Learning Based Controlled-Concurrency Testing

Getting Started Guide

Requirements

- VirtualBox: While any recent version should suffice, we built and tested our artifact using version 6.1.10.

Setting up

- In VirtualBox, select `File -> Import Appliance`, then select the `QL.ova` file from the extracted contents. After the import is completed, you should see `QL` as one of the listed VMs in your VirtualBox app.
- Run the `QL` virtual machine. The username is `ql` and the password is `qltest`.
- In the home folder (`/home/ql`), there is a `oopsla/psharp-ql` directory which contains all the sources (P# tester with the implementation for `QL`, and the non-proprietary benchmarks), along with scripts for running the experiments.
- Open a terminal (from Favorites or `Ctrl + Alt + T`), and run `powershell`. Note that we assume that all subsequent commands are run from `powershell`.
- We have implemented QL as an additional testing strategy on top of the P#-testing framework. To build the P#-tester (including the sources for QL), run `./build.ps1` in `oopsla/psharp-ql`.
- Go to the Benchmarks directory: `cd Benchmarks`
- Build the benchmarks, along with the program to drive the experiments (`EvaluationDriver`) using `./build-benchmarks.ps1`

Everything is now setup!

Quick Test

To quickly test the setup, run the command

```
./run-benchmarks.ps1 -mode "Test" -numEpochs 1
```

from the Benchmarks folder. Each mode runs the tests which are configured in the corresponding folder. For example, the previous command will run the `Raft-v1` benchmark, and the configuration for the test (number of iterations, max-steps, etc) are present in the JSON configuration file under `Benchmarks/Test`. By default, the P#-tester will be invoked 100 times to obtain the B-100 metric, which is the number of times the tester finds the bug out of 100 invocations (the `NumEpochs` value is set to 100 in the JSON).

The `-numEpochs 1` overrides this value, to report B-1 instead, for quick evaluation. Finally, a csv file summarizing the results is created under `Benchmarks`. This table reports how many times each scheduler exposed a bug in `Raft-v1` out of the `numEpochs` number of invocations (in this case, 1) of the P#-tester. The prefix of the csv file matches the mode (in this case, the file name will be `test...csv`).

Artifact Structure

If you open Visual Studio Code in the artifact (added to Favorites), then the `oopsla/psharp-ql` should be loaded automatically (if not, load the folder by `File -> Open Folder -> /home/ql/oopsla/psharp-ql`).

The locations of the various scheduling strategies are available in the folder `Source/TestingServices/Runtime/Scheduling/`. The variants of QL are available under `Learning`, Greedy, Random and PCT are available under `Probabilistic`, while IDB is available under `DelayBounded`.

For any scheduling strategy, the important methods are as follows:

- `GetNext`: returns the next action to be executed depending on the exploration strategy
- `GetNextBooleanChoice`: control the non-deterministic boolean choice in a statement.
- `GetNextIntegerChoice`: control the random integer generation in a statement.
- `PrepareForNextIteration`: perform cleanup operations before the next iteration begins. In Ql, for example, this method actually propagates the rewards and updates q-values for each observed state-action pair.

The benchmarks are located under `psharp-ql/Benchmarks`. We provide the Protocols and Threading benchmarks under the respective folders. We omit the production benchmarks as they are Microsoft proprietary.

The Benchmarks folder also contains `EvaluationDriver`, which is a project to automatically drive the experiments, running the different schedulers in parallel.

Step-by-Step Instructions

P# Testing

Our implementation builds upon P#, which is an open-source controlled concurrency testing framework. Given a concurrent program, the P#-tester takes over the scheduling decisions, and serializes the program execution. The scheduling decisions are driven by a particular exploration strategy. Each invocation of the P#-tester requires the following inputs: - Path to the dll under test - Name of method to be tested (`-method:x`) - Scheduling strategy to be used (`-sch:x`) - Number of iterations explored (`-i:x`) - Maximum length of each serialization (`-max-steps:x:x`) - Output directory where buggy traces are dumped (`-o:x`)

For QL, we have the additional argument `-abstraction-level` which determines the state abstraction used during exploration (`default`, `inbox-only` or `custom`).

The `EvaluationDriver` project takes as input a JSON configuration file which provides values for the arguments above. In fact, the configuration takes a *set* of schedulers, and runs them in parallel.

Bug Finding

Command

```
./run-benchmarks.ps1 -mode "Bugfinding" -numEpochs 100
```

Description

This command runs all the benchmarks to compute the B-100 metric, and generates a csv file (under the Benchmarks folder, named `bugfinding...csv`).

Note that 100 epochs will take a very long time (several hours) to run. Reviewers can opt for a smaller value for `numEpochs` instead (such as 10).

Timed Bug Finding

Command

```
./run-benchmarks.ps1 -mode "Bugfinding" -numEpochs 100 -timeout 300
```

Description

This command runs all the benchmarks to compute the B-100 metric, with a fixed-time budget, instead of a fixed iteration budget. This generates a csv file under the Benchmarks folder, named `bugfinding...csv`.

Note that 100 epochs will take a very long time (several hours) to run. Reviewers can opt for a smaller value for `numEpochs` instead (such as 10).. The timeout value is in seconds. The reviewers can experiment with a smaller timeout value as well. Note that the results of the timed run will depend on external factors such as machine configuration, current CPU load, etc.

Handling Data-Nondeterminism

Command

```
./run-benchmarks.ps1 -mode "DataNondet" -numEpochs 100
```

Description

QL is the first scheduling strategy which accounts data-nondeterminism. The experiment runs QL and QL-NDN (QL with handling of data non-determinism turned off) on the Raft-v1, Raft-v2, Paxos and Chord benchmarks. It outputs a csv file under Benchmarks, named `datanondet...csv`.

Note that 100 epochs will take a very long time (several hours) to run. Reviewers can opt for a smaller value for `numEpochs` instead (such as 10).

State Hashes

Command

```
./run-benchmarks.ps1 -mode "StateHash" -numEpochs 100
```

Description

This command runs the Protocols, and the SafeStack application, with various state abstractions (or hashes). Note that this experiment does not perform any aggregation, but the results can still be read-off easily. For each benchmark, a folder is created corresponding to the state abstraction (`default`, `inbox-only` or `custom`) under `StateHash/out`, and a `results.json` file is created in each of them. These files contain all the necessary details.

Note that 100 epochs will take a very long time (several hours) to run. Reviewers can opt for a smaller value for `numEpochs` instead (such as 10).

Performance

Command

```
./run-benchmarks.ps1 -mode "Perf" -numEpochs 1
```

Description

This experiment compares the time taken for exploration by the different scheduling strategies. For this, we create versions of the Protocol benchmarks with the bugs disabled (these are placed in the `Protocols_BugsDisabled` project). The experiment then measures the time to perform a *single* invocation of P# tester with 10,000 iterations, for the different schedulers. This experiment generates a csv file under the Benchmarks folder, named `perf...csv`.

Note that the numbers obtained may differ significantly over runs, due to external factors such as machine configuration, current CPU load, etc. The running times are all in seconds.

State Coverage

Command

```
./run-benchmarks.ps1 -mode "StateCoverage"
```

Description

This experiment compares the coverage achieved by the different schedulers. Like the Perf experiment, the experiment runs on Raft-v1, FailureDetector and SafeStack benchmarks with the bugs turned *off*, since we are purely interested in measuring the coverage. For each benchmark, we aggregate the number of unique states explored for each of the schedulers in a csv file under the StateCoverage folder.