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. This demo presents LEAP: an efficient technique as well as a tool prototype to deterministically replay concurrent Java programs on multiprocessors without any changes to the host's environment. During execution, LEAP records the thread access orders w.r.t. each shared memory location. The same thread access orders are then enforced in the replay execution to drive the program to the same states. The replay determinism of this approach is underpinned by formal models and a replay theorem developed in this work. Compared to the related approaches, LEAP records much less information, and thus much more efficient.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Reliability

1. INTRODUCTION

The ability to deterministically reproduce a software failure is of critical importance for software bug understanding, bug fix, and bug verification. However, reproducing concurrency-related failures is quite challenging, due to the huge thread-interleaving space. This demo presents LEAP, an efficient deterministic replay technique that fully reenacts the problematic execution of concurrent Java programs on multi-processors, thus giving the programmers both the context and the history information to dramatically expedite the debugging process.

LEAP is based on a new type of local order w.r.t. the shared memory locations and concurrent threads. The main insight is that, given the same program input, it is sufficient to deterministically replay the program execution by recording the partial thread access information local to the individual shared variables. We use the example in Figure 1 as an illustration. The program contains a race condition that triggers an `ERROR` at line 4 following the interleaved execution order `<1,5,2,6,7,3,4>`. During execution, LEAP uses two **access vectors** (`x.vec` and `y.vec`) for the shared variables `x` and `y` and records `<t1,t2,t1>` and `<t2,t1,t2>` respectively. During replay, LEAP associates `x` and `y` with conditional variables to enforce the access order of threads be identical to what is recorded in their respective access vectors.

The replay determinism of this approach is formally proved in our full paper [1]. Powered by this idea, LEAP is able to support the deterministic replay by recording much less information compared to the related approaches [2, 9]. To further address the efficiency problem, LEAP uses a field-based approach to statically identify shared variables, thus, avoiding the cost of runtime identification. In addition, LEAP makes 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.

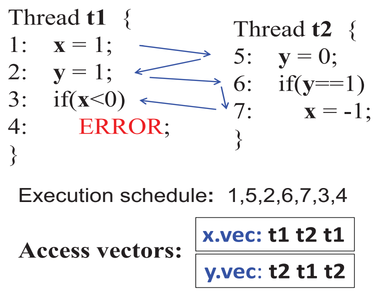
The deterministic replay of concurrent programs on multiprocessors is challenging because, at any time, there might be multiple concurrently executing threads on the same platform. Existing techniques either require special hardware support [10, 7, 3], incur significant runtime overhead [2, 9] or do not provide determinism [4]. LEAP addresses both the recording efficiency and the replay determinism. Our experiments show that LEAP is 2x-10x faster than the use of other related approaches [2, 9, 4]. The average runtime overhead of LEAP is less than 10% on Tomcat and Derby. Moreover, 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 [6], and 100% of the randomly injected concurrency bugs.

In this demo, we present the core techniques of LEAP and show how it works using command line options.

2. LEAP

This section briefly describes the technique and the implementation of LEAP. More details appear in [1].

To locate and to identify shared variable accesses, LEAP first employs a static escape analysis [5] to compute a complete over-approximation of all the shared memory locations in the program. To maintain a consistent identification of shared variables across runs, LEAP proposes a static field-based shared variable identification scheme applied to all the field variables and synchronization monitors (called SPE) in the program. After obtaining all the SPEs, LEAP assigns *offline* to each of them a numerical index as its runtime identifier. The static field-based shared variable identifica-



```
Thread t1 {                Thread t2 {
1:   x = 1;                5:   y = 0;
2:   y = 1;                6:   if(y==1)
3:   if(x<0)               7:       x = -1;
4:       ERROR;            }
}
```

Execution schedule: 1,5,2,6,7,3,4

Access vectors: x.vec: t1 t2 t1
y.vec: t2 t1 t2

Figure 1: Example code with races

tion remains consistent across runs. And since the whole procedure is applied statically, it does not incur runtime overhead. To match the thread identity at record and replay, LEAP introduces additional synchronization operations at the thread creation time to ensure the same thread creation order across runs.

The implementation of LEAP is based on the Soot [8] framework. LEAP is fully automatic and without any user intervention. The LEAP architecture (Figure 2) consists of three main components: 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. For both versions, a LEAP monitoring API invocation is inserted before each SPE access. And both the API call and the SPE access are protected by a dedicated lock specific to ensure LEAP collects the right thread accessing order seen by the SPE.

Started by a record driver, the recorder collects the access vector for each SPE during the execution of the record version. The recorder also adds the parent thread ID to a thread creation order list, once a new thread is created. When the recording stops, the recorder saves the access vectors as well as the thread creation order information and generates a replay driver.

To replay, the 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, directed by the recorded access vectors, to enforce the same thread execution order w.r.t. the SPEs compared with the recording phase. To enable the user-level thread scheduling, the replayer associates each

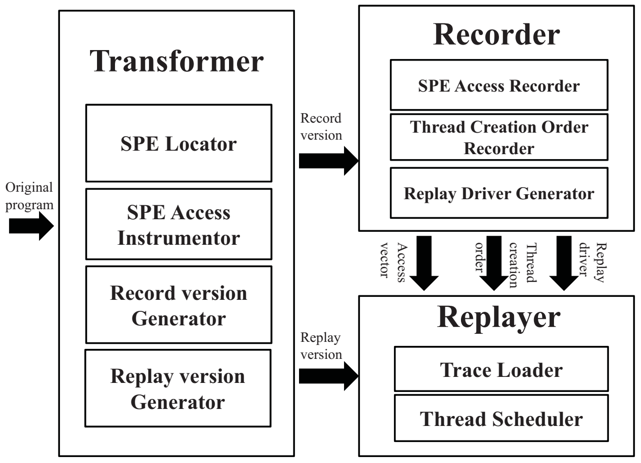


Figure 2: LEAP overview

thread in the replay with a semaphore maintained in a global data structure, so that each thread can be suspended and resumed on demand. 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. And before a new thread is created, the ID of the parent thread is compared to the ID in the thread creation order list, to ensure the identification of each thread is the same as that of the recording phase.

3. DEMO OUTLINE

We have two main objectives in this demo:

a. We show the replay determinism supported by LEAP and show that LEAP can reproduce real concurrency bugs.

b. We show the superior performance of LEAP over the other related approaches.

We will first use a GUI application to render an intuitionistic feeling of LEAP and show how it works. We then use a multi-threaded program constructed by an anonymous FSE reviewer to demonstrate the replay determinism supported by LEAP. After that, we use a real concurrency bug from `Derby` to show that LEAP is able to reproduce failures in real complex multi-threaded applications. Finally, we conduct a performance comparison between LEAP and the use of the global-order based techniques [2].

4. CONCLUSION

We have presented LEAP, a new local-order based approach that deterministically replays concurrent program executions on multi-processors with low overhead. LEAP is publicly available at `http://www.cse.ust.hk/prism/leap/`.

5. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for providing valuable feedback on this paper. This work has been supported by RGC GRF grant 622208.

6. REFERENCES

[1] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, 2010.
[2] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT*, 1998.
[3] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multi-processing. In *ASPLOS*, 2009.
[4] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Softw. Pract. Exper.*, 2004.
[5] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT*, 2007.
[6] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. *IPDPS*, 2003.
[7] Derek Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, 2008.
[8] http://www.sable.mcgill.ca/soot/.
[9] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 1987.
[10] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multi-processor execution efficiently. In *ISCA*, 2008.