Parallel Data Race Detection for Task Parallel Programs with Locks

Adarsh Yoga
Department of Computer Science
Rutgers University
adarsh.yoga@cs.rutgers.edu

Santosh Nagarakatte
Department of Computer Science
Rutgers University
santosh.nagarakatte@cs.rutgers.edu

Aarti Gupta
Department of Computer Science
Princeton University
aartig@cs.princeton.edu

ABSTRACT

We describe `PTRacer`, a tool that performs parallel data race detection for task parallel programs written using the Intel Threading Building Blocks C++ library. `PTRacer` supports the use of synchronization operations in these task parallel programs. `PTRacer` detects all races without false positives in our test suite of 120 unit tests that include racy and non-racy programs with and without locks. The performance overhead of `PTRacer` is similar to SPD3, the state-of-the-art race detector for task parallel programs, while providing stronger guarantees. The artifact contains our tool, the test suite used for checking the effectiveness of the detection, applications from PARSEC and PBBS benchmark suites for performance experiments, and implementations of existing detectors, which can used to reproduce the results reported in the paper.

1. DESCRIPTION

1.1 Insightful

Timely. `PTRacer` is a tool for detecting data races in task parallel programs. A program exhibits a data race when there are multiple accesses to a shared memory location, at least one of them is a write, and there is no ordering between these accesses. Data races are symptomatic of errors in parallel programs. We are specifically interested in detecting data races in task parallel programs. Task parallelism has become mainstream with multiple language and library based frameworks such as Intel Threading Building Blocks (TBB), Cilk, X10, and Java Fork-Join tasks.

Addresses gap in prior work. Although data race detectors have been actively studied for multithreaded programs, detectors for task parallel programs have received limited attention. Existing detectors for task parallel programs either run the program serially [1, 2] and/or do not handle programs with locks [3]. Intel Parallel Studio ships with a serial data race detector that has significant performance overheads for even simple applications. Ideally, a race detector should detect races not only in the schedule observed in execution but also in all other schedules for the same input, which eliminates the need for interleaving exploration. `PTRacer` is a design point which combines static instrumentation and dynamic data race detection to (1) detect apparent races in the observed schedule and also in other schedules for a given input, (2) detect races in programs that use locks, and (3) runs in parallel leveraging the multiple cores. Races detected by `PTRacer` are apparent races, which are races that the seem to appear considering only the parallel spawn/sync constructs without considering the actual computation performed by individual tasks.

1.2 Useful

Serves a purpose otherwise impossible. `PTRacer` is a data race detector for task parallel programs using the Intel TBB library. It is usable with large applications. It detects data races when these applications use locks. We were not able to run the data race detector that ships with the Intel Parallel Studio with these applications.

Apart from the above features, `PTRacer` combines static instrumentation and dynamic data race detection in a novel way to detect races not only in the observed schedule but also in other schedules for a given input, which can reduce/eliminate the need for interleaving exploration. When computations in critical sections influence branch statements, the schedule observed during dynamic analysis will be different depending on the scheduling of critical sections. Such branch statements are called schedule sensitive branches (SSB). In the presence of SSBs, the dynamic trace observed by the analysis will likely not have memory operations from all schedules (*e.g.*, from the path not-taken at the branch statement). `PTRacer` can also detect races in the presence of SSBs by recording accesses that can occur in the not-taken path of a SSB (Refer to our paper for further details).

`PTRacer` also includes a constraint solving component to check whether the races involving SSBs are feasible. `PTRacer` generates per-task constraints from an execution and checks, using an SMT solver, if the branch condition at a schedule sensitive branch can be inverted.

Effectiveness. Our artifact demonstrates that `PTRacer` detects races in the presence of locks and SSBs, while running the program in parallel. In contrast, `SPD3` reports false positives in the presence of locks and misses races in programs that have SSBs. Our artifact also demonstrates that `PTRacer` is usable with long-running applications and has performance overhead similar to `SPD3` while detecting races in a larger class of programs.

1.3 Usable

Our artifact is publicly available on GitHub. It checks for data races in any task parallel program that uses the Intel TBB library. We provide scripts to automate the installation and execution of `PTRacer`. We provide detailed instructions on how install and use the tool from scratch.

2. ARTIFACT

Our artifact is publicly available through GitHub at the following URL https://github.com/rutgers-apl/PTRacer. The artifact is structured as follows: (1) **tdebug-lib** contains the `PTRacer` dynamic data race detection library for TBB programs, (2) **PTRacer-solver** contains the constraint generator written in python that constructs the first-order logic formula to check the feasibility of a sched-

ule sensitive branch, (3) **spd3-lib** contains our implementation of the SPD3 dynamic data race detector for TBB programs, which is the state-of-the-art, (4) **tdebug-llvm** contains the compiler pass in Clang+LLVM 3.7 that instruments the TBB program with calls to the data race detection library, (5) **tbb-lib** contains the modified Intel TBB library to enable data race detection, and (6) **test_suite** contains 120 unit tests that include racy and non-racy programs with and without locks and schedule sensitive branches.

The benchmarks that we use are available for download at http://bit.ly/29i3OYL. The entire artifact including the tools and the benchmarks requires approximately 6 GB of storage space.

2.1 Setup

Run-time Environment. `PTRacer` has been developed and tested on a 4.00GHz four-core Intel x86-64 i7 processor, with 64 GB of memory running 64-bit Ubuntu 14.04.3. `PTRacer` works on C++ programs that use Intel TBB library. `PTRacer` requires the C++ programs to be compiled with the Clang+LLVM compiler provided with the artifact.

Software Dependencies. Our artifact uses `CMake` to compile the Clang+LLVM sources. `CMake` can be downloaded from https://cmake.org/download/. To install `CMake` on Ubuntu use

```
$ sudo apt-get install cmake
```

`PTRacer` uses Z3 to check the satisfiability of the generated formula. Z3 is available on GitHub at the URL https://github.com/Z3Prover/z3. To install Z3 use

```
$ cd <Z3_base_directory>
$ python scripts/mk_make.py
$ cd build; make
$ export PYTHONPATH =
  <path_to_Z3_base_directory>/build
```

Our artifact uses `jgraph`, a postscript graphing tool, to generate the performance graph. The `jgraph` tool is available as a package for installing on Ubuntu. To install *jgraph* on Ubuntu,

```
$ sudo apt-get install jgraph
```

To convert the postscript graph generated by `jgraph` to a pdf we use `epstopdf`. To install `epstopdf` on Ubuntu,

```
$ sudo apt-get install texlive-font-utils
```

Installation. We provide two bash scripts to automate the installation of `PTRacer` and SPD3: `build_PTRacer.sh` and `build_SPD3.sh`. We use $< PT\_ROOT >$ to refer to the base directory of our artifact.

To build `PTRacer` run the `build_PTRacer.sh` shell script.

```
$ cd <PT_ROOT>
$ source build_PTRacer.sh
```

To build `SPD3` run the `build_SPD3.sh` shell script.

```
$ cd <PT_ROOT>
$ source build_SPD3.sh
```

Note, the shell scripts have to be sourced at the command-line. The installation will fail if they are run as executables (with ./ command).

Download the benchmarks from http://bit.ly/29i3OYL. To unpack the benchmarks,

```
$ cd <PT_ROOT>
$ tar -xvf <path_to_benchmarks.tar.gz>
```

2.2 Usage

Test-suite. The unit tests can be compiled by running `make` in the `test_suite` directory. The unit tests are compiled using the Clang+LLVM compiler provided in the artifact which instruments the unit tests with calls the data race detection library. To execute each unit test use

```
$ ./<unit_test>
```

Alternatively, we provide a python script `run_tests.py` to execute all the test programs and generate a test report. To run `PTRacer` on the unit tests use

```
$ python run_tests.py -d ptracer > report.txt
```

To run SPD3 use

```
$ python run_tests.py -d spd3 > report.txt
```

Note, to execute `PTRacer` on the unit tests, first run the build script for `PTRacer` and then run `run_tests.py`. Similarly, for SPD3, first run the build script for SPD3. This is necessary since the build scripts setup the appropriate paths to run the specific data race detector.

Benchmarks. We provide a python script that executes `PTRacer` and `SPD3` on the benchmarks. Since the benchmark applications are long running we suggest using the `nohup` command to run the script. To run the benchmarks,

```
$ cd benchmarks
$ nohup python run_bmarks.py > report.txt &
```

Note, to execute `PTRacer` on the unit tests, first run the build script for `PTRacer` and then run `run_bmarks.py`. Similarly, for SPD3, first run the build script for SPD3.

2.3 Expected Results

Test-suite. The python script `run_tests.py` reports the number of unit tests that succeed or fail. A unit test succeeds if the data race detection tool reports all the data races that exist in the unit test and does not report any false positives. For all the unit tests that failed, the python script reports the cause of the failure, whether it was due to a missed data race or a false positive. All the unit tests are expected to succeed when executed with `PTRacer`. In contrast, `SPD3` is expected to miss races and report false positives.

Benchmarks. The python script `run_bmarks.py` executes `PTRacer` and `SPD3` on each benchmark application and generates a bar graph called `Slowdown_graph.pdf`. This graph shows the relative slowdown of executing the benchmark application with the data race detection tool over the baseline execution of the application without instrumentation. The mean slowdown of `PTRacer` over all the benchmarks on a x86-64 bit 4.00GHz machine with four cores and 64GB RAM (with simultaneous multithreading disabled) is expected to be 6.7× plus or minus 1×. The mean slowdown of `SPD3` is expected to be 5.4×.

References

[1] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in cilk programs that use locks. SPAA, 1998.

[2] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. SPAA, 1997.

[3] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. PLDI, 2012.