GoodRun: Enforcing Good Runs in Parallel Programs

Silviu Andrica, Cristian Zamfir, George Candea
EPFL (Lausanne, Switzerland)

We define a program's run to be good if its execution is characterized by *performance* and *output correctness* and *deadlock freedom*. A bad run is one that is not at par, performance-wise, with other runs, causes wrong outputs, or deadlocks.

When a program runs in production, it encounters two different classes of executions: previously seen and previously unseen executions. When encountering a previously seen execution, GoodRun knows how the program will behave with respect to the three execution characteristics, i.e. GoodRun is able to foretell if the execution the program is engaging in is a good or a bad one. Seen executions are gathered from test runs or runs in production, or, GoodRun has theoretically explored this execution, as it is described later. The current execution, defined by its thread schedule, is known by GoodRun if it represents a prefix of a saved thread schedule. When this is not the case, the execution is marked as unseen. Such an execution cannot be classified neither as good, nor as bad.

GoodRun transforms previously unseen executions into executions with known effects and enforces thread schedules that are good for the system, i.e. GoodRun enforces all three characteristics of a good run.

Performance: Under certain schedules, the program might run considerably faster than under other schedules. The explanation lies in either a difference in mutex contention, or in the order in which threads acquire locks. For the latter, suppose there are two threads T_1 and T_2 that both wish to acquire lock L. Further suppose that thread T_1, after acquiring lock L, performs an I/O operation—which takes a long time to finish—before releasing L. On the other hand, thread T_2 only updates in-memory data-structures—a quick operation—and releases L afterwards. GoodRun will allow T_2 to get the lock L before T_1, because the system will progress faster compared to the other case, since T_2 will not have to wait for T_1, and T_1 is only "slightly" delayed.

Correctness: Due to potential data races present in the code, the outcome of a program's run is dependent on the thread schedule. To preserve correctness and offer consistent results, GoodRun will enforce the access to shared variables to be always safe.

Deadlock Freedom: GoodRun will enforce the same order with respect to lock acquisitions and releases. Thus, no deadlocks occur due to lock-inversion.

The two latter restrictions imposed by GoodRun might incur high overhead or not be valid in the current run due to input dependencies. To bypass the impediment, the order of accessing different shared variables or locks can vary if there is no dependency edge between them in the happens-before graph describing the program's execution. To further reduce the imposed overhead, GoodRun performs only localized thread schedule enforcement. Using profilers, GoodRun can detect the hotspots on the application and devise a thread schedule that reduces contention. The respective thread schedule will be enforced only when reaching the hotspots. For the rest of the program's execution, the thread schedule is controlled by the normal scheduler. The same technique can be applied for locks. A bit trickier is to deal with shared variables. In a gross approximation, objects that are not created locally in a thread and kept privately are considered as shared. GoodRun will control the access to such objects.

The project is a dual to Dimmunix [3]. Whereas Dimmunix finds and avoids bad schedules leading to deadlocks, in GoodRun we find and enforce good schedules. When encountering an unseen execution, GoodRun has two choices:

Morph the current execution into one that is known to be good. We use two instrumentation mechanisms:

- Library Instrumentation: GoodRun gains control over thread scheduling and can enforce the desired schedule by instrumenting the *pthreads* library or the *Java Virtual Machine*.
- Program Instrumentation: The program is modified to enforce the desirable schedule. A first option is to inline GoodRun in the program as a set of assertions placed before locks acquisitions, shared

variables accesses, or slow operations. However, a change to the system, e.g. adding new functionality, requires the inlining process to be performed again. An alternative is to keep GoodRun and the controlled program separate and delegate thread schedule decisions to GoodRun, through lightweight instrumentation. Although this solution incurs a higher overhead compared to inlining, the gains are in ease of maintenance, i.e. only added or modified parts of the program need be instrumented.

Increase Surveillance: GoodRun performs no thread schedule enforcement. Instead, the goal is to collect enough diagnosis information to categorize the execution as good, and subsequently morph other executions into this one, or as bad, and avoid it next time. Because GoodRun is uncertain of the effects of this execution, the program will run in "quarantine". Its execution will be considered a transaction that succeeds if the program terminates and has a correct output. If the execution causes a failure, GoodRun will save it as a bad execution to be avoided and the run's effects on the entire system will be removed. GoodRun will also profile the application to discover possible performance improvements.

The challenges we expect to encounter include:

Schedule tracing: GoodRun needs a lightweight method for recording thread schedules. We currently use the *JikesRVM* Java Virtual Machine whose thread schedule can be easily tracked and controlled since it is embedded into the VM.

Schedule classification: For a given set of schedules, GoodRun must decide which good schedules are "better" than others. Since correctness and deadlock-freedom are required for all good executions, GoodRun ranks good executions based on their performance. GoodRun employs several techniques to rank related thread schedules based on their performance.

- **Static analysis:** By inspecting the program's binary, GoodRun infers the types of operations a thread will do. Assigning variable costs to these operations quantifies the impact on performance and enables GoodRun to take an informed decision.

- **Trace analysis:** Based on tracing program execution, GoodRun can identify opportunities for better scheduling by analyzing performance hotspots and devise a performance improved, localized thread schedule.

- **Profiling:** System performance is measured under the different schedules. GoodRun then concludes which specific features of the schedule have most influence on the performance (i.e., correlation analysis).

Inferring Good schedules: Using symbolic execution [1] and partial-order reduction [2, 4], GoodRun can extract valid thread schedules equivalent to a good schedule, creating a family of good thread schedules. Each family member can substitute any other member. Out of a family of thread schedules, GoodRun will aim to enforce the thread schedule promising the highest performance. When beginning execution of a program, out of all the good thread schedules, GoodRun will enforce the one offering the highest performance. However, program execution depends on user inputs which can invalidate the currently enforced thread schedule. GoodRun will then pick another family of thread schedules, choose the best performer and enforce it. With every condition branch taken, the set of available families of thread schedules changes—it reduces its size when the program is branching and increases its size when control flow paths merge. This suggests the threads should be classified as graphs resembling the program's control flow graph.

Helpers and optimizations: Application performance is not always influenced purely by the synchronization behavior, therefore GoodRun would have to find a way to distinguish the effects of, e.g., involuntary preemptions from the effects of synchronization. GoodRun could also explore the benefits of doing static analysis on the code beforehand, if the source is available.

Possible extensions include applying GoodRun to contention managers in software transactional memories. E.g., DSTM2 ships with 6 different contention managers, each one trying to be clever about how transaction conflicts are resolved.

References

[1] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[2] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *SIGPLAN*, 2005.

[3] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[4] R. Mittermayr and J. Blieberge. Static partial-order reduction of concurrent systems in polynomial time. In *ISoLA*, 2008.