Data Races

by

Jared Hall

Program Determinism

The question of whether or not a program is deterministic is a straightforward question when it's asked of a sequential program. The general rule of how to discover if a program is deterministic is to ask: if given the same input, is the same output always produced? As I said, it's usually easy to reason about sequential programs, but things can become confusing when reasoning about parallel programs because parallel programs make use of threads. The confusion is due to the programmer not controlling when the threads perform their work. This can create a scenario where different executions of the same program compute the same work in a different order. It is even possible that *different* work will be performed in a *different* order. The bottom line is that the programmer gives up some amount of control by parallelizing the program. Because of this loss of control, it is tempting to think that parallel programs can't be deterministic. This temptation is based on good reasoning, but is misleading. The way we judged if a program was deterministic was: if given a certain input, is the same output always generated? That measure is completely independent of the inner workings of the program. Under this rule, a threaded program can prove to be just as deterministic as a sequential program. Dividing up the work into concurrent pieces may or may not affect the outcome. In many cases, dividing up the work is just an attempt to do the same work faster by using extra cores, but will have absolutely no affect on the output. The key here is to always consider the input to output relationship.

(next: data races)

Data Races

A data race is defined as a scenario when:

> *A shared variable is subject to concurrent access by at least 2 threads, where at least one thread is making a write.*

As you can see, the definition has nothing to do with inputs and outputs to the program. It is best to think of a data race as the *potential* for something bad to happen. It's just a scenario, but the nature of the scenario is such that program determinism is at stake. The order of reads and writes by the concurrent threads could produce varying outputs for the same input. If a program has a data race, it is not deterministic. However, a program with a data race may display deterministic-esque properties for many executions, and may mislead the programmer into believing that the program is deterministic. This is because some data races may never produce a situation where different outputs are created for the same inputs.

Imagine a scenario where one thread is executing a conditional statement on a shared variable that another thread is incrementing. Let's say that the variable is initialized to 0. The first thread will execute the conditional code if the variable is 1, but skip the code if the variable is still 0. We can see that this is a data race scenario. If the incrementing thread increments the variable before the conditional statement resolves, then the conditional code will get to run, but otherwise it will not run. There is no way of knowing which outcome will be the case before runtime. It is possible that in 99 out of 100 runtimes, the conditional is true and is false just once. In this case the program may appear to be deterministic, but we know that it is not.

To rigorously analyze a program for data races, you have to imagine the program as having a schedule. The schedule determines when every line of code is executed. A sequential program that doesn't utilize concurrency will just have one schedule. However, concurrent programs will have an exponential amount of schedules based on the degree of parallelization. The more the program is made parallel, the more possible schedules (in an exponential

relationship). Using locks is a way of constraining the number of possible schedules.

If a debugger attempting to find data races tried to exhaustively enumerate all of these schedules, it would take too much time to be practical. The programmer could take it upon themselves to prevent data races by locking every variable that is used, but this greatly decreases potential parallelization, and so isn't ever used. A dynamic(runtime) approach and algorithm are outlined in the next lecture notes. It will simply detect when a variable has been associated with a lock, and find instances where the variable is not associated with a lock. In those instances, an error will be raised.