ORDER: Object centRic DEterministic Replay for Java

ZheMin Yang, Min Yang, Lvcai Xu, Haibo Chen and Binyu Zang
Parallel Processing Institute, Fudan University

2011 USENIX Annual Technical Conference (USENIX ATC’11)
Debugging

Buggy Execution

T1
T2
T3
T4

Bug
Crash

Run again...

T1
T2
T3
T4

Normal Run
Deterministic Replay

**Record Mode**

- T1: Checkpoint A
- T2: Checkpoint B
- T3: Checkpoint C
- T4: Checkpoint A

**Replay Mode**

- Replaying from log B, C
- Read Checkpoint B

Crash
State-of-the-art

Mostly focus on native systems

Address-based dependency tracking
Special hardware support (FDR ISCA’03, Bugnet ISCA’05, Lreplay ISCA’10, etc.)
Software approach: large overhead, inscalable (SMP-Revirt, VEE’07, etc.)

Replay for managed runtime
Not counting data race (JaRec, SPE’04)
Not cover external dependency, large overhead (Leap, FSE’10)
Not cover non-determinism inside managed runtime
Contribution

Key observations
  False positive in garbage collection
  Access locality in object level

ORDER
  Record and replay at object-level
    Eliminate false positive in GC
    Good locality and less contention
    Scalable performance (108% for JRuby, SpecJBB, SPECJVM)
  Cover more non-determinisms than before
    Good bug reproducibility
Outline

Why object centric deterministic replay?

Recording object access timeline

Non-determinism mitigated

Optimizations

Evaluation Result
Java Runtime Behavior

Garbage Collection
  Movement of object is quite often

Object-oriented design
  Inherently good access locality
Address-based dependency tracking

• Ordering shared memory accesses:
  – Two instructions are tracked if:
    1) They both access the same memory
    2) At least one of them is a write
    3) They are operated in different threads
Dependences Introduced by GC

- Write operations in GC introduce dependencies...
  - Two instructions are tracked if:
    1) They both access the same memory
       GC operates on the same heap space as the original application
    2) At least one of them is a write
       Huge write operations in GC
    3) They are operated in different threads
       GC threads are always different from Java threads
Dependencies Introduced by GC

- They **DO** affect the address-based dependency tracking system
  - Root cause: object movement
  - So they can not be ignored
False Positives by GC

8X more dependency by GC

16-core
16-threads
Interleaving of Object Accesses

Java programs are commonly designed around objects

Objects accessed by a thread are very likely to be accessed by the same thread soon
## Interleaving of Object Accesses

**Object level interleaving rate: All less than 7%!**

<table>
<thead>
<tr>
<th>Case</th>
<th>Interleaving</th>
<th>Access</th>
<th>Rate(%)</th>
<th>Case</th>
<th>Interleaving</th>
<th>Access</th>
<th>Rate(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler.compiler</td>
<td>53997073</td>
<td>3678311937</td>
<td>1.46</td>
<td>compress</td>
<td>448683851</td>
<td>34015732971</td>
<td>1.31</td>
</tr>
<tr>
<td>comiler.sunflow</td>
<td>159104781</td>
<td>7589140476</td>
<td>2.09</td>
<td>crypto.aes</td>
<td>3725080365</td>
<td>59999629461</td>
<td>6.21</td>
</tr>
<tr>
<td>fft.small</td>
<td>6281</td>
<td>12283085730</td>
<td>&lt;0.01</td>
<td>crypto.rsa</td>
<td>135072884</td>
<td>21917377595</td>
<td>0.62</td>
</tr>
<tr>
<td>fft.large</td>
<td>3447</td>
<td>16312951356</td>
<td>&lt;0.01</td>
<td>crypto.signverify</td>
<td>33185584</td>
<td>23327050394</td>
<td>0.14</td>
</tr>
<tr>
<td>lu.small</td>
<td>6500</td>
<td>34325013828</td>
<td>&lt;0.01</td>
<td>derby</td>
<td>2444646763</td>
<td>49325408866</td>
<td>4.95</td>
</tr>
<tr>
<td>lu.large</td>
<td>3311</td>
<td>277302000000</td>
<td>&lt;0.01</td>
<td>mpegaudio</td>
<td>922855001</td>
<td>63774976691</td>
<td>1.45</td>
</tr>
<tr>
<td>sor.small</td>
<td>4446</td>
<td>24581389638</td>
<td>&lt;0.01</td>
<td>serial</td>
<td>315661230</td>
<td>17466253036</td>
<td>1.80</td>
</tr>
<tr>
<td>sor.large</td>
<td>3358</td>
<td>104319000000</td>
<td>&lt;0.01</td>
<td>xml.validation</td>
<td>96681920</td>
<td>6296521288</td>
<td>1.53</td>
</tr>
<tr>
<td>sparse.small</td>
<td>4201</td>
<td>29899769674</td>
<td>&lt;0.01</td>
<td>xml.transform</td>
<td>1409648652</td>
<td>65924269984</td>
<td>2.13</td>
</tr>
<tr>
<td>sparse.large</td>
<td>3055</td>
<td>104576000000</td>
<td>&lt;0.01</td>
<td>SPECjbb2005</td>
<td>78856923</td>
<td>1.88456E+15</td>
<td>&lt;0.01</td>
</tr>
<tr>
<td>monte.carlo</td>
<td>3503</td>
<td>96019240895</td>
<td>&lt;0.01</td>
<td>JRuby</td>
<td>161801036</td>
<td>1.34541E+12</td>
<td>0.01</td>
</tr>
</tbody>
</table>
Object Centric Deterministic Replay

Reveal new granularity: object
  Reduction of GC dependencies
  Reduced contention of synchronization
  Improved locality
Outline

Why Object centric deterministic replay?

Recording object access timeline

Non-determinism mitigated

Optimizations

Evaluation Result
Design of ORDER

Dynamic Instrumentation in Java compilation pipeline
Handle dynamic loaded library and external code by default

Extend object header with accessing information
Object identifier (OI)
Accessing thread identifier (AT)
Access counter (AC)
Object level lock
Read-write flag
Recording Object Access Timeline

Thread 1(t1)  Thread 2(t2)
1: if(entry.klass.get()==this
   && name.equals(entry.name))
   3: entry.method=...
2: entry.method.get()
Recording Timeline

Thread 1(t1)
1: if(entry.klass.get()==this && name.equals(entry.name))
2: entry.method.get()
3: entry.method=...

Thread 2(t2)

CTID: -
timeline (entry)

header
AT: 1
AC: 1

(t1,2)
(t2,1)
(t1,1)

After Garbage Collection
Replaying timeline

Thread 1(t1)  Thread 2(t2)
1: if(entry.klass.get()==this
    && name.equals(entry.name))

2: entry.method.get()

3: entry.method=...

2: entry.method.get()

CTID: 1
header
AT: 1
AC: 0

(t1,1)

Inconsistent
Outline

Why Object centric deterministic replay?

Recording object access timeline

Non-determinism mitigated

Optimizations

Evaluation Result
Handling Non-determinisms

Interleaved object accesses
Lock acquirement
Garbage collection

Recording object access timeline
Recording interfaces between GC/Java threads

In paper:
- Signal
- Program Input
- Library invocation
- Configuration of OS/JVM
- Adaptive Compilation
- Class Initialization
Outline

Why Object centric deterministic replay?

Recording object access timeline

Non-determinism mitigated

Optimizations

Evaluation Result
Opt: Unnecessary Timeline Recording

Thread-local objects
   Identified by Escape Analysis [OOPSLA’99]

Assigned-once objects
   Continuous write operations during initialization
   After initialization, no thread will write to the fields of these objects
   Identified by modifying the Escape Analysis
Outline

Why Object centric deterministic replay?

Recording object access timeline

Non-determinism mitigated

Optimizations

Evaluation Result
Evaluation Environments

Implemented in Apache Harmony
   By modifying the compilation pipeline

Machine setup
   16-core Xeon machine (1.6GHz, 32G Memory)
   Linux 2.6.26

Benchmarks
   SPECjvm2008, Pseudojbb2005, JRuby
Evaluation Questions

How much overhead ORDER incurs in record and replay?
• How does it compare to the state-of-the-art?

How large is the log size?

How about the bug reproducibility?
Evaluation Results: Record Slowdown

About 2x slowdown, overhead most comes from tracing timeline in memory
Record slowdown (compared to LEAP)

1.5x to 3x faster than LEAP

ORDER records more non-determinism
Scalability (Record Phase)

(from 1 thread to 16 threads)

Almost scalable
Replay Slowdown

(from 1 thread to 16 threads)
## Log size

<table>
<thead>
<tr>
<th>Case</th>
<th>Log Size (timeline)</th>
<th>Log Size (others)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler.compiler</td>
<td>88(m/h)</td>
<td>35(m/h)</td>
</tr>
<tr>
<td>compiler.sunflow</td>
<td>61(m/h)</td>
<td>58(m/h)</td>
</tr>
<tr>
<td>scimark.fft.small</td>
<td>0.60(m/h)</td>
<td>10(m/h)</td>
</tr>
<tr>
<td>scimark.fft.large</td>
<td>0.47(m/h)</td>
<td>7(m/h)</td>
</tr>
<tr>
<td>scimark.lu.small</td>
<td>0.37(m/h)</td>
<td>6(m/h)</td>
</tr>
<tr>
<td>scimark.lu.large</td>
<td>0.35(m/h)</td>
<td>5(m/h)</td>
</tr>
<tr>
<td>scimark.sor.small</td>
<td>2(m/h)</td>
<td>40(m/h)</td>
</tr>
<tr>
<td>scimark.sor.large</td>
<td>0.68(m/h)</td>
<td>11(m/h)</td>
</tr>
<tr>
<td>scimark.sparse.small</td>
<td>2(m/h)</td>
<td>36(m/h)</td>
</tr>
<tr>
<td>scimark.sparse.large</td>
<td>0.56(m/h)</td>
<td>10(m/h)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Log Size (timeline)</th>
<th>Log Size (others)</th>
</tr>
</thead>
<tbody>
<tr>
<td>scimark.monte-carlo</td>
<td>0.013(m/h)</td>
<td>0.22(m/h)</td>
</tr>
<tr>
<td>compress</td>
<td>4(m/h)</td>
<td>44(m/h)</td>
</tr>
<tr>
<td>crypto.aes</td>
<td>1.4(m/h)</td>
<td>9(m/h)</td>
</tr>
<tr>
<td>crypto.rsa</td>
<td>26(m/h)</td>
<td>6(m/h)</td>
</tr>
<tr>
<td>crypto.signverify</td>
<td>10(m/h)</td>
<td>8(m/h)</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>511(m/h)</td>
<td>2(m/h)</td>
</tr>
<tr>
<td>serial</td>
<td>1553(m/h)</td>
<td>121(m/h)</td>
</tr>
<tr>
<td>xml.validation</td>
<td>632(m/h)</td>
<td>31(m/h)</td>
</tr>
<tr>
<td>Pseudojbb</td>
<td>1085(m/h)</td>
<td>550(m/h)</td>
</tr>
<tr>
<td>JRuby</td>
<td>0.8(m/h)</td>
<td>170(m/h)</td>
</tr>
</tbody>
</table>
### Bug Reproducibility

Real-world concurrent bugs reproduced by ORDER. Each of them comes from open source communities and causes real-world buggy execution.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Category</th>
<th>Bug description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JRuby-931</td>
<td>atomic</td>
<td>Non-atomic traversing of container triggers ConcurrentModification-Exception.</td>
</tr>
<tr>
<td>JRuby-1382</td>
<td>atomic</td>
<td>Non-atomic read from memory cache causes system crash.</td>
</tr>
<tr>
<td>JRuby-2483</td>
<td>atomic</td>
<td>Concurrent bug caused by using thread unsafe library code.</td>
</tr>
<tr>
<td>JRuby-879</td>
<td>order</td>
<td>List threads before thread is registered causes non-deterministic result.</td>
</tr>
<tr>
<td>JRuby-2380</td>
<td>violation</td>
<td></td>
</tr>
<tr>
<td>JRuby-2545</td>
<td>dead lock</td>
<td>Lock on the same object twice causes deadlock.</td>
</tr>
</tbody>
</table>
Bug reproducibility(JRuby-2483)

Concurrent bug caused by thread unsafe library HashMap

Non-determinism in Library is also important

Some discussion before:

HashMap.get() can cause an infinite loop!

Jul 25th, 2005
by plightbe.

Yes, it is true. HashMap.get() can cause an infinite loop. Everyone I’ve talked to didn’t believe it either, but yet there it is — right in front of my very eyes. Now, before anyone jumps up and shouts that HashMap isn’t synchronized, I want to make it clear that I know that. In fact, here is the paragraph from the JavaDocs:

Note that this implementation is not synchronized. If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map.
Conclusion

Java Deterministic Replay is unique
Two observations on Java Runtime Behavior

Object centric deterministic replay
   Reveal new granularity: Object
   Cover more non-determinisms than before
   Record timeline

Performance
   About 108% performance slowdown, and scalable.
Thanks

ORDER

Object-centric Deterministic Replay for Java

Questions?

Parallel Processing Institute
http://ppi.fudan.edu.cn
Backup Slides
Comparison with Leap

LEAP uses static instrumentation
   Cannot reproduce concurrent bugs caused by external code
      such as libraries or class files dynamically loaded during runtime.

LEAP does not distinguish between instances of the same type
   may lead to large performance overhead when a class is massively instantiated
In dependency-based replay, 2→3 or 3→2 is normally recorded. Shared-memory(entry.method) is accessed in both 2 and 3. One of them (instruction 3) is a write.
Use soot to annotate such objects offline
  Reduce record/replay overhead as well as log size
  Static analysis is imprecise, so further log reduction is necessary

Use a log compressor to eliminate the remaining thread local/assigned once objects after recording
  – Used to reduce replay overhead as well as log size
Handling Other Non-Det (1/2)

Signal
- Usually wrapped to wait, notify, and interrupt operations for thread
- Records return values and status of the pending queue

Program Input
- Log the content of input

Library invocation
- E.g., `System.currentTimeMillis()`, `Random/SecureRandom` classes
- Logs return values of these methods
Handling Other Non-Det (2/2)

Configuration of OS/JVM
records the configuration of OS/JVM

Class Initialization
Records initialization thread identifier
Forces same thread initialize same class in replay

Adaptive Compilation
Not supported yet, can be done similarly as Ogata et al.
OOPLSA’2006