Kivati: Fast Detection and Prevention of Atomicity Violations

Lee Chew  David Lie

University of Toronto
MOTIVATION

- Multi-core computers are ubiquitous, and programs must become multithreaded → more concurrency bugs
- Despite existing research, they remain an open and difficult problem
- A recent study by Lu et al. (ASPLOS 2008) has shown atomicity violations cause 65% of all concurrency bugs
PROBLEM (INTUITIVELY)

Thread 1

if (shared_ptr != NULL) [READ]

b = shared_ptr->data [READ]

Thread 2

shared_ptr = NULL [WRITE]
• atomicity violation occurs when two threads access a shared variable unserializably → no equivalent serial execution

Thread 1

if (shared_ptr != NULL) [READ]

shared_ptr = NULL [WRITE]

b = shared_ptr->data [READ]

Thread 2

if (shared_ptr != NULL) [READ]

b = shared_ptr->data [READ]

if (shared_ptr != NULL) [READ]

b = shared_ptr->data [READ]

if (shared_ptr != NULL) [READ]

b = shared_ptr->data [READ]

shared_ptr = NULL [WRITE]
GOALS

• an online system to detect and prevent atomicity violations
  – complement offline bug-finders
  – re-order *unserializable* interleavings into *serializable* interleavings
  – only new interleavings that were possible in the original program

• incur low overhead
  – existing solutions either use software monitoring (up to 72X slowdown) or custom hardware
  – we achieved an average of 19% overhead on commodity hardware
SYSTEM ARCHITECTURE

- simple static analyser
- label possible violations

- modified Linux kernel
- detect and prevent actual violations

Source Code ➔ ANNOTATOR ➔ Regular Compiler ➔ PREVENTION ENGINE

Fixed Program
Report of All Fixes
• annotate sections of code where violations may occur
  – atomic region = code path between pair of accesses to shared variable
  – defined by `begin_atomic()` and `end_atomic()` annotations

• conservative
  – many atomic regions are unnecessary: either atomicity isn’t required or atomicity can’t be violated
  – DOES NOT affect correctness, only impacts performance
ANNOTATION EX: SIMPLE

1: tmp1 = shared;
2: tmp1 = tmp1 + 1;
3: shared = tmp1;

1:  begin_atomic(1, &shared);
2:  tmp1 = shared;
3:  tmp1 = tmp1 + 1;
4:  shared = tmp1;
5:  end_atomic(1);
1: if (shared) {
2:     shared = 0;
3: }
4: tmp = shared;

1: begin_atomic(1, &shared);
2: if (shared) {
3:     begin_atomic(2, &shared);
4:     shared = 0;
5:     end_atomic(1);
6: }
7: tmp = shared;
8: end_atomic(1);
9: end_atomic(2);
PREVENTION ENGINE

- use watchpoints to detect and prevent violations
  - hardware registers present on Intel/AMD processors
  - efficient method of monitoring memory accesses

- `begin_atomic()` and `end_atomic()`
  - custom system calls
  - start/stop monitoring memory location
PREVENTION MODE

- detect and prevent atomicity violations

Thread 1:

```c
begin_atomic(1, &global_num);
a = global_num;
........
........
b = global_num;
end_atomic(1);
```

Thread 2:

```c
global_num = 5;
........
........
global_num = 5;
global_num = 5;
global_num = 5;
global_num = 5;
```

Table:

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_num</td>
<td>1000</td>
</tr>
<tr>
<td>Watchpoint</td>
<td>1000</td>
</tr>
</tbody>
</table>
Bug-Finding Mode

- Same as Prevention Mode + increase likelihood of atomicity violations occurring
  - "gap" between pair of accesses is usually small
  - therefore, inject delay between them

```c
begin_atomic();
a = global_num;
b = global_num;
end_atomic();
```
THE NEVER-ENDING REGION?

- `end_atomic()` may not occur → use a timeout

Branching:
```c
begin_atomic(1, &global);
if (global >= 0)
{
    global = -1;
    end_atomic(1);
}
```

Forced Ordering:
```c
begin_atomic(1, &global);
global = 5;
flag = 1;
while (flag == 1); // wait
while (flag != 1);
global = 10;
flag = 0;
tmp = global;
end_atomic(1);
```
SOLUTION: `end_atomic()` no longer drops into kernel

```
b = global_num;
end_atomic(1);

........
begin_atomic(2, other_shared);
```

<table>
<thead>
<tr>
<th></th>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_num</td>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>

Watchpoint #1

<table>
<thead>
<tr>
<th></th>
<th>clear_wp</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>wp_num</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
CHALLENGE 2: PER-CORE WATCHPOINTS

SOLUTION: Lazily propagate watchpoint settings

Watchpoint 1

Core 1

1000

Core 2

Watchpoint 1

1000

Core 3

Watchpoint 1

1000

Core 4

Watchpoint 1

1000
CHALLENGE 3: TRAPS AFTER ACCESS

SOLUTION: Undo and re-apply access

Thread 1

begin_atomic();
a = global_num;

............... 
b = global_num;
end_atomic();

Thread 2

global_num = 5;

<table>
<thead>
<tr>
<th>global_num</th>
<th>OLD_VALUE</th>
<th>NEW_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
EVALUATION

- tested using 2.13 GHz Core 2 Duo machine with 2GB of RAM
- tested in Prevention Mode

<table>
<thead>
<tr>
<th>Application</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSS (Firefox module)</td>
<td>Running the included test suite</td>
</tr>
<tr>
<td>VLC Media Player</td>
<td>Transcoding a video using the x264 codec</td>
</tr>
<tr>
<td>Apache</td>
<td>Webstone 2.5</td>
</tr>
<tr>
<td>Apache &amp; MySQL</td>
<td>TPC-W with 10 browsers</td>
</tr>
<tr>
<td>SPEC2001 OMP</td>
<td>Running itself</td>
</tr>
</tbody>
</table>
OVERHEAD (%)

• isolate factors causing overhead
  – dropping into kernel vs. in-kernel processing
• null syscall = just dropping into kernel and returning
  – but overhead is still high → it comes from dropping into kernel

<table>
<thead>
<tr>
<th>Application</th>
<th>Base</th>
<th>Null syscall</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSS</td>
<td>32.4</td>
<td>25.3</td>
<td>19.7</td>
</tr>
<tr>
<td>VLC</td>
<td>18.0</td>
<td>14.3</td>
<td>13.0</td>
</tr>
<tr>
<td>Apache</td>
<td>27.9</td>
<td>22.6</td>
<td>16.5</td>
</tr>
<tr>
<td>TPC-W</td>
<td>53.7</td>
<td>40.9</td>
<td>29.5</td>
</tr>
<tr>
<td>SPEC2001 OMP</td>
<td>30.0</td>
<td>24.6</td>
<td>19.0</td>
</tr>
</tbody>
</table>
DOMAIN CROSSINGS (k/sec)

- decrease in # of domain crossings tracked by performance improvements

<table>
<thead>
<tr>
<th>Application</th>
<th>Base</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSS</td>
<td>1403</td>
<td>821 (41%)</td>
</tr>
<tr>
<td>VLC</td>
<td>730</td>
<td>492 (33%)</td>
</tr>
<tr>
<td>Apache</td>
<td>1114</td>
<td>608 (45%)</td>
</tr>
<tr>
<td>TPC-W</td>
<td>2359</td>
<td>1220 (48%)</td>
</tr>
<tr>
<td>SPEC2001 OMP</td>
<td>1315</td>
<td>788 (40%)</td>
</tr>
</tbody>
</table>
## MISSED ARs VS. # of WATCHPOINTS

- # of watchpoints < 4: steep increase in missed ARs
- # of watchpoints > 4: diminishing returns

<table>
<thead>
<tr>
<th>Application</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSS</td>
<td>57%</td>
<td>39%</td>
<td>5.7%</td>
<td>3.6%</td>
<td>1.4%</td>
<td>0.32%</td>
<td>0.0007%</td>
</tr>
<tr>
<td>VLC</td>
<td>34%</td>
<td>15%</td>
<td>5.2%</td>
<td>1.6%</td>
<td>0.01%</td>
<td>0.0006%</td>
<td>0%</td>
</tr>
<tr>
<td>Apache</td>
<td>51%</td>
<td>29%</td>
<td>4.9%</td>
<td>3.0%</td>
<td>0.58%</td>
<td>0.42%</td>
<td>0.027%</td>
</tr>
<tr>
<td>TPC-W</td>
<td>59%</td>
<td>44%</td>
<td>9.1%</td>
<td>6.1%</td>
<td>1.8%</td>
<td>1.0%</td>
<td>0.39%</td>
</tr>
<tr>
<td>SPEC2001 OMP</td>
<td>66%</td>
<td>9.1%</td>
<td>4.8%</td>
<td>3.5%</td>
<td>1.3%</td>
<td>0.022%</td>
<td>0.001%</td>
</tr>
</tbody>
</table>
## BUG-DISCOVERY TIMES

<table>
<thead>
<tr>
<th>App</th>
<th>Bug #</th>
<th>Prevention</th>
<th>Bug-finding (20ms)</th>
<th>Bug-finding (50ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Apache</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>66:59</td>
<td>8:01</td>
<td>8:23</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Didn’t Happen</td>
<td>13:30</td>
<td>17:20</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Didn’t Happen</td>
<td><strong>4:49</strong></td>
<td><strong>7:33</strong></td>
</tr>
<tr>
<td><strong>NSS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>12:25</td>
<td>2:59</td>
<td>2:05</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1:40</td>
<td>0:16</td>
<td>0:17</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4:41</td>
<td>2:21</td>
<td>3:09</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2:00</td>
<td>0:33</td>
<td>0:56</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Didn’t Happen</td>
<td><strong>10:19</strong></td>
<td><strong>7:40</strong></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>52:45</td>
<td>9:27</td>
<td>7:33</td>
</tr>
<tr>
<td><strong>MySQL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>8:53</td>
<td>1:50</td>
<td>1:26</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>11:15</td>
<td>2:44</td>
<td>3:20</td>
</tr>
</tbody>
</table>
CONCLUSION

• Kivati is an online system that detects and prevents atomicity violations
  – re-order unserializable interleavings into serializable ones

• we have low overhead and are effective
  – 19% overhead on average
  – detected all bugs in our collection

• novel use of watchpoints to monitor accesses
  – overcame challenges: costly trips into the kernel, per-core watchpoints and traps occurring after accesses
Thank-you.
FALSE POSITIVES

Prevention Mode

Bug-finding Mode

NSS

TPC-W

NSS

TPC-W

1st Run
2nd Run
3rd Run
4th Run
RELATED WORK

Atomicity Violation Detectors
- SVD – approximates some atomic regions
- AVIO – use training runs to identify malignant violations
- CTrigger – injects delays to increase chance of violations

Atomicity Violation Prevention Systems
- Atom-Aid – provides probabilistic protection by grouping and atomically executing certain instructions
- Constrained interleavings (Yu & Narayanasamy) – ensure only correct interleavings observed during training runs are allowed during regular execution