Cache Guided Deterministic Replay for Concurrent Java Programs
What is it about?

- Concurrent programs are difficult to debug.
- Use deterministic replay
- Search-based
  - Small log, small record cost, incomplete
  - Best-effort exhaustive state space search
- Order-based
  - Record dependences among key events: R/W, (un)lock
  - Huge logs: STRIDE 30MB/s → Performance degradation
  - Easy replay
CARE

- **Cache guided deterministic replay**
- Key Idea: Take advantage of thread locality
  - i.e. no need to record access to same variable by same thread twice.
- Only record dependencies among different Threads
- Cache miss → other thread did access same variable
- Cache miss detected by value prediction cache
Value Prediction Cache

T1
x = 1;
y = 2;
read x;

VPC:
\[ x = 1 \]
\[ y = 2 \]

Real cache:
\[ x = 1 \]
\[ y = 2 \]

T2
z = 2;

VPC:
\[ z = 2 \]

Real cache:
\[ z = 2 \]
Value Prediction Cache

T1
x = 1;
y = 2;
read x;

<table>
<thead>
<tr>
<th>VPC:</th>
<th>Real cache:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>x = Invalid</td>
</tr>
<tr>
<td>y = 2</td>
<td>y = 2</td>
</tr>
</tbody>
</table>

T2
x = 2;

VPC:  x = 2
Real cache: x = 2
Value Prediction Cache

T1
x = 1;
y = 2;

read x;

VPC:
x = 1
y = 2

Real cache:
x = 2
y = 2

T2
x = 2;

VPC:
x = 2

Real cache:
x = 2

Cache miss detected
### Value Prediction Cache

**T1**
```
x = 1;
y = 2;
read x;
```

<table>
<thead>
<tr>
<th>VPC:</th>
<th>Real cache:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>x = invalid</td>
</tr>
<tr>
<td>y = 2</td>
<td>y = 2</td>
</tr>
</tbody>
</table>

**T2**
```
x = 1;
```

<table>
<thead>
<tr>
<th>VPC:</th>
<th>Real cache:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>x = 1</td>
</tr>
</tbody>
</table>
Value Prediction Cache

T1
x = 1;
y = 2;
read x;

VPC:
x = 1
y = 2

Real cache:
x = 1
y = 2

Cache miss not detected

T2
x = 1;

VPC:
x = 1

Real cache:
x = 1
Figure 1: Illustration of missing dependences

Figure 2: Illustration of refetching
Algorithm 1: read

\[d \leftarrow \text{heap}(v)\]
\[\text{if } \text{cache}(v) \neq d \text{ then} \]
\[\text{synchronized } v\]
\[d \leftarrow \text{heap}(v)\]
\[H \leftarrow H \cup (\text{last}(v), r)\]
\[G \leftarrow G \cup \{r\}\]
\[\text{last}(v) \leftarrow r\]
\[\text{cache}(v) \leftarrow d\]

G: set of read actions with cache miss
H: inter thread dependences
r = \langle \text{tid}, \text{read}, v, \text{uniqueId} \rangle
Algorithm 2: write

```java
synchronized v
    heap(v) ← d
    if last(v).t ≠ t then
        H ← H ∪ (last(v), w)
        last(v) ← w
    last(v) ← w
    cache(v) ← d

G: set of read actions with cache miss
H: inter thread dependences
w = <t, write, v, uniqueId>
```
Algorithm 3: lock

acquire(v)

if last(v).t ≠ t then
  H ← H ∪ (last(v), l)
  last(v) ← w

G: set of read actions with cache miss
H: inter thread dependences
w = <t, acquire, v, uniqueld>
Cache Organization

- **Big cache**
  - Less unnecessary cache misses → smaller log
  - Disables the garbage collection mechanism → drains memory

- **Small cache**
  - Unnecessary cache misses

- **Optimal cache**
  - Efficient updates and queries
  - Moderate memory consumption
  - High cache hit rate
(a) Study of runtime overhead against cache organization

(b) Study of log size against cache organization
Heuristics to still get SC replay

- Try to schedule read actions first
  - If desired value is inconsistent with the one in the heap suspend thread
  - Immediately after desired value is written resume
- Add sequence number to groups of variables
  - Sequence numbers define dependences between variables

---

**Figure 1: Illustration of missing dependences**
## Performance

Table 1: Comparison of CARE and LEAP under benchmark programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CARE</th>
<th>LEAP [18]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Overhead (x)</td>
<td>Log Size (/s)</td>
</tr>
<tr>
<td>Avrora</td>
<td>1.52</td>
<td>2.18MB</td>
</tr>
<tr>
<td>Batik</td>
<td>1.49</td>
<td>1.51KB</td>
</tr>
<tr>
<td>H2</td>
<td>18.5</td>
<td>24.2MB</td>
</tr>
<tr>
<td>Lusearch</td>
<td>3.41</td>
<td>6.53MB</td>
</tr>
<tr>
<td>Sunflow</td>
<td>64.9</td>
<td>886MB</td>
</tr>
<tr>
<td>Tomcat</td>
<td>4.76</td>
<td>7.80MB</td>
</tr>
<tr>
<td>Xalan</td>
<td>7.18</td>
<td>13.6MB</td>
</tr>
<tr>
<td>Tsp</td>
<td>2.79</td>
<td>1.84MB</td>
</tr>
<tr>
<td>Moldyn</td>
<td>11.9</td>
<td>24.1MB</td>
</tr>
</tbody>
</table>
**Performance**

Table 2: Comparison of CARE and Stride with normalized values

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CARE</th>
<th>Stride [37]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Overhead</td>
<td>Log Size</td>
</tr>
<tr>
<td>Avrora</td>
<td>16.0%</td>
<td>8.97%</td>
</tr>
<tr>
<td>Batik</td>
<td>39.5%</td>
<td>65.1%</td>
</tr>
<tr>
<td>H2</td>
<td>29.5%</td>
<td>88.3%</td>
</tr>
<tr>
<td>Lusearch</td>
<td>37.9%</td>
<td>14.2%</td>
</tr>
<tr>
<td>Sunflow</td>
<td>16.7%</td>
<td>14.7%</td>
</tr>
<tr>
<td>Tomcat</td>
<td>40.0%</td>
<td>33.2%</td>
</tr>
<tr>
<td>Xalan</td>
<td>59.2%</td>
<td>9.52%</td>
</tr>
<tr>
<td>Tsp</td>
<td>2.51%</td>
<td>0.32%</td>
</tr>
<tr>
<td>Moldyn</td>
<td>23.8%</td>
<td>7.94%</td>
</tr>
</tbody>
</table>
Conclusion

• CARE records only inter-thread dependencies
• Takes use of cache: cache miss = dependency
• Good performance: small log, low overhead