Safe optimisations for high-level concurrent programming languages

Francesco Zappa Nardelli
francesco.zappa_nardelli@inria.fr

Moscova Project-team, INRIA Paris-Rocquencourt, France

Compiler optimisations preserve the semantics of sequential code, but can introduce subtle bugs in concurrent code. Consider the unoptimised program on the left, where two threads share two memory locations `x` and `y` which initially hold `0`:

Thread 0	Thread 1
x = 1;	if (x ==1) {
if (y == 1)	x = 0;
print x;	y = 1; }

$\longrightarrow$

Thread 0	Thread 1
x = 1;	if (x ==1) {
if (y == 1)	x = 0;
print 1 ;	y = 1; }

This program cannot print `1`. However if the compiler performs constant propagation, then the original code is rewritten into the code on the right, which always prints `1`. Examples can be more complicated, involving several optimisations at a time. For instance in the Java program below, two threads share the memory locations x and y, which initially hold 0:

Thread 0	Thread 1
r1 = x	r2 = y
y = r1	x = (r2==1)?y:1
	print r2

$\longrightarrow$

Thread 0	Thread 1
r1 = x	x = 1
y = r1	r2=y
	print r2

The unoptimised code (on the left) cannot print 1. However Sun Hotspot and Gcj rewrite the code into the optimized version (on the right), which always print 1. (Can you guess the sequence of optimisations applied?)

The memory model of a programming language is the contract between the programmer and the compiler that precisely describes which reorderings the programmer must take into account when reasoning about the correctness of a program. The new revision of the C++ standard, called C++11, and the forthcoming revision of C define a memory model that enables efficient low-level programming via an escape mechanism called low-level atomics. A mathematical description of the C/C++ memory model can be found in [1] and [2].

The safety of compiler optimisations has been studied for the SC and the DRF memory models in [3]. However the interaction between low-level atomics and compiler optimisations is unclear and subtle: in this stage we propose to extend the study of [3] to cover low-level atomics. At the time of writing, it is easy to write code involving low-level atomics which is miscompiled by `gcc` (including the development branch), while the `llvm` optimiser seems over conservative. In parallel to the theoretical investigation we will analyse the behavior of mainstream compilers and, whenever possible, suggest lines for development of report bugs: a successful internship might thus have a big impact on the practice of concurrent programming.

References

[1] Foundations of the C++ Concurrency Memory Model, by Boehm and Adve
`http://www.hpl.hp.com/techreports/2008/HPL-2008-56.html`.

[2] Mathematizing C++ Concurrency, by Batty, Owens, Sarkar, Sewell, and Weber
`http://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf`.

[3] Safe optimisations for Shared-Memory Concurrent Programs, by Sevcik
`http://www.cl.cam.ac.uk/~js861/papers/transsafety.pdf`