Automated Java Bytecode Replay for Performance Evaluation and Tuning,

Stephen Lennon,

Project specification for Masters in Advanced Software Engineering, UCD, Dublin,

21 September 2015.

Document Control

Date	Change	Revision
21 September 2015	Initial	1
8 October 2015	Revisions to scope and objectives	2

Table of Contents

Description

Java Bytecode Replay is the technique of causing a Java Virtual Machine Instance to re-execute a previously recorded sequence of bytecode instructions. Typically reasons for replaying bytecode centre around reproducing an order of execution in a multithreaded program where the recorded sequence of bytecode manifests a bug. Merely executing the program again will result in different input (Input taken from the filesystem may be identical, but input from the system clock, over the network or from the random number generator for example will be different) and also a different execution order of the threads in an multithreaded application may prevent the bug from manifesting in a re-execution unless these varying factors are also induced to be the same. Typically bytecode replay has two main objectives:

- Replay the input to the application, complete with delays and events;
- Replay the exact ordering of execution of threads in the application.

Machado, Lucia and Rodrigues (2015) propose the most recent application of this technique found in the preparation of this document. The techniques applied are a form of bytecode modification to produce thread execution schedules which are then used to locate the cause of concurrency bugs.

Schuppan, Baur and Biere (2005) describe the format of a thread schedule that can achieve deterministic replay, but merely mention that the schedule could accommodate a record of the input to the program.

Steven, Chandra, Fleck and Podgurski (2000) provide the prior work that has the greatest overlap with the objectives of this project in that they have defined and implemented a capture scheme based on substituting parts of the Java standard library with alternate implementations. This scheme is insufficiently efficient for console (Nion-GUI) applications and does not benefit from more recent advancements in the JVM by introduction of Java Agents or from the field of Bytecode Modification that has been popularised since its publication.

Orso and Kennedy (2005) describe a scheme of *Selective Capture* that proposes to address the volume of information that must be captured in order to replay execution of a program, though the objective of their efforts is again for the purposes of testing and dynamic analysis. The SCARPE tool mentioned by Orso and Kennedy (2005) is described at length in the later paper by Joshi and Orso (2007). The technique described involves defining groups of classes and monitoring method calls that cross the boundary of the defined group, with a view to reproducing these calls for replay, thereby potentially sidestepping much of the gritty specifics addressed by Steven et al. (2000) in their interaction with native methods and substitutions into the Java standard library to produce jRapture. This advantage is achieved by the use of the BCEL (Bytecode Engineering Library), thereby emphasising the potential to use bytecode implementation to simplify the difficult tasks that the authors of earlier papers encountered.

Objective

This project proposes to provide a framework for experimental investigating of the effects of JVM parameter selection on the performance of Java Programs against key metrics. Such metrics are to be defined but include concerns such as memory usage, runtime, pauses caused by Garbage Collection (Which may reduce perceived responsiveness), and the like with focus on selecting parameters that affects the User's experience in using the application. Such parameters are likely to be limited to a selection of parameters passed to the Java Virtual Machine at application startup, such as those affecting heap and stack size and configuration. Part of the objective of the project is to define these performance metrics and external parameters.

While it is possible to re-run a Java application with identical input but different external parameters it is not possible, without preparation and suitable tools, to exactly duplicate the execution path of an application because not all inputs can be accurately simulated, such as system time, user interaction, and input data that originates from other sources that are not controllable. Files stored locally on the filesystem are controllable and reproducible as input, for example, but input obtained over the network from other systems may be difficult or impossible to replicate exactly. It would be of great utility to be able to capture all inputs to a Java application and replay them exactly a very large number of times to experiment with variations in JVM parameters.

In practice the difficulty of reproducing an execution path makes even experiments of small scope cumbersome. The viability of experimenting with external parameters further decreases when the number of parameters available for selection is large, which is the case with JVM parameters, because the number of sensible combinations of parameters increases exponentially, and often judgement and heuristic substitute for rigorous method for lack of time and patience to experiment.

Additionally several factors can invalidate the selection of parameters, and it may be infeasible to repeat the experiment. This can happen:

- When the character of the system's inputs change, such as when an application graduates from load testing with simulated data to production service with real-world input data;
- When the character of inputs changes because of a change in an upstream system;
- When the hardware platform underlying the application changes or is upgraded;
- When the JVM underlying the application execution is upgraded;
- When a part of the application itself changes;
- When the performance expectations of the system are re-evaluated such as to accommodate more users or new or different downstream systems.

For the purposes of the performance experiments envisioned it is not desired to interfere with normal thread scheduling of the Java Virtual Machine as to do so would invalidate the results. It is necessary, however to be aware of the thread construction tasks in the JVM so that input can be recorded in a way that facilitates replay in a multithreaded environment.

Thereafter input to a defined set of methods can be captured, and later replayed with the original processing delays intact. This aspect of the endeavour overlaps significantly with the objectives of bytecode replay as described in the current literature.

Provided the set of methods selected for record and playback is carefully chosen it should be possible to encircle critical input channels from outside the JVM, such as *java.io.InputStream* and similar classes and record methods selectively to later replay that input. The selection of the methods and the scheme for complete capture of relevant inputs is within the scope of the project.

One area of Software Development practice that struggles with the effects of change is that of Continuous Delivery. In Continuous Delivery software ecosystems software is delivered into production very frequently, possibly dozens of times per day. Because it is infeasible to perform manual validation of every build and delivery, a scheme of automated testing is typically performed. This may include running Unit Tests, Integration Tests, and System Tests automatically as the software is rebuilt and delivered into production. While beyond the scope of this document to discuss this practice of Continuous Delivery at length, attention is drawn to the two factors of this practice that are central to its utility: *Reproducibility* and *Automation*.

Because testing steps are written once, and reproduced without further intervention every time the software is built, a well written test suite can detect when any changes introduced to the application break its contract with the outside world, or cause internal malfunction, without any further manual effort.

If we were to apply these principles of reproducibility and automation to performance testing we could envision defining a performance testing schedule that could be run automatically to select new JVM parameters when the application or external factors have, of necessity, changed. If it were possible to run the test suite with a minimum of manual intervention then it could be performed frequently with little overhead. It could also be run without any direct impetus to confirm that the parameters previously selected are still optimal, thereby accounting for undetected change, such as changes in the performance of upstream systems or user behaviour.

Approach

The approach taken is one of Prototype Development, with reference to prior work in the field. Learnings will be documented in a project report to accompany the software prototype.

The software prototype will consist of:
- A tools to replay the execution of Java Programs;
- Automation to generate tests of the applications in a way that facilitates tuning the performance of the application with a focus on factors of performance that affect the experience of end users;
- A visualisation component to promote understanding of the results of the experiments performed,

subject to the schedule of deliverables below.

Deliverables

Mandatory Deliverables

The project will produce a functioning codebase capable of recording and replaying the execution of a variety of selected Java console applications. Such replay will have a sufficiently small overhead that performance evaluation on the results remains meaningful. Replay will be limited to recording and replay of input and related concerns to achieve this reproducible execution, but will specifically exclude any attempt to reproduce thread execution schedules as this is contrary to the objective of measuring varying performance with varying JVM parameters.

A project report will accompany the deliverables.

Discretionary Deliverables

To build upon the mandatory deliverables, a wide selection of console applications will be demonstrated to work when recorded and replayed. Relevant JVM parameters will be charted and a means provided to create and execute meaningful replay schedules. Collection of metrics upon repeated replays of the application will be automated.

Exceptional Deliverables

Replay will function against complex console applications including application servers with multiple class loaders. Data collected during execution will be presented graphically such that the best performing JVM parameters are easy to identify.

Timeline

Phase One, Weeks 1, 2, 3 to 10 October

- Research and examination of similar implemented techniques.
- Devise implementation strategy.
- Construct delivery mechanism, Java Agent or otherwise.
- Bytecode implementation prototype or alternative.
- Proof of concept simple record and replay operational on a small console application.

Phase Two, Weeks 4, 5, 6 to 31 October

- Chart meaningful JVM parameters and types and ranges of values.
- Collect metrics with JMX or similar metics APIs.

Provide a framework for automated execution with varying parameters and automated collection of results.

Phase Three, Weeks 7, 8, 9 to 21 November

- Apply all devised techniques to progressively more complicated console Java software, including some application servers, such as Tomcat and JBoss and determine how increased complexity and multiple class loaders affect the prototype.
- Adjust the prototype to accommodate new learnings as possible.
- Give consideration to usability and presentation of results.

Phase Four, Weeks 10, 11, 12 to 12 December

- Author project report.
- Gather experimental examples of the application of the prototype.

References

Machado, N., Lucia, B., & Rodrigues, L. (2015). Concurrency Debugging with Differential Schedule Projections.

Schuppan, V., Baur, M., & Biere, A. (2005). Jvm independent replay in java. *Electronic Notes in Theoretical Computer Science*, *113*, 85-104.

Steven, J., Chandra, P., Fleck, B., & Podgurski, A. (2000). jRapture: A capture/replay tool for observation-based testing (Vol. 25, No. 5, pp. 158-167). ACM.

Orso, A., & Kennedy, B. (2005, May). Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes* (Vol. 30, No. 4, pp. 1-7). ACM.

Joshi, S., & Orso, A. (2007, October). Scarpe: A technique and tool for selective capture and replay of program executions. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on* (pp. 234-243). IEEE.