Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine

Michael Bond
Milind Kulkarni
Man Cao
Meisam Fathi Salmi
Jipeng Huang

Purdue
Microsoft

Ohio State
Nondeterminism is problematic

• Reproduce and Debug

• Replication for fault-tolerance

• Record and Replay!
  • RecPlay, M. Ronsse, et al, 1999
  • Respec, D. Lee, et al, 2010
  • DoublePlay, K. Veeraraghavan, et al, 2011
  • Chimera, D. Lee, et al, 2012
  • CLAP, J. Huang, et al, 2013
  And many others...
Record and Replay Types

- Offline
  - Debugging

- Online
  - Fault tolerance
  - Distribute dynamic analysis
Record and Replay Challenges

• Single-thread, *external* sources of nondeterminism
  • I/O, time, SysCall, etc.
  • Garbage collection, hash code
  • Adaptive compilation and dynamic classloading
    • Even harder for metacircular JVM (Jikes RVM)

• Multithreaded, *internal* nondeterminism
  • Thread interleaving
  • Hard and expensive to capture or control
Contributions

Handle both nondeterminisms

• External (VM, system, I/O)
  • Non-trivial engineering effort
  • Novel methodology to sidestep nondeterminisms
    • Fork-and-recompile

• Internal (multithreading)
  • Two dynamic analyses
    • RECORD
    • REPLAY
  • Low overhead
  • Fewer limitations
Handling internal nondeterminism
Easy case: data-race-free execution

T1
synchronized(m) {
    wr o.f
}
synchronized(m) {
    wr p.g
}

T2
synchronized(m) {
    rd o.f
    rd p.g
}
Hard case: executions with data races

• Unfortunately, most real-world programs have data races
  • Instrument all potentially racy memory accesses to capture *cross-thread data dependences*
  • Add many synchronization operations, very expensive!
Cross-thread data dependences

Same cross-thread data dependences  Deterministic replay
Limitations of existing multithreaded record and replay approaches

• High overhead for recording cross-thread dependences
• OR do not handle racy execution
• OR support only offline or online replay, but not both
• OR rely on speculation and extra cores
• OR need custom hardware

*Our approach overcomes all of these limitations at the same time!*
RECORD

• Builds on our prior work “Octet”
• Octet tracks cross-thread dependences at object granularity

• Each object has an ownership state
  • Analogous to cache coherence protocol

• For simplicity, consider
  \[ o\text{.state} \in \{ \text{Wr}_T, \text{Rd}_T \} \]
• Cross-thread dependence => state transition
State transition example

Initially o.state = $\text{Wr}_{T_1}$, p.state = $\text{Rd}_{T_2}$
State transition example

T1
- write check
- wr o.f

T2
- read check

Wr_{T1}
State transition example

- **T1**:
  - Write check
  - Safe point

- **T2**:
  - Read check
  - Safe point

**Time**: $T_1 \xrightarrow{wr_{T_1}} safe\ point \xrightarrow{rd_{T_2}} T_2$
State transition example

- \( T_1 \):
  - Write check
  - \( wr_{T_1} \)
  - Safe point

- \( T_2 \):
  - Read check
  - \( rd_{T_2} \)

How to record this cross-thread dependence?
State transition example

How to record this cross-thread dependence?
RECORD Design

• What?
  • Response edge
  • Per-thread response counter

• Where in execution?
  • Dynamic Program Location (DPL)

• Per-thread log file
RECORD example

T1
write check
wr o.f
safe point
T2
read check
rd o.f

T1.resp++
T1.record (DPL, RESPONSE)

T2.record (DPL, REQUEST, T1, T1.resp)
RECORD example

T1.resp++
T1.record (DPL, RESPONSE)

T1.record (DPL, REQUEST, T2, T2.resp)

T2.resp++
T2.record (DPL, RESPONSE)
RECORD observation

• Adds low overhead
  • Most accesses (>99%) are same-state
  • Only one load and one if check
REPLAY

• Goal: enforce recorded edges

• Instrument every possible edge source and sink
  • Check if DPL matches

• Edge source
  • Increment counter

• Edge sink
  • Wait for counter to reach recorded value
REPLAY example

Time

DPL ✗
T1
- replay check
- wr o.f

DPL ✓
T2
- replay check
- rd o.f
REPLAY example

T1
- replay check
- wr o.f

T2
- replay check
- rd o.f

while(T1.resp < recorded_T1_resp) {
    mem_fence
}

DPL ✅ while(T1.resp < recorded_T1_resp) {
    mem_fence
}
mem_fence
REPLAY example

```
while(T1.resp < recorded_T1_resp) {
    mem_fence
} mem_fence
```
REPLAY example

DPL  
mem_fence  
T1.resp++

DPL

while(T1.resp < recorded_T1_resp) {
    mem_fence
}
mem_fence
REPLAY example

while (T1.resp < recorded_T1_resp) {
  mem_fence
}

while (T2.resp < recorded_T2_resp) {
  mem_fence
}

DPL

while (T1.resp < recorded_T1_resp) {
  mem_fence
}
mem_fence

DPL

while (T2.resp < recorded_T2_resp) {
  mem_fence
}
mem_fence

DPL

DPL
REPLAY example

DPL ×
while(T2.resp < recorded_T2_resp) {
    mem_fence
}
mem_fence

DPL ✓
while(T1.resp < recorded_T1_resp) {
    mem_fence
}
A more complicated case

- **T1**: Write check
- **T2**: Read check
- **T3**: Read check
- **T4**: Read check

Events:
- **wr o.f**: write operation finishes
- **rd o.f**: read operation finishes
- **safe point**: safe point
- **read check**: read check
- **write check**: write check
- **RdSh**: read shared
- **Wr**: write
- **Rd**: read

Diagram:
- T1: Write check
  - wr o.f
  - Safe point
- T2: Read check
  - rd o.f
- T3: Read check
  - rd o.f
- T4: Read check
  - rd o.f

Events:
- Wr\(_T1\)
- Rd\(_T2\)
- RdSh
- RdSh
REPLAY observations

• Do not track object’s state
  • Only per-thread or global counters
• How about program synchronization?

RECORD

```
T1
synchronized(m){
o.f = ...;
...
// safepoint
}
```

REPLAY

```
T1
synchronized(m){
o.f = ...;
...
// safepoint
}
```

T2
```
synchronized(m){
  ...
  // wait for T1
  ... = o.f;
}
```

REPLAY

```
T2
synchronized(m){
  // wait for T1
  ...
  ... = o.f;
}
```

Deadlock!
Elide program synchronization

• Necessary
  • RECORD does not track synchronization
  • Otherwise deadlock

• Side effect: more parallelism (see Evaluation)
Handling external nondeterminism
Goal: Application-level determinism

• No need to track JVM’s cross-thread dependences!
  • Jikes RVM

• Contexts for compiled code
External nondeterminisms

• Handled nondeterminisms
  • Stop-the-world GC, record and replay DPLs of GC points
  • Deterministic hash code
  • Deterministic “logical time”
  • Deterministic I/O
  • etc.

• Adaptive compilation and dynamic classloading
  • Most challenging (esp. in Jikes)!
  • Fork-and-recompile
Fork-and-recompile
Evaluation

• Implementation in Jikes RVM
  • Publicly available (http://sourceforge.net/p/jikesrvm/research-archive/49/)
• DaCapo 2006 & 2009, SPEC JBB 2000 & 2005
• 64 cores (AMD Opteron 6272)
## REPLAY Efficacy

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>REPLAY Success Rate</th>
<th>Default w/ value logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb6</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>lusearch6</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>xalan6</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>avrora9</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>jython9</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>luindex9</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>lusearch9</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>pmd9</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>sunflow9</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>xalan9</td>
<td>60%</td>
<td></td>
</tr>
<tr>
<td>pjbb2000</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>pjbb2005</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>
REPLAY Efficacy

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>REPLAY Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Default w/ value logging</td>
</tr>
<tr>
<td>hsqldb6</td>
<td>100%</td>
</tr>
<tr>
<td>lusearch6</td>
<td>100%</td>
</tr>
<tr>
<td>xalan6</td>
<td>100%</td>
</tr>
<tr>
<td>avrora9</td>
<td>100%</td>
</tr>
<tr>
<td>jython9</td>
<td>100%</td>
</tr>
<tr>
<td>luindex9</td>
<td>100%</td>
</tr>
<tr>
<td>lusearch9</td>
<td>100%</td>
</tr>
<tr>
<td>pmd9</td>
<td>100%</td>
</tr>
<tr>
<td>sunflow9</td>
<td>100%</td>
</tr>
<tr>
<td>xalan9</td>
<td>60%</td>
</tr>
<tr>
<td>pjbb2000</td>
<td>100%</td>
</tr>
<tr>
<td>pjbb2005</td>
<td>100%</td>
</tr>
</tbody>
</table>
## RECORD logging throughput

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RECORD Logging MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb6</td>
<td>0.7</td>
</tr>
<tr>
<td>lusearch6</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>xalan6</td>
<td>7.7</td>
</tr>
<tr>
<td>avrora9</td>
<td>2.5</td>
</tr>
<tr>
<td>jython9</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>luindex9</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>lusearch9</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>pmd9</td>
<td>0.1</td>
</tr>
<tr>
<td>sunflow9</td>
<td>0.1</td>
</tr>
<tr>
<td>xalan9</td>
<td>9.7</td>
</tr>
<tr>
<td>pjbb2000</td>
<td>1.1</td>
</tr>
<tr>
<td>pjbb2005</td>
<td>4.8</td>
</tr>
<tr>
<td>Benchmark</td>
<td>RECORD Logging MB/s</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>hsqldb6</td>
<td>0.7</td>
</tr>
<tr>
<td>lusearch6</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>xalan6</td>
<td>7.7</td>
</tr>
<tr>
<td>avrora9</td>
<td>2.5</td>
</tr>
<tr>
<td>jython9</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>luindex9</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>lusearch9</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>pmd9</td>
<td>0.1</td>
</tr>
<tr>
<td>sunflow9</td>
<td>0.1</td>
</tr>
<tr>
<td>xalan9</td>
<td>9.7</td>
</tr>
<tr>
<td>pjbb2000</td>
<td>1.1</td>
</tr>
<tr>
<td>pjbb2005</td>
<td>4.8</td>
</tr>
</tbody>
</table>
Conclusion

• Handle both external and internal nondeterminisms
  • Metacircular JVM
  • fork-and-recompile

• Efficient record and replay
  • Low overhead in RECORD
  • More parallelism in REPLAY

• Overcome many limitations simultaneously
  • Online/offline, software-only, no speculation, etc.