LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs

Jeff Huang, Peng Liu, and Charles Zhang
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{smhuang, lpxz, charlesz}@cse.ust.hk

ABSTRACT

The technique of deterministic record and replay aims at faithfully reenacting an earlier program execution. For concurrent programs, it is one of the most important techniques for program understanding and debugging. The state of the art deterministic replay techniques face challenging efficiency problems in supporting multi-processor executions due to the unoptimized treatment of shared memory accesses. We propose LEAP: a deterministic record and replay technique that uses a new type of local order w.r.t. the shared memory locations and concurrent threads. Compared to the related work, our technique records much less information without losing the replay determinism. The correctness of our technique is underpinned by formal models and a replay theorem that we have developed in this work. Through our evaluation using both benchmarks and real world applications, we show that LEAP is more than 10x faster than conventional global-order based approaches and, in most cases, 2x to 10x faster than other local-order based approaches. Our recording overhead on the two large open source multi-threaded applications Tomcat and Derby is less than 10%. Moreover, as the evidence of the deterministic replay, LEAP is able to deterministically reproduce 7 out of 8 real bugs in Tomcat and Derby, 13 out of 16 benchmark bugs in IBM ConTest benchmark suite, and 100% of the randomly injected concurrency bugs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids; Tracing; Diagnostics*

General Terms

Algorithms, Performance, Reliability

1. INTRODUCTION

As concurrency becomes the major programming model of the performance improvement of software in the multi-core era, it is also the culprit of many so-called *Heisenbugs*, such as data races, deadlocks, and atomicity violations, that are easy to hatch but difficult to detect and to fix. One of the most effective ways for combating these bugs is the technique of record and replay [5, 14, 10, 25, 24, 1, 23, 7, 16, 12, 17, 19, 27]. The record and replay technique aims at fully reenacting the problematic execution of concurrent programs, thus giving the programmers both the context and the history information to dramatically expedite the process of bug fix.

A crucial design factor of record and replay solutions is the degree of recording fidelity, i.e., the amount of data to be recorded, for the sufficient reproduction of problematic program executions. Simply speaking, the degree of recording fidelity is proportional to the degree of faithfulness in replay, unfortunately, also to the runtime overhead of using the technique. This characteristic is less problematic for the hardware-based record and replay solutions [30, 20, 16, 12, 17], in which special chips share the cost of the recording computation. For the software-only solutions [19, 27] on uni-processors, the replay of concurrent programs can be achieved deterministically with low overhead by capturing the thread scheduling decisions. However, for software-only solutions on *multi-processors*, making the best trade-off between how much to record and how faithful to replay is still a very challenging problem, drawing intense research attentions [10, 25, 5, 14, 24, 1, 23].

Our research is also concerned with the software-only record and replay solutions. Our general observation is that the state of the art does not achieve the recording efficiency and the replay determinism[1] at the same time. Conventional deterministic multi-processor replay techniques usually incur a significant runtime overhead of 10x to 100x [14, 5, 7, 6], making them unattractive for production use or even for testing purposes. For instance, Dejavu [5] is a *global clock* based approach that is capable of deterministically replaying concurrent systems on multi-processors by assigning a global order to all "critical events", including both the synchronization points and the shared memory accesses. As indicated by the authors, the enforcement of the global order on variable accesses across multiple threads incurs a large runtime overhead on multi-processors. The research of lightweight record and replay techniques [10, 25, 24, 1, 23] has successfully lowered the recording overhead, however, at the cost of sacrificing the determinism in reproducing buggy runs. JaRec [10] and RecPlay [25] abolish the idea of global ordering and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-60558-791-2/10/11 ...$5.00.

[1]We define replay determinism as the faithful reenactment of all program state transitions experienced by a previous execution. A more complete and formal model is presented in Section 3.

use *Lamport clock* [13] to maintain partial thread access orders w.r.t. only the monitor entry and exit events, thus, making the recording process lightweight. However, without tracking the shared memory accesses, their approaches cannot deterministically reproduce problematic runs for the fact that a large majority of shared memory accesses are not synchronized, either due to programming errors or because they are harmless [21].

As also pointed out in [26], to deterministically replay a concurrent system on multi-processors, it is necessary to record the thread access orders of the shared memory locations, a method commonly believed to be too expensive to be practical [10, 25, 24, 1, 23]. In this paper, we demonstrate that it is possible to achieve efficiency in this approach by observing that, given the same program input, it is sufficient to deterministically replay the program execution by recording partial thread access information local to the individual shared variables. Based on this observation, we have designed and implemented LEAP, a replay tool that addresses both the recording efficiency and the replay determinism. The replay determinism is underpinned by a semantic model and formal theorems. To achieve efficiency, we use a field-based approach to statically identify shared variables, thus, avoiding the cost of runtime identification. In addition, we make extensive use of static analysis to provide a close approximation of the necessary program locations that need to be monitored and, thus, to prune away a large percentage of otherwise redundant recording operations.

The idea of the local-order based recording can be rooted back to InstantReplay [14], which enables the deterministic replay by recording the access history of all the shared objects w.r.t. a particular thread. This technique does not suit our design objectives of being both deterministic and efficient. First, InstantReplay requires the unique identification of shared objects dynamically, a task hard to be efficiently and correctly implemented in practice. Second, InstantReplay uses a complex computation model based on the CREW protocol, making the recording process very costly. Third, there are important soundness issues with the local-order based approaches that must be formally proved. Another local-order based approach is the use of Lamport clock that tracks the partial order of critical events that *each thread sees* [10, 25]. Our technique tracks the order of thread accesses that *each shared variable sees*, which is operationally simpler than the use of Lamport clock.

We evaluate the runtime performance of LEAP by comparing to the related techniques including the use of global clock, InstantReplay, and Lamport clock. Our micro-benchmark shows that LEAP is more than 10x faster than the global clock based approach, more than 5x faster than InstantReplay, and at least 2x faster than the use of Lamport clock. On real world large open source multi-threaded applications such as Tomcat and Derby, LEAP is 5x to 10x faster than the related approaches, measured by third-party benchmarks. The average runtime overhead of LEAP is less than 10% on Tomcat and Derby. Moreover, as the evidence of the replay correctness, LEAP is able to deterministically reproduce 7 out of 8 real concurrency bugs in Tomcat and Derby, 13 out of 16 benchmark bugs in IBM ConTest benchmark suite [8], and 100% of the randomly injected concurrency bugs.

In summary, this paper makes the following contributions:

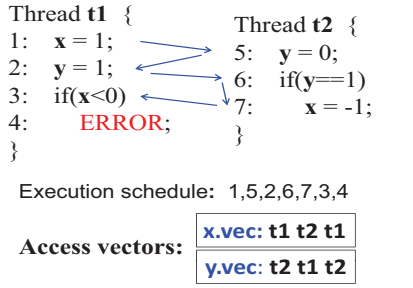
1. We present a new local-order based deterministic and efficient record and replay technique, LEAP. We provide a formal model of the concurrent program execution and use it to prove the soundness of our technique.

2. We describe the implementation of LEAP that uses static analysis and bytecode instrumentation to transparently provide the capability of the deterministic replay for Java programs without any user intervention.

3. We evaluate LEAP by first quantifying its differences compared to the state of the art recording techniques, including global clock, InstantReplay and Lamport clock. We then conduct thorough experiments to evaluate the correctness of LEAP by reproducing real and randomly injected bugs, using popular and computation-intensive concurrent applications.

The rest of the paper is organized as follows: Section 2 presents the technical details of LEAP; Section 3 presents the semantic model and proofs; Section 4 describes the implementation of LEAP; Section 5 evaluates LEAP; Section 6 reviews related work and Section 7 summarizes this paper.

Figure 1: Example code with races



2. LEAP: LOCAL-ORDER BASED DETERMINISTIC REPLAY

LEAP provides a general technique for the deterministic replay of concurrent programs on multi-processors. The main idea of LEAP is that each shared variable tracks the order of thread accesses it sees during execution. In this section, we present the core techniques of LEAP.

2.1 LEAP overview

We first use a simple example to show the main technique of LEAP and draw its differences as compared to the conventional global-order based approach to the deterministic replay. In Figure 1, we show a race condition that triggers an `ERROR` at line 4 following the interleaved execution order `<1,5,2,6,7,3,4>`. The global-order based approaches record this schedule and use it to re-execute the program at the cost of 7 global synchronization operations. Our observation is that thread accesses to different shared variables need not to be tracked together. Instead of enforcing a global order, we claim that it is sufficient to record the thread access order that each shared variable sees. In our example, instead of the global order vector, we use two **access vectors** (`x.vec` and `y.vec`) for the shared variables `x` and `y` and record `<t1,t2,t1>` and `<t2,t1,t2>` respectively. We require zero global synchronization operations and two groups of local synchronization operations executed in parallel. During replay, we associate `x` and `y` with conditional variables to enforce the access order of threads be identical to what is recorded in their respective access vectors.

Although our technique can be easily illustrated, to ensure

determinism and efficiency, there are many tough challenges that we must tackle:

1. *Static shared variable localization.* How to effectively locate shared variables statically? What will happen if we miss some shared variables, or some local variables are mistakenly recognized to be shared?

2. *Consistent shared variable and thread identification across runs.* How to match the identities of shared variables and of threads between the recording run and the replay run? For example, the deterministic replay would fail if the shared variable `x` at record is incorrectly recognized as `y` at replay, or the thread `t1` is mistakenly recognized as `t2`.

3. *Non-unique global order.* Keen readers may point out that, by only recording the thread access orders each variable sees, LEAP will permit a global thread schedule that is different from the recording run. For instance, in our example, LEAP also permits the global order `<5,1,2,6,7,3,4>`. Will this affect the faithfulness of the replay?

In the rest of the section, we focus on discussing the first two issues. The soundness of our approach associated with the third issue is fundamental to our technique. In Section 3, we provide a formal semantic model and proofs to show this phenomenon does not affect the faithfulness of the replay.

2.2 Locating shared variable accesses

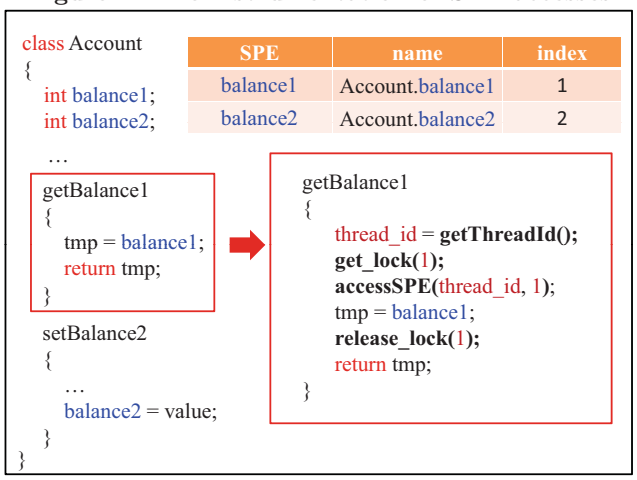
Precisely locating shared variables is generally undecidable [4]. We therefore compute a complete over-approximation using a static escape analysis in the Soot[2] framework called *ThreadLocalObjectAnalysis* [11]. *ThreadLocalObjectAnalysis* provides on demand answers to whether a variable can be accessed by multiple threads simultaneously or not. However, there are a few important issues with this analysis. First, static analysis is inherently conservative, as local variables might be reported as shared. We show in Section 3 (Corollary 1) that this type of conservativeness does not affect the correctness of the deterministic replay. Second, *ThreadLocalObjectAnalysis* does not distinguish between read and write accesses. For shared immutable variables, of which the values never change after initialization, we do not need to address them for they cannot cause nondeterminism. Third, we discover that static variables are all conservatively reported as escaped in *ThreadLocalObjectAnalysis*. Since the static variables might also be accessed only by one thread, we wish them to be analyzed in the same way as the instance variables, in order to obtain a more precise result. Thus, we make two enhancements to the *ThreadLocalObjectAnalysis*: 1. we further refine the analysis results of *ThreadLocalObjectAnalysis* so that we do not record accesses to shared immutable variables; 2. we modify *ThreadLocalObjectAnalysis* to treat static variables in the same way as instance variables.

2.3 Field-based shared variable identification

For Java programs, since the standard JVMs do not support the consistent object identification across runs, we cannot use the default object hash-code. We use a static field-based shared variable identification scheme, applied to the following three categories of variables, which are collectively referred to as the *shared program elements* (SPE): 1. variables that serve as monitors; 2. class variables; 3. thread escaped instance variables. These SPEs include both Java monitors and shared field variables that may cause nondeterminism. SPEs are uniquely named as follows: for category 1, it is the name of the declaring type of the object variable; for category 2 and 3, it is the variable name, combined with the name of the class, in which the variable is declared.

[2]http://www.sable.mcgill.ca/soot

Figure 2: The instrumentation of SPE accesses



After obtaining all the SPEs in the program, LEAP assigns *offline* to each SPE a numerical index as its runtime identifier. For example, in Figure 2, suppose the two field variable `balance1` and `balance2` of the `Account` class are identified as shared, they are mapped to the numerical IDs `1` and `2`.

The static field-based shared variable identification remains consistent across runs and does not incur runtime overhead. Moreover, compared to the object level identification approaches [14], this approach is more fine-grained as different fields of the same object are mapped to different indices. Consequently, accesses to different fields of the same object do not need to be serialized at the runtime.

There are a few issues with our field-based shared variable identification. First, our approach does not statically distinguish between different instances of the same type. As a result, accesses to the same shared field variable of different instances of the same type would be serialized and recorded into the same access vector. For this concern, we formally prove in Section 3 (Corollary 2) that the deterministic replay is also guaranteed, if the thread accesses to different shared variables are recorded globally into a single access vector. Second, we cannot uniquely identify scalar variables that are alias to shared array variables. To deal with this issue, we perform an alias analysis for all of the scalar array variables in the program and represent all the aliases with the same SPE, ignoring the indexing operations. This treatment guarantees that the nondeterminism caused by array aliases can be correctly tracked, however, at the cost of reducing the degree of concurrency. Fortunately, in our experiment, we find very few such cases in large Java multi-threaded applications. A good object-oriented program rarely manipulates shared array data directly, so they are rarely escaped.

2.4 Unique thread identification

Since the thread identity is the only information recorded into the access vectors, we must make sure that a thread at

the recording phase is correctly recognized during replay. A naive way is to keep a mapping between the thread name and the thread ID during recording and use the same mapping for replay. However, different parent threads can race with each other on creating their child threads. Therefore, the thread ID assignment is not fixed across runs.

To address this problem, LEAP introduces additional synchronization operations at the thread creation time to ensure the same thread creation order across the recording run and the replay run. More specifically, LEAP maintains a list that records the IDs of the parent threads in the global order of their thread creation operations. The list is used to direct the thread creation order at replay.

2.5 Handling early replay termination

Our local-order based approach permits different global schedules for threads that do not affect each other's program states. One caveat of this approach is that it gives rise to the possibility of early termination: a program *crash* action, compared to the original one, might be performed "earlier" in the replay execution, thus, making the replayed run not fully identical to the recording run in terms of its behavior. To faithfully replay all the thread execution actions, we ensure that every thread in the replay execution performs the same number of SPE accesses as it does in the recording execution. Consequently, we guarantee that the replay execution does not terminate until all the recorded actions in the original execution are performed, thus making the final state of the replayed execution the same as that of the original one.

3. A THEOREM OF LOCAL ORDERING

In this section we formally prove the soundness of our local-order based approach for the deterministic replay of shared memory races. We also use two corollaries to show the soundness the field-based shared variable identification approach and the soundness of using an unsound but complete static escape analysis for the deterministic replay.

3.1 Modeling concurrent program execution

To provide a basis for our proof, we first define the execution semantics of concurrent programs in a style similar to [9]. We consider a program comprised of a set of concurrently executing threads $\mathbb{T} = \{t_1, t_2, ...\}$, communicating through a global store σ. The global store consists of a set of variables $\mathbb{S} = \{s_1, s_2, ...\}$ that are shared among threads, and we use $\sigma[s]$ to denote the value of the shared variable s on the global store. Each thread has also its own local store π, consisting of the local variables and the program counter to the thread. Each thread executes by performing a sequence of actions on the global store and its own local store. Each action α is a single indivisible access[3] on a single variable. We use $\Gamma(\alpha)$ to denote the owner thread of action α and $var(\alpha)$ the variable accessed by α. If $var(\alpha)$ is a shared variable, we call α a *global action*, otherwise it is a *local action*. The program state is defined as $\Sigma = (\sigma, \Pi)$, where σ is the global store and Π is a mapping from thread identifiers t_i to the local store π_i of each thread.

[3]The access could be a read, a write, a lock acquisition, a lock release, a message sending or a message receiving [22]. Nevertheless, in our setting, we do not necessarily need to distinguish between different access types.

The program execution is modeled as a sequence of transitions defined over the program state Σ. Let α^k be the kth action in the global order of the program execution and Σ^{k-1} be the program state just before α^k is performed (Σ^0 is the initial state), the state transition sequence is:

$$\Sigma^0 \xrightarrow{\alpha^1} \Sigma^1 \xrightarrow{\alpha^2} \Sigma^2 \xrightarrow{\alpha^3} \dots$$

Given a concurrent system described above, we next formally define the execution semantics of action α. To give a precise definition, we first introduce some additional notations similar to [9]:

- $\sigma[s := v]$ is identical to σ except that it maps the variable s to the value v.
- $\Pi[t_i := \pi_i]$ is identical to Π except that it maps the thread identifier t_i to π_i.

Let the relation $\sigma \xrightarrow{\alpha} \sigma'$ models the effect of performing an action α on the global store σ, and $\pi \xrightarrow{\alpha} \pi'$ models the effect of performing α on the local store π. The execution semantics of performing α are defined as:

$$\frac{var(\alpha) \notin \mathbb{S} \qquad \Gamma(\alpha) = t_i \qquad \pi_i \xrightarrow{\alpha} \pi'_i}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma, \Pi[t_i := \pi'_i])}$$

$$\frac{var(\alpha) = s \in \mathbb{S} \quad \Gamma(\alpha) = t_i \quad \pi_i \xrightarrow{\alpha} \pi'_i \quad \sigma[s] \xrightarrow{\alpha} \sigma'[s]}{(\sigma, \Pi) \xrightarrow{\alpha} (\sigma[s := \sigma'[s]], \Pi[t_i := \pi'_i])}$$

These semantics simply take different kinds of actions into consideration. The first case means that when a local action is performed by a thread, only the local store of the thread is changed to a new state determined by its current state. The global store and the local stores of other threads remain the same. The second case means that when a global action is performed by a thread t_i on the shared variable s, only s and π_i are changed to new states. The states of other shared variables on the global store as well as the local stores of other threads remain the same. Also, consider the action sequence local to each thread, since the program counter is included in the local store, the next action of t_i should be determined by t_i's current local store π_i.

The execution semantics defined above conform to a general concurrent execution model with deterministic input. Although dynamic thread creation and dynamic shared variable creation are not explicitly supported by the semantics, they can be modeled within the semantics in a straightforward way [9].

3.2 Equivalence of execution schedules

The action sequence $\langle \alpha^k \rangle$ of a program execution is called an *execution schedule* denoted by δ. Suppose there is an execution schedule δ of size N that drives the program state to Σ^N, our goal is to have another execution schedule δ' that is able to produce the same program state as Σ^N. Obviously, this can be achieved if $\delta' = \delta$ holds. However, this is too strong a condition. We show a relaxed and sufficient condition based on the access vectors of all the shared variables. To state precisely, let δ_s be the sequence of actions w.r.t. s projected from δ, τ_s be the sequence of thread identifiers picked out from δ_s, and τ be the mapping from s to τ_s for $\forall s \in \mathbb{S}$ (τ is the access vectors of all the shared variables), we prove:

Theorem 1. *Under the execution semantics defined in Section 3.1, two execution schedules δ and δ' of the same concurrent program have the same final state $\Sigma^N = \Sigma'^N$ if $\Sigma^0 = \Sigma'^0 \wedge \tau = \tau'$.*

The core of the proof is to prove the following lemma:

Lemma 1. *For any action α'^k $(k \leq N)$ in the replay execution δ', suppose it is the pth action on a shared variable s, then α'^k is equal to the pth action on s in the original execution δ.*

For two actions to be equal here, they need to read and write the same values, not just do the same operation on the same shared variable. Next, we first define a notion of "happened-before", and then we prove Lemma 1 using this notion.

Consider the *"happened-before"* order of the original execution. The "happened-before" relation is defined as follows:

(a) if action α_i immediately preceded action α_j in the same thread, then α_i happened-before α_j;

(b) if action α_i and action α_j by different threads are consecutive actions on a shared variable s, without any intervening actions on s, then α_i happened-before α_j;

(c) "happened-before" is reflexive and transitive.

More accurately, rules (a) and (b) define *"happened-immediately-before"* and "happened-before" is the reflexive transitive closure of "happened-immediately-before".

Proof. Let's say the "happened-before" tree of an action is the tree of all the actions that "happened-before" it, we next prove Lemma 1 by induction on the depth of the "happened-before" tree.

Base case: Consider an action on the shared variable s, with a "happened-before" tree of depth 1. This means that the current action does not depend on anything that happened-before it involving shared variables. Because the first action on a shared variable is performed by the same thread in both the original and the replay execution, and because that thread is deterministic, the replay action should be identical to the one in the original execution.

Induction: Now assuming that Lemma 1 holds for all actions with happened-before depth $\leq n$, we prove it for $n+1$. Consider an action α_i on a shared variable s, where α_i has a tree of happened-before depth $n+1$. Let's say α_i is the pth action on s. The $(p\text{-}1)^{th}$ action on s has a lower happened-before depth so it is an equal action in both the original and the replay execution. Additionally, every action α_j that "happened-immediately-before" α_i has a happened-before tree of depth n, therefore it is equal to a similarly numbered action in the original execution (i.e., if α_j is the kth action on a shared variable v, then α_j is equal to the kth action on v in the original execution). Now action α_i only depends on all the α_j actions. So, since our approach enforces that the pth action on s is performed by the same thread in both executions, and since the thread is deterministic and every value that α_i can depend on has to be equal to each other, it follows that action α_i is also equal in the original and replay executions.

Lemma 1 is proved. If we apply Lemma 1 to the last action α'^N in the replay execution, we can get $\Sigma'^N = \Sigma^N$. Thus, Theorem 1 is proved. □

With Theorem 1, we have proved the soundness of local-order based approaches for the deterministic replay that is able to reach the same program state as the original execution, by only recording the access vectors for all the shared variables.

While $\tau = \tau'$ is a rather relaxed condition, we can surely add more information that also guarantees the deterministic replay. For example, if the local variable accesses are recorded, the deterministic replay is still guaranteed as long as we do not miss any shared variable accesses. Following we derive two corollaries:

Corollary 1. *The deterministic replay holds as long as $\tau = \tau'$, regardless of whether accesses to local variables are recorded or not.*

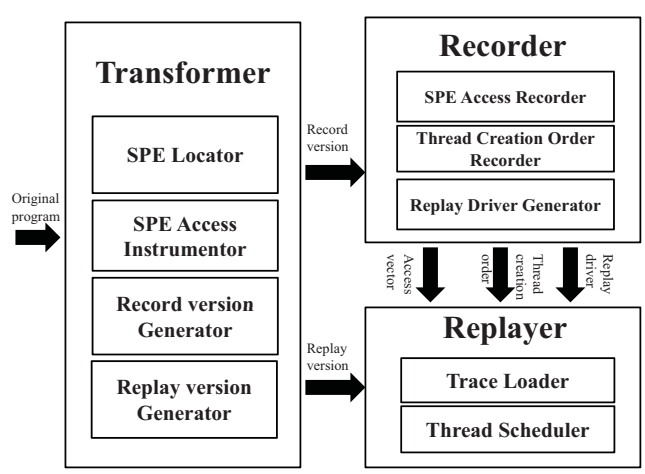
Corollary 2. *Recording different shared variable accesses into a single access vector does not affect the correctness of the deterministic replay.*

As noted in Section 2.2, the static escape analysis is conservative such that local variables might be mistakenly categorized as shared. Corollary 1 ensures that this conservativeness does not affect the correctness of the deterministic replay as long as all the shared variables are correctly identified. Corollary 2 is easy to understand as the thread access orders on different shared variables can be considered as a global order on a single variable abstracted from these shared variables. To be more clear, assuming all thread accesses are recorded into a global access vector, it is a global order of the execution schedule and the determinism must hold. As noted in Section 2.3, Corollary 2 ensures the soundness of our field-based shared variable identification.

4. LEAP IMPLEMENTATION

We have implemented LEAP using the Soot[2] framework. Figure 3 shows the overview of the LEAP infrastructure, consisting of the transformer, the recorder, and the replayer. The transformer takes the bytecode of an arbitrary Java program and produces two versions: the record version and the replay version. Started by a record driver, LEAP collects the access vector for each SPE during the execution of the record version. When the recording stops, LEAP saves both the access vectors and the thread creation order information

Figure 3: The overview of LEAP infrastructure



and generates a replay driver. To replay, the LEAP replayer uses the generated replay driver as the entry point to run the replay version of the program, together with recorded information. The replayer takes control of the thread scheduling to enforce the correct execution order of the threads w.r.t. the SPEs. We now introduce each of the components in turn.

4.1 The LEAP transformer

The LEAP transformer performs the instrumentation on Jimple, an intermediate representation of Java bytecode in the three-address form. For the record version, after locating all the SPEs in the program, the transformer visits each Jimple statement and performs the following tasks:

Instrumenting SPE accesses If the SPE is not a Java monitor object, we insert a LEAP monitoring API invocation before the Jimple statement to collect both the thread ID and the numeric SPE ID. Both the API call and the SPE access are wrapped by a lock specific to the accessed SPE to ensure we collect the right thread accessing order seen by the SPE. If the SPE is a Java monitor object, we insert the monitoring API call *after* the `monitorentry` and *before* the `monitorexit` instructions. The API call is also inserted *before* `notify/notifyAll/thread start` operations and *after* `wait/thread join` operations. Figure 2 shows a source-code equivalent view of the instrumentation on the read/write accesses to the shared field variables. The box on the left shows the original method `getBalance1`, inside of which the shared variable `balance1` is read. The box on the right shows the transformed version of `getBalance1`. For multiple shared variable accesses in a method, the thread ID needs only to be obtained once. Also, to remove the unnecessary recording overhead, we do not need to instrument the SPEs that are always protected by the same monitor.

Instrumenting thread initialization To capture the thread identity information, as described in Section 2.4, the transformer inserts the instrumentation code inside the Thread constructor to synchronize and to collect the thread creation order.

Instrumenting recording end points To enable the deterministic replay, we insert the recording end points to save the recorded runtime information and to generate the replay driver. Currently, LEAP supports three types of recording end points. First, we add a *ShutDownHook* to the JVM Runtime in the record driver as a recording end point. When the program ends, the ShutDownHook will be invoked to perform the saving operations. Second, we insert a try-catch block into the `main` thread and the `run` method of each Java Runnable class. We then add a method invocation in the catch block to capture the uncaught runtime exceptions as the recording end points. Third, LEAP also supports the user specified recording end points by allowing the annotation-based specification of end points. During the traversal of the program statements, the transformer will replace the annotation with a method invocation, indicating the end of recording.

To generate the replay version, the transforming process is largely identical to the record version with a few differences: 1. since the order of synchronization operations on each SPE is controlled by the LEAP replayer during replay, we need to insert the API call *before* the original synchronization operations in the program, i.e, `monitorenter` and `wait`, to avoid deadlock; 2. the inserted API call is bound to a different implementation from the one used during the recording phase; 3. since we need to ensure that the replay execution does not terminate until all recorded actions in the original execution have been executed (See Section 2.5), we insert extra API invocations *after* each SPE access so that we can check whether a thread has performed all its recorded actions in the original execution or not.

4.2 The LEAP recorder

When executing the record version of the target program, the LEAP monitoring API will be invoked on each critical event to record the ID of the executing thread into the access vector of the accessed SPE. To reduce the memory requirement, we use a compact representation of the access vectors by replacing consecutive and identical thread IDs with a single thread ID and a corresponding counter. For example, suppose the access vector of a SPE contains `<t1,t1,t2,t2,t2>`, it is replaced by `<t1,t2>` and a corresponding counter `<2,3>`. This compact representation produces much smaller log size compared to the related approaches in our experiment.

Once a new thread is created, we add the parent thread ID to the thread creation order list. Once a program end point is detected, the LEAP recorder will then save the recorded data, i.e, the recorded access vectors, and the thread creation order list, and generate the replay driver.

4.3 The LEAP replayer

The LEAP replayer controls the scheduling of threads to enforce a deterministic replay using both the access vectors and the thread identity information. To enable the user level thread scheduling, the replayer associates each thread in replay with a semaphore maintained in a global data structure, so that each thread can be suspended and resumed on demand.

To replay, the replay driver first loads both the saved access vectors and the thread creation order list and starts executing the replay version of the program. Before a new thread is created, the ID of the parent thread is compared to the ID at the head of the thread creation order list. If they are the same, the new thread is allowed to be created and the head of the list is removed. In this way, the identification of each thread is guaranteed to be the same as that of the recording phase. Before each SPE access, the threads use their semaphores to coordinate with each other in order to obey the access order defined in the access vector of the SPE. Also, to make sure that the replay execution does not terminate "early", the thread also counts the total number of SPE accesses it has performed so far after each SPE access. The thread suspends itself if it finds that it has already executed all its SPE accesses in the original execution, as recorded in the access vector, until all threads have finished their recorded actions. Since the threads accessing different SPEs can execute in parallel, the replaying process is also faster than that of a global order scheduler, which can only execute one thread each time.

5. EVALUATION

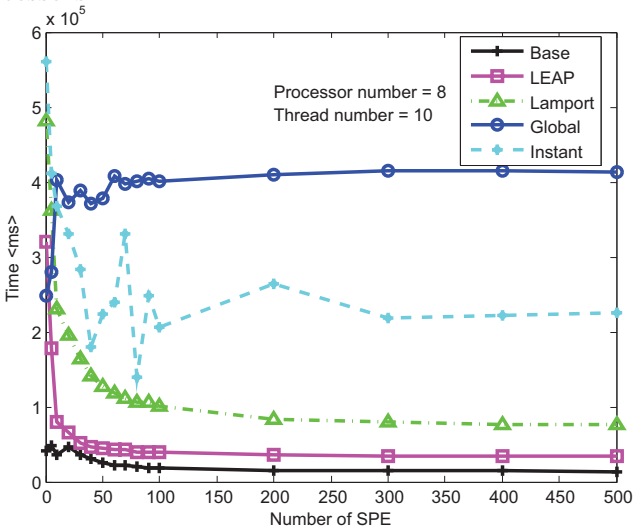
5.1 Evaluation methodology

We assess the quality of LEAP by quantifying both its recording overhead and the correctness of the deterministic replay. To properly compare our technique to the state

of the art, we have also implemented the following techniques: the Dejavu approach based on the global clock [5], the technique presented by InstantReplay [14], and the JaRec approach based on the Lamport clock [13]. Because none of these tools are publicly available, we faithfully implemented them according to their representative publications. Since JaRec is not a deterministic replay technique, we extended its capability to tracking shared memory races, in order to make it comparable to our technique. Our implementations are publicly available[4].

For the evaluation, we first design a micro-benchmark to conduct controlled experiments for quantifying various runtime characteristics of the evaluated techniques. We then use real complex Java server programs and third-party benchmarks to assess the recording overhead of LEAP in comparison to the related approaches. We use the bug reproducibility as a way to verify if our technique can faithfully and deterministically reproduce problematic concurrent runs. All experiments are conducted on two 8-core 3.00GHz Intel Xeon machines with 16GB memory and Linux version 2.6.22. We now present these experiments in detail.

Figure 4: The runtime characteristic of LEAP and other techniques on our microbenchmark with the number of SPE ranges from 1 to 500. The microbenchmark starts 10 threads running on 8 processors.



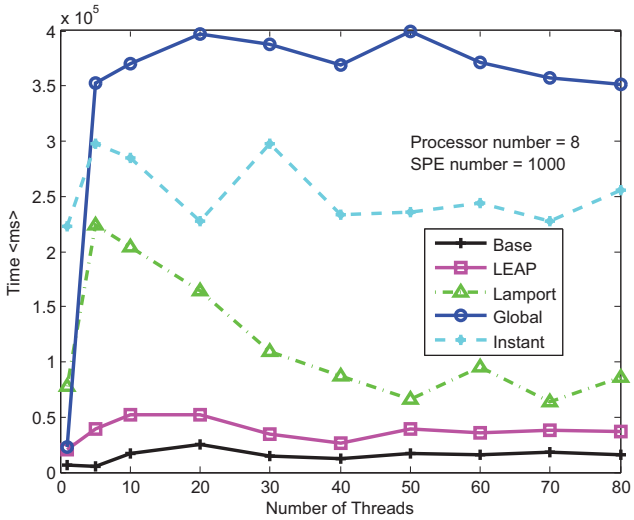
5.1.1 Micro-benchmarking

We have designed a micro-benchmark to quantify the runtime characteristics of LEAP and the related record and replay techniques. The benchmark consists of concurrent threads that randomly update shared variables in a loop. For each experiment, we can control the number of threads and shared variables. In our experiments, we set the number of threads ranging from 1 to 100, and the number of shared variables ranging from 1 to 1000, we then measure the time needed for all the threads to finish a fixed total number of updating operations under different settings.

Figure 4 and 5 show the runtime characteristics of LEAP and the related techniques on our micro-benchmark. In the figures, *Base* refers to the native execution. *Global*, *Lamport* and *Instant* refer to the recorded execution using global clock, Lamport clock and InstantReplay respectively. Figure 4 shows that the performance of the LEAP instrumented version is close to the base version. By fixing the number of threads to 10, as the number of SPE increases from 10 to 500, LEAP is more than 10x faster than global clock, more than 5x faster than InstantReplay, and at least 2x faster than Lamport clock. Global clock is the slowest among the four techniques. The main reason is that the use of global clock requires a global synchronization on every shared variable access, which significantly affects the degree of concurrency. Figure 5 shows a similar performance trend as the number of threads increases from 10 to 80 and the number of SPEs is fixed to 1000.

[4]http://www.cse.ust.hk/prism/leap

Figure 5: The runtime characteristic of LEAP and other techniques on our microbenchmark with the number of threads ranges from 1 to 80 running on 8 processors. The number of SPE is set to 1000.



5.1.2 Benchmarking with third-party systems

To perform the unbiased evaluation, we first use LEAP on two widely used complex server programs, Derby and Tomcat, with the PolePosition[5] database benchmark and the SPECWeb-2005[6] web workload benchmark. Each benchmark starts with 10 threads and we measure the time for finishing a total number of 10000 operations. We also selected a suite of third-party benchmarks, among which Avrora and Lusearch are from the dacapo-9.12-bach benchmark suite[7], and MolDyn, MonteCarlo and RayTracer are from the Java Grande multi-thread benchmark suite.

Table 1 shows some of the relevant static attributes of the benchmarked programs as well as the associated runtime overhead of the evaluated record and replay techniques. We report the total number of field variable accesses in the program (*Total*), the total number instrumented SPE accesses (*SPE*), the number of SPEs (*SPESize*), the log size (KB/sec) of the related approaches (*Log*), the log size of LEAP (*LogCmp*), and the runtime overhead (*LEAP*, *Lamport*, *Instant* and *Global*). Overall, the percentage of SPE

[5]http://polepos.sourceforge.net
[6]http://www.spec.org/web2005
[7]http://dacapobench.org

Table 1: The runtime overhead of LEAP and the state-of-the-art techniques

Application	LOC	Total	SPE	SPESize	Log	LogCmp	LEAP	Lamport	Instant	Global
Avrora	93K	16003	1725(11%)	113	30623	796	626%	1697%	1821%	1036%
Lusearch	69K	11497	1140(9.9%)	75	7485	632	74%	308%	379%	227%
Derby	1.51M	48356	1433(3.0%)	264	18545	113	9.9%	68%	113%	52%
Tomcat	535K	23046	654(2.6%)	163	15351	51	7.3%	39%	44%	34%
MolDyn	864	821	634(77%)	66	110761	37760	64%	2776%	3567%	9960%
MonteCarlo	3128	427	104(24%)	18	70384	1994	7.5%	7.9%	8.6%	9.1%
RayTracer	1431	442	223(50%)	19	124239	35878	18%	39%	43%	94%

accesses over the total number of field variable accesses varies from less than 3% on Derby and Tomcat to around 10% on Avrora and Lusearch. As MolDyn (77%), MonteCarlo (24%) and RayTracer (50%) are relatively small applications dedicated for multi-threaded benchmarking, the percentage of their SPE accesses is large.

Log size By using our compact representation of the access vectors, the log size of LEAP is much smaller than the related approaches, from 3x in MolDyn to as large as 164x in Derby. We recognize that the log size in LEAP is still considerable ranging from 51 to 37760 KB/sec. With the increasing disk capacity and disk write performance, as also observed by other researchers [24], moderate log size does not pose serious problem. For long running programs, we can reset logs through the use of checkpoints.

Recording overhead LEAP is the fastest on all the evaluated applications. It is even more than 150x faster than global clock on MolDyn. For Derby and Tomcat, LEAP is 5x to 10x faster than all the related approaches. The sheer runtime overhead of LEAP on Derby and Tomcat is less than 10% (9.9% and 7.3% respectively). LEAP's overhead is large on Avrora (626%), the reason is that there are several SPEs in Avrora that are frequently accessed in hot loops.

5.1.3 Concurrency bug reproduction

One of the major motivating forces for the record and replay technique is to help reproducing so-called Heisenbugs. We believe that the ability of deterministically reproducing a concurrency-related bug is a strong indicator of the replay correctness, because it requires the program state to be correctly restored for the bug to be triggered. To compare the bug reproducibility, we have also implemented JaRec for the comparison. We first compare LEAP and JaRec for their capabilities of reproducing real-world concurrency bugs happened in complex server systems as well as a number of benchmark bugs widely used in concurrency testing. To proper quantify the bug reproducibility, we have also designed a bug injection technique that injects atomic set violations into our micro-benchmark. We then assess how many of the violations can be deterministically replayed by LEAP and JaRec.

Random bug injection

Our bug injection technique is based on the problematic thread interleaving patterns presented in [29]. We introduce 10 dummy shared variables into the program and divide them into 5 groups, each group representing an atomic set as defined in [29]. During the recording phase, on each critical event, the thread also *randomly* performs a *write* or *read* access on one of the introduced variables. We use the same random seed for each thread across record and replay. After each random access, if one of the problematic thread interleaving patterns occurs, the program stops and the replay data are exported. Given the same program input, a deterministic replay technique should be able to recreate the occurred bug pattern.

To compare the concurrency bug reproducibility between LEAP and JaRec, we use 100 different random seeds to inject 100 concurrency bugs into our micro-benchmark. For each run, we initialize 10 threads in the program. LEAP is able to deterministically reproduce 100% of these bugs, while JaRec cannot deterministically reproduce any of them. The reason is that JaRec does not record shared memory races, while all these bug patterns are generated on shared memory accesses.

Table 2: Summary of the evaluated real bugs

Bug Id	Version	LOC	Exception Type
Derby230	Derby-10.1	1.34M	DuplicateDescriptor
Derby1573	Derby-10.2	1.52M	NullPointerException
Derby2861	Derby-10.3	1.51M	NullPointerException
Derby3260	Derby-10.2	1.52M	SQLException
Tomcat728	Tomcat-3.2	150K	NullPointerException
Tomcat4036	Tomcat-3.3	184K	NumberFormatException
Tomcat27315	Tomcat-4.1	361K	ConcurrentModification
Tomcat37458	Tomcat-5.5	535K	NullPointerException

Table 3: Summary of the evaluated benchmark bugs

Bug Name	LOC	Bug Description
BubbleSort	362	Not-atomic, Orphaned-Thread
AllocationVector	286	Weak-reality, two stage access
AirlineTickets	95	Not-atomic interleaving
PingPong	272	Not-atomic
BufferWriter	255	Wrong or no-Lock
RandomNumbers	359	Blocking-Critical-Section
Loader	130	Initialization-Sleep Pattern
Account	155	Wrong or no-Lock
LinkedList	416	Not-atomic
BoundedBuffer	536	Notify instead of notifyAll
MergeSort	375	Not-atomic
Critical	73	Not-atomic
Deadlock	135	Deadlock
DeadlockException	255	Deadlock
FileWriter	311	Not-atomic
Manager	236	Not-atomic

Real and benchmark concurrency bugs

Table 2 and 3 show the description of the real concurrency bugs and the benchmark bugs used in our experiments. All the 8 real bugs in Table 2 are extracted from the Derby and

Tomcat bug repositories[8] that were reported by users. The 16 benchmark bugs in Table 3 are from the IBM ConTest benchmark suite [8], which cover the major types of concurrency bugs, including data races, atomicity violation, order violation, and deadlocks. We also run both JaRec and LEAP on these buggy programs to compare the bug reproducibility between them.

For the 8 real world concurrency bugs, LEAP is able to deterministically reproduce 7 of them (88%), except the bug `tomcat4036`, and JaRec reproduced none of them. For the 16 benchmark bugs, LEAP can reproduce 13 of them (81%), except `BufferWriter`, `Loader`, and `DeadlockException`, while JaRec can only reproduce one of them (`Deadlock`). The reason for LEAP to miss `tomcat4036` is that the bug is triggered by races of the internal data of the underlying JDK library `java.text.DateFormat`, which LEAP does not instrument. And because all these real bugs are related to shared memory races, JaRec are not able to reproduce any of them. For the three benchmark cases LEAP cannot reproduce, two of them are related to random numbers and the other one makes LEAP `OutOfMemory` because too many threads (>5000) are involved in loops.

5.2 Discussion

The evaluation results have clearly demonstrated the superior runtime performance of LEAP as well as its much higher concurrency bug reproducibility, compared to existing approaches. Through our experiments with real world large multi-threaded applications, we observed several limitations of LEAP that we plan to address in our future work:

Input nondeterminism As LEAP only captures the nondeterminism brought by thread inter-leavings, it may not reproduce executions containing input nondeterminism, e.g., programs with nondeterministic I/O. The two benchmark bugs that LEAP cannot reproduce both contain random number generators that use the current system time as the random seed. Since it is not likely to keep the random numbers the same across record and replay without saving them, LEAP may not reproduce executions that contain such random issues. A way to overcome these issues is to save the program states of some key nondeterministic events, e.g., the value of random seeds. We set this as our future work.

JDK library LEAP does not record shared variable accesses in the underlying JDK library. If an execution contains races of the internal data of these APIs, LEAP might not be able to reproduce it. The bug `tomcat4036` is an example of this limitation. In fact, we can also instrument the underlying Java Runtime, but as the JDK library is used frequently, it would incur large runtime overhead. An implementation of LEAP on the JVM should relieve this issue as the JVM environment enables efficiently tracing the internal data of the JDK library. We also set this as our future work.

Long running programs LEAP currently has to replay from the beginning of the program execution. For long running programs, it might not be convenient to replay the whole program execution concerning the long replay time and the large log size. We plan to extend LEAP to use a lightweight checkpoint scheme that only replays the program from the last checkpoint to the recording end point.

[8]https://issues.apache.org

6. RELATED WORK

As the deterministic replay of concurrent programs is of such significant importance, there have been enumerable research efforts on this topic. In this section we briefly review some of the other key software-only related work.

Record/replay PRES [24] and ODR [1] are two recent projects that use record/replay for the reproduction of concurrency bugs. PRES proposes a novel technique that uses a feedback replayer to explore thread interleaving space, which reduces the recording overhead at the price of more replay attempts. ODR proposes a new concept, output-deterministic replay, that focuses on replaying the same program output, and uses a similar idea as PRES that depends on offline inference to help recording less online. SMP-ReVirt [7] makes use of hardware page protection to detect shared memory accesses, aiming at replaying multi-processor virtual machines, but its overhead can increase upto 10x on multi-processors. To avoid the overhead of recording memory races, RecPlay [25] and Kendo [23] provide deterministic multi-threading of concurrent programs that perfectly synchronized using locks. Unfortunately, most real world concurrent applications may contain benign or harmful data races, making these approaches unattractive. Though RecPlay and Kendo both use a data race detector during replay to ensure the deterministic replay up until the first race, they suffer from the limitation that they cannot replay past the data race. For instance, while debugging using a replayer, a programmer might want to understand the after effects of a benign data race, which is not possible with RecPlay and Kendo.

Deterministic by default There are also approaches to the nondeterminism in concurrency by making concurrent programs deterministic by default. In this direction, there have been language design approaches [3, 2] as well as hardware ones [6, 31]. For example, languages such as DPJ [3] guarantee deterministic semantics by providing a type and effect system to perform compile-time type checking. The problem with language level approaches is that they often require nontrivial programmer annotations or have a limited class of concurrency semantics. Hardware approaches such as DMP [6] make inter-thread communication fully deterministic by imposing a deterministic commit order among processors. PSet [31] eliminates untested thread inter-leavings by enforcing the runtime to follow a tested interleaving via processor support. Because hardware approaches rely on non-standard hardware support, they are limited to proprietary platforms. Though DMP [6] also proposes a software-only algorithm, its overhead is more than 10x.

Code analysis tools Another line of approaches is to use code analysis tools [15] or model checkers [18] to try to eliminate concurrency bugs offline. Code analysis tools suffer from inaccuracies and false positives. Model checkers statically explore all thread schedules, which is hard to scale to large programs. Though CHESS [18] employs a context-bounded way to reduce the search space, it may miss most of the concurrency bugs in theory. RaceFuzzer [28] is another representative technique that given a potential race pair it controls a race directed random thread scheduler to actively create real races. As RaceFuzzer has only partial information of the races, it suffers from the limitation of nondeterminism.

7. CONCLUSION

We have presented LEAP, a new local-order based approach that deterministically replays concurrent program executions on multi-processors with low overhead. Our basic idea is to capture the thread access history of each shared variable, and we use theoretic models to guarantee its correctness. We have implemented LEAP as an automatic program transformation tool that provides the deterministic replay support to arbitrary Java programs. To evaluate our technique, we make use of both benchmarks and real world concurrent applications. We extensively quantified the runtime overhead of using LEAP as well as the correctness of the LEAP-based replay through reproducing concurrency bugs. Our evaluation shows that, compared to the state of the art, LEAP incurs much lower runtime overhead and has much superior capability of correctly reproducing concurrency bugs. For real world applications that we evaluated, the overhead of using LEAP is under 10%, exhibiting the great potential for the production use. We have also discussed some limitations that we have observed during our experimentation, and these limitations are the focus of our future work.

8. ACKNOWLEDGEMENT

The authors wish to express deep appreciation to the anonymous FSE reviewers for their insightful and constructive comments on an early draft of this paper. This research has been supported by RGC GRF grant 622208. The authors are very grateful for this support.

9. REFERENCES

[1] Gautam Altekar and Ion Stoica. Odr: output deterministic replay for multicore debugging. In *SOSP*, pages 193–206. ACM, 2009.

[2] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA*, pages 81–96. ACM, 2009.

[3] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA*, pages 97–116. ACM, 2009.

[4] Eric Bodden and Klaus Havelund. Racer: effective race detection using aspectj. In *ISSTA*, pages 155–166. ACM, 2008.

[5] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT*, pages 48–59. ACM, 1998.

[6] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multi-processing. In *ASPLOS*, pages 85–96. ACM, 2009.

[7] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, pages 121–130. ACM, 2008.

[8] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS*, page 286.2. IEEE Computer Society, 2003.

[9] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267. ACM, 2004.

[10] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Software Practice and Experience*, 34(6):523–547, 2004.

[11] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT*, pages 353–364. IEEE Computer Society, 2007.

[12] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, pages 265–276. IEEE Computer Society, 2008.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[14] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[15] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, pages 103–116. ACM, 2007.

[16] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multi-processor execution efficiently. In *ISCA*, pages 289–300. IEEE Computer Society, 2008.

[17] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multi-processor replay. In *ASPLOS*, pages 73–84. ACM, 2009.

[18] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[19] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS*, pages 216–227. ACM, 2006.

[20] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, pages 284–295. IEEE Computer Society, 2005.

[21] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *PLDI*, pages 22–31. ACM, 2007.

[22] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP*, pages 167–178. ACM, 2003.

[23] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, pages 97–108, New York, NY, USA, 2009. ACM.

[24] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multi-processors. In *SOSP*, pages 177–192. ACM, 2009.

[25] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM TOCS*, 17(2):133–152, 1999.

[26] Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9):62–67, 2003.

[27] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *PLDI*, pages 258–266. ACM, 1996.

[28] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21. ACM, 2008.

[29] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345. ACM, 2006.

[30] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, pages 122–135. ACM, 2003.

[31] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, pages 325–336. ACM, 2009.