Do C/C++ compilers correctly compile concurrent programs?

Contribution 1
Characterisation of optimisations sound in an arbitrary concurrent context

On the code above, the optimised trace exhibits two extra accesses, including one introduced memory write. Introduction of observable memory writes is provably unsound in the C11 memory model: the optimised trace cannot be justified from the reference trace by a sequence of correct transformations. We detect a concurrency compiler bug.

In general, it is possible to characterise which optimisations are correct in a concurrent setting by observing how they eliminate, reorder, or introduce, memory accesses in the traces of the sequential code with respect to a reference trace. As an example, the instance of loop-invariant code motion on the right eliminates redundant memory accesses and reorder independent accesses: it is provably correct.

Contribution 2
Design and implementation of a random-testing tool that checks if a compiler applies only optimisations sound in arbitrary contexts

The cmmtest tool that we designed and implemented puts this strategy at work:
- it generates a well-defined sequential C program using a modified version of Csmith
- it invokes the optimising compiler under test
- it traces the global (potentially shared) memory accesses of the optimised code
- it compares the optimised trace against a reference trace for the source program, building on our theory of sound optimisations for the C11/C++11 memory model
- if the traces cannot be matched, it performs test-case reduction

Several concurrency compiler bugs and unexpected behaviours detected by cmmtest in the latest release of GCC

http://www.di.ens.fr/~zappa/projects/cmmtest/