CS575 Paper Analysis Project

Threads Cannot Be Implemented As a Library

Chiu-Chun, Chen

Email: chenchiu@oregonstate.edu

June 7, 2022

GENERAL THEME

The general theme of this paper is about thread problems that can't be solved with the Java programming language due to the compiler or the thread theory itself. Threads cannot be implemented as a library, as the title implies. This paper aims to explain why threads can't operate properly under certain scenarios if the library is only used for parallel programming.

THE AUTHOR

Hans-J. Boehm, the author, has been a software engineer at Google in Palo Alto, California, since March 2014. He now focuses on concurrent programming difficulties in general and Android in particular. He is a past Chair of ACM SIGPLAN and an ACM Fellow (2001-2003). He chaired the ISO C++ Concurrency Study Group (WG21/SG1) until late 2017, and he continues to participate actively in the group. From 1978 to 1983, he received his master's and doctoral degrees in computer science from Cornell University, and from 1984 to 1989, he worked as an assistant and associate professor at Rice University. His homepage (https://hboehm.info/) contains further personal information.

ARTICLE SUMMARY

The article provides several examples of how multithreading may not operate correctly in high-level programming languages like Java, C#, or Ada. Because of the compiler and the logic itself, the essay concludes that multithread programming cannot be utilized as a library. Take me for an example, I used to use multithread in C# and Python when learning at this class, and I'm aware that there are some drawbacks to utilizing multithread, such as Python's GIL, which prevents it from using multithread. This work, therefore, gives me a better understanding of why high-level programming languages have parallel programming restrictions.

According to the article, multi-threaded routines that are not built using thread support, such as C/C++, instead of using a "multi-thread library," may result in mistakes. The use of these libraries produces erroneous results that may lead programmers astray. As a result, this article will discuss why the matter is so significant. Multi-threading is vital because it allows separate logic to run at the same time during concurrent interactions, such as when you open a browser while also opening a calculator. In addition, multiprocessors are the norm. With greater computation, it even

uses less power. According to the author, all of the advantages can only be realized by developing intentionally multi-threaded programs.

The importance of multi-threaded applications is that they allow multiple threads to share memory. However, experience has shown that employing multi-thread functions in languages like C# and Java can lead to errors. These impairments are often sporadic, making them difficult to detect during testing. As a result, it's critical to address the issues that lead to unreliable programs.

THE PTHREADS APPROACH TO CONCURRENCY

According to this paper, if a programming language supports concurrency, it must support multi-threaded execution semantics, particularly the memory model. Typically, concurrent executions might set all variables to zero at the start. Consider the following scenario:

Thread 1: x = 1; r1 = y;
Thread 2: y = 1; r2 = x;

Thread 2 must execute before the x value can be set. As a result, r1 and r2 can both be 1. In the experiment, however, both r1 and r2 will be zero. The author gives two reasons: the first is that compilers may reorder memory, which has no meaning for each thread and is isolated. Another reason could be that the hardware reorders the codes, resulting in a store result in the buffer for other threads to see.

The problem can be solved by employing a mutex lock, which is uncommon in Java but very useful in C/C++. The pthread mutex lock function in C/C++ can solve the problem by ensuring "synchronized memory" and preventing reordering. The function can also avoid compiler reordering because the value may be read or written by the compiler. The solution works the most of the time, but it may still create incorrectness, and it may also fail occasionally. The reason for this is that it does not specify threads and hence does not constrain the compiler sufficiently. In addition, the solution may impose a limit on the finest algorithmic approaches that break the preceding criteria.

CORRECTNESS ISSUES

Another consideration is the problem of correctness, according to this paper. For example, concurrent modification allows programs to access memory that is being used by other threads. Consider the following scenario:

```
if ( x == 1 ) ++y;
if ( y ==1 ) ++x;
```

If our compiler is allowed to convert sequential code that does not contain calls to pthread operations in any way that retains sequential correctness, the above could be transformed to:

```
++y; if (x != 1) --y;
++x; if (y != 1) --x;
```

According to the compiler, there is a race, thus x == 1 and y == 1 might be the result. The issue is that the compiler is unaware of the existence of threads. Even though we employ mutex lock, the remaining two issues are even more serious. It is unavoidable that one is complier could introduce a race. Any additional fields that share memory with this one will be read and rewritten by the compiler. To demonstrate the issue, the author provides a thorough example. Promotion is another issue. The result may not be valid if we introduce extra reads and writes while the lock is not held. These examples show how compilers must be aware of threads and deal with thread-specific semantic difficulties.

PERFORMANCE ISSUES

Finally, the author brings up the subject of performance issues. Because the Pthread library forbids changing shared variables, performance will be limited and sanctioned unless memory operations are isolated. Furthermore, it is difficult to write and debug. The mutex lock is a hardware atomic that allows several threads to share memory. This instruction is frequently used while using the Pthread library. As a result, the Pthread library builds on them at a substantial cost of performance.

This library may leverage hardware primitives for less restrictive use of the shared variable in some circumstances to avoid deadlock issues, which may reduce performance. However, there are lock-free and wait-free programming strategies to avoid locks; in our instances, they all entail races. These strategies, as far as we know, only work for a tiny percentage of multi-threaded programs, and they also require lower-level libraries, which could affect the performance of authors who are unfamiliar with them. These strategies tinker with shared variables using standard load and store instructions; some of them function as expected, but the majority of them suffer performance degradation due to reordering limitations.

Another concern is that synchronization reduces performance. The author uses a million list updates as an example. Any thread will not invoke set if the index exceeds 10000. As a result of being visible to other threads, threads are particularly weak in ordering operations. In general, as the number of threads increases, the speed-ups decrease. The author provides evidence to show that using the Pthread mutex primitives without synchronization would not improve performance. On other processors, such as the Pentium 4, synchronization consumes more resources, maybe because the single-threaded version appears to be essentially ideal, or because the memory system is already saturated.

SOLUTIONS

Members of the author's team are working on solutions to these issues for the C++ standard. They recently predicted that a solution based on the Java Memory Model might overcome all of the concerns stated above. However, in a variety of ways, it was modified to the design aims of other languages:

1. It is unnecessary to properly specify the semantics of all data races due to the lack of a safety assurances type. Instead, use the volatile keyword to prevent "data races" and to share variable access amongst library routines. As a result, the threads approach is preserved. When a data race exists, this would necessitate a more precise definition.
2. Some type-safety and security issues aren't as serious as others. For example, in the context of a type-unsafe language, most of the work on causality is unnecessary.
3. In some circumstances, the performance of the Java memory model was sacrificed for simplicity, which may or may not be correct in this text.

4. The compiler must incorporate stores in the case of C++ bit-fields, resulting in the potential of races that were not included in the sources. Modern architectures may be able to impose constraints on adjacent bit-fields.

POSSIBLE FLAWS

One issue I noticed is that the author only highlights potential concerns that are currently occurring without conducting any research. Certain questions may have emerged, such as whether the author's highlighted issues will occasionally occur on a bug-free programming language. Could we see whether there's a programming language that can solve this study's problem? If I were the researcher, I'd explain why threads can't be made available as a library for individuals to evaluate on their own. Also, as the conclusion may change over time, it would be preferable if we could change the title to "cannot be used due to..." so that others can research or address the issues.

CONCLUSION

Parallel programming in high-level programming languages may not perform as planned due to a lack of explanation and programming in low-level programming. Even if we can enclose functions for certain parallel programming conditions, it's difficult to define every feature of each program with our encapsulation; in fact, pthread has already done so, yet it's still far from matching realistic parallel programming. Both the software logic and the hardware instructions' mechanism are considered in parallel programming. Therefore, high-level programming languages still require a significant amount of effort to execute effective parallel programming in the way that we would like.