CS 636 Semester 2020-2021-2: Assignment 1

28th January 2021

Due Your assignment is due by Feb 16, 2021, 11:59 PM IST.

Policies

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

Submission

- Submit a report with name "`assign1-report.pdf`" on mooKIT. Mention the group members in the document.
- The "`assign1-report.pdf`" file should contain results that have been asked for in the following problem. Explain the results to help the evaluators.
- The assignment requires you to fork an existing Git repository. After forking, invite all the TAs to the forked Git repository to help with the evaluation.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.
- You are encouraged to use the LaTeX typesetting system to generate the PDF report.

Evaluation

- Write your code such that all the strategies and the EXACT output format are respected.
- We will evaluate your implementations on a Unix-like system, for example, a recent Debian-based distribution.
- We will evaluate the implementations with our OWN inputs and test cases, so remember to test thoroughly.

Problem Description [100 points]

RoadRunner is a dynamic analysis framework written in Java. The goal of this assignment is to use RoadRunner and extend some of the analyses it provides.

Why RoadRunner? Other alternatives for implementing dynamic program analyses for Java programs are the HotSpot compiler in OpenJDK and Jikes RVM). But RoadRunner provides easier abstractions at the cost of performance.

RoadRunner infrastructure – FORK the RoadRunner infrastructure from the following public Git repository: `https://git.cse.iitk.ac.in/swarnendu/cs636-roadrunner`.

Follow the instructions in the files `RoadRunner-README.txt`, `CS636-README.md`, and `INSTALL.txt` to build and configure RoadRunner on a GNU/Linux system.

Read the paper "C. Flanagan and S. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs, PASTE 2010." to know more about the infrastructure. The paper is included in the repository and provides a brief overview of the design of RoadRunner. The RoadRunner infrastructure has evolved since the paper was published, so every description in the paper may not match the source code.

Study RoadRunner – Browse the HB (`src/tools/hb/*`) and FastTrack (`src/tools/FastTrack`) implementations and relevant RoadRunner source. You can start by studying the following helper classes:

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

Use FastTrack – Use the FastTrack analysis and the accompanying tool `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. RoadRunner should work successfully with the following benchmarked applications: avrora, batik, fop, h2, jython, luindex, lusearch, pmd, sunflow, tomcat, and xalan.

We will use the following five DaCapo benchmarks for our study: avrora, luindex, lusearch, sunflow, and xalan. Run each benchmark a few (for e.g., ≥ 10) times. The goal is to use a dynamically sound analysis like FastTrack to generate race reports. The benchmark versions included with RoadRunner are known to have a few data races. The following shows a sample RoadRunner report, which is reported by RoadRunner at the end of the execution.

```
<field>
    <name> avrora/sim/radio/Medium$Transmission.lastBit_J </name>
    <error> <name> FastTrack </name> <count> 100 </count> </error>
</field>
<field>
    <name> avrora/sim/radio/Medium.Pr_D </name>
    <error> <name> FastTrack </name> <count> 100 </count> </error>
</field>
<field>
    <name> avrora/sim/radio/Medium.Pn_I </name>
    <error> <name> FastTrack </name> <count> 100 </count> </error>
</field>
```

The report provides information about the classes and the fields which are involved in a data race. For e.g., `avrora/sim/radio/Medium$Transmission` is the class name, `lastBit` is the field name, and `J` is the type of the field (`long`). Some other analysis framework may format the class name as `avrora.sim.radio.Medium$Transmission`.

You can check the Type Descriptions page for more information.

Note that the race reports that you get might differ. Furthermore, some executions may lead to nonzero race reports. Try executing the benchmarks many times in such cases.

FT checks for data races in its methods. Extend the FT code to keep track of the source locations. Whenever a race is detected, you can form race pairs with your source locations.

Collate the *unique* race reports from the different runs for each benchmark. You should consider two race pairs `A` $\Leftrightarrow$ `B` and `B` $\Leftrightarrow$ `A` to be the same. We will use these race reports as the input for the next tasks.

Task 1: Implement sampling support in FastTrack – Extend the FastTrack data race detection analysis to include support for sampling. Name the new analysis `FT2S`.

Sampling 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 100^{th} access per-thread or globally. LiteRace [3], Pacer [1], and DataCollider [2] are some existing techniques that implement some form of sampling.

The `FT2S` tool should take one new command line parameter `-samplingrate` which indicates the proportion of accesses that are to be sampled. For example, `-samplingrate=10` indicates a sampling rate of 10% (i.e., sample every tenth shared memory access).

Synchronization operations are usually far less frequent compared to shared memory accesses. Hence, we will apply sampling to only reads and writes.

Evaluate the impact of sampling on the performance and data race coverage of FastTrack for five benchmarks avrora, luindex, lusearch, sunflow, and xalan.

1. Include a bar graph in the associated report that compares the performance of FT2 with FTS with 50% sampling rate and with FTS with 10% sampling rate. The x-axis should represent benchmarks and the y-axis should represent normalized performance in terms of time (lower bars are better). That is, there should be at least three bars: FT2, FTS w/ 50% sampling, and FTS w/ 10% sampling for each benchmark.

   FT2 should be similar to FTS with 100% sampling. To get reliable performance numbers, run each configuration a few times and take the average to mitigate noise. You should normalize the running time of each tool with the empty N tool. Suppose the N tool with lusearch takes 10 s and FT2 takes 20 s, then the height of the bar corresponding to FT2 in your plot would be 2.

2. Report the data race coverage (i.e., number of unique data races detected) of FT2S. Your table may look like Table 1. List details (e.g., program location) about the data races that are reported by each of the three variants.

3. Introduce a second command line parameter `-samplingscheme`. This option takes at least two values: `count` and `adaptive`. The `count` scheme refers to the above access-count-based strategy. The `adaptive` option refers to an adaptive strategy, where the sampling rate starts high but then decays with the number of accesses. For example, an idea can be to maintain a per-program-location frequency and vary the sampling rate based on the frequency. The intuition is that the likelihood of a bug manifesting at a given location decreases with increasing frequency (cold region hypothesis).

   You are encouraged (optional) to try out additional heuristics to improve the data race coverage in presence of sampling, by adding additional choices to the `-samplingscheme` parameter. Include a brief description of the sampling algorithms in your report.

	FT2	FT2S w/ 50% sampling	FT2S w/ 10% sampling
bench1	xx	yy	zz
bench2	-	-	-
bench3	-	-	-
...			

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

Task 2: Implement support for schedule steering and perturbation in FastTrack – Extend the FastTrack data race detection analysis to include support for schedule steering and perturbation to try and improve data race coverage. The goal is to break any spurious happens-before edge during execution with perturbation. Name the new analysis `FT2SS`.

Read the paper "B. Kasikci et al. RaceMob: Crowdsourced Data Race Detection, SOSP 2013." which innovates on the basic FastTrack algorithm to reduce performance overheads. Sections 3.2-3.4 in the paper discusses the idea of schedule steering to improve data race detection coverage.

The `FT2SS` tool will not instrument all shared memory reads and writes. Instead, the tool will selectively instrument only the potential racy accesses. You should pass the list of potential racy pairs via a new command line parameter `-racepair`. The parameter should point to an ASCII file that lists the pair of source locations to be monitored during the execution. The `FT2SS` tool should avoid instrumenting other source locations (also called program sites).

Given a race pair `A` $\Leftrightarrow$ `B`, try to ensure that the `FT2SS` approach implements the schedule steering method and perturbs the execution as much as possible.

Here is a minimal list of ideas you should implement.

- You should make the delays adaptive. For example, you may want to increase or decrease the delay at a site `A` based on whether the race is still undetected and also based on the frequency of dynamic accesses.
- Given a race pair `A` $\Leftrightarrow$ `B`, you should try to enforce that `B` executes first in order to perturb the execution as much as possible.
- Try and introduce delays before synchronization acquire and after synchronization release operations. You may not want to make a thread wait within a critical section.

You are free to experiment with additional ideas. Be careful to avoid situations where the threads get stuck indefinitely. Use microbenchmarks to test the correctness of your implementation.

Your report should include the following:

- Brief explanation of the heuristics used in your "schedule steering and perturbation" implementation.
- Brief description of your implementation (for e.g., pause times, adaptive conditions).
- Report whether your implementation was able to detect all the data races normally reported by FastTrack. Did schedule steering or perturbation help discover additional data races in the benchmarks?

References

[1] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional Detection of Data Races. In *PLDI*, pages 255–268, 2010.

[2] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, pages 1–16, 2010.

[3] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, 2009.