Automatic generation of Synchronization Primitives for Multithreaded Programs

Short Description

Reasoning about synchronization mechanisms in multithreaded software is hard. This work aims to create an approach that would free developers from this hard challenge by generating optimal synchronization mechanisms for the software under consideration. During this work, it will be necessary to create an approach that reasons about concurrency bugs, potential synchronization mechanisms, and developer's synchronization intentions. It will be necessary to create a conceptual solution for this challenge, an algorithm for generation of synchronization mechanisms, and a prototype of a tool that demonstrates the benefits of this approach.

Abstract

In the era of multithreaded software, threads execute concurrently and independently. Input data and the interaction between concurrent threads determine the result of the concurrent software. In order for software to perform correctly, it is necessary to synchronize the interaction between threads. However, due to the nondeterministic nature of multicore hardware (e.g. shared cache, shared memory bus); it is hard to predict order of interactions between threads, as they do not progress uniformly. Developers find it very challenging to reason about necessary synchronization between the threads, as it is an np-hard problem. Eventually, developers either over synchronize their software (downgrading parallel programs by introducing many sequential sequences), or do not properly synchronize thread operations (leading to concurrency bugs).

The main issue behind synchronization perils is transformation of developer's synchronization intentions into synchronization mechanisms. Developers, at all time, need to understand two things: i) which memory locations are shared among threads, and ii) how threads can access shared memory locations (different execution paths, different operations). While developers should understand what memory locations thread share, it is very hard for them to understand possible interleavings of thread operations on shared memory. Existing approaches that try to solve this problem (either try to find concurrency bugs, or try to introduce automatic synchronization) start by identifying shared memory. This is not an easy task because: i) static analysis often introduces false warnings, and ii) dynamic analysis is often not complete for reasoning about shared memory. Even if an approach answers these two questions, the challenge still remains: Which synchronization mechanism is optimal (in terms of responsiveness, performance, and memory consumption) for accessing a certain shared memory location?

The idea of this work is to offer developers interfaces and functions for accessing shared memory. Developers would use these to express their intentions for accessing shared memory. Then, it will be necessary to trace the execution of these interfaces and functions to create execution traces and reason which threads share which memory location, and reason about operations they perform on these locations. Finally, the idea is to process performed operations and generate optimal synchronization mechanisms for different threads when accessing shared memory. The assumption is that developers, whenever they are trying to access shared memory,

would have to use these interfaces. In order to generate optimal synchronization mechanisms, it will be necessary to reason about code coverage (or other type of analysis, e.g., static analysis) that can guarantee appropriate completeness of trace (or other representation of software).

The interfaces solution would solve the challenge with automatic identification of shared memory locations. Furthermore, once all accesses to shared memory are running through a known set of interfaces and functions, it is easy to either create a tracing mechanism within the interfaces or use external tools (e.g., binary instrumentation) to trace those executions. One of the problems is that sometimes developers change their threads and their shared locations so that previously shared memory becomes exclusive to a single thread. However, because they are unaware of this, they still keep synchronization mechanisms, leading to performance overhead. The approach proposed in this work would solve this challenge as it would generate synchronization mechanisms based on profiling the application. Developers can choose between many synchronization mechanisms (e.g., lock-free, mutex, barriers, static scheduling). However, they are not always aware of the influence of these mechanisms on the responsiveness, performance, and memory consumption of their applications. With this approach, we will offer a possibility to generate synchronization mechanisms for a specific quality property of the software under consideration.

List of actions:

- Understand different synchronization mechanism types.
- Understand concurrency bugs.
- Design generic interface for accessing shared memory location (regardless of data type) in C/C++.
- Generate execution trace of these interfaces.
- Apply existing algorithms for identification of shared memory between threads (e.g., Lockset memory model).
- Create an algorithm for the analysis of execution trace in order to generate optimal synchronization between threads (do this considering these parameters: average performance, responsiveness, memory consumption).
- Select and prepare benchmarks to test (e.g., remove existing synchronization mechanisms).
- Apply the solution on the benchmarks.
- Use existing tools (e.g., Thread Sanitizer, Helgrind) in order to find concurrency bugs (compare original version with existing synchronization mechanisms, and the new one generated using this approach in terms of: concurrency bugs, average performance, responsiveness, and memory consumption).

Expected contributions:

- Interfaces for accessing shared memory (Read/Write).
- Automatic generation of optimal synchronization mechanisms.
- Discussion of a necessary code coverage and completeness of execution traces and their relation with the completeness of the approach.

Keywords: Multithreaded Software, Synchronization Mechanisms, Concurrency Bugs.