Report 2: Threads Cannot be Implemented as a Library

Dalia Camacho García-Formentí,
Instituto Tecnolgico Autnomo de México (ITAM)
Mexico City, Mexico

I. Summary

In the article 'Threads Cannot be Implemented as a Library' [1], Boehm presents some of the main issues that arise when multithreaded programming is approached at a library level, while taking foregranted the compiler. This article is focused on the Pthreads in C/C++.

The problems approached in this article are related to the order in which the compiler (or hardware) restructures the program for its execution. If code reordering is managed at a single thread level, then, data races may arise in multithreaded programs, even when locks are correctly implemented in the source code.

Pthreads library was created in a way that would be easy for programmers to work with multiple threads without the need of handling memory models. This is done by the usage of primitives such as `pthread_mutex_lock()` and `pthread_mutex_unlock()`. The first function adds barriers to the hardware that avoid instruction reordering. The function `pthread_mutex_lock()` is also an opaque function, which means that the compiler expects it to read or modify some global variables, but without knowing what to expect precisely. Usually, this prevents the compiler from reordering the following instructions. However, this approach may fail and the user should be aware of the corresponding issues.

The first problem the author considers is concurrent modification. This problem may occur when the compiler reorders the code, then variable modification can be done concurrently in an unexpected manner, which alters the behaviour of the original program.

The second issue has to do with storing variables that are not written explicitly in the source code. This problem is called rewriting of adjacent data. The memory location of a variable that is not defined in the source code will not be protected by the primitives of a library. Thus, any thread can modify this memory location without any barriers.

The last problem Boehm explores is register promotion. This problem is illustrated with an example of a for loop that continuously accesses and modifies a variable x. If another thread may have access to x, then x is protected. If the compiler considers the latter rarely occurs, then x can be saved as a register, which is faster to access, but this increases the number of unprotected reads and writes, which increases the hazard of data races to occur.

After showing the possible concurrency issues, Boehm considers the performance of the Pthreads library. The main problems are the overhead of calling the primitives; the barriers that prevent code reordering execution; and the amount of cycles needed for atomic operations which require modifying the values in cache or in memory instead of using registers.

To evaluate the performance of programs that implement the Sieve of Erathostenes and HTP4 are used to compare the performance of *mutex* locks, *spin* locks, *volatile* access, and unsafe access for one, two and four threads. The implementation with mutex locks was the slowest program, spin locks performed a little better, but the best performance was achieved by using unsafe access. In this particular example having multiple threads increased the performance, in every case. However, having multiple threads with synchronisation had a similar performance than that of a single thread in the unsafe access for the Sieve. While, the performance on HTP4 was better with a single thread in the unsafe scenario compared to multiple threaded programs.

The author proposes treating the problem at the compiler level taking the Java Memory Model as a basis, but with the proper modifications for C++.

II. Analysis

The issues addressed in this article appear mostly at a compiler level. More specifically techniques that improve performance such as code reordering and speculative execution may generate data races, if threads are considered independent from each other. The locks implemented by Pthreads try to prevent code reordering and speculation, however, the compiler can still have ill behaviours that affect the final result. Moreover, if Pthreads does prevent code reordering, performance is severely affected, because without reordering the hardware cannot make the most of the pipelineing technique.

If we are concerned about data races and performance, according to the author's experiment, the best approach would be using a single thread with the unsafe access. In other words, we would use sequential programming, which underutilises computational resources. Even more, the latter is one of the main reasons to consider multithreaded programming instead of sequential programming. But if we simply use multithreaded programming without synchronisation, we may obtain ill results, which make the program useless.

Thus, Boehm's proposed solution seems as an accurate approach. This solution will solve the issues at a compiler level and if done correctly data races can be avoided while taking advantage of techniques such as code reordering and speculative execution, which increase the performance. With this solution reordering of instructions and out of order execution done by the hardware could still lead to data races, however a solution at a compiler level is a good first approach.

References

[1] H.-J. Boehm, "Threads cannot be implemented as a library," *SIGPLAN Not.*, vol. 40, pp. 261–268, June 2005.