Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model

Robin M., Pankaj P., Francesco Z.

Presented by
Akshay Gopalakrishnan

December 20, 2021

Introduction

Testing sequential programs to hunt for compiler bugs is well established at this time. However, testing concurrency bugs still remains a hard problem. This is majorly due to ill understood specification of the memory consistency model, and lack of information on the various transformations that preserve concurrent program semantics. This work designs a strategy to reduce the complexity of finding concurrency bugs in C11/C++11 via testing sequential code. The source code trace is compared with that of the final end code trace. Previous theoretical work establishing soundness of various local transformations in concurrent execution is utilized to achieve this.

Example of Compiler optimisation

Consider the following example C code, where *th_1* and *th_2* are run by two different threads.

```c
int g1 = 1; int g2 = 0;

void *th_1(void *p) {
  for (int l = 0; (l != 4); l++) {
    if (g1) return NULL;
    for (g2 = 0; (g2 >= 26); ++g2)
      ;
  }
}

void *th_2(void *p) {
  g2 = 42;
  printf("%d\n",g2);
}
```

The above code should print the value If $g2$ to be 42.
However, the above code when compiled using gcc 4.7.0 with -O2 enabled on x86-64 machine, the result of printf is 0.

Example of Compiler optimisation

Consider the following example C code, where *th_1* and *th_2* are run by two different threads.

```
int g1 = 1; int g2 = 0;

void *th_1(void *p) {
  for (int l = 0; (l != 4); l++) {
    if (g1) return NULL;
    for (g2 = 0; (g2 >= 26); ++g2)
      ;
  }
}

void *th_2(void *p) {
  g2 = 42;
  printf("%d\n",g2);
}
```

The above code should print the value lf $g2$ to be 42.
However, the above code when compiled using gcc 4.7.0 with -O2 enabled on x86-64 machine, the result of printf is 0.

The reason why it should be 42 is that the value of $g2$ is never changed in *th_1*. Notice that $g1$ has 1, so the conditional will always return *NULL*. It is only *th_2* that writes to $g2$.

Behind the scenes

The above code is optimized using Loop invariant code motion in *th_1*. However, the code that results due to it in x86-64 format is as below:

```
th_1:
    movl  g1(%rip), %edx    # load g1 (1) into edx
    movl  g2(%rip), %eax    # load g2 (0) into eax
    testl %edx, %edx        # if g1 != 0
    jne   .L2               #   jump to .L2
    movl  $0, g2(%rip)
    ret
.L2:
    movl  %eax, g2(%rip)    # store eax (0) into g2
    xorl  %eax, %eax        # return 0 (NULL)
    ret
```

Notice the last *movl* before return. Here the value of $g2$ is restored to be 0. This is sound if the program was sequential, however, in a concurrent setting this is clearly unsafe.

Behind the scenes

The above code is optimized using Loop invariant code motion in *th_1*. However, the code that results due to it in x86-64 format is as below:

```
th_1:
    movl  g1(%rip), %edx     # load g1 (1) into edx
    movl  g2(%rip), %eax     # load g2 (0) into eax
    testl %edx, %edx         # if g1 != 0
    jne   .L2                #   jump to .L2
    movl  $0, g2(%rip)
    ret
.L2:
    movl  %eax, g2(%rip)     # store eax (0) into g2
    xorl  %eax, %eax         # return 0 (NULL)
    ret
```

Notice the last *movl* before return. Here the value of $g2$ is restored to be 0. This is sound if the program was sequential, however, in a concurrent setting this is clearly unsafe.

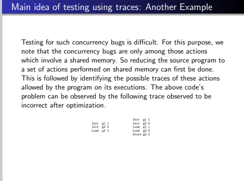
Here, the optimization tries to eliminate the inner loop. However, the process introduces some redundant writes, like that of $g2$ to be present. Thus, redundant write Introduction in this case becomes unsafe. This in turn makes such a variant of loop invariant code motion unsafe.

Main idea of testing using traces: Another Example

Testing for such concurrency bugs is difficult. For this purpose, we note that the concurrency bugs are only among those actions which involve a shared memory. So reducing the source program to a set of actions performed on shared memory can first be done. This is followed by identifying the possible traces of these actions allowed by the program on its executions. The above code's problem can be observed by the following trace observed to be incorrect after optimization.

```
Init  g1 1
Init  g2 0
Load  g1 1
```

```
Init  g1 1
Init  g2 0
Load  g1 1
Load  g2 0
Store g2 0
```



The memory model of C11 by this time was defined using traces / execution based semantics. It was more of utilizing this to do practical testing of C programs.

To summarize the memory model, each shared memory access has a memory order. Each action has an access, memory order and thread id.

$$\text{mem\_ord}, \mu \ ::= \ \text{NA} \mid \text{SC} \mid \text{ACQ} \mid \text{REL} \mid \text{R/A} \mid \text{RLX}$$
$$\phi \ ::= \ \text{R}_\mu\, l\, v \mid \text{W}_\mu\, l\, v \mid \text{Lock}\, l \mid \text{Unlock}\, l \mid \text{Fence}_\mu \mid \text{RMW}_\mu\, l\, v_1\, v_2$$
$$\text{actions} \ ::= \ aid, tid{:}\phi$$

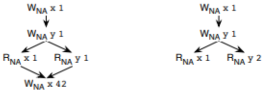
Useful terminology used for optimization

- Every concurrent program has various executions possible.
- These executions involve various shared memory *actions* that are done (even those due to multiple iterations of loop).
- A collection of these actions is defined to be an *opsem*.
- *opsem* also retains the syntactic order between actions belonging to same thread.
- All possible *opsems* for a program is called *opsemset*.

An example is as below;

```
x = 1; y = 1; if (x == y){x = 42;}
```

has, among others, the two opsems below:



Effect of an optimization

- The effect of an optimization is directly seen at the *opsem* level.
- For an *opsem*, an optimization has the effect of reordering, eliminating and introducing actions.

The following example showcases this:

```
for (i=0; i<2; i++) {           t = y; x = t;
  z = z + y + i;          ⇒     for (i=0; i<2; i++) {
  x = y;                          z = z + t + i;
}                               }
```

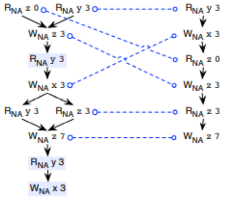


Figure 3: effect of loop invariant code motion (LIM) on an opsem

Soundness of Optimisations

- Every *opsem* represents an execution.
- The outcome of the execution can be characterized as the final values in memory and the final read values.
- These two are represented together as an *obs* or observable behavior.
- A pair $opsem, obs$ represent a candidate execution.
- An optimization is sound if the set of *obs* of the transformed program is a subset of that of the source program .

Types of Optimizations considered

- Elimination - eg: Read-after-Write, Write-after-Write, etc.
- Reordering - eg: Read-Write, Read-Read, etc.
- Introduction - Read / Write.

Using above results for testing

Most bugs found in *gcc* were that where writes were introduced in the program. Some bugs were based on illegal reordering across atomic(SC) actions. The figure below showcases such a program

```
atomic_uint a; int32_t g1, g2;
int main (int, char *[]) {
  a.load () & a.load ();
  g2 = g1 != 0;
}
```

Trace analysis showcasing illegal reordering

gcc test version disallowed any reorderings across atomic actions. However, on trace analysis it was observed that the load of $g1$ was moved above the two atomic loads to a.

ALoad	a	0	4	Load	g1	0	4
ALoad	a	0	4	ALoad	a	0	4
Load	g1	0	4	ALoad	a	0	4
Store	g2	0	4	Store	g2	0	4

Although in this case such a reordering does not result in any new outcome, introducing other concurrent threads running might result in some new behaviors.

Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model

└─Trace analysis showcasing illegal reordering

Trace analysis showcasing illegal reordering

gcc test version disallowed any reorderings across atomic actions. However, on trace analysis it was observed that the load of $g1$ was moved above the two atomic loads to a.

Although in this case such a reordering does not result in any new outcome, introducing other concurrent threads running might result in some new behaviors.

The most simple example is that of message passing, where in this case a is considered as an atomic flag variable and $g1$ as the message containing the data. Reordering these two would result in a case where $g1$ has some data before the flag was set.

Yet another interesting transformation

The following program gives us an interesting view of "bug" caused by transformation

```
atomic_int a; uint16_t g;
void func_1 () {
  for (; a.load () <= 0; a.store (a.load () + 1))
    for (; g; g--);
}
```

On trace analysis, we note that a load to g has been replaced by a store to g.

ALoad	a	0	4		ALoad	a	0	4
Load	g	0	2	?	Store	g	0	2
ALoad	a	0	4		ALoad	a	0	4
AStore	a	0	4		AStore	a	0	4
ALoad	a	1	4		ALoad	a	1	4

Is it a bug?

The above situation was caused by the following transformation

```
for (; g; g--);        →    g = 0;
```

The source program itself would not change the value of g, leaving it to be the deafult value 0. In that sense, the transformation above does make sense, as a load is replaced by a redundant write to 0. However, whether such a transformation should be allowed or not is a debate. Note that the introduced write could lead to cache contention where there shouldn't be any, leading to performance dipping down.

Is it a bug?

The above situation was caused by the following transformation

`for (; g; g--);` $\longrightarrow$ `g = 0;`

The source program itself would not change the value of g, leaving it to be the deafult value 0. In that sense, the transformation above does make sense, as a load is replaced by a redundant write to 0. However, whether such a transformation should be allowed or not is a debate. Note that the introduced write could lead to cache contention where there shouldn't be any, leading to performance dipping down.

Its interesting that a transformation can in essence involve removing and inserting actions as an atomic step. Definitely, the class of transformations one can do is infinte. :o

Thank you

Questions?