX*</td>
<td>wx*</td>
<td>R+WX*</td>
<td>true</td>
<td>I</td>
<td>WX*</td>
<td>r+wx*</td>
</tr>
<tr>
<td>R+WX*</td>
<td>r+</td>
<td>R+</td>
<td>false</td>
<td>R+WX*</td>
<td>wx*</td>
<td>R+</td>
<td>true</td>
<td>I</td>
<td>R+WX*</td>
<td>r+wx*</td>
<td>R+</td>
</tr>
<tr>
<td>R+WX*</td>
<td>r+</td>
<td>WX*</td>
<td>true</td>
<td>II</td>
<td>R+WX*</td>
<td>wx*</td>
<td>WX*</td>
<td>true</td>
<td>III</td>
<td>R+WX*</td>
<td>r+wx*</td>
</tr>
<tr>
<td>R+WX*</td>
<td>r+</td>
<td>R+WX*</td>
<td>true</td>
<td>II</td>
<td>R+WX*</td>
<td>wx*</td>
<td>R+WX*</td>
<td>true</td>
<td>I</td>
<td>R+WX*</td>
<td>r+wx*</td>
</tr>
</tbody>
</table>

**No race** = \( T_1 \) and \( t_2 \) operations are serializable

- **Race case:**
  - I: XwR
  - II: WrW
  - III: RwW
  - IV: XrwX

- Any race conditions among \( K > 2 \) threads can always be reduced to one of the four race cases
Talk outline

• Overview and scope
• Asymmetric races
• The oracle ToleRace
• Pin-ToleRace
• Evaluation of Pin-ToleRace
• Ideal software ToleRace
• Summary
The oracle ToleRace mechanics

- Upon acquiring a lock protecting V, T1 creates two private copies of V: V' and V''
- T1 operates on V'; t2 on V; V'' is a clean copy
- Before unlocking, execute the resolution function f
  - Detecting and/or tolerating the race at this point
**Tolerace resolution function**

- Given a shared variable:
  \( V = \text{global operable}; V' = \text{local operable}; V'' = \text{local clean} \)

- Tolerate races by enforcing serial execution

- Race cases:

<table>
<thead>
<tr>
<th>Case</th>
<th>Tolerate &amp; Detect</th>
<th>F() = V</th>
<th>F() = V'</th>
<th>F() = V''</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: XwR</td>
<td>Tolerate</td>
<td>V != V''</td>
<td>V == V''</td>
<td>V != V''</td>
</tr>
<tr>
<td>II: WrW</td>
<td>Detect</td>
<td>V != V''</td>
<td>V == V''</td>
<td>V != V''</td>
</tr>
<tr>
<td>III: RwW</td>
<td>Detect</td>
<td>V != V''</td>
<td>V == V''</td>
<td>V != V''</td>
</tr>
<tr>
<td>IV_A: XrwX</td>
<td>Tolerate</td>
<td>V != V''</td>
<td>V == V''</td>
<td>V != V''</td>
</tr>
<tr>
<td>IV_B: WrwX</td>
<td>Tolerate</td>
<td>V != V''</td>
<td>V == V''</td>
<td>V != V''</td>
</tr>
<tr>
<td>IV_C: RwW</td>
<td>Detect</td>
<td>V != V''</td>
<td>V == V''</td>
<td>V != V''</td>
</tr>
</tbody>
</table>

- Tolerate races by enforcing serial execution

\[ F(v) = V \]

\[ t_1 t_2 \quad t_2 T_1 \quad T_1 t_2 \quad T_1 t_2 \quad t_2 T_1 \quad \text{N/A} \]

- Detect & Tolerate

- Detect

- N/A
Analogous to transactional memory, ...
Comparison with Transactional Memory

- Uses lazy versioning and lazy conflict detection
- Never aborts or rolls back
- Does not need contention management
- Can handle I/O and overlapped critical sections
Talk outline

• Overview and scope
• Asymmetric races
• The oracle ToleRace
• Pin-ToleRace
• Evaluation of Pin-ToleRace
• Ideal software ToleRace
• Summary
Pin-ToleRace

• Implemented software ToleRace on top of Pin, a dynamic instrumentation tool from Intel

• Work directly on executables

• Motivation for software ToleRace:
  – Can be deployed immediately
  – Gauges the worst case overhead by performing all analyses and decisions at runtime
Pin-ToleRace specific details

- x86/Linux platform
- Parallel pthread-based programs
- pthread_mutex_lock/unlock pair defines a critical section
Oracle ToleRace versus Pin-ToleRace

- **Oracle ToleRace**
  - Protects shared variables
  - All protected variables known
  - Atomic copy
  - Not realizable

- **Pin-ToleRace**
  - Protects shared memory
  - All protected locations determined on-the-fly
  - Non-atomic copy
  - Implementable
Tolerate races with Pin-ToleRace

- Pin-ToleRace knows all the shared accesses in the safe thread, but cannot distinguish between intervening \textit{rw} and \textit{w} sequences from other threads.

- Comparison of oracle ToleRace with Pin-ToleRace

<table>
<thead>
<tr>
<th>race type</th>
<th>Tolerable</th>
<th>Oracle ToleRace</th>
<th>Pin-ToleRace</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>XwR</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>II</td>
<td>WrW</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>III</td>
<td>RwW</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>IV\textsubscript{A}</td>
<td>RrwR</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>IV\textsubscript{B}</td>
<td>WrwX</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>IV\textsubscript{C}</td>
<td>RrwW</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
Pin-ToleRace evaluation

• Microbenchmark stress tests
• Real applications
Benchmarks

• 3 microbenchmarks for stress tests
  – Scalar, static array, and dynamic array

• 13 real applications
  – SPLASH2: four kernels and four applications
  – PARSEC: one kernel and four applications

• All benchmarks compiled and run on Intel 32-bit system with 4-core 2.8 GHz P4-Xeon
Stress tests

- Demonstrate race toleration with microbenchmarks
  - Safe thread: increments shared counters each iteration
  - Unsafe threads: impart random writes to shared counters
- Overhead is very high
  - Almost always executing inside critical sections

### Graph

- **Dynamic array**
- **Native**
- **Pin**
- **Pin-Tolerace**

- **Normalized execution time**

- **Number of iterations**
  - 5M
  - 7.5M
  - 10M

- **Normalized execution times**
  - 74x
  - 6.6x
Critical section characteristics

- Small number of unique critical sections
- Infrequently executing inside critical sections

<table>
<thead>
<tr>
<th></th>
<th>unique</th>
<th>nested CS</th>
<th>dynamic number of instrs per CS (user)</th>
<th>% dynamic instrs in CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky</td>
<td>14</td>
<td>no</td>
<td>29</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>fft</td>
<td>10</td>
<td>no</td>
<td>17</td>
<td>&lt; 0.01%</td>
</tr>
<tr>
<td>lu</td>
<td>7</td>
<td>no</td>
<td>17</td>
<td>&lt; 0.01%</td>
</tr>
<tr>
<td>radix</td>
<td>9</td>
<td>no</td>
<td>17</td>
<td>&lt; 0.01%</td>
</tr>
<tr>
<td>barnes</td>
<td>10</td>
<td>no</td>
<td>94</td>
<td>0.18%</td>
</tr>
<tr>
<td>ocean</td>
<td>26</td>
<td>no</td>
<td>17</td>
<td>&lt; 0.01%</td>
</tr>
<tr>
<td>radiosity</td>
<td>36</td>
<td>yes</td>
<td>18</td>
<td>0.11%</td>
</tr>
<tr>
<td>water-spatial</td>
<td>16</td>
<td>no</td>
<td>13</td>
<td>&lt; 0.01%</td>
</tr>
<tr>
<td>dedup</td>
<td>7</td>
<td>yes</td>
<td>600</td>
<td>0.42%</td>
</tr>
<tr>
<td>facesim</td>
<td>5</td>
<td>yes</td>
<td>46</td>
<td>&lt; 0.01%</td>
</tr>
<tr>
<td>ferret</td>
<td>4</td>
<td>yes</td>
<td>690</td>
<td>1.59%</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>11</td>
<td>no</td>
<td>13</td>
<td>0.40%</td>
</tr>
<tr>
<td>x264</td>
<td>2</td>
<td>no</td>
<td>11</td>
<td>&lt; 0.01%</td>
</tr>
</tbody>
</table>
Critical section characteristics

- The table shows unique accesses to possibly shared locations per critical section.
- This number is less than five except for barnes, dedup, facesim, and ferret.

<table>
<thead>
<tr>
<th>Function</th>
<th>unique accesses</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky</td>
<td></td>
<td>4.78</td>
<td>0.38</td>
</tr>
<tr>
<td>fft</td>
<td></td>
<td>1.37</td>
<td>0.04</td>
</tr>
<tr>
<td>lu</td>
<td></td>
<td>2.99</td>
<td>0.01</td>
</tr>
<tr>
<td>radix</td>
<td></td>
<td>2.82</td>
<td>0.19</td>
</tr>
<tr>
<td>barnes</td>
<td></td>
<td>19.13</td>
<td>0.03</td>
</tr>
<tr>
<td>ocean</td>
<td></td>
<td>3.00</td>
<td>0.00</td>
</tr>
<tr>
<td>radiosity</td>
<td></td>
<td>4.92</td>
<td>0.23</td>
</tr>
<tr>
<td>water-spatial</td>
<td></td>
<td>2.62</td>
<td>0.01</td>
</tr>
<tr>
<td>dedup</td>
<td></td>
<td>80.87</td>
<td>3.52</td>
</tr>
<tr>
<td>facesim</td>
<td></td>
<td>7.70</td>
<td>1.14</td>
</tr>
<tr>
<td>ferret</td>
<td></td>
<td>72.89</td>
<td>33.83</td>
</tr>
<tr>
<td>fluidanimate</td>
<td></td>
<td>5.00</td>
<td>0.00</td>
</tr>
<tr>
<td>x264</td>
<td></td>
<td>2.16</td>
<td>0.02</td>
</tr>
</tbody>
</table>
Pin-ToleRace performance

- On average, about 2X and 24% slowdown compared to the native and Pin run, respectively
- Approximate upper bound on overhead

Normalized execution time of Pin-ToleRace
Ideal software ToleRace performance

- On average, only 7% slowdown
- Most applications run with less than 1% overhead
- Approximate lower bound on overhead
Summary

• Asymmetric races are an important class of parallel programming errors

• We presented ToleRace, a theoretical framework for detecting and *tolerating* asymmetric races

• We showed that an implementable software ToleRace system based on Pin has a 2X overhead

• Aim to further improve ToleRace by
  – Providing a stronger isolation guarantee
  – Lowering the software ToleRace overhead
Backup Slides
The oracle ToleRace

- A *theoretical* framework for handling asymmetric races in lock-based parallel programs
  - Creates local copies of shared variables upon CSEnter() 
  - Detects changes to shared data at critical CSExit() 
  - Propagates the appropriate copy to hide races
- Dynamically detects the race and also *tolerates* it whenever possible
- Incurs overhead only at critical section execution
Pin-ToleRace general framework

- Defines safe memory as the region that holds local copies of shared memory locations
- Once in a critical section, instruments each executed instruction touching shared locations
- Searches the safe memory for shared locations
  - If found: accesses are redirected to the safe memory
  - If not: create a new node in the safe memory and redirect accesses
The safe memory region

- Contains three main data structures:
  - Safemem list
  - Tid-lock table
  - Safemem header