How a Rogue Optimization Breaks C11 Memory Consistency

A compiler transformation that caches a global variable introduces a data race

Abstract

A widely used compiler optimization contains a serious error that breaks C11's memory consistency model by introducing a shared memory data race. Although the optimization is perfectly valid for single-threaded programs, it has the potential to break multi-threaded programs based on both C11's thread model and the commonly used pthread model.

By Marcel Beemster

(c) Copyright 2016 by Solid Sands B.V., Amsterdam, the Netherlands
SuperTest™ is a trademark of Solid Sands B.V., Amsterdam, The Netherlands

The Test Program

The error is easy to invoke with the following simple test program (caveat: nothing related to memory consistency is 'easy' or 'simple'):

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

#define RANGE  10000
static int sharedVar;

static void *finder (void *fromIndexp) {
    int fromVal = *(int *)fromIndexp;
    for (unsigned int i = fromVal; i < fromVal+RANGE; i++) {
        if (i * i == 81) {  /* Only true once for T1 */
            sharedVar = i;  /* 2nd write to sharedVar by T1 */
        }
    }
    return NULL;
}

static void multiFinder (int i) {
    pthread_t t2;
    int argT1 = 0;
    int argT2 = RANGE;
    sharedVar = 0;          /* First write to sharedVar */
    pthread_create (&t2, NULL, &finder, &argT2); /* Start T2 */
    finder (&argT1);        /* T1 and T2 in parallel here */
    pthread_join (t2, NULL);/* T2 terminated */
    if (sharedVar != 9) {   /* Final read of sharedVar by T1 */
        printf ("Loop %d, FAIL: 9 != %d\n", i, sharedVar);
        exit (1);
    }
}

int main (void) {
    for (int i = 0 ; i < 100 ; i++) {
        multiFinder (i);
    }
    printf ("Success\n");
    return 0;
}
```

After initialization of global variable **sharedVar** by the main thread (T1), the program creates a new thread T2. T2 executes, in parallel with T1, the same function as T1 but with a different argument. Only T1 writes to **sharedVar** while the two execute in parallel, and neither thread reads from **sharedVar**. This means there is no data race. T1 waits with a 'join' for T2 to terminate. After that T1 verifies the expected value of **sharedVar**. If this fails, the program terminates with an error.

All of the actions before are in a loop that continues until a failure of the expected value check occurs, or 100 iterations did not fail. If the incorrect optimization is present, failure typically appears within 5 iterations.

To demonstrate, compile the program with optimization **`-O1`** on a GGC-4 compiler, for example gcc-4.9 (check the compiler version with **`gcc -v`**; install 4.9 on Linux using **`sudo apt-get install gcc-4.9`**), and run it. Note that **`-std=c11`** is not required to trigger the error, but is technically required to provide C11's memory consistency model:

```
gcc-4.9 -O1 -std=c11 test.c -lpthread && ./a.out
```

There is good chance you will get this output (perhaps at a different iteration):

```
Loop 2, FAIL: 9 != 0
```

You can run it more than once and get different iterations reported. The question is: Why did thread T1 not write to the shared variable in that iteration?

Let us first check that the program can work as intended by compiling it without optimization:

```
gcc-4.9 -std=c11 test.c -lpthread && ./a.out
```

The resulting output is:

```
Success
```

The error is the result of a rogue compiler optimization of the loop in the function **`finder`**. This is the reasoning behind the compiler's optimization:

- A global variable **`sharedVar`** is written inside the loop
- Writing to global variables is expensive
- Repeatedly writing to the same global variable in a loop is more expensive
- Inside a loop, the write to a global variable can be cached in a register so that multiple expensive writes are avoided
- Let's cache **`sharedVar`** in a register!

And this is the compiler transformation that is applied. Original code:

```
for (i = fromVal; i < fromVal + MAXITERS; i++) {
      if (i*i == 81) { sharedVar = i; }
}
```

Optimized code:

```
cachedVal = sharedVar;
for (i = fromVal; i < fromVal + MAXITERS; i++) {
      if (i*i == 81) { cachedVal = i; }
}
sharedVar = cachedVal;
```

This is a useful optimization. It is perfectly valid if that code would run in a sequential program. It would also be valid if **sharedVar** would be written at least once inside the loop. But that is not the case in the test program: it is multi-threaded and the loop in thread T2 does NOT write to **sharedVar**.

Before the transformation, only thread T1 writes exactly one time to **sharedVar** in the loop. After the optimization, both threads simultaneously read from and write to **sharedVar**:

THE COMPILER HAS INTRODUCED A DATA RACE!

With that data race after optimization, the following sequence (schedule) of events erases the write to **sharedVar** by T1. Other schedules are possible and not all of them result in a failure:

```
T2:  cachedVal = sharedVar;   // cachedVal (of T2) = -1
T1:  cachedVal = sharedVar;   // cachedVal (of T1) = -1
T1:  cachedVal = i;           // cachedVal (of T1) = 9
T1:  sharedVar = cachedVal;   // sharedVar = 9
T2:  sharedVar = cachedVal;   // sharedVar = -1 (!)
```

As you can see, the write-back of the cached value by T2 has erased the expected value of 9 in **sharedVar**. The reason is that the compiler optimization did not anticipate the possibility that the shared variable is not written at all in the loop.

One may question if the compiler is wrong or right to perform the optimization. Clearly the program is not working as intended, but should the developer not have been more careful with the handling of the shared variable? That is a valid question to which the answer is 'NO': the program is correct and the compiler is wrong. More on this below.

A correct optimization is the following:

```
hasWritten = 0;
for (i = fromVal; i < fromVal + MAXITERS; i++) {
    if (i*i == 81) {cachedVal = i; hasWritten = 1;}
}
if (hasWritten == 1) {sharedVar = cachedVal;}
```

In this way, the write is protected by a condition if it did not happen in the loop. At the same time, no initial read is necessary any more. This is indeed the code that GCC-5 produces.

Lessons

An important lesson from this error is that there exist optimizations that are perfectly acceptable in sequential programs, but that are incorrect in parallel or multi-threaded programs. In principle this means that a compiler has more freedom to optimize a sequential program than a multi-threaded program.

In fact, the C11 standard warns about such incorrect optimizations in NOTE 13 in Section C11:5.1.2.4p27:

> *Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this standard, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race.*

A second lesson is that testing for memory consistency is non-trivial. Almost by definition the test has to be multi-threaded and any errors have a non-deterministic behavior.

Thirdly, although the example demonstrates with GCC-4, the optimization can be present in other compilers. Developers that rely on shared memory consistency should check their development tools.

Background: Memory Consistency Models

A memory consistency model of a programming language or system is important when multiple threads communicate via shared memory. The memory consistency model defines what programmers can expect from variables (objects in shared memory) that are shared between multiple threads.

There are many different memory consistency models, in particular at the level of hardware. Their differences can have a huge impact on the possible style of multi-threaded programming.

The easiest to understand model is that of sequential consistency. With sequential consistency, all threads observe memory transactions in the same order and they all have the same view of the memory. It is needed to implement "Dekker's Algorithm" to synchronize between threads without hardware synchronization instructions. Sequential consistency is the easiest to understand but the hardest to implement in multi-processor hardware. In a multi-level cache architecture, reads and writes to memory have to be managed judiciously and can be localized (which is necessary for efficiency) only when they are not shared between processors. The Intel x86's Total Store Order model comes close to sequential consistency, but it is not quite the same (`http://blog.regehr.org/archives/898`).

Sequential consistency is costly to implement in hardware in terms of speed and power efficiency. So, most multi-processor hardware architectures (such as the ARM architecture) choose a much weaker memory consistency model that allows caching of reads and writes without having to check with remote caches. Instead, the shared memory is only synchronized when an explicit memory synchronization action is performed, such as a cache flush.

To implement a sequential memory abstraction on top of a weak memory consistency model is possible but very performance inefficient because it requires many explicit memory synchronization operations.

Therefore, C11 (and C++11) have adopted a 'weak memory consistency model'. It makes it possible to efficiently implement C11 on top of architectures that support only weak consistency models. The price to be paid is that developers cannot assume sequential consistency and cannot write multi-threaded programs that contain data races. A data race occurs when two threads run in parallel without some form of synchronization and one of them writes a shared variable, while the other reads or writes the same shared variable.

You may ask how shared variables are still useful for communication when this is not allowed. The answer is in the clause 'without some form of synchronization'. If there is explicit synchronization between the two threads, then one thread can write to the shared variable before the synchronization point (also known as a 'barrier'), and the second can read from it after the synchronization point, or overwrite it.

In Practice

In practice, the C programming language was designed to be an Operating System programming language. Operating systems are almost by definition multi-threaded shared-memory programs and C has served its purpose well. Before C11, C did not have a memory consistency model at all. This was not a big issue when the hardware memory behavior was quite transparent and C compilers were not so aggressive with optimizations. When issues would surface, developers were careful with shared variables (by protecting them with 'volatile', for example). Also, despite the failure of our test program, compilers are typically prudent with the introduction of writes to memory.

But the tides are shifting. The ARM architecture has a weaker memory consistency model than the x86, allowing more freedom in the implementation of the memory hierarchy. Making variables 'volatile' is not sufficient any more because that only affects the compiler, not the cache hierarchy. This implies that developers have to be more cautious with shared data manipulation. And they have to be able to rely on the compiler to behave as specified by the C11 language standard: when it comes to memory consistency, C99 is not sufficient.

Compilers themselves also become more aggressive when it comes to optimizations due to much improved static analysis techniques. Not all compiler engineers are aware that optimizations can be harmful to multi-threading. For many years, a debate is ongoing about the proper optimization level to compile the Linux kernel. See for example:

> **http://unix.stackexchange.com/questions/153788/linux-cannot-compile-without-gcc-optimizations-implications**

and this one about permissible optimizations related to undefined behavior:

> **https://gcc.gnu.org/bugzilla/show_bug.cgi?id=71892**

The Test Program and the C11 Language Standard

Given this long introduction, we can now understand what the C11 standard has to say about the test program.

First, the standard defines the notion of 'conflicting' actions (C11:5.1.2.4p2):

> "*Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location.*"

The term *conflicting* is not entirely intuitive, as it seems to imply something is wrong. This is not the case. It just means that the actions are related through a common memory location.

Next, the standard defines *data race* and specifies that a program with a data race is not valid (C11:5.1.2.4:p25):

> "*The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.*"

The *data race* clause provides three techniques to avoid that two conflicting actions turn into a data race. The first is to ensure that the conflicting actions are in the same thread. The second technique is to use atomic, which we will ignore here because the test program does not use it. The third technique is to establish that the actions have a specific order.

In the test program, there are three conflicting actions that are relevant to the error: the initial write to **`sharedVar`**, the write to **`sharedVar`** in the loop, and the read of **`sharedVar`** after the loop. These actions are all performed by T1. This implies there is no data race for these conflicting actions (technique 1: all conflicting actions are in the same thread). The fact that T2 runs simultaneously with T1 for a while does not affect the order of the conflicting actions since T2 does not read or modify the shared variable.

Technically, the program contains more conflicting actions that need to be analyzed. For example, there is variable **`argT2`** that is written by T1 and read by T2. Such analysis will show that all conflicting actions are in the same thread or properly ordered according to the C11 standard.

Hence it is concluded that the test program is a valid C11 program. There is no undefined behavior and the compiler is wrong to introduce read and write actions of **`sharedVar`** in T2 in an optimization.

History and Further Reading

The error was first reported in 2007 by the author for the GCC OpenMP compiler and discussed at length on the OpenMP mailing list. Unlike C at the time, OpenMP did have a memory consistency model. Unfortunately that OpenMP list is no longer on-line and only some fragments of the discussion remain in the internet's archive, for example here:

http://web.archive.org/web/20090108032918/http://openmp.org/pipermail/omp/2007/000845.html

The report did make it into GCC's Bugzilla, but it was not resolved at the time:

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=31862

With the approval of C11 in 2012, C gained a precise model of memory consistency. The error then became an error in the implementation of C11 too, in addition to OpenMP. The error was finally fixed in GCC-5, in 2015, while the last GCC-4 update, GCC 4.9.3 in June 2015, still contains the rogue optimization. This common optimization may also be present in other compilers.

For more reading into memory consistency, the works of Hans-J. Boehm are worth studying (**http://hboehm.info/pubs.html**). A good start is this report by Boehm:

http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf

The title of this report from 2004, *Threads Cannot Be Implemented As a Library*, precisely refers to the kind of optimization discussed here: a compiler has to be aware of the memory consistency model, and the language standard needs to define such a model if it is used for shared memory multi-threaded programming.