ORDER: Object centRic DEterministic Replay for Java

ZheMin Yang, Min Yang, Lvcai Xu, Haibo Chen and Binyu Zang

Parallel Processing Institute, Fudan University

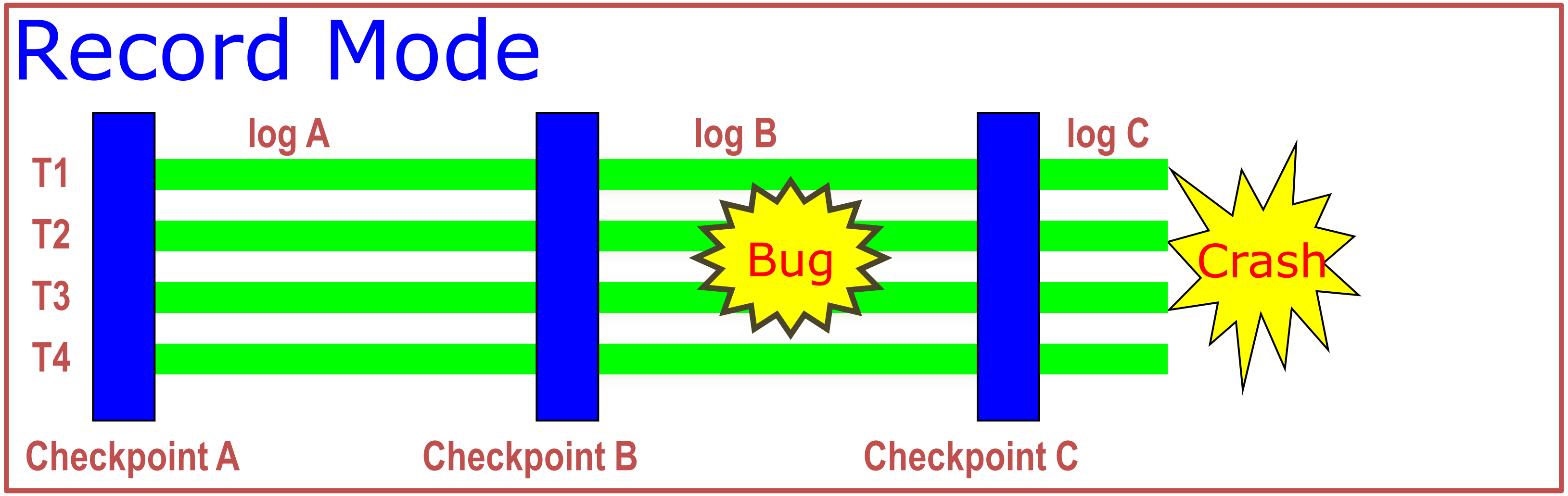
2011 USENIX Annual Technical Conference (USENIX ATC'11)

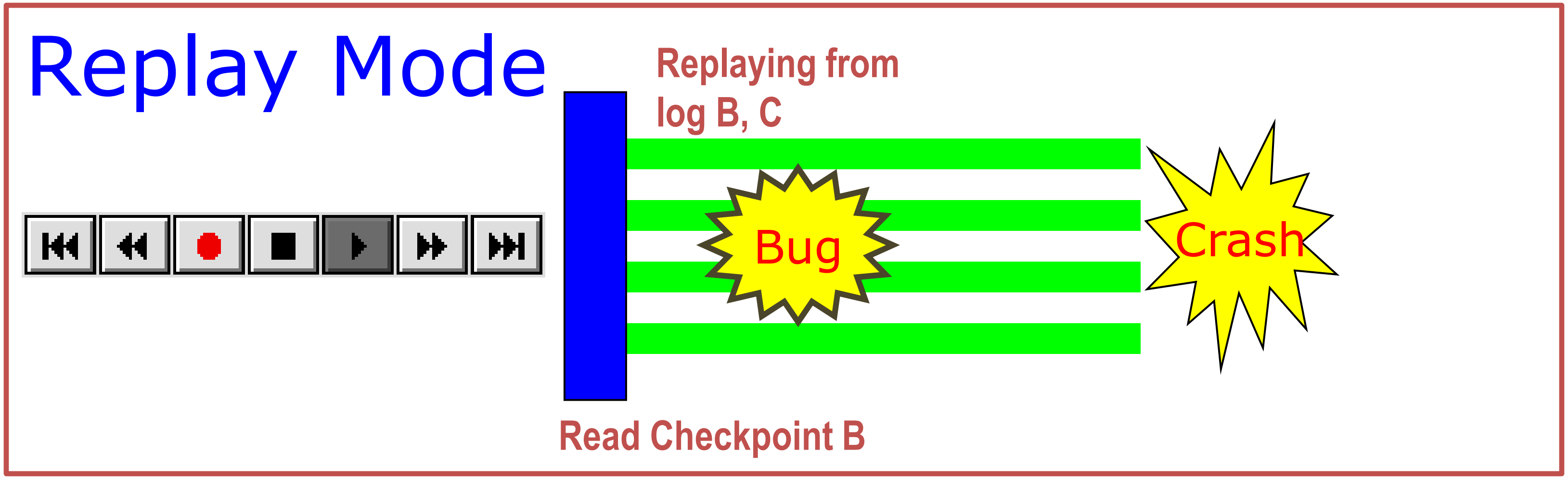
Debugging


Buggy Execution
T1
T2
T3
T4
Bug
Crash

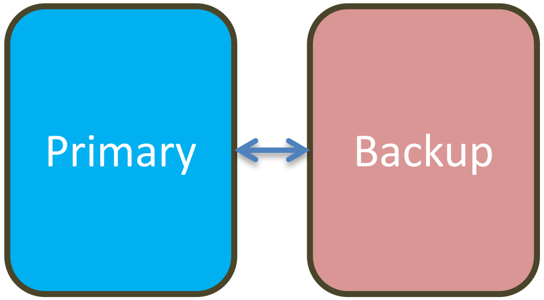

Run again…
T1
T2
T3
T4
Normal
Run

Deterministic Replay





FAULT-TOLERANCE



PROGRAM ANALYSIS

Deterministic Replay

DEBUGGING

INTRUSION ANALYSIS

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

Dependencies 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

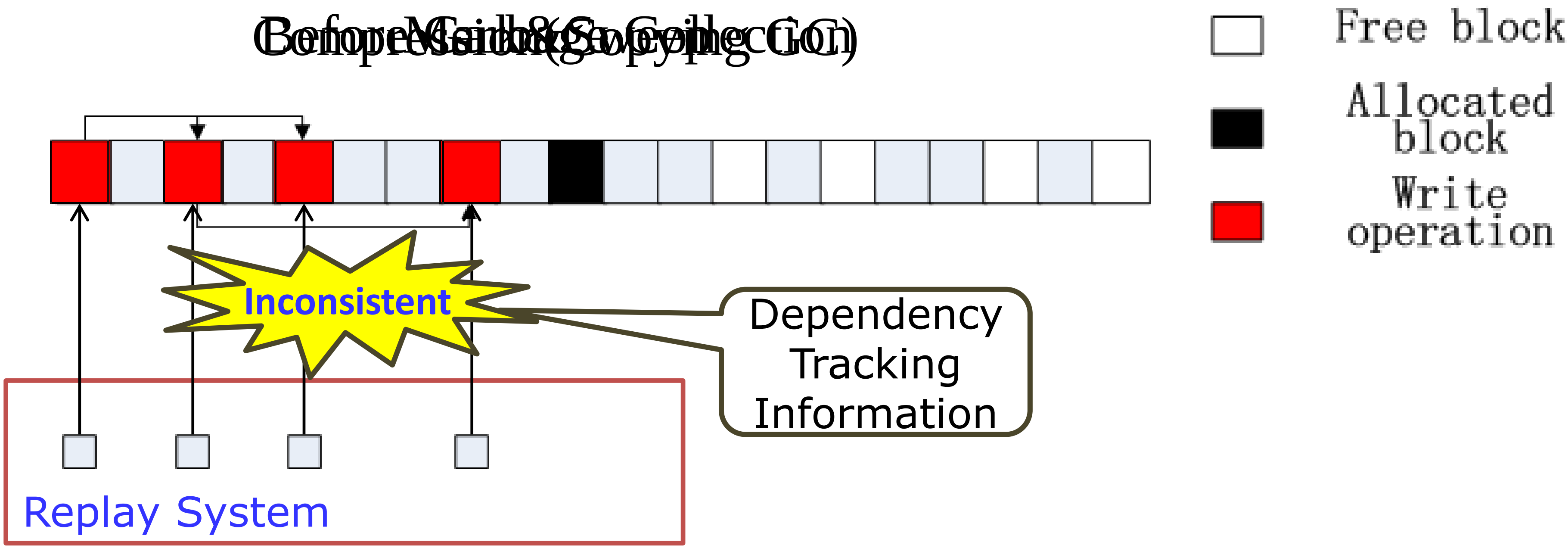
       Huge write operations in GC

    3) They are operated in different threads

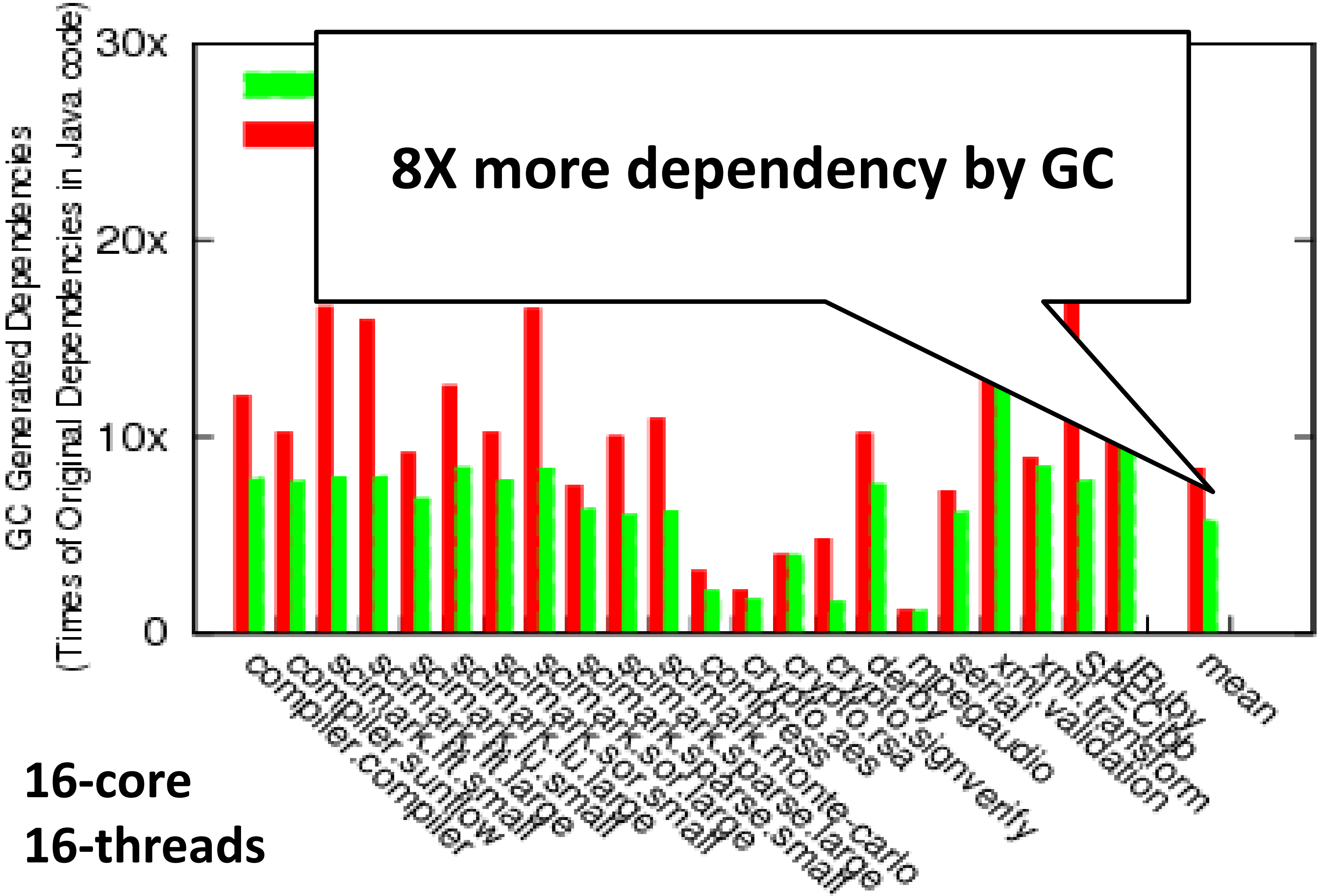
       GC threads are always different from Java threads

Dependencies Introduced by GC

- They **<u>DO</u>** affect the address-based dependency tracking system
  - Root cause: object movement
  - So they can not be ignored



False Positives 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

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

Object level interleaving rate: All less than 7%!

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

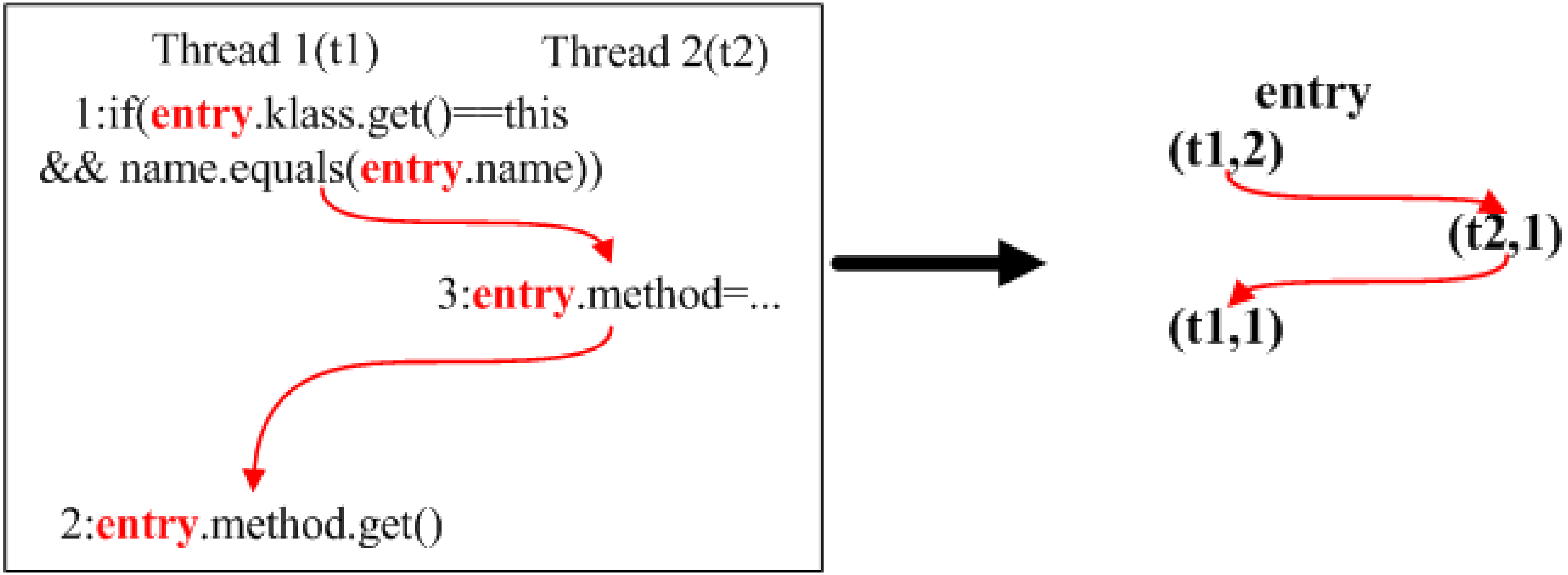
Optimizations

Evaluation Result

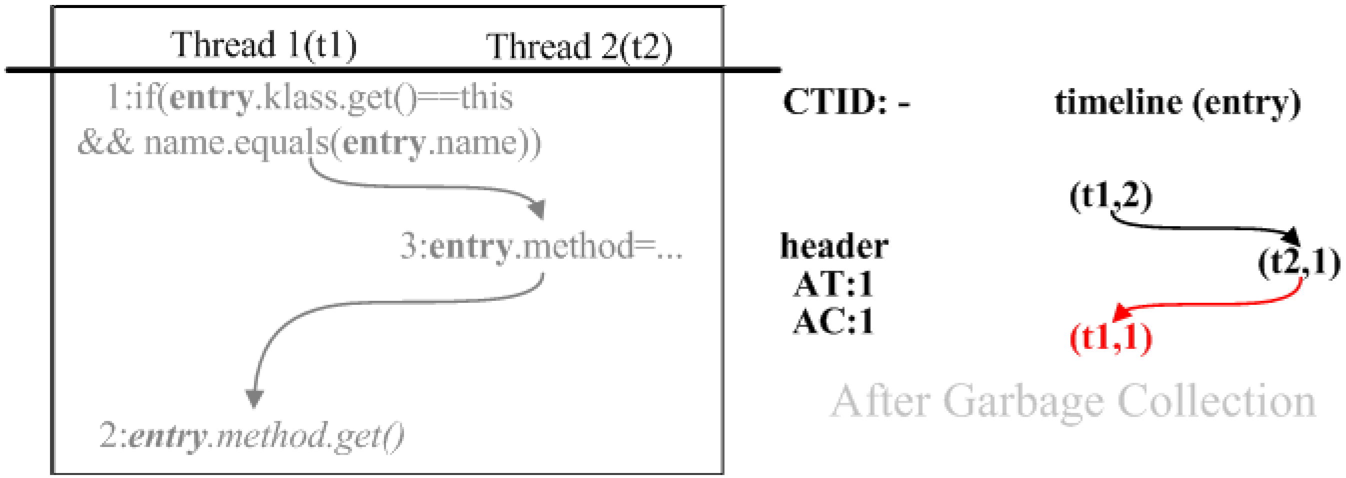
Design of ORDER

- 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

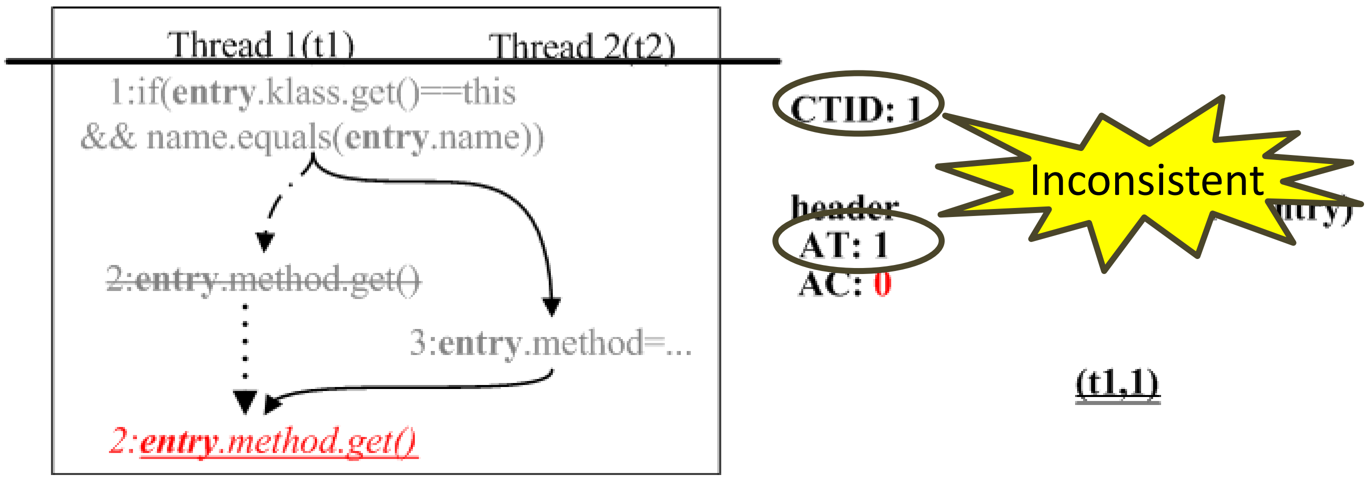
Recording Object Access Timeline



Recording Timeline



Replaying timeline



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 (Interleaved object accesses, Lock acquirement)

Recording interfaces between GC/Java threads (Garbage collection)

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?

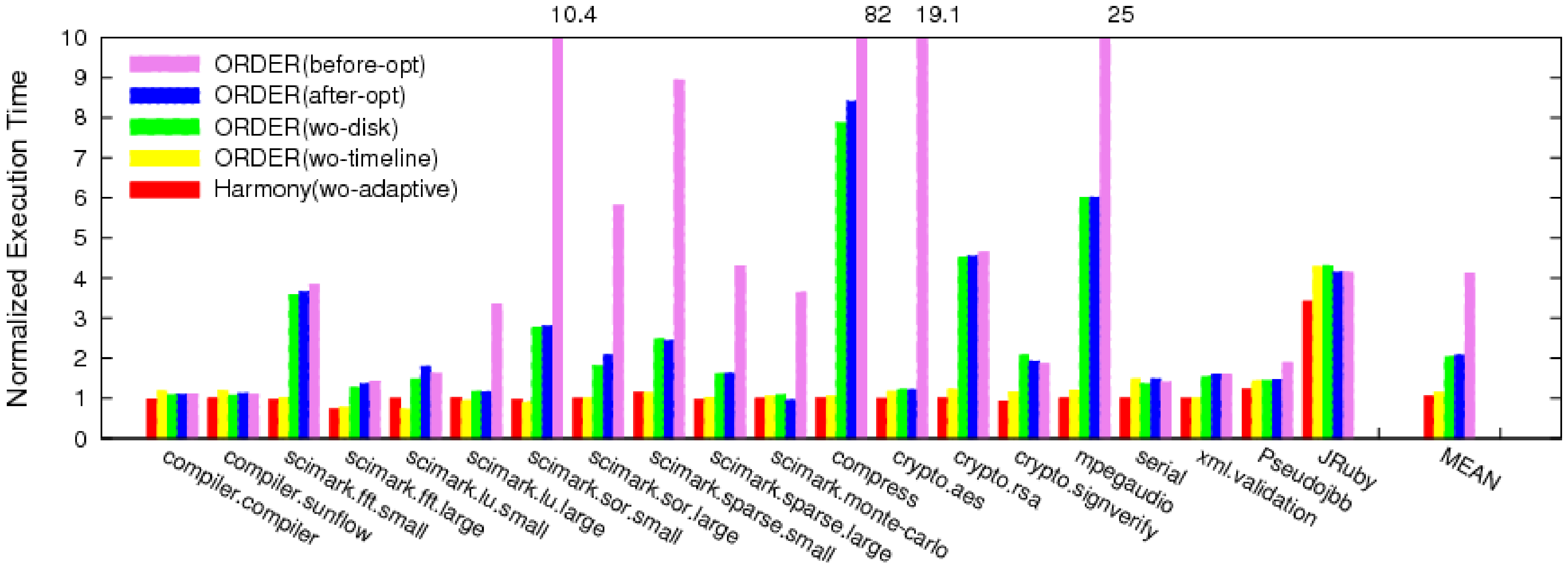
- How does it compare to the state-of-the-art?

How large is the log size?

How about the bug reproducibility?

Evaluation Results: Record Slowdown

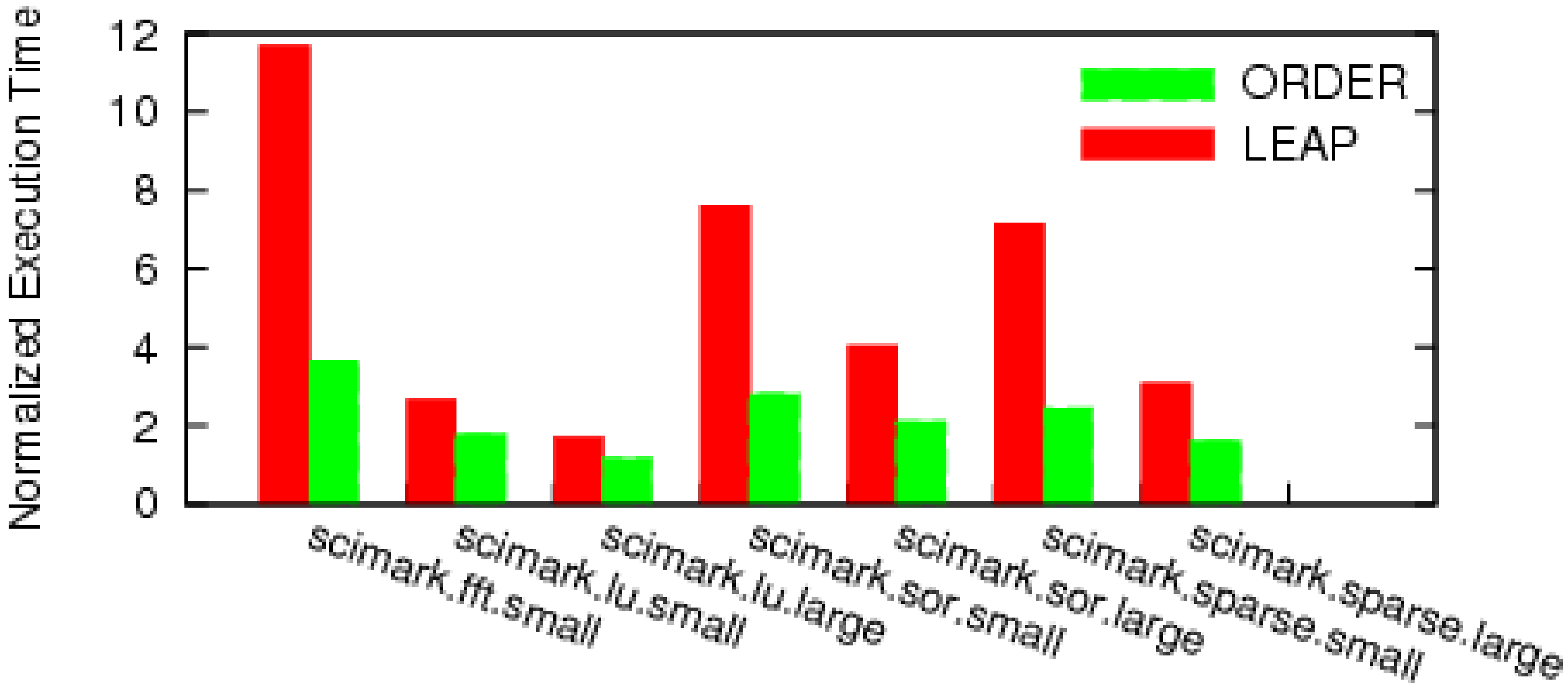
16-threads



About 2x slowdown, overhead most comes from tracing timeline in memory

Record slowdown(compared to LEAP)

16-threads

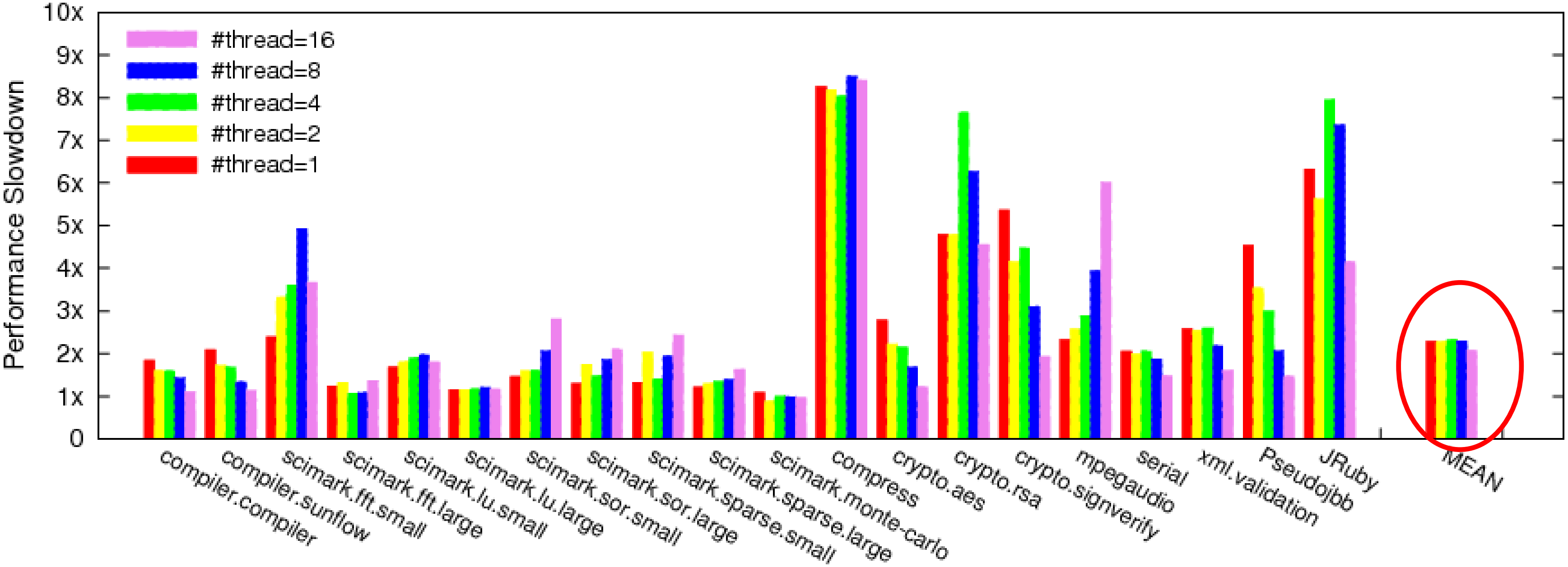


1.5x to 3x faster than LEAP

ORDER records more non-determinism

Scalability(Record Phase)

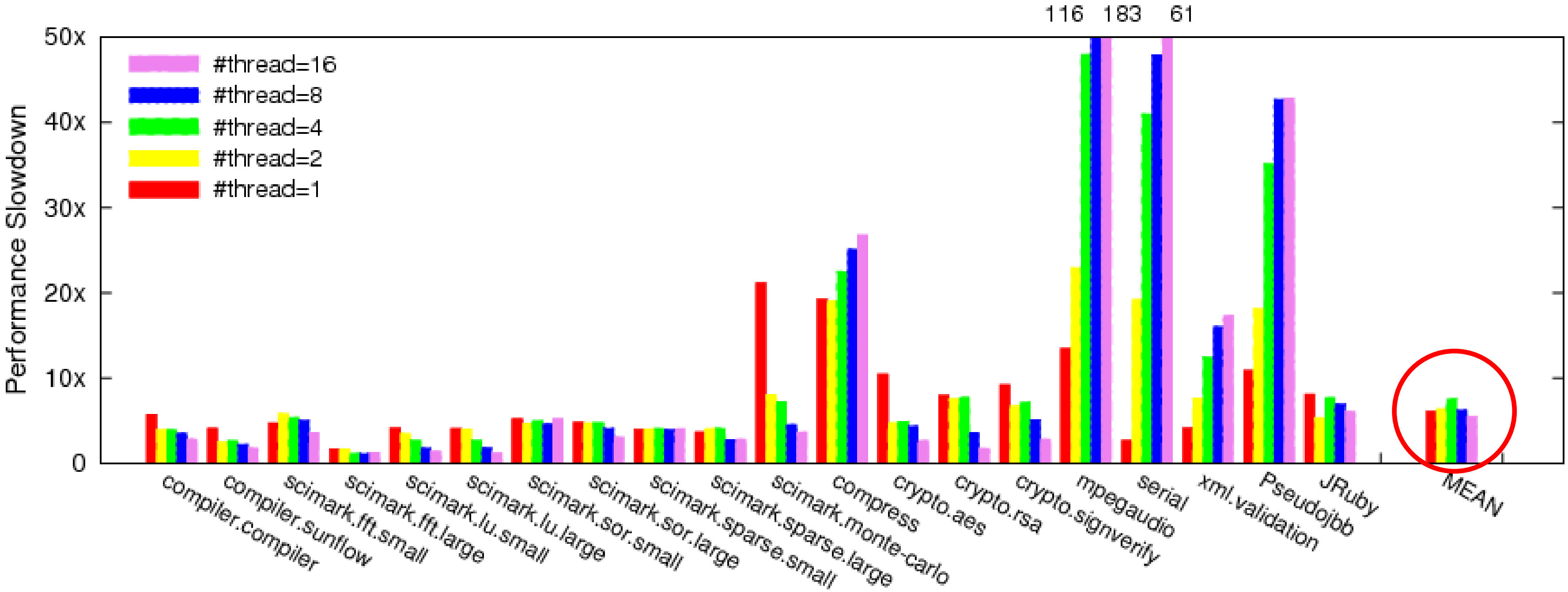
(from 1 thread to 16 threads)



Almost scalable

Replay Slowdown

(from 1 thread to 16 threads)



Log size

Case	Log Size (timeline)	Log Size (others)	Case	Log Size (timeline)	Log Size (others)
compiler.compiler	88(m/h)	35(m/h)	scimark.monte-carlo	0.013(m/h)	0.22(m/h)
compiler.sunflow	61(m/h)	58(m/h)	compress	4(m/h)	44(m/h)
scimark.fft.small	0.60(m/h)	10(m/h)	crypto.aes	1.4(m/h)	9(m/h)
scimark.fft.large	0.47(m/h)	7(m/h)	crypto.rsa	26(m/h)	6(m/h)
scimark.lu.small	0.37(m/h)	6(m/h)	crypto.signverify	10(m/h)	8(m/h)
scimark.lu.large	0.35(m/h)	5(m/h)	mpegaudio	511(m/h)	2(m/h)
scimark.sor.small	2(m/h)	40(m/h)	serial	1553(m/h)	121(m/h)
scimark.sor.large	0.68(m/h)	11(m/h)	xml.validation	632(m/h)	31(m/h)
scimark.sparse.small	2(m/h)	36(m/h)	Pseudojbb	1085(m/h)	550(m/h)
scimark.sparse.large	0.56(m/h)	10(m/h)	JRuby	0.8(m/h)	170(m/h)

Bug Reproducibility

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

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

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 plightbo.

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

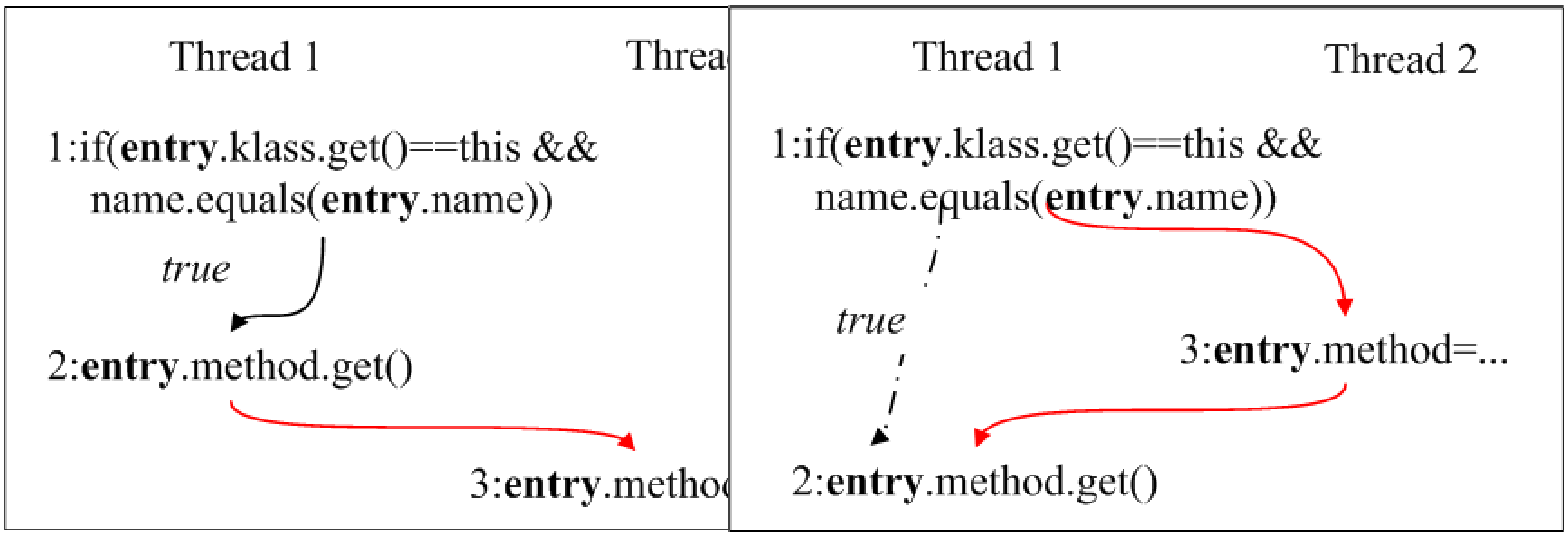
Dependency-based Deterministic Replay: JRuby

Whether 1->3 is recorded depends on:

Whether 1 and 3 access a shared memory

Depends on the record granularity

Correct



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 write

Opt: Unnecessary Timeline Recording

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.getCurrentTimeMillis(), 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