CS 636 Semester 2018-2019-II: Assignment 2

February 13, 2019

DUE Your assignment is due by Thursday Feb 28 2019 11:59 PM IST.

POLICIES

- You can do this assignment in GROUPS OF UP TO TWO students.
- Do not plagiarize or turn in reports from other groups. Members from all involved groups will be PENALIZED if caught, without considering the SOURCE and SINK.

SUBMISSION

- Submit a report with name "<roll-no>-assign2-report.pdf".
- The "<roll-no>-assign2-report.pdf" file should contain results that have been asked for in the following problem. You are encouraged to use LaTeX typesetting system for generating the file.
- The assignment requires you to fork an existing Git repository. After forking, invite the TAs to YOUR forked Git repository to help with the evaluation.
- Submitting your assignments late will mean losing points automatically. You will lose 10% for each day that you miss, for up to three days.

Problem Description **[100 points]**

RoadRunner is a dynamic analysis framework written in Java. The goal of this assignment is to learn about RoadRunner, and use and extend some of the analyses that it provides. Future assignments MAY involve development with RoadRunner.

Why RoadRunner? There are other alternatives (for e.g., the HotSpot compiler in OpenJDK and Jikes RVM) to RoadRunner for implementing dynamic program analyses for Java programs. But RoadRunner is simpler to learn since it provides better abstractions.

RoadRunner infrastructure – Fork the RoadRunner infrastructure from the following Git repository.

`git clone git@git.cse.iitk.ac.in:swarnendu/cs636-roadrunner.git`.

Follow the instructions in the files `README.md`, `README-RoadRunner.txt`, and `INSTALL.txt` to build and install it on a GNU/Linux system. I have not used Windows/MacOS, but it might be possible to set up RoadRunner on those platforms.

You do not need to submit anything for this work item.

Paper reading – Read the following paper "C. Flanagan and S. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs, PASTE 2010."

The paper provides a brief overview of the design of RoadRunner. The paper is kind of dated, and the infrastructure has evolved since the paper was published, so not every description in the paper may match the source exactly. Feel free to search the Internet for more information on RoadRunner.

You do not need to submit anything for this work item.

Study RoadRunner – Browse the HB and FastTrack implementations and the associated RoadRunner source, and write a brief report on the source code of `HB` and `FT2`.

You should assume that the readers know HB and FastTrack but they have no knowledge of RoadRunner. Your job is to help them quickly understand and pickup the RoadRunner code (like writing a manual for the tools). Your report should probably include pointers about the general control flow of the infrastructure, the role of important classes and the HB and FastTrack tools in particular.

There is no bound on the length of the description. It is fine to include small code snippets if you think it is important, but in general AVOID JUST FILLING IN PAGES WITH COPIED CODE.

The following are a FEW useful classes to start looking at:

- `Tool`
- `ShadowThread`
- `ShadowLock`
- `AccessEvent`

A thorough study will probably help you in your future assignments.

Use existing analysis like FastTrack – Use the FastTrack analysis provided as part of the RoadRunner framework to familiarize yourself with the infrastructure. The accompanying tool name is FT2. Run the analyses on several microbenchmarks and benchmarks, and study the results.

You should try and run FastTrack with microbenchmarks like `test/Test.java` or your own. In addition, RoadRunner should work successfully with the following benchmarked applications: avrora, batik, fop, h2, jython, luindex, lusearch, pmd, sunflow, tomcat, and xalan.

Implement sampling support in FastTrack – Extend the FastTrack data race detection analysis to include support for sampling. Name the new analysis `FT2-S`.

Sampling generally means using some predicate to *conditionally* run heavyweight analysis like data race detection on certain events (like shared-memory reads and writes). The goal of using sampling is to improve performance. For example, a simple algorithm could sample every 10^{th} shared-memory access. LiteRace [3], Pacer [1], and DataCollider [2] are some existing techniques that implement some form of sampling. The exact nature of the sampling algorithm you want to implement for this assignment is up to you, this is an opportunity to innovate.

Students who come up with innovative sampling algorithms will receive bonus points at the discretion of the evaluators.

The new tool you are going to write `FT2-S` should take one command line parameter `-sampling_rate` which will be a number indicating the rate of sampling. For example, `-sampling_rate=10` indicates a sampling rate of 10%.

Include a brief description of the sampling algorithm in your report. Evaluate the impact of sampling on the performance and data race coverage of FastTrack for three benchmarks avrora, lusearch, and xalan.

1. Include a bar graph in the associated report that compares the performance of FT2 with FT-S with 50% sampling rate and with FT-S with 5% sampling rate. The x-axis should represent benchmarks and y-axis should represent normalized performance in terms of time (lower bars are better). For each benchmark, there should be at least three bars: FT2, FT-S w/ 50% sampling rate, and FT-S w/ 5% sampling rate.
2. Represent the data race coverage as a table. Your table may look like Table 1. Also list details (e.g., program location) about the data races that are exposed by each of the three tools.

	FT2	FT-S w/ 50% sampling	FT-S w/ 5% sampling
avrora	xx	yy	zz
lusearch	-	-	-
xalan	-	-	-

Table 1: Impact of sampling on the data race coverage of FastTrack

References

[1] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 255–268, New York, NY, USA, 2010. ACM.

[2] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2010.

[3] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 134–143, 2009.