CS 575 -- Spring Quarter 2017

Paper Analysis Project

Tanya Khemani

Topic: “Threads Cannot Be Implemented as a Library”

Theme

The paper talks about the approach to threads by taking the example of Pthreads (in context of C) approach. It explains how and why it works. The author quotes "A lot of multi-threaded code is developed in languages that were designed without thread support." So, in order to handle concurrency issues the programmers are directed to the use of libraries such as Pthreads.
In this paper, the author points out some issues that he claims makes assuring a multi-threaded program's correctness not possible.
The paper also states that in order to reason about the correctness of a multi-threaded program there must be a specified memory model that describes what happens when code referring to the same variables runs concurrently or which variable assignment are visible to other threads. Languages like C and C++ lack a specified memory model. Pthreads attack this problem by stating that all write accesses to shared variables must be done between calls to the library primitives; pthread_mutex_lock() and pthread_mutex_unlock() or similar functions. The idea being that since the compiler knows nothing of the functions and must treat them as opaque functions(logically treats primitives like function calls accessing global data), thus the compiler must make sure all variables must be written back to memory before the function call is made and not move memory operations around such function calls.
There are three potential issues regarding program correctness which are presented in the paper are **concurrent modification**, **rewriting of adjacent data** and **register promotion**.

About the author

The author of the paper is Hans-J. Boehm. He is from Palo Alto, California. He works as a software engineer at Google where he has been since March, 2014. Past that, he worked as a Researcher, Research Manager at HP Labs(1999-2014). He also chairs the ISO C++ Concurrency Study Group (WG21/SG1). He is an ACM Fellow, and a past Chair of ACM SIGPLAN (2001-2003). Also he chaired a variety of SIGPLAN conferences. He has also been a Software Engineer at SGI(1996-1999) and a Researcher at Xerox PARC(1989-1996). He was an

Assistant and Associate Professor at Rice University(1984-1989) and an Assistant Professor at University of Washington(1982-1984).

Talking about his interests and project work,

He implemented the arithmetic evaluation engine for the default Android Calculator which avoids cumulative errors, and provides arbitrarily scrollable, accurate, results. He worked on understanding how to program systems with **non-volatile byte addressable memory** and it will impact our basic programming model. He helped to uncover and remove obstacles to writing **reliable multithreaded code** by participating in the revision of the Java "memory model", and more recently led a successful effort to properly define shared variable semantics in C and C++. Also he worked on Understanding the **compiler optimization consequences** of the preceding work. He participated in the C++ **transactional memory** standardization effort (WG21/SG5) and worked on "always on" **data-race detection.** The area of Conservative Garbage Collection interested him. He started working on this at Rice University where he was involved on the implementation of the Russell programming language, which was jointly developed with Alan Demers and then with the help of many other contributors at Xerox PARC, SGI, and HP which finally resulted in a generally available and widely used garbage collector library. Fast multiprocessor synchronization interested him and he worked on fast lock implementations for Java virtual machines. He also coauthored a paper on practical implementation of monitor locks without hardware support. Hans also reimplemented Xerox Cedar environment both in C at Xerox and in a very different form for C++ at SGI.

Experiments

Three potential issues regarding program correctness are presented in the paper.

1. **Concurrent modification**
   The Pthreads specification prohibits access to a shared variable while another thread is modifying it or in other words, Pthreads mandates protecting shared variables, but

whether this can be assured or not depends on the semantics of the underlying language of which the library specification has no control. What variables should be protected is determined by memory model, which is unspecified. Since a language designed without concurrency might transform code in such a way that the semantics of the program remain the same in a single threaded context. As an example, two threads each executing one of the following statements where x and y are initially zero:

Initial state
x=y=0

Thread 1:

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

Thread 2:

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

could be transformed to:

Interleaving:

Thread 1:

```
        ++y  [y==1]
        if (x != 1)
           --y [y==0]
```

Thread 2:

```
        ++x  [x==1]
        if (y != 1)
          --x [x==0]
```

In a single threaded environment this would not change the semantics, but when there are two threads involved there is a race since the shared variables are altered without using the locking primitives of the Pthreads library.

2. **Rewriting of Adjacent Data**

   This problem occurs due to adjacent data transformations when partially updating C structs. The issue brought up emphasizes the fact that the compiler might write to adjacent memory locations while storing a variable to memory. In fact, in some situations it might be unavoidable; consider a struct containing bit fields of the form:

   ```
   struct { int a:17; int b:15 } x;
   ```

   An assignment to the variable x.a is likely to be implemented as a 32 bit wide store instruction causing the value of x.b to be rewritten, which is alright for sequential code, but if there is a concurrent update to x.b between the old value of x is read and the new value of x.a is written, the update of x.b is lost. For bit fields this is a well recognized fact and does not actually violate the Pthreads standard since only concurrent writes to "memory locations" is prohibited by it. But, in general there is nothing that prohibits a compiler to rewrite adjacent data in other situations as well.

   Adjacency Transformation (locking a but not b as it is not shared between threads) and Interleaving:

Thread 1:	Thread 1:	x.a =1
x.a =1	x.a =1	x.a =0
	x.b = 0	x.b=3
		x.b=0
Thread 2:	Thread 2:	
x.a=2	x.a = 2	
x.b=3	x.b=3	

3. **Register promotion**

   Optimizing compilers often promote frequently accessed variables to a register sometimes for speedup. Example: promoting a variable in a loop to a register because profiling determined that the loop usually runs without threads. Example:

```
for (...) {
    ...
    if (mt) pthread_mutex_lock(...);
    x = ... x ...
    if (mt) pthread_mutex_unlock(...);
}
```

   For performance reasons the programmer wants to avoid the extra overhead of calling the lock functions if there is only one active thread. Since x is accessed inside a loop and the compiler might speculate that mt is most often false, this could be transformed to the following:

```
r = x;
for (...) {
    ...
    if (mt) {
        x = r; pthread_mutex_lock(...);
    r = x;
    }
     r = ... r ...
     if (mt) {
      x = r; pthread_mutex_unlock(...);
      r = x;
  }
}
x = r;
```

```
Initial state
x=0
f() {x++}                f() is like lock()
```

```
Thread 1:
for(…)
f ()
x = 2*x
```

```
Thread 1:
r=0
for (…)
x=r
f ()
r=x
r=2*r
```

Extra reads and writes to a shared variable have been introduced outside the lock and the code is no longer safe as threads could interleave such lockless code.

Conclusion

- Threading should be a part of a formal language specification, rather than a user library.
- Failing to do so results in incorrect compilation and the inability to express lock-free algorithms, resulting in poor performance.

Insights

After referring the paper, what I found insightful was that the programs which rely on the memory ordering without explicit synchronization are extremely hard to write and debug, Mutex locks typically require one hardware atomic memory update instruction per library call and deadlocks may be introduced in lock-based programming.

Flaws

One flaw that I found in the paper was in the Pthreads approach to Concurrency. Talking about the sequential consistency, the author says that if we started in an initial state in which all variables are zero, and one thread executes:

Thread 1:

x=1; r1=y;

while another executes

Thread 2:

y =1; r2 = x;

Sequential Reordering and Interleaving:

Thread 1:	Thread 1:	
x=1	r1 = y	r1= y[0]
r1=y	x=1	r2 = x[0]
Thread 2:	Thread 2:	
y=1	r2=x	x=1
r2=x	y=1	y=1

For Thread 1, the compiler could eliminate the redundant read of X, replacing r2=X with r2=r1. This allows deducing that r1==r2 is always true, making the write of Y unconditional. Then the compiler may move the write to before the read of X since no dependence is violated. Sequential consistency would allow both the reads of X and Y to return 1 in the new but not the original code. This outcome for the original code appears to violate causality since it seems to require a self-justifying write of Y. It must, however, be allowed if compilers are to perform the common optimization of redundant read elimination.

Initially x=y=0

```
Thread 1:        |   Thread 2:
                 |
r1= X;           |   r3 = Y;
                 |
r2= X;           |   X = r3;
                 |
if(r1 == r2)     |
                 |
Y =1;            |
```

After compilergtransformation

```
Thread 1:        |   Thread 2:
                 |
Y =1;            |   r3 = Y;
                 |
r1=X;            |   X = r3;
                 |
r2=r1;           |
                 |
if(true);        |
```

Future scope

We can make use of Artificial Intelligence with the current approaches of multithreading to detect and modify the issues caused by library based threads. Though this is a difficult step since it involves usage of AI based algorithms but eventually we can make it happen. By using this, the authors of the code may know the issues caused by library based threads.