

# ORDER: Object centRiC DEterministic Replay for Java

Zheming Yang  
*Fudan University*

Min Yang  
*Fudan University*

Lvcai Xu  
*Fudan University*

Haibo Chen  
*Fudan University*

Binyu Zang  
*Fudan University*

## Abstract

Deterministic replay systems, which record and replay non-deterministic events during program execution, have many applications such as bug diagnosis, intrusion analysis and fault tolerance. It is well understood how to replay native (e.g., C) programs on multi-processors, while there is little work for concurrent java applications on multicore. State-of-the-art work for Java either assumes data-race free execution, or relies on static instrumentation, which leads to missing some necessary non-deterministic events.

This paper proposes the ORDER framework to record and reproduce non-deterministic events inside Java virtual machine (JVM). Based on observations of good locality at object level for threads and frequent object movements due to garbage collection, ORDER records and replays non-deterministic data accesses by logging and enforcing the order in which threads access objects. This essentially eliminates unnecessary dependencies introduced by address changes of objects during garbage collection and enjoys good locality as well as less contention, which may result in scalable performance on multicore. Further, by dynamically instrumenting Java code in the JVM compilation pipeline, ORDER naturally covers non-determinism in dynamically loaded classes.

We have implemented ORDER based on Apache Harmony. Evaluation on SPECjvm2008, PseudoJBB2005, and JRuby shows that ORDER only incurs 108% performance overhead on average and scales well on a 16-core Xeon testbed. Evaluation with a real-world application, JRuby, shows that several real-world concurrency bugs can be successfully reproduced.

## 1 Introduction

*Deterministic replay* has many applications such as diagnosing (concurrency) bugs [4, 8, 12, 18, 25, 26, 29], facilitating fault tolerance [2], forensic analysis [11] and offloading heavyweight dynamic program analysis [10]. Essentially, it works by recording non-deterministic

events such as data access interleavings and interactions with external environments of a program during normal execution, and ensuring the same order of program execution by replaying the recorded events and enforcing constraints within the events. Currently, many deterministic replay systems for native code (e.g., C or C++ based programs) use a dependency-based approach that enforces the accessing order to a specific shared address at different granularities such as word [6], cache line [17, 22], or page [12, 16].

While the approaches to replaying native code have been studied extensively and relatively well understood, it is still unanswered question how to efficiently replay concurrent Java applications on multicore in a scalable and efficient way. Unlike native code, Java code usually needs to cooperate with the Java virtual machine (JVM) to achieve automatic garbage collection and to interact with the native code. Such runtime features introduce new non-determinism and more design considerations to implement a scalable and efficient deterministic replay system for Java code.

Many state-of-the-art deterministic replay systems for Java applications record *Logical Thread Schedule* [28, 9, 30], which assumes that applications are running on uni-processor platforms. Such a strategy is unsuitable for replaying concurrent Java applications running on multi-processor platforms. JaRec [13] records non-determinism in lock acquisition, but cannot reproduce buggy execution caused by data race. LEAP [15] uses static instrumentation for Java code to replay interleaved data accesses, thus it cannot reproduce non-determinism introduced by external code, such as libraries or class files dynamically loaded during runtime. Furthermore, LEAP does not distinguish different instances of the same class, and false dependencies between different objects of the same class may lead to large performance overhead when a class is massively instantiated.

Record time and log size are two critical performance metrics for deterministic replay systems, which can typ-

ically be optimized by applying transitive reduction in dependency-based address tracking approaches [6, 12, 17, 23]. However, these techniques may not be suitable for Java applications, due to frequent object movements by garbage collector. According to our evaluation results, with word or cache-line level address tracking approaches, garbage collection of JVM will introduce 7 times more unnecessary dependencies for SPECjvm2008 and SpecJBB2005. Furthermore, many Java programs have good locality on accessing a single object for Java threads.

Based on the observation above, this paper proposes ORDER, *Object centRiC DEterministic Replay*, to identify data access dependencies at object granularity. Such an object-centric technique can avoid recording massive unnecessary dependencies introduced by object movements from garbage collector, reduce contention on accessing shared metadata due to the low probability of object-level interleavings, and improve recording locality by inlining shared-memory access information within object headers. By dynamically instrumenting Java code during JVM compilation pipeline, ORDER naturally covers non-determinism caused by dynamically loaded classes and libraries.

We have implemented ORDER based on Apache Harmony, to record and replay non-deterministic events for concurrent Java applications on multicore. To further improve the performance of ORDER, we have also implemented a compiler analysis algorithm based on Soot [31] to avoid tracking dependencies for thread-local and assigned-once objects. Besides, we implement an offline log compressor algorithm to filter out remaining unnecessary dependencies from thread-local and assigned-once objects caused by imprecise compiler analysis.

Performance evaluation results show that ORDER has relatively good and scalable performance on a 16-core Intel machine for SPECjvm2008, PseudoJBB2005, and JRuby. The average overhead for recording non-determinism is around 108%. ORDER is also with good scalability on a 16-core platform. Performance comparison with LEAP [15] shows that ORDER is 1.4X to 3.2X faster than LEAP. We also show that ORDER can successfully reproduce several real-world concurrency bugs in JRuby.

In summary, the contribution of this paper includes:

- Two observations (i.e., GC-introduced dependencies and object access locality) for deterministically replaying Java applications based on a study of Java runtime behavior.
- The case for *object-centric deterministic replay*, which leverages the object granularity to record non-deterministic data access events using dynamic instrumentation.

- The implementation and evaluation of ORDER based on a real-world JVM platform, which demonstrate the efficiency and effectiveness of ORDER.

This paper is organized as following. In next section, we will present our study with evaluation results on the runtime behavior of Java programs on multicore platforms. In Section 3, we describe the main idea and design of our object-centric deterministic replay approach. The implementation details of our prototype ORDER are presented in Section 4. Section 5 shows the evaluation results in terms of performance, scalability, log size and bug reproducibility of ORDER. Finally, section 6 summarizes related work in deterministic replay and section 7 concludes our work with a brief overview of possible future work.

## 2 Java Runtime Behavior

In Java runtime environment, *garbage collection (GC)* is commonly used to automatically reclaim *non-reachable* memory spaces. The use of GC enables automatic memory management and avoids many memory-related bugs such as *dangling pointers*, *double free*, and *memory leakage*. GC usually requires moving or modifying objects in heap, which may cause additional dependencies for deterministic replay. In this section, we evaluate the impact of GC and describe two major observations that may affect the scalability and performance of deterministic replay systems.

### 2.1 Environment Setup and Workloads

The experimental results listed below are all generated on a machine with 4 quad-core Xeon processors (1.6GHz) and 32 GB physical memory. The Linux kernel version is 2.6.26 and the version of Apache Harmony is m12. We evaluate 21 parallel Java applications from *SPECjvm2008*, *SPECjbb2005*, and *JRuby*. *SPECjvm2008* is a general-purpose benchmark suite composed of a number of multithreaded applications. We omit the result for *sunflow* because it failed to be compiled by Apache Harmony m12 on our evaluation platform. *SPECjbb2005* is a server-side Java application that simulates an online marketing system. It emulates a common 3-tier system, and focuses on the business logic and object manipulation. *JRuby* is a Java implementation of the Ruby Programming language and provides a Ruby Interpreter entirely written in Java.

Each benchmark of *SPECjvm2008* is configured to run a single iteration, which ensures a fixed workload. Results of JRuby are collected on the most recent stable version of JRuby (JRuby 1.6.0) and a multi-threaded Ruby benchmark provided by it (i.e., *bench\_threaded\_reverse*). If the number of threads is not specifically mentioned, all results are collected with 16 threads. All tests are tested three times and we report the average of them.

## 2.2 Dependency-based Replay

Many dependency-based deterministic replay techniques record data dependencies according to data addresses. They record data dependencies when two instructions access the same address [6], cache line [17], or page [12, 16]. *Dependencies, conflicts or constraints* [6, 12, 17, 23]  $a \rightarrow b$  in these systems indicate that 1) instruction  $a$  and  $b$  both access the same memory location; 2) at least one of them is a write; and 3)  $a$  happens before  $b$ . To make execution deterministic, a replay run ensures that  $b$  does not happen until  $a$  has been executed.

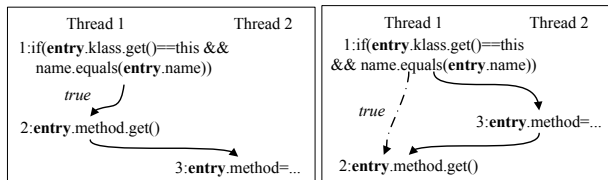


Figure 1: A real-world concurrency bug reported in JRuby community (JRuby-1382). Application crashes when statement 3 is executed between statements 1 and 2.

Figure 1 shows a real-world bug in JRuby. In this example, JRuby maintains a method cache. Thread 1 checks whether the required method resides in the cache. In correct execution (1 to 2 to 3), after the comparison code (statement 1) returns true, JRuby obtains the corresponding method from the cache, which is supposed to be the required method. However, if the content of the global variable “entry” is modified between statements 1 and 2 (1 to 3 to 2), the method obtained in statement 2 may be unexpected and crash the program. Suppose we treat reads/writes to each object as accesses to the same memory location, then there are two data conflicts in this example, 1,3 and 2,3. Dependency-based systems record the order of conflicted data instead of the order of all executed statements. For example, in Figure 1, if the application executes correctly, (2->3) is recorded. In buggy execution, (1->3) and (3->2) is recorded.

The number of recorded dependencies relies on the granularity chosen by deterministic replay systems. For example, if a deterministic replay system traces data dependencies according to the *real address* of data, dependency (1->3) in the given example may not be recorded. Specifically, statement 1 and 3 do not access the same memory location of data because *entry.klass* and *entry.name* have a different address from *entry.method*. Likewise, whether dependency (1->3) is recorded in page-level dependency-based replay relies on whether *entry.klass* or *entry.name* resides in the same page as *entry.method*. Though the granularity of recording dependency does not affect the correctness of a replay strategy,

large performance overhead will be introduced if it is either too small (large disk operations), or too large (false sharing).

Instead of directly recording dependencies, BugNet [22] and PinPlay [27] log the value of load instruction after another thread modifies the same location. In these deterministic replay systems, the number of logged values depends on the number of conflicts occurred in the execution of programs. Although their logging approaches are different from recording dependencies, their performance is also affected by the Java runtime behavior we list below.

## 2.3 Observation 1: Dependencies from GC

In JVM, GC is triggered if the memory management scheme indicates that performing GC is beneficial. Each time GC is triggered, it will scan the entire heap space, mark the reachable objects, remove non-reachable objects from heap, and possibly move reachable objects to achieve better cache locality and fewer heap fragments. Both marking and moving reachable objects will introduce a large amount of write operations in the heap. Thus, when using data addresses or cache-lines to identify dependencies in Java applications, there are a lot of extra dependencies introduced by GC. Most of these dependencies do not truly affect the behavior of Java applications, thus they are not necessary to be recorded. Furthermore, the dependency boost will cause a long pause time in GC and deteriorate application performance.

Figure 2 shows the ratio of dependencies generated by two widely-used garbage collectors to those generated by an application itself. In Apache Harmony, when using the popular generational garbage collector, dependencies introduced by garbage collection are about 8 times the dependencies introduced by original Java applications. The dependency boost even exceeds 16 times the dependencies introduced by original application in *scimark.sor.large*, *xml.validation*, *scimark.fft.small*, and *SPECjbb2005*. Results of Mark-Sweep garbage collector are similar to those of Generational GC, which indicates that such a phenomenon is likely to be a common case.

GC itself is a non-deterministic event in the JVM. Object allocation order, total heap size, garbage collection algorithm, and many other runtime events will affect the behavior of GC. Thus, deterministic replay system cannot ignore the influence from GC. However, recording garbage collection behavior may cost much and worsen program performance.

## 2.4 Observation 2: Interleavings of Object Accesses

Within JVM, *object*, a new candidate of granularity for recording dependencies, is introduced by the managed

memory strategy. According to our experiments, objects accessed by a thread are very likely to be accessed by the same thread at the next time. Hence, interleavings<sup>1</sup> seldom happen at object level. As depicted in Table 1, the ratio of interleavings at object level is less than 7%. This ratio is extremely low in scientific applications (*fft*, *lu*, *sor*, *sparse*, *monte\_carlo*). This implies that recording dependencies among threads at object level will likely result in better locality and less contention.

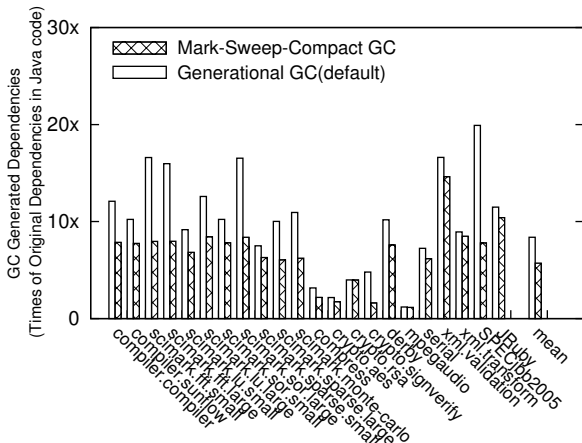


Figure 2: Dependencies introduced by GC when using data address to identify dependencies in Java applications. The base line is the number of dependencies introduced by Java applications themselves. Number of dependencies is calculated according to the CREW protocol in SMP-Revirt [12].

### 3 Object-centric Deterministic Replay

Based on the two observations, we propose an Object centric DEterministic Replay (ORDER) scheme to record and replay concurrent Java applications. ORDER uses object as the granularity to record interleavings of data accesses. In the rest of this section, we first discuss why object would be a proper granularity of tracking dependencies, and then illustrate the sources of non-determinism within JVM and how ORDER handles them.

#### 3.1 Why Object Centric?

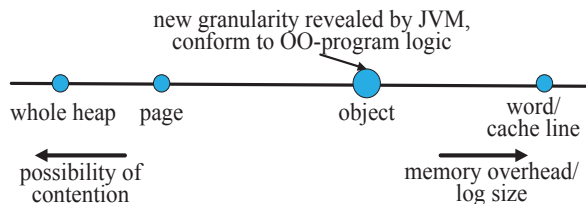


Figure 3: Spectrum of granularity in deterministic replaying Java applications.

Case	Interleaving	Access	Rate(%)
compiler.compiler	53997073	3.7E+9	1.46
compiler.sunflow	159104781	7.6E+9	2.09
fft.small	6281	1.2E+10	<0.01
fft.large	3447	1.6E+10	<0.01
lu.small	6500	3.4E+10	<0.01
lu.large	3311	2.87E+11	<0.01
sor.small	4446	2.5E+10	<0.01
sor.large	3358	1.0E+11	<0.01
sparse.small	4201	3.0E+10	<0.01
sparse.large	3055	1.1E+11	<0.01
monte_carlo	3503	9.6E+10	<0.01
compress	448683851	3.4E+10	1.31
crypto.aes	3.73E+9	6.0E+10	6.21
crypto.rsa	135072884	2.2E+10	0.62
crypto.signverify	33185584	2.3E+10	0.14
derby	2.44E+9	4.9E+10	4.95
mpegaudio	922855001	6.4E+10	1.45
serial	315661230	1.7E+10	1.80
xml.validation	96681920	6.3E+9	1.53
xml.transform	1.41E+9	6.6E+10	2.13
SPECjbb2005	78856923	1.9E+15	<0.01
JRuby	161801036	1.3E+12	0.01

Table 1: Ratio of interleavings at object level: the second and third column show the number of interleavings and total object accesses accordingly. The forth column shows the percentage of interleavings among the total number of object accesses.

In object-oriented programming languages like Java, applications are usually designed around objects. Figure 3 shows the spectrum of design consideration on the granularity of deterministic replay for Java. This implies several advantages of ORDER:

**Elimination of GC dependencies:** Our first observation above shows that massive extra dependencies will be raised by GC if tracking dependencies using addresses. However, such extra dependencies will naturally be eliminated when tracking dependencies at object level, as the movement of an object does not change its content.

**Reduced contention of synchronization:** When recording data dependencies, data content should be protected through synchronization to avoid possible data races. Our second observation indicates an object will likely be accessed consecutively by one thread. This implies less contention over the synchronization construct protecting the metadata information of an object.

**Improved locality:** Furthermore, Java does not support pointer arithmetic, and the memory layout of Java applications is managed by JVM. By embedding metadata information into object headers, there will be good locality for accessing such metadata information.



### 3.2 Recording Data Access Interleavings

As discussed in previous work [4, 8, 12, 18, 25, 26, 29], many bugs introduced by non-deterministic events in multi-threaded applications are caused by concurrent data accesses, i.e., the order that different threads access the same data. In Java applications, data are usually grouped as objects. Thus, access to a memory location can be considered as access to the corresponding object. This is a major source of non-determinism.

Instead of recording conflict access pairs, ORDER only records the object access timeline. The recording/replaying scheme in ORDER is depicted in Figure 4. ORDER records how many times a thread has accessed an object before this object is accessed by another thread. ORDER maintains such access timeline and enforces it during a replay run.

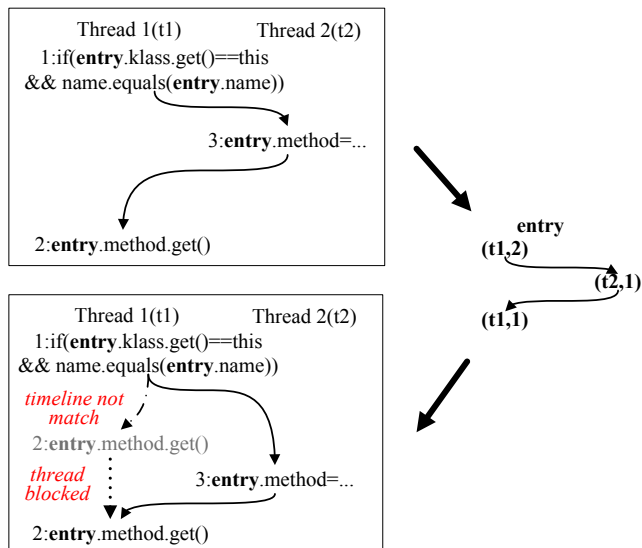


Figure 4: The recording/replaying scheme in ORDER. Each tuple  $\langle tn, x \rangle$  represents a timeline record, which indicates that the object is accessed by thread  $tn$  for  $x$  times.

#### 3.2.1 Metadata

We extend the header of each object with five fields to record the timeline of object accesses:

**Object identifier (OI):** Information in the original object header cannot uniquely identify an object. ORDER extends the original 32-byte hash-code to 64-byte so that it can uniquely identify an object in long-term execution. This new identifier is generated according to the identifier of the allocating thread and the index that indicates the object allocation order in this thread.

**Accessing thread identifier (AT) and access counter (AC):** Accessing thread identifier and access counter maintain current status of object access timeline. Every

timeline recorded by ORDER can be interpreted as “*this object (OI) is accessed by some thread (AT) for some times (AC)*”. In record phase, these two fields record: (1) which thread is now accessing this object; (2) how many times this thread has accessed it. In replay phase, they maintain: (1) which thread is expected to access this object; (2) how many times this thread will further access such an object before an expected interleaving is encountered.

**Object-level lock:** When recording/replaying object accesses, the accessed content as well as access thread (AT)/access counter (AC) should be synchronized. ORDER uses an object-level lock to protect the whole object (including fields, array elements and object header) when a certain access to this object is recorded/replayed. Using object granularity, our approach only needs to synchronize the accessed object instead of the whole heap or page, which may reduce the strength of synchronization.

**Read-Write flag:** The Read-Write flag records whether the current timeline record is *read-only* or *read-write*. This information is used in the timeline filter to reduce log size.

#### 3.2.2 Recording/Replaying Object Access Timelines

ORDER contains two modes corresponding to the record and replay phase of deterministic replay systems respectively. Each mode contains an instrumentation action added to compilation pipeline of JVM. We also modify garbage collector to record the final state of timelines.

**Record mode:** Figure 5 illustrates how ORDER records object access timeline. In record mode, when an object is about to be accessed by a certain thread, ORDER compares AT in object header with the identifier of current accessing thread (CTID). If this access is a continuous access ( $AT == CTID$ ) (Figure 5.a), ORDER updates the access counter ( $AC = AC + 1$ ). When an interleaving is encountered ( $AT != CTID$ ) (Figure 5.b and 5.c), ORDER puts the timeline record to log and resets timeline record in the object header ( $AT = CTID, AC = 1$ ). When JVM is terminated, or objects are collected by GC, ORDER records the final timeline record of each object to log (Figure 5.d). Besides, the record operations as well as the original object access are enclosed by an object-level lock acquire/release pair, which ensures the atomicity of the record process. The Read-Write flag is set to *read-write* if a write operation is encountered.

**Replay mode:** Figure 6 illustrates how ORDER reproduces the recorded timeline. In replay mode, when ORDER is about to reproduce timeline record for a certain object, it loads the timeline record (AT and AC) into the object header. When a thread is about to access this object, the code instrumented by ORDER will compare its identifier (CTID) with expected thread identifier (AT) in object header. If the requesting thread is the expected ac-

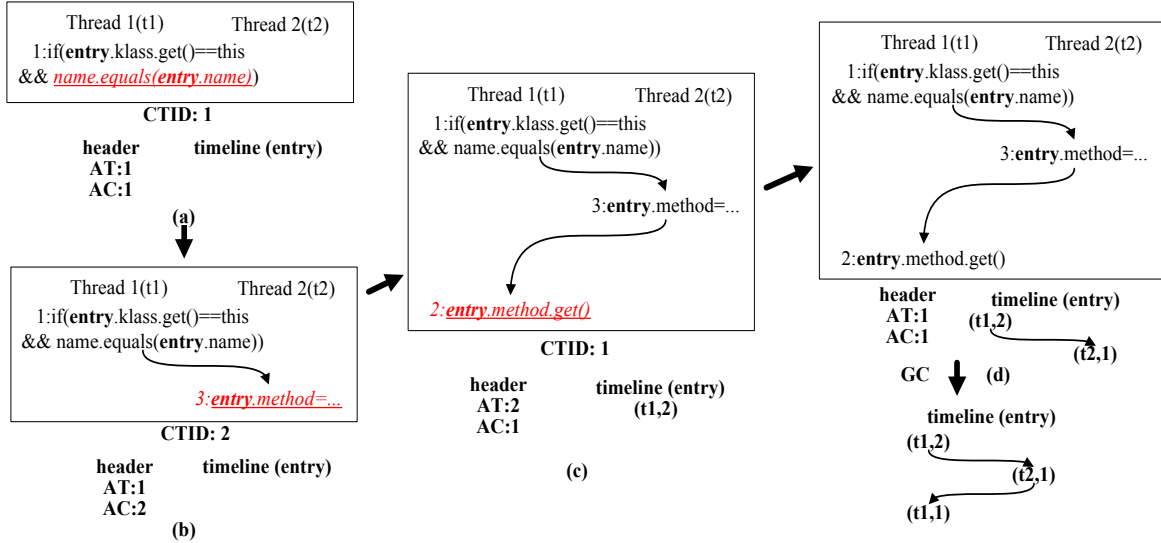


Figure 5: Record mode of ORDER.

cess thread ( $CTID == AT$ ) (Figure 6.a, 6.b), ORDER updates the access counter ( $AC = AC - 1$ ). When a recorded interleaving is about to occur ( $AC == 0$ ), ORDER loads the next timeline record into the object header. If a requesting thread is not the expected access thread ( $CTID \neq AT$ ), it will be blocked until a recorded interleaving updates the timeline record (Figure 6.c). If the blocked thread no longer violates the recorded timeline, it will be woken up and continue its execution (Figure 6.d). Like the record mode, a similar object-level lock is used to protect the whole object when certain access to this object is replayed.

### 3.2.3 Eliminating Unnecessary Timeline Records

Though object-level locks cause less contention than page/heap-level locks, recording object access timelines still incurs notable performance slowdown. According to our study, much of the recording overhead comes from instrumentation to the following objects, which never cause non-determinism:

**Thread-local objects:** Many objects allocated by JVM are thread-local objects. These objects are accessed in a certain thread and never shared with other threads.

**Assigned-once objects:** Assigned-once objects have continuous write operations in their initialization methods. After initialization, the assigned-once objects are shared among different threads but no thread will write the fields of these objects. Such objects do not really introduce non-determinism to Java applications. According to our evaluation, assigned-once objects are very common in the JVM. For example, the switch table objects generated by Javac, class objects, string objects, and most of the final arrays are assigned-once objects.

To eliminate such unnecessary recording of object ac-

cess timelines, an offline preparation phase is introduced to analyze the target application and annotate the Java bytecode. Recording/replaying phase of ORDER can adopt annotations from bytecode and eliminate unnecessary recordings.

Accesses to these two kinds of objects can be identified by inter-procedure analysis. We use escape analysis [7] to find thread-local objects. The original escape analysis algorithm introduces three escape states. Each state represents a certain kind of object: *NoEscape* means that objects allocated by a certain allocation site are method-local objects, which do not escape outside this method. *ArgEscape* represents the objects that are exposed to other functions, but they are not visible to different threads. *GlobalEscape* means that objects are shared among threads. However, because *GlobalEscape* does not contain read/write information, the original escape analysis cannot identify assigned-once objects. In ORDER, we apply the following modifications to the original algorithm so that assigned-once objects can be identified:

- A new escape state *Shared-Write* is introduced. Object nodes in the connection graph are set to such a state when: 1) Objects allocated by the corresponding new site can be global escaped; 2) Write operations may be applied to these objects after they are shared among threads.
- The read/write states of phantom object nodes, which represent the reference to object nodes that are not allocated in the local method, are traced. It can be *Read-Only* or *Read-Write* which means whether the corresponding objects can be written

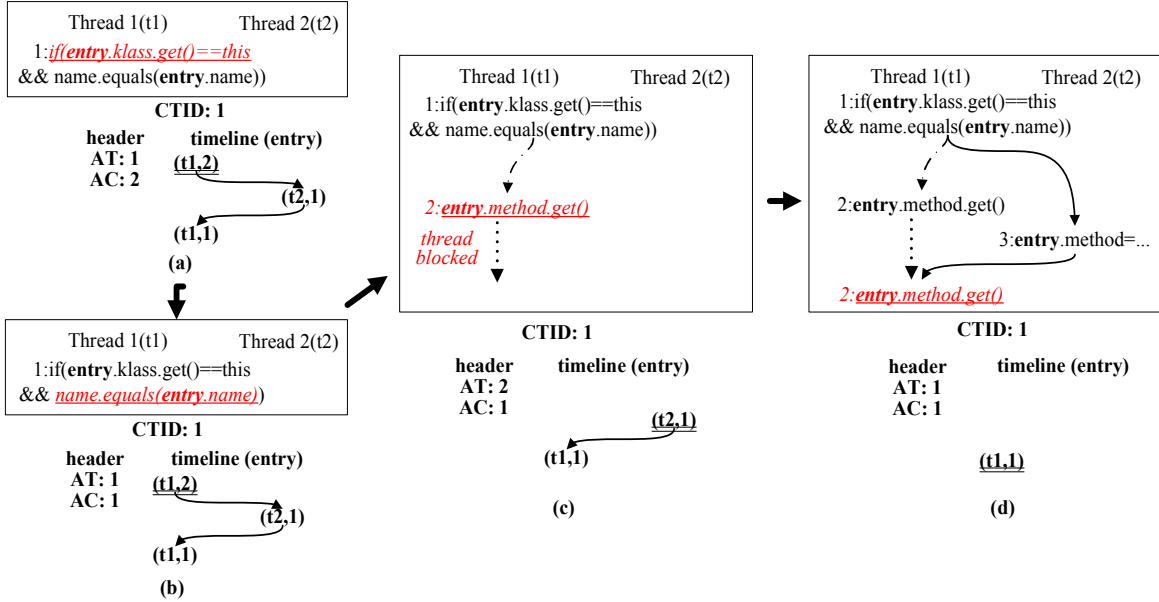


Figure 6: Replay mode of ORDER.

in the current method. When a new phantom object is created, its read/write state is set to *Read-Only*. In intra-procedure analysis, if a write operation is applied to  $p$ , the read/write states of all phantom object nodes pointed by  $p$  are switched to *Read-Write*. Because read/write state is only attached to phantom object nodes, they are finally marked as assigned-once objects if their read/write states are *Read-Write*.

- The read/write information is used to identify assigned-once objects. It affects the escape state in the following way. First, we modify the transfer function of escape state in intra-procedure analysis that if a certain node is already *Global-Escape* and a write operation is encountered, its escape state is set to *Shared-Write*. Second, in the inter-procedure merge function, when a *Read-Write* phantom object node is merged to a *Global-Escape* normal object node, which means that globally shared objects can be modified outside the allocation function, the escape state of this normal object node is changed to *Shared-Write*.

To avoid eliminating necessary timeline records, our offline analysis algorithm is conservative. Specifically, when an object node might possibly be exposed to external code, its escape state is automatically set to *Shared-Write*.

### 3.2.4 Log Compression

The raw log recorded by ORDER contains object access timelines of all recorded objects. Although inter-

procedure escape analysis can eliminate some unnecessary timeline records, ORDER still records many thread-local or assigned-once objects due to the imprecision of the analysis algorithm. Timeline records of such objects does not help to correct deterministic replay in the replay phase. Thus, we apply a timeline filter to eliminate these unnecessary object access timelines. The timeline filter analyzes the timeline information in the log and filters out the following ones: a) Timeline that has only one occurrence; b) Timeline that has only one occurrence of read-write in the beginning. Besides, timelines which have several occurrences of read-write in the beginning are partially eliminated. We reserve the leading read-write occurrences and eliminate the following read-only ones. The log compressor costs short time and can be processed by an idle core, and it can be applied either offline or online in GC. Currently, ORDER applies it offline and we plan to apply this filter online in the future.

## 3.3 Other Non-determinism in JVM

Some sources of non-determinism are common for both Java and native code, while the JVM additionally introduces sources of non-determinism such as garbage collection, adaptive compilation and class initialization. In the following, we describe each source of non-determinism and how ORDER handles it.

### 3.3.1 Common Non-determinism in Java and Native Code

**Lock acquisition:** When two threads are competing on the same lock, the order of lock acquisition is an important source of non-determinism. Thus, it is neces-

sary to record/replay the lock acquisition/release order. The Java programming language provides two basic synchronization idioms: synchronized methods and synchronized statements, each of which is ensured by maintaining a corresponding critical section in the JVM. In ORDER, entering a critical section is treated as an access to the corresponding object. Besides, JVM also provides explicit locks and atomic operations in package *java.util.concurrent*. Likewise, ORDER treats these operations as accesses to the lock objects.

**Signal:** Similar to C/C++ programs, signals also cause non-deterministic behavior in Java applications. In JVM, they are usually wrapped to wait, notify, and interrupt operations for threads. Leveraging their non-preemptive nature, ORDER records the return values and status of the pending queue instead of the triggering time. Java developers can write preemptive signal handlers by using the “sun.misc.SignalHandler” interface, though they are neither officially supported according to Sun, nor supported by many state-of-the-art Java Virtual Machines. By tracing and recording the timeline status of the current active object, which is the last accessed object in the current thread, non-determinism in such libraries can also be reproduced by ORDER.

**Program Input:** Different input may generate different program behavior. ORDER records this non-determinism by logging the content of input to Java programs.

**Library invocation :** Some methods of the Java library, like *System.currentTimeMillis()* and methods in *Random/SecureRandom* classes, generate non-deterministic return values. ORDER logs the return values of these methods to ensure determinism. Besides, non-deterministic events in native libraries are exposed to Java applications through lock acquisition, signal, input, garbage collection, class initialization, and the non-deterministic Java libraries. They are handled in ORDER as we covered in the corresponding sources of non-determinism.

**Configuration of OS/JVM :** To ensure that the environment setting of a record run is consistent with that of a replay run, ORDER records the configuration of OS/JVM and reproduces the recorded configuration in the replay run.

### 3.3.2 Unique Non-determinism in JVM

**Garbage collection:** Garbage collection is another source of non-determinism in the JVM. In multi-threaded Java applications, different order of object allocations across threads may cause different heap layout in memory, which then causes different collector behavior. As a result of the object-centric recording of data accesses, ORDER does not need to record dependencies introduced by GC. However, GC can affect the behavior of

Java applications through several interfaces. To record such non-determinism, ORDER logs interfaces between GC threads and Java threads. Since most interface invocations are triggered by JVM, non-determinism should be recorded according to the JVM inner mechanism. We will discuss the detail later in section 4.

**Adaptive Compilation :** Adaptive compilation, which is also known as incremental compilation, recompiles methods if they are frequently invoked. Behavior of adaptive compilation relies on the profiling result of program execution. Because the profiling result varies in different executions, the behavior of adaptive compilation is also non-deterministic. Reproducing non-determinism caused by adaptive compilation can be supported by recording virtual machine states and profiling results introduced by Ogata et al. [24].

**Class Initialization :** When JVM resolves and initializes a class, static fields of this class are commonly initialized by the thread resolving the class. Thus, which thread first invokes the class resolution method may affect the behavior of Java applications. ORDER records the resolution and initialization thread identifier, and ensures that the same thread first enters the class resolution method in the replay run.

## 3.4 Discussions

**Coverage of non-determinism:** To our knowledge, ORDER is the first deterministic replay system which records non-determinism introduced by the Java runtime, such as GC, class initialization, etc. Moreover, unlike JaRec and LEAP, ORDER not only captures non-deterministic lock acquisition or data access interleaving outside JVM library, but also records non-deterministic events inside the library. Although recording such non-deterministic events incurs additional runtime overhead, we believe they are necessary and essential to deterministic replay systems. As discussed in LEAP, some bugs may not be reproducible due to the ignoring of these non-deterministic events. More importantly, loss of such non-determinism may unexpectedly deadlock normal program execution due to the inconsistent execution between recorded execution and replaying one. When replaying long-running Java applications, the replay system may deadlock itself before the buggy instruction is encountered.

**Transitive log reduction:** The timeline recorded by ORDER is already the smallest set of object access interleavings, which does not need to be further optimized by transitivity reduction [23]. Currently, ORDER does not separate conflicts from read-read dependencies. Our evaluation results in section 5 show that tracing timeline incurs much more overhead than swapping logs to disk. Though identifying conflicts from read-read dependencies can further reduce log swapping overhead, which



is already very small, it will notably increase the complexity of timeline tracing logic. Thus, ORDER does not apply complex conflict-based reduction algorithm like SMP-Revirt [12].

## 4 ORDER Implementation

We have implemented a prototype of ORDER based on Apache Harmony [1]. We add several new instrumentation phases into Harmony compilation pipeline to support deterministic replay. Besides, the default garbage collector of Harmony (Generational GC with default configuration for each object space) is modified to record final timeline of each object. We also modify Soot[31] to annotate thread-local or assigned-once object accesses in methods. Such annotation is attached to Java bytecode as a new attribute, which can be simply discarded if the target JVM does not support ORDER. Thus, it does not affect the portability of original Java application. Because preemptive handlers are currently not implemented in Apache Harmony, the current version of ORDER does not handle the corresponding non-determinism. In the following sections, we discuss how ORDER records non-determinism and cooperates with native code in Harmony.

### 4.1 Modification to Harmony Compilation Pipeline

Harmony uses *pipelines* to manage compilation configuration of Java methods. In Harmony, every *pipeline* contains a set of *actions* each of which represents a single analysis or optimization of Java methods. The instrumentation processes of record and replay phase are implemented as two *actions* separately in ORDER. Besides, if adaptive compilation is enabled, two or more *pipelines* can be assigned to a single method. Then, when a method is frequently invoked, it can be recompiled with a more aggressive *pipeline*. Whether adaptive compilation is enabled or not only affects the performance of Java application and does not affect the bug reproducibility of ORDER. Thus, the current prototype of ORDER disables adaptive compilation to reduce engineering effort and uses a single compilation pipeline. Type/copy propagation, constant folding, dead/unreachable code elimination, devirtualization and all platform dependent optimizations except peephole and fast array filling [3] are enabled in the selected pipeline.

### 4.2 Recording GC in Harmony

Although garbage collection is an important source of non-determinism in the JVM, it rarely affects the behavior of Java applications. ORDER does not record garbage collection activity in Harmony, but only records the following interfaces between garbage collection threads and Java threads:

1. After garbage collection, dead objects that have finalization methods should be finalized. The order that the finalization methods are invoked depends on heap layout and garbage collection algorithm. In Harmony, finalizable objects are enqueued to a specific object queue after garbage collection. Finalizing threads extract objects from the queue and invoke their *finalize* methods. ORDER records the order they are extracted from the queue and reproduces the recorded order in replay run.
2. In Java, weak/soft/phantom reference represents several strengths of "non-strong" object instances, and they are collected in GC according to the memory usage. After GC, JVM notifies the queue of weak reference objects that the status of corresponding weak objects may be changed by garbage collection. Likewise, queues of soft/phantom reference objects are notified in the same way. The size of weak/soft/phantom reference set depends on runtime heap status. Like finalization, Harmony maintains a *references\_to\_enqueue* queue to link the *reference\_enqueue* thread with Java threads. The order these objects are extracted from *references\_to\_enqueue* queue is recorded.
3. Java applications can explicitly invoke method *Runtime.freeMemory* to query the size of free memory from JVM. Different heap layouts result in different free memory sizes during execution. Because this method has no side effect, ORDER only records the return value of it.

### 4.3 Cooperating with JVM Native Code

ORDER uses dynamic instrumentation to guarantee that the replay run has the same object access timeline as the recorded run has. Although execution behavior of Java code is deterministic with the help of instrumentation, the execution behavior of JVM native code is still non-deterministic. ORDER records non-deterministic events of JVM native code that will cause non-deterministic behavior of Java applications, and ignores those not really affecting Java applications. ORDER should cooperate properly with ignored JVM native code so that the inconsistency between them will not introduce deadlock.

The internal suspend-resume mechanism of Harmony does not affect application behavior. Thus, it is ignored by ORDER. In Harmony, when enumerating the root set of the Java heap, GC threads suspend Java threads in order to get a consistent snapshot of the Java heap. When Java threads are about to enter *enable-suspend* state (named safe-point or safe-region), it must record the status of the current stack frame so that GC threads can obtain a complete set of live objects. However, frequently recording stack frame information is costly

and worsens performance. In Harmony, safe-point/safe-region is only invoked at call sites or certain system calls (e.g., sleep). ORDER blocks Java threads when they are about to violate the recorded object access timeline. When the blocked time of a Java thread is beyond a threshold (500ms), it prepares its own stack frame information and enters the *enable-suspend* state. When a Java thread enters the *enable-suspend* state, it can be easily suspended by other threads. The Java thread exits the *enable-suspend* state when it does not violate recorded timeline further.

## 5 Evaluation Results

In this section, we evaluate the performance slowdown of ORDER. We use an Intel Xeon machine with 4 quad-core 1.6Ghz CPUs and 32 GB physical memory, which runs a Linux with kernel version 2.6.26. We show the results for the SPECjvm2008 suite except *sunflow*, *derby* and *xml.transform*. One of them, *sunflow*, failed to be compiled by original Apache Harmony m12, and two of them (*derby* and *xml.transform*) failed to be compiled because adaptive compilation is disabled. The original *SPECjbb2005* runs for a fixed amount of time and is not suitable for evaluating performance slowdown. To ensure a fixed workload, we evaluate a variant one called *Pseudojbb2005*, which runs for a fixed number of transactions (100000).

### 5.1 Slowdown in Record Phase

Figure 7 depicts the performance slowdown of record phase in ORDER. All benchmarks are run with 16 threads to evaluate the performance slowdown. To show the effect of eliminating unnecessary timeline recording, performance before and after this optimization are both presented (before-opt vs. after-opt). In the raw recording system, although most of the applications have performance slowdown of less than 8 times and overhead of some benchmarks is even lower than 100%, the slowdown rises up to 82x in an extreme case (*compress*). Compared to the raw system, performance after eliminating unnecessary timeline recording is much better. As shown in Figure 7, after optimization, the record overhead of ORDER is less than 8 times in all cases, even in *compress*. ORDER incurs an average of 108% overhead compared to the original execution run. Actually, the optimized version of ORDER introduces less than 100% overhead for most of the benchmarks, which means that it may be efficiently used for many applications. If not specifically mentioned, all results below are collected under optimized ORDER system.

Moreover, we evaluated three more configurations of ORDER to investigate the source of overhead in ORDER: raw Apache Harmony without adaptive compilation (wo\_adaptive), ORDER without recording timeline

(wo\_timeline), and ORDER without swapping timeline log to disk, i.e., the timeline records are stored only in memory (wo\_disk). As shown in Figure 7, the performance of most applications (except *JRuby*) after disabling adaptive compilation is very close to the original JVM. On average, there is a 5.8% performance difference, which means performance impact of adaptive compilation is insignificant in most cases. Besides, the performance of ORDER is similar whether disabling disk operations for timeline logs or not. These two configurations introduce a slowdown of about 104% and 108% correspondingly. This shows that overhead of disk operation is also small. Further, when disabling timeline recording as a whole, ORDER introduces only 16% overhead on average. This confirms that the major performance overhead of ORDER comes from tracing timeline in memory.

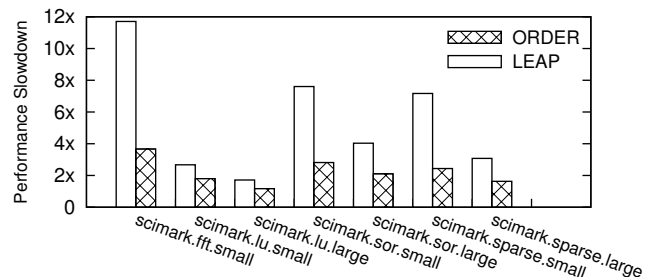


Figure 8: Performance slowdown of record phase in ORDER and LEAP. 16 threads are used to evaluate the performance slowdown.

Figure 8 depicts the performance of ORDER compared to LEAP[15]. Because the static instrumentation approach does not support reflection, LEAP cannot properly instrument Java code of original SPECjvm2008 or SPECjbb2005. To evaluate the performance of LEAP, we manually replace the reflection mechanism in *scimark*, *mpegaudio*, and *compress* with direct method invocation. We found that when recording *mpegaudio*, *compress*, *scimark.monte\_carlo* or *scimark.fft.large*, LEAP either throws an *OutOfMemoryError* or does not finish in two hours. Results of *JRuby* are also not presented here because LEAP throws *NullPointerException* in static instrumentation phase. The performance result just serves as a reference because LEAP records non-determinism in neither library code nor Java runtime. As shown in Figure 8, although ORDER records more non-deterministic events than LEAP, ORDER is still 1.4x to 3.2x faster than LEAP in the evaluated benchmarks.

By reducing the strength of synchronization, ORDER notably improves the scalability of recording interleaved object accesses in Java applications. Figure 9 shows the performance slowdown of ORDER when the number of threads varies from 1 to 16. Overall, ORDER scales well.

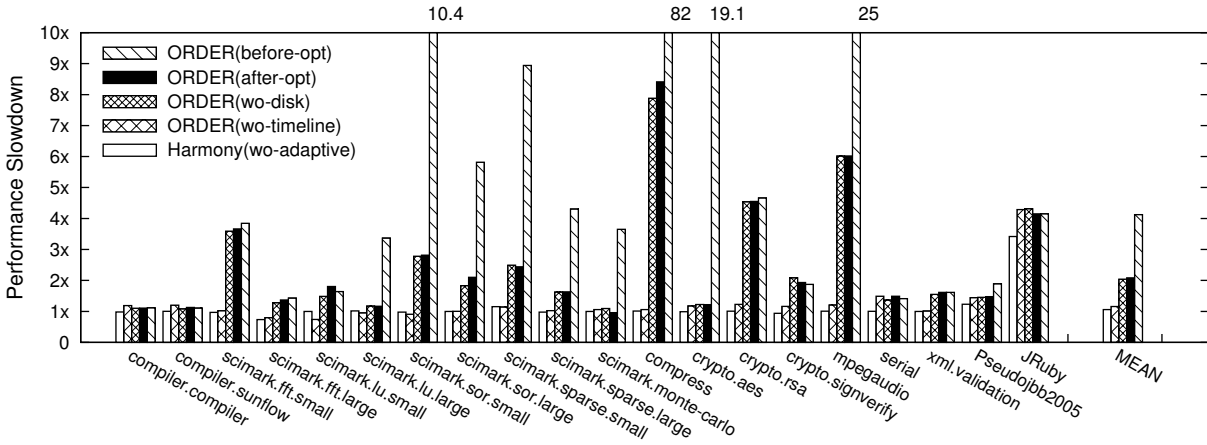


Figure 7: Performance slowdown of record phase in ORDER. 16 threads are used to evaluate the performance slowdown.

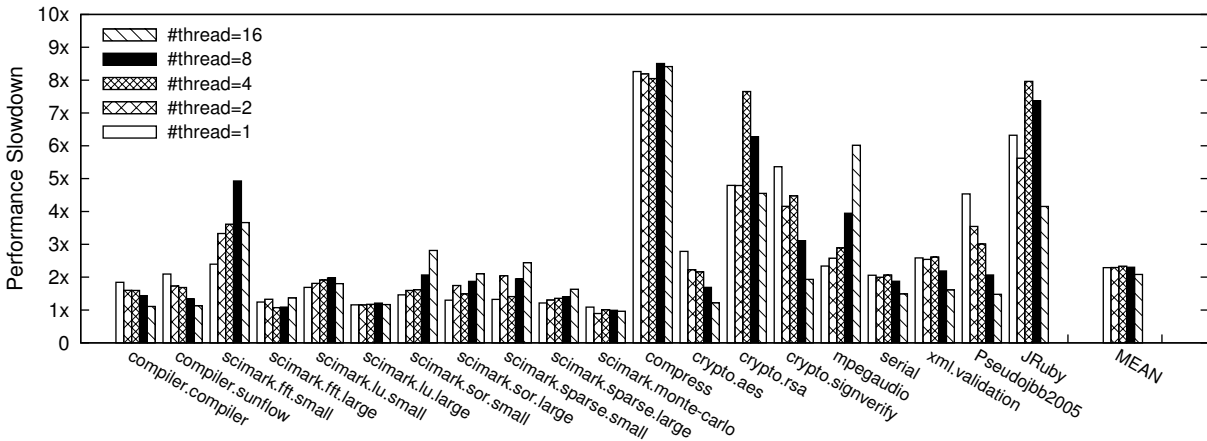


Figure 9: Performance slowdown of record phase when the number of threads varies from 1 to 16.

Most of the cases have similar performance slowdown while the number of threads increases. With the number of threads increasing, only one case in SPECjvm2008 (*mpegaudio*) has an obvious increase of performance slowdown. The increased number of object access interleavings degrades the performance in this case. Meanwhile, we observed that there are still many assigned-once objects recorded in this case, which is caused by the conservativeness of assigned-once analysis, and they can be further eliminated by improving the precision of analysis algorithm. For some applications such as *compiler* and *crpto.aes*, when we increase the number of threads, the recording overhead even decreases. Despite the reduced contention of ORDER, this anomaly is also affected by the following two reasons: 1) As discussed in Yi et al. [34], some benchmarks themselves are not scalable, like *compiler*; 2) Instrumentation of ORDER increases the complexity of intermediate representation, thus introduces additional overhead to analysis and optimizations in Harmony; such overhead is amortized when the number of threads increases.

## 5.2 Slowdown in Replay Phase

Similarly, ORDER uses dynamic instrumentation in Harmony to implement the replay phase. Thus, instrumentation of replay phase causes similar performance slowdown to record phase. Besides, blocking threads to ensure correct timeline will introduce additional overhead. Figure 10 depicts the replay slowdown of ORDER. For most of the selected benchmarks, the performance of replay phase is scalable from 1 to 16 threads, with four exceptions (*xml.validation*, *serial*, *mpegaudio*, and *Pseudojbb2005*). In these four applications, with the number of threads increasing, performance slowdown also increases because Java threads are frequently blocked. Currently the prototype of ORDER uses a naive implementation of thread scheduler, which can be further enhanced to speedup replay phase performance.

## 5.3 Log Size

Besides the performance overhead, many state-of-the-art deterministic replay systems also suffer from large space overhead. To record interleaved data accesses on

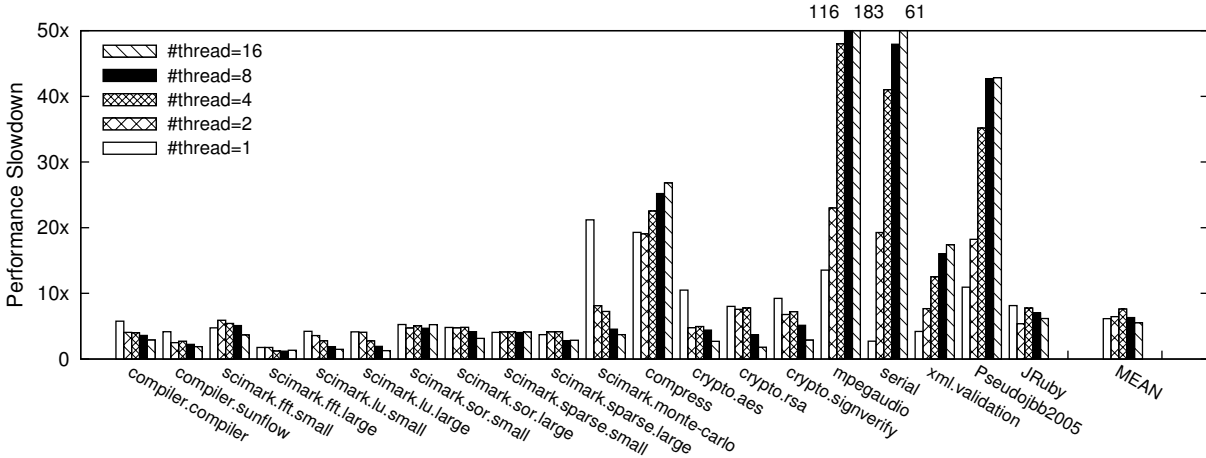


Figure 10: Performance slowdown of replay phase when the number of threads varies from 1 to 16.

multi-processor architecture, deterministic replay systems usually need to record tens of Gigabytes logs per Hour(g/h) [12, 15]. Although the disk capacity today is large enough to store log files, it is hard to share such large data on network. If we want to report a concurrency bug to the corresponding community, uploading a log file with tens of gigabytes is obviously not attractive.

Table 2 shows that the log size of ORDER is very small. In most cases, it generates a log file less than 100 Megabytes per Hour(m/h), which is considerably smaller than those reported in other deterministic replay systems[12, 15]. Only two cases (*serial* and *Pseudobb2005*) generate log files that are greater than 1 Gigabytes per Hour. We notice that most of the logged interleavings of *serial* relate to contention for a single global buffer in the original application, which is introduced by a producer-consumer scenario. Most of the logged interleavings in *Pseudobb2005* are caused by the false sharing between different static fields in the same class, which occurs because prototype of ORDER does not distinguish accesses to the same class object. However, such a log size is still much smaller than those reported in previous literatures [12, 15].

## 5.4 Concurrency Bug Reproducibility

To confirm the reproducibility of ORDER on concurrency bugs, we reproduce six real-world concurrency bugs from open source projects with ORDER. The characteristics of these concurrency bugs are listed in Table 3. These cases cover three major categories of concurrency bugs reported by Lu et al. [18]. By replaying the recorded logs, the buggy executions are successfully reproduced in replay phase. Among the bugs, JRuby-2483 is caused by using thread unsafe library code, which fails to be reproduced in a static instrumentation approach [15].

Case	Log Size (timeline)	Log Size (others)
compiler.compiler	88(m/h)	35(m/h)
compiler.sunflow	61(m/h)	58(m/h)
scimark.fft.small	0.60(m/h)	10(m/h)
scimark.fft.large	0.47(m/h)	7(m/h)
scimark.lu.small	0.37(m/h)	6(m/h)
scimark.lu.large	0.35(m/h)	5(m/h)
scimark.sor.small	2(m/h)	40(m/h)
scimark.sor.large	0.68(m/h)	11(m/h)
scimark.sparse.small	2(m/h)	36(m/h)
scimark.sparse.large	0.56(m/h)	10(m/h)
scimark.monte-carlo	0.013(m/h)	0.22(m/h)
compress	4(m/h)	44(m/h)
crypto.aes	1.4(m/h)	9(m/h)
crypto.rsa	26(m/h)	6(m/h)
crypto.signverify	10(m/h)	8(m/h)
mpegaudio	511(m/h)	2(m/h)
serial	1553(m/h)	121(m/h)
xml.validation	632(m/h)	31(m/h)
Pseudobb2005	1085(m/h)	550(m/h)
JRuby	0.8(m/h)	170(m/h)

Table 2: Log size of ORDER, in 16-thread execution.

## 6 Related Work

**State-of-the-art deterministic replay for Java:** State-of-the-art deterministic replay systems for Java applications use the strategy called “logical thread scheduling” to record multi-threaded Java execution [28, 9, 30]. As mentioned in Dejavu [9], “logical thread scheduling” is based on a global clock (i.e., time stamp) for the entire application. This strategy works efficiently in uni-processor platforms. However, global clock among cores needs to be synchronized frequently, which imposes contention to a single global lock. There are currently no scalable deterministic replay systems based-on such an approach for multi-processor platforms.

JaRec [13] assumes that Java applications are data-



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	Concurrency bug caused by using thread unsafe library code.
JRuby-879 JRuby-2380	order violation	Listing threads before thread is registered causes non-deterministic result.
JRuby-2545	deadlock	Lock on the same object twice causes deadlock.

Table 3: Real-world concurrency bugs reproduced by ORDER. Each of them comes from open source communities and causes real-world buggy execution.

race free programs and records only the lock acquisition order, which cannot be used to reproduce concurrency bugs caused by data races. LEAP [15] records non-determinism introduced by data accesses through static recompilation and instrumentation, which cannot cover external code, such as libraries or class files dynamically loaded during runtime. Thus, it cannot reproduce concurrency bugs caused by these missing parts. None of the existing deterministic replay systems can reproduce bugs caused by non-determinism inside JVM. Furthermore, LEAP does not distinguish instances of the same class, and the false sharing between different objects may lead to large performance overhead when a class is massively instantiated.

There are also several proposed approaches to improve efficiency and scalability of deterministic replaying native code written in C/C++. They can be grouped into two sets according to how they record non-deterministic data accesses:

**Software deterministic replay for native code:** Uniprocessor deterministic replay systems [5, 33] record interrupt boundaries and input payloads, which are proven useful in bug diagnosis and intrusion detection. However, state-of-the-art systems use multiple CPUs with shared memory data access. Such an additional source of non-determinism makes efficient recording difficult for software.

Several approaches are proposed to reduce the synchronization overhead and performance slowdown introduced by memory race recording. Transitive Reduction [23] is proposed to reduce the log size by applying transitivity-based log reduction to log files generated by deterministic replay system. It can also reduce the

synchronization overhead in the replay phase. However, such an approach still needs to use global clock and cannot reduce synchronization overhead in record phase.

SMP-Revirt [12] modifies the page protection mechanism for recording non-deterministic data access events. By using page as the granularity to track dependencies, SMP-Revirt achieves a low performance overhead in architectures with 1 to 2 cores. However, because they record a very large granularity of data sharing, its performance drops significantly when number of cores is increased to 4. In order to mitigate thrashing caused by frequent transfers of page ownership, SCRIBE [16] defines a minimal ownership retention interval and disallows ownership transitions until the interval expires. Although it notably relieves the contention among threads, the extended interval of page ownership makes it difficult to capture atomic violation bugs in record run.

PRES and ODR [26, 4] record partial information in record phase and use an offline reproducer to infer the race occurred in the record phase. Because only a part of the execution information is recorded, they can achieve a low performance slowdown. However, the reproducibility depends heavily on how much information is recorded, and the proper scheme to record information is hard to decide. Although recording less information can reduce performance overhead, the recorded execution may not be reproducible.

**Hardware-assisted deterministic replay:** Since software based deterministic replay systems usually impose large performance overhead, hardware-assisted deterministic replay systems [6, 14, 17, 19, 20, 21, 22, 32] are proposed to modify hardware components for recording data access conflicts efficiently. Many such systems apply optimizations to further reduce record overhead and log size. However, they impose non-trivial hardware complexity and there are still no commercially available processors built with these features.

## 7 Conclusion and Future work

This paper presented ORDER, the first object-centric deterministic replay system for concurrent Java applications on multicore. ORDER recorded interleaved data accesses in Java applications by tracking how each thread accesses each object and enforced such a constraint during replay. By dynamically instrumenting Java code in the compilation pipeline, ORDER naturally covered non-determinism in dynamically loaded classes and libraries. Evaluation results showed that ORDER achieved good performance and scalability for a range of benchmarks, which notably outperformed LEAP, a state-of-the-art deterministic replay system for Java. Bug reproducibility study further showed that ORDER successfully reproduced several real-world concurrency bugs.

While ORDER has demonstrated the efficiency and

effectiveness of recording and replay concurrent Java programs, there are still plenty of optimization spaces, which will be our future work. First, timeline filter of unnecessary dependencies is currently applied offline, and applying it online can reduce space overhead and eliminate unnecessary disk operations in the recording phase. Second, we plan to combine ORDER with techniques that cover the non-determinism in adaptive optimization [24] to enable adaptive optimization for JVM. Finally, we plan to combine ORDER with an object-level checkpointing mechanism to further reduce log size, and extend ORDER with some selective tracking mechanisms to focus on only interested objects, to further reduce performance overhead.

## Acknowledgments

We thank our shepherd Keith Adams and the anonymous reviewers for their insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 61003002 and 90818015, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100, IBM X10 Innovation Faculty Award, a research grant from Intel as well as a joint program between China Ministry of Education and Intel numbered MOE-INTEL-09-04, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

## References

- [1] Apache harmony. <http://harmony.apache.org>.
- [2] Fault tolerance in vmware. [http://www.vmware.com/files/pdf/perf-vsphere-fault\\_tolerance.pdf](http://www.vmware.com/files/pdf/perf-vsphere-fault_tolerance.pdf).
- [3] Optimizations in harmony. <http://wiki.apache.org/harmony/Jitrino.OPT>.
- [4] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proc. SOSP*, pages 193–206, 2009.
- [5] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proc. VEE*, pages 154–163, 2006.
- [6] Y. Chen, W. Hu, T. Chen, and R. Wu. Lreplay: a pending period based deterministic replay scheme. In *Proc. ISCA*, pages 187–197, 2010.
- [7] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proc. OOPSLA*, pages 1–19, 1999.
- [8] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. PLDI*, pages 258–269, 2002.
- [9] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proc. SIGMETRICS symposium on Parallel and distributed tools (SPDT)*, pages 48–59, 1998.
- [10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. USENIX ATC*, pages 1–14, 2008.
- [11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI*, pages 211–224.
- [12] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proc. VEE*, pages 121–130, 2008.
- [13] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Softw. Pract. Exper.*, 34(6):523–547, 2004.
- [14] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proc. ISCA*, pages 265–276, 2008.
- [15] J. Huang, P. Liu, and C. Zhang. Leap: The lightweight deterministic multi-processor replay of concurrent java programs. In *Proc. FSE*, 2010.
- [16] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. SIGMETRICS*, pages 155–166, 2010.
- [17] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proc. MICRO*, pages 564–575, 2009.
- [18] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS*, pages 329–339, 2008.
- [19] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proc. ISCA*, pages 289–300, 2008.
- [20] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proc. ASPLOS*, pages 73–84, 2009.
- [21] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *Proc. ASPLOS*, pages 229–240, 2006.
- [22] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proc. ISCA*, pages 284–295, 2005.
- [23] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. ACM/ONR workshop on Parallel and distributed debugging (PADD)*, pages 1–11, 1993.
- [24] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. Nakatani. Replay compilation: improving debuggability of a just-in-time compiler. In *Proc. OOPSLA*, pages 241–252, 2006.
- [25] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proc. ASPLOS*, pages 25–36, 2009.
- [26] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proc. SOSP*, pages 177–192, 2009.
- [27] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proc. CGO*, pages 2–11, 2010.
- [28] K. Ravi, S. Harini, and C. Jong-Deok. Deterministic replay of distributed java applications. In *Proc. IPDPS*, page 219, 2000.
- [29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, 1997.
- [30] V. Schuppan, M. Baur, and A. Biere. Jvm independent replay in java. *Elsevier Electron. Notes Theor. Comput. Sci.*, 113:85–104, 2005.
- [31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proc. conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 13–, 1999.
- [32] M. Xu, R. Bodik, and M. D. Hill. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. In *Proc. ISCA*, pages 122–135, 2003.
- [33] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and V. Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.
- [34] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In *Proc. OOPSLA*, pages 361–376, 2009.