SigRace: Signature-Based Data Race Detection

A. Muzahid, D. Suárez, S. Qi, J. Torrellas
Background

- A data race is when two threads access the same variable **concurrently** and at least one is a **write**.

- Data races are a growing problem
  - Becoming more common
  - Hard to detect and reproduce
Previous Work

- **Software Detectors**
  - 10-50x execution slow down
  - Disrupt the thread timing

- **Hardware Detectors**
  - Modify cache to track accesses & cache coherence protocol
  - Lose information on invalidation
  - Affect critical path
SigRace Overview

• Use **hardware address signatures** and **happened before approach**
  – No cache modifications
  – Catches more data races
Happened Before - Logical Clocks

- Determine an ordering of threads

- Threads keeps a **vector clock**: timestamp for each thread

- Synchronization event: $V_{c,t}[t]++$

- Acquire a shared object: $V_{c,u}[i] = \max(V_{c,u}[i], V_{c,t}[i])$
**Happened-Before**

T’s epoch **happened before** U’s if:

\[ VC_t[t] < VC_u[t] \& VC_t[u] < VC_u[u] \]

- Ex [1,0] and [2,1]

Concurrent = Neither epoch happened before the other

Detect data races by finding concurrent accesses
Hardware Address Signature

• History of memory accesses
• Generated using a bloom filter:
  – Hash and accumulate memory address
• Can create false positives do to aliasing
Normal Mode – Processor

Each processor tracks:

1. Current Timestamp (TS)
2. Read Hardware Signature (R)
3. Write Hardware Signature (W)

At the end of a block, send TS, R, W to the Race Detection Module (RDM) for analysis.

Block:
- Epoch or
- Fixed number of instructions
Normal Mode - Detection

- Compare new TS to the history of other processor’s TS

- If intersection not null flag a data race

- Save the conflict signature and conflicting threads
Re-execution

- **Rollback** to nearest checkpoint before data race
  - Require fine-grained **checkpointing**
- **Re-execute** threads until the epoch of the race
  - Epoch by epoch to recreate exact thread scheduling
  - Uses TS log and additional hardware
Analysis Mode

- Determine exact location of the data race
- For conflicting threads: intersect load and store addresses with the conflict signature

If there is a hit:
Log the address & instruction
Analysis Mode

• End of epochs: Compare logs
  – Common addresses = Data race
  – Else false positive
SIGRACE IMPLEMENTATION
**Hardware Implementation**

- Contains BlockHistoryQueue[.] which stores past time stamps and signatures for all the processors

- Stores the current epoch timestamp instructions
- Stores flags giving information about the mode of operation and controller to determine when to dump the information into RDM
## SOFTWARE IMPLEMENTATION

<table>
<thead>
<tr>
<th>Operation</th>
<th>Implementation</th>
<th>Code</th>
</tr>
</thead>
</table>
| Unlock    | Conventional   | UNLOCK{
unlock($1);} |
|           | SigRace        | SN_UNLOCK{
sync_reached;
timestamp = TS;
unlock($1.lock);} |
| Lock      | Conventional   | LOCK{
lock($1);} |
|           | SigRace        | SN_LOCK{
Lock($1.lock);
sync_reached;
TS=generateTS(TS, $1.timestamp);
timestamp = TS;
unlock($1.lock);} |
SigRace Implementation

**Hardware Changes**

**Modified**
- Extra Race Detection Module (RDM)
- Additions to private L1 cache

**Unmodified**
- The cache coherence protocol.
- Data Arrays

**Software Changes**

**Modified**
- Compiler level changes

**Unmodified**
- Application Code
Sigrace Implementation: Virtualization

(a) Vector Clock

(b) Race Detection Module (RDM)
### False Positive Analysis: Signature Configuration

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Address Partition</th>
<th>LSB</th>
<th>USB</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>8 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>10 22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>16 16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Config</th>
<th>No. of Bloom Filters</th>
<th>Bits per Bloom Filter</th>
<th>Sig Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>16</td>
<td>256</td>
<td>4k bit</td>
</tr>
<tr>
<td>S2</td>
<td>16</td>
<td>128</td>
<td>2k bit</td>
</tr>
<tr>
<td>S3</td>
<td>16</td>
<td>64</td>
<td>1K bit</td>
</tr>
<tr>
<td>S4</td>
<td>8</td>
<td>512</td>
<td>4k bit</td>
</tr>
<tr>
<td>S5</td>
<td>8</td>
<td>256</td>
<td>2K bit</td>
</tr>
<tr>
<td>S6</td>
<td>8</td>
<td>128</td>
<td>1K bit</td>
</tr>
</tbody>
</table>
Block and Block History Size Effects

![Graph showing lost detection window (%) and average number of comparisons per BHQ[i] entries for different block sizes and history sizes.]

- **Lost Detection Window (%)**
  - 2,000 Inst/Block
  - 8,000 Inst/Block
  - Sync/Block
  - 4,000 Inst/Block
  - 16,000 Inst/Block

- **Number of BHQ[i] Entries**
  - 16
  - 32
  - 64

- **Average Num of Comparisons per BHQ[i]**
  - Values for different block sizes and history sizes.
## SigRace Effectiveness

<table>
<thead>
<tr>
<th>Application</th>
<th>Finding Existing Races</th>
<th>Finding Injected Races</th>
<th>Finding Injected Races</th>
<th>Racy Runs</th>
<th>Static Races Found</th>
<th>W-ReEnact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SigRace</td>
<td>W-ReEnact</td>
<td>SigRace</td>
<td>W-ReEnact</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stat</td>
<td>Dyn</td>
<td>Stat</td>
<td>Dyn</td>
<td>Racy Runs</td>
<td></td>
</tr>
<tr>
<td>Barnes</td>
<td>11</td>
<td>719</td>
<td>6</td>
<td>419</td>
<td>1/25</td>
<td>3</td>
</tr>
<tr>
<td>Ocean</td>
<td>1</td>
<td>29</td>
<td>1</td>
<td>6</td>
<td>7/25</td>
<td>8</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>12</td>
<td>14307</td>
<td>12</td>
<td>436</td>
<td>6/25</td>
<td>7</td>
</tr>
<tr>
<td>Fluidanimate</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12/25</td>
<td>95</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>24</strong></td>
<td><strong>15055</strong></td>
<td><strong>19</strong></td>
<td><strong>861</strong></td>
<td><strong>25/100</strong></td>
<td><strong>113</strong></td>
</tr>
</tbody>
</table>
On average instruction overhead due to re-execution is 22% and about 2/3 of it is caused by false positives.

On average bandwidth overhead is 63 byte per thousand committed instructions.
Conclusion

• Uses hardware signatures for race detection
  – Does not modify cache state or coherence
• Catches more bugs than current detectors
  – 29% more static races
  – 150% more dynamic races
Questions?
Discussion Questions

1. In a fine-grained locking program which will preform better SigRace or a cache based detectors?

2. Can SigRace be used to tolerate data race?

3. Is it justifiable to have a higher instruction overhead do to false positives?
Extra: Re-execution

Each thread executes one epoch at a time

TRT = Local view
holds the TS of the epoch

**next** to be executed

GRT = Global view
Last epoch each thread has executed

Next thread obeys:
grt[i] > trtp[i], i ≠ p

Thread Re-Execution Timestamp (TRT)

Global Re-Execution Timestamp (GRT)