Java Replay for Dependence-based Debugging

Jan Lönnberg, Mordechai Ben-Ari and Lauri Malmi

Aalto University, School of Science
Weizmann Institute of Science, Department of Science Teaching

2011-07-17
Introduction

Setting

- Concurrent Programming course at Aalto University
- Java used in lectures, exercises and assignments

Goals of our research

- Assist students in understanding concurrency in Java.
- Assist students and teaching assistants in understanding what the students’ programs do.
Introduction

Approach

- Develop and introduce a tool to show what happens in a concurrent Java program.

- We base the tool on post-mortem analysis of execution trace file to make it possible for e.g. a TA to give a failing execution to a student.

- We use **dynamic dependence analysis** to make it easier to trace interactions between parts of the program.
Introduction

Approach

- Replay is used to avoid the need to try to reproduce an execution to collect more information.
- Support a slicing debugging strategy:
  - Start with a symptom of the failure (e.g. a deadlock, incorrect output or similar).
  - Work backwards along a chain of incorrect behaviour until you reach the correct behaviour.
- Make it easier to trace unexpected interactions between threads and different parts of the program.
Overview of Atropos

**Instrumenter** Adds code to gather traces to the user’s program.

**JVM** The instrumented code is executed in a normal JVM, producing a trace.

**Replay and dependence analysis** The trace is re-executed in a dependence analyser, forming a DDG.

**Visualisation** The DDG can be explored graphically.
Collecting execution information

Approach

- Bytecode instrumentation:
  - Compatibility: no need to implement a full JVM with associated libraries and keep that up to date
  - Portability: runs on any JVM
  - Keeping the execution environment as similar to the underlying JVM as possible
- Traces saved to files for later use.
Collecting execution information

Implementation

- Each thread collects information on its own execution.
- Only operations that may interact with other threads are recorded.
  - Shared memory (fields) operations
  - Concurrency primitives
Collecting execution information

Implementation

- Happens-before relationships allow the instrumentation can safely transfer data between threads.
  - If \( x \) happens-before \( y \), store an identifier (thread and operation id) for \( x \) in a field associated with the interaction before \( x \); read this after \( y \) and put it in the execution trace.
  - Locks and volatile variables get their own fields.

- Consistency in unlock/lock pairs guaranteed by lock itself.
- Consistency in volatile variables is guaranteed by replacing the field with a reference to an identifier/value pair.
Limitations

- Access to locks and fields from uninstrumented code cannot be tracked.
- Instrumenting `Object` leads to crashes, so locking can only be tracked for instances of selected classes.
- If many writes in a data race wrote the value that was read, one cannot say which write was the one that the read read its value from.
  - Since the value is known, this does not affect replay.
  - The read could have read any one of these values, so they are all relevant to exploring the program’s (mis-)behaviour.
Collecting execution information

Limitations

- Objects need to be identified uniquely when the trace is saved.
  - Assign objects unique ids from a global table when writing to disk.
  - This leads to a trade-off between memory use and precision.
- Effects of instrumentation highly JVM-dependent
Replay and dependence analysis

Approach

- Dependence analysis is based around replaying the trace in a JVM that calculates vector timestamps and dependency relations while executing.

- Replay is performed in an order consistent with happens-before.

- In the case of data races, execution order becomes unclear, possibly even circular. In this case, the data dependency must be determined later.

- Control dependencies can be replaced with simpler constructs.
Visualisation

Approach

- Obvious representations exist for vertices and edges.
- If \( x \) happens-before \( y \), \( x \) is placed above \( y \).
- Use executions of lines of code as a suitable level of detail.
- Visualise only the part of the DDG relevant to the task at hand; let the user explore the graph along dependency edges.
Evaluation

Performance

- Performance loss by factor of about 5
- Trace file sizes manageable with ZIP compression
- Size of in-memory representation of DDG problematic

Usability

- The DDG visualisation itself is reasonably easy to understand.
- Finding the relevant information in the DDG is problematic.
Conclusion

Future work

- Making execution traces shorter
  - Model checking or manual execution instead of testing
  - Shorter test runs
- Less memory-intensive DDG calculation
- Storing execution traces more efficiently
  - Eliminate race-free reads and writes
- Additional navigation
  - Forwards and backwards
  - Alternative visualisations
- More explicit guidance on how to use Atropos