How to Miscompile Programs with Benign Data Races

Hans. J. Boehm

Presented by
Akshay Gopalakrishnan

May 29, 2024

Introduction

- Programs with Data Races are problematic as their behavior is unpredictable.
- Identifying data races are important.
- Certain data races are not "harmful" from hardware standpoint, they are "benign" and do not result in program misbehavior.
- However, in languages like C++, such benign races are also harmful due to Compilers.
- This paper shows for each category of benign race, how a compiler can optimize code resulting in unpredictable behaviors.

Data Race

A data race is when two memory operations accessing the same memory, of which at least one of them is a write, occur simultaneously.

Two memory operations occur simultaneously if they are next to each other in an interleaved execution order.

Benign Data Race

- Lazy Initialization.
- Reading Old and New values.
- Redundant Writes.
- Bit Manipulations.

Reordering across Lazy Initalizations

```
if (!init_flag) {
    lock();
    if (!init_flag) {
        my_data = ...;
        init_flag = true;
    }
    unlock();
}
tmp = my_data;
```

```
if (!init_flag) {
    lock();
    if (!init_flag) {
        my_data = ...;
        init_flag = true;
    }
    unlock();
}
tmp = my_data;
```

If you want to set the init value to *true*, you can instead do it by double checking the initial value of true. This is just to avoid too many writes to the same value of *true* by multipel threads. It rather helps to read twice, which is much less expensive than one write, which will also trigger cache lines to be invalidated, and so on. The proper way to do this however, would be to acquire a global lock, and check once again inside the value of init before changing it. The outer check before lock acquisition will first ensure if the init is updated, do not compete for lock. While the inner check is to save on cycles on repeating writes by different threads.

```
if (!init_flag) {
    lock();
    if (!init_flag) {
        my_data = ...;
        init_flag = true;
    }
    unlock();
}
tmp = my_data;
```

```
if (!init_flag) {
    lock();
    if (!init_flag) {
        my_data = ...;
        init_flag = true;
    }
    unlock();
}
tmp = my_data;
```

In the original program, nothing prevents the compiler from reordering the write to data before the write to init flag. In a concurrent context, this would violate certain message passing litmus. However, the argument would be that this is done within a lock primitive, so it should be okay. But this would be fine only if the reads to both init and data are done only after acquisition of the same lock. Even if this is the case, the compiler can also reordering the read to data above the outer conditional block; thereby changing its value again by introducing a read within the inner conditional block, after data changes. This kind of hoisting firstly creates a security problem, wherein temp has the old value of data even if it is not meant to always see it.

Introducing Reads to Enable Register Spilling

```
{
    int my_counter = counter;  // Read global
    int (* my_func) (int);

    if (my_counter > my_old_counter) {
        ... // Consume data
        my_func = ...;
        ... // Do some more consumer work
    }
    ... // Do some other work
    if (my_counter > my_old_counter) {
        ... my_func(...) ...
    }
}
```

```
{
    int my_counter = counter;  // Read global
    int (* my_func) (int);

    if (my_counter > my_old_counter) {
        ... // Consume data
        my_func = ...;
        ... // Do some more consumer work
    }
    ... // Do some other work
    if (my_counter > my_old_counter) {
        ... my_func(...) ...
    }
}
```

A read write race is bad. However, if there is only on write (apart from initialized write), the read-write race can be considered harmless. On many occasions, it is in fact okay to simply read either old or new value. On others, it is also okay to read either init or non-init, so in that sense, read-write race is kinda irrelevant and not harmful.

```
{
    int my_counter = counter;  // Read global
    int (* my_func) (int);

    if (my_counter > my_old_counter) {
        ... // Consume data
        my_func = ...;
        ... // Do some more consumer work
    }
    ... // Do some other work
    my_counter = counter;  // Reread global!
    if (my_counter > my_old_counter) {
        ... my_func(...) ...
    }
}
```

How to Miscompile Programs with Benign Data Races

```
{
    int my_counter = counter;  // Read global
    int (* my_func) (int);

    if (my_counter > my_old_counter) {
        ... // Consume data
        my_func = ...;
        ... // Do some more consumer work
    }
    ... // Do some other work
    my_counter = counter;  // Reread global!
    if (my_counter > my_old_counter) {
        ... my_func(...) ...
    }
}
```

As part of register spilling, which is intended to reuse/optimize use of register memmory, the compiler can introduce a read before the second conditional block. This can be done with the assumption that the shared memory value read does not change. And in this case too we have read-write race, where it is considered benign as the read only returns a binary result (either init or non-init). But in this program, we could have an additional behavior, one where the conditional expression for both blocks diverge.

Introducing Writes via To Avoid Repeated Writes to Shared Memory

```
f(x)
{
    ...;
    for (p = x; p != 0; p = p -> next) {
        if (p -> data < 0) count++;
    }
}
```

How to Miscompile Programs with Benign Data Races

Introducing Writes via To Avoid Repeated Writes to Shared Memory

Introducing Writes via To Avoid Repeated Writes to Shared Memory

```
f(x)
{
    ...;
    for (p = x; p != 0; p = p -> next) {
        if (p -> data < 0) count++;
    }
}
```

Write of the same existing value in memory does not affect the correctness of code. Thus, in this sense redundant write-write data race is harmless.

```
reg = count;
for (p = x; p != 0; p = p -> next) {
    if (p -> data < 0) reg++;
}
count = reg;
```

How to Miscompile Programs with Benign Data Races

```
reg = count;
for (p = x; p != 0; p = p -> next) {
    if (p -> data < 0) reg++;
}
count = reg;
```

As part of Loop code motion, and save cycles on repeated write to shared memory count, the compiler can save the initial value of count into some local register memory. This would introduce a read to count outside the loop. The modifications to count within the loop is now modifications to register, which is later written to count after the loop terminates. This program, effectively introduces a read and potentially a write to count if the data in p is never negative. Introduction of such reads/writes in a concurrent context leads to security breach (reading the old value of count) as well as violate coherence (write introduction is unsafe).

Reordering Disjoint Bit Manipulations

```
x.bits1 = 1;
switch(i) {
    case 1:
        x.bits2 = 1;
        ...;
        break;
    case 2:
        x.bits2 = 1;
        ...;
        break;
    case 3:
        ...;
        break;
    default:
        x.bits2 = 1;
        ...;
        break;
}
```

How to Miscompile Programs with Benign Data Races

Reordering Disjoint Bit Manipulations

Reordering Disjoint Bit Manipulations

```
x.bits1 = 1;
switch(i) {
    case 1:
        x.bits2 = 1;
        ...;
        break;
    case 2:
        x.bits2 = 1;
        ...;
        break;
    case 3:
        ...;
        break;
    default:
        x.bits2 = 1;
        ...;
        break;
}
```

There can be data races between two memory operations where the programmer knows for sure that the two operations use or modify different bits in a shared variable. Programmers tend to use multiple bits in the same variable in order to optimize for performance.

```
x.bits1 = 1;
tmp = x.bits2;
x.bits2 = 1;
switch(i) {
    case 1:
        ...;
        break;
    case 2:
        ...;
        break;
    case 3:
        x.bits2 = tmp;
        ...;
        break;
    default:
        ...;
        break;
}
```

How to Miscompile Programs with Benign Data Races

```
x.bits1 = 1;
tmp = x.bits2;
x.bits2 = 1;
switch(i) {
    case 1:
        ...;
        break;
    case 2:
        ...;
        break;
    case 3:
        x.bits2 = tmp;
        ...;
        break;
    default:
        ...;
        break;
}
```

The compiler can hoist the writes to $x.bits2$ outside the switch, after holding its old value in *tmp*. The value is restored if Case 3 is satisfied within the switch. This optmization effectively introduces a read and a write to $xbits.2$, if we know the value of i to always be just 3. Thus, in a concurrent context, the bit manipulations on both parts of x would interlave, potentially causing a harmful data race.

Conclusion

- Benign data races are ideally not meant to cause any programs to misbehave.
- However, compilers can optimize code in many ways, making the benign data races to actually cause unpredictable behaviors.
- Thus, data races which are harmless from a hardware perspective, is not so in the software stack.
- Correctness of such optimizations must be observed from a concurrent perspective more carefully.