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

Robin Morisset
ENS & INRIA

Pankaj Pawan
IIT Kanpur & INRIA

Francesco Zappa Nardelli
INRIA

Abstract

Compilers sometimes generate correct sequential code but break the concurrency memory model of the programming language: these subtle compiler bugs are observable only when the miscompiled functions interact with concurrent contexts, making them particularly hard to detect. In this work we design a strategy to reduce the hard problem of hunting concurrency compiler bugs to differential testing of sequential code and build a tool that puts this strategy to work. Our first contribution is a theory of sound optimisations in the C11/C++11 memory model, covering most of the optimisations we have observed in real compilers and validating the claim that common compiler optimisations are sound in the C11/C++11 memory model. Our second contribution is to show how, building on this theory, concurrency compiler bugs can be identified by comparing the memory trace of compiled code against a reference memory trace for the source code. Our tool identified several mistaken write introductions and other unexpected behaviours in the latest release of the `gcc` compiler.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors – compilers; D2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

Keywords C11/C++11 memory model; compiler testing

1. Random testing for concurrency compiler bugs

The C and C++ languages were originally designed without concurrency support: threads were available via external libraries, yielding unexpected behaviours and misunderstandings between programmers and compiler writers.[1] The recent revision of the C and C++ standards [6] does provide a precise semantics for threads (formalised in [4]): well-synchronised programs must exhibit only sequentially consistent behaviours, racy programs can have any behaviour, and an escape mechanism with a complex semantics, called *low-level atomics*, enables programmers to write high-performance but portable concurrent code. The resulting model is intricate and the interactions with compiler optimisations are not entirely understood. Today's C and C++ compilers, whose optimisers were initially developed in absence of any well-defined memory model, are being extended to support the new concurrency standard. This is a hard task. Past research showed that all production-quality C compilers used to generate incorrect code for accessing volatile variables [10]: the memory model defined by the volatile modifier is trivial compared to the new C11/C++11 model. Correctly supporting the new memory model in today's compilers will be an error-prone enterprise, requiring significant development effort. It is thus vital to investigate techniques to build assurance in widely used implementations of C and C++.

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

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);
}

int main() {
  pthread_t th1, th2;
  pthread_create(&th1, NULL, th_1, NULL);
  pthread_create(&th2, NULL, th_2, NULL);
  (void)pthread_join (th1, NULL);
  (void)pthread_join (th2, NULL);
}
```

Figure 1: `foo.c`, a concurrent program miscompiled by `gcc` 4.7.0.

Consider for instance the C program in Figure 1. Can we guess its output? This program spawns two threads executing the functions `th_1` and `th_2` and waits for them to terminate. The two threads share two global memory locations, `g1` and `g2`, but a careful reading of the code reveals that the inner loop of `th_1` is never executed and `g2` is only accessed by the second thread, while `g1` is only accessed by the first thread. According to the C11/C++11 standards this program should always print `42` on the standard output: the two threads do not access the same variable in conflicting

[1] as an example, this recent discussion illustrates the mismatch and tensions between what Linux kernel developers expect from compilers and what `gcc` does: `http://gcc.gnu.org/ml/gcc/2012-02/msg00005.html`.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

ways, so the program is *data-race free*, and its semantics is defined as the interleavings of the actions of the two threads.

If we compile the above code with the version 4.7.0 of `gcc` on an x86_64 machine running Linux, and we enable some optimisations with the `-O2` flag (as in `g++ -std=c++11 -lpthread -O2 -S foo.c`) then, sometimes, the compiled code prints `0` to the standard output. This unexpected outcome is caused by a subtle bug in `gcc`'s implementation of the loop invariant code motion (LIM) optimisation. If we inspect the generated assembly we discover that `gcc` saves and restores the content of `g2`, causing `g2` to be overwritten with `0`:

```
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
```

This optimisation is sound in a sequential world because the extra store always rewrites the initial value of `g2` and the final state is unchanged. However, as we have seen, this optimisation is unsound in the concurrent context provided by `th_2`, and the C11/C++11 standards forbid it.

How to search for concurrency compiler bugs? We have a *compiler bug* whenever the code generated by a compiler exhibits a behaviour not allowed by the semantics of the source program. Differential random testing has proved successful at hunting compiler bugs. The idea is simple: a test harness generates random, *well-defined*, source programs, compiles them using several compilers, runs the executables, and compares the outputs. The state of the art is represented by the Csmith tool by Yang, Chen, Eide and Regehr [24], which over the last four years has discovered several hundred bugs in widely used compilers, including `gcc` and `clang`. However this work cannot find *concurrency compiler bugs* like the one we described above: despite being miscompiled, the code of `th_1` still has correct behaviour in a sequential setting.

A naive approach to extend differential random testing to concurrency bugs would be to generate *concurrent* random programs, compile them with different compilers, record *all* the possible outcomes of each program, and compare these sets. This works well in some settings, such as the generation and comparison of litmus tests to test hardware memory models; see for instance the work by Alglave et al. [3]. However this approach is unlikely to scale to the complexity of hunting C11/C++11 compiler bugs. Concurrent programs are inherently *non-deterministic* and optimisers can compile away non-determism. In an extreme case, two executables might have disjoint sets of observable behaviours, and yet both be correct with respect to a source C11 or C++11 program. To establish correctness, comparing the final checksum of the different binaries is not enough: we must ensure that all the behaviours of a compiled executable are allowed by the semantics of the source program. The Csmith experience suggests that large program sizes (∼80KB) are needed to maximise the chance of hitting corner cases of the optimisers; at the same time they must exhibit subtle interaction patterns (often unexpected, as in the example above) while being well-defined (in particular data-race free). Capturing the set of all the behaviours of such large concurrent programs is tricky as it can depend on rare interactions between threads; computing all the behaviours allowed by the C11/C++11 semantics is even harder.

Despite this, we show that differential random testing can be used successfully for hunting concurrency compiler bugs. First, C and C++ compilers must support separate compilation and the concurrency model allows any function to be spawned as an independent thread. As a consequence compilers must always assume that the *sequential* code they are optimising can be run in an *arbitrary concurrent context*, subject only to the constraint that the whole program is well-defined (race-free on non-atomic accesses, etc.), and can only apply optimisations that are sound with respect to the concurrency model. Second, 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. Combined, these two remarks imply that testing the correctness of compilation of concurrent code can be reduced to validating the traces generated by running optimised sequential code against a reference (unoptimised) trace for the same code.

We illustrate this idea with program `foo.c` from Figure 1. Traces only report accesses to global (potentially shared) memory locations: optimisations affecting only the thread-local state cannot induce concurrency compiler bugs. On the left below, the reference trace for `th_1` initialises `g1` and `g2` and loads the value `1` from `g1`:

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

On the right above, the trace of the `gcc -O2` generated code performs an extra load and store to `g2` and, since arbitrary store introduction is provably incorrect in the C11/C++11 concurrency model we can detect that a miscompilation happened. Figure 2 shows another example, of a randomly generated C function together with its reference trace and an optimised trace. In this case it is possible to match the reference trace (on the left) against the optimised trace (on the right) by a series of sound eliminations and reordering of actions.

Contributions This approach to compiler testing crucially relies on a theory of sound optimisations over executions of C11/C++11 programs. Building on the work by Ševčík [17, 19] for an idealised DRF model, our **first contribution** is the study and correctness proof of several criteria for sound optimisations in the C11/C++11 model, covering all the optimisations we observed in testing real compilers (with the exception of irrelevant read introductions and merging of accesses), presented in Section 3. The **second contribution** of this work is a strategy to perform differential testing of compilers against a memory model, reducing the hard problem of hunting concurrency compiler bugs to matching memory traces realised by sequential code. The `cmmtest` tool, which we designed and developed, puts the testing strategy to work building on our theory of C11/C++11 optimisations. These are presented in Section 4. Our **third contribution**, in Section 5, is a report on some concurrency compiler bugs and other unexpected behaviours caught by preliminary testing of `gcc`. The `gcc` developers promptly fixed all the reported bugs and are integrating our soundness criteria with the `gcc` optimiser. We begin in Section 2 by introducing our theory of semantic optimisations and recalling the C11/C++11 memory model and conclude with a discussion of related work. Complete proofs and an evaluation release of `cmmtest` are on the web [1].

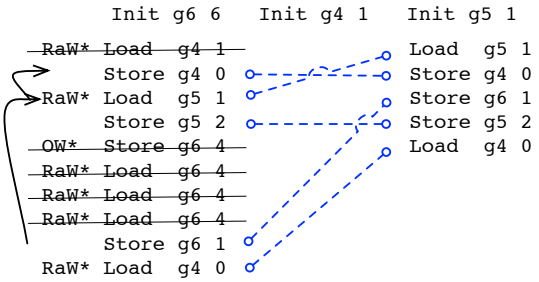
2. Program Transformations and the C11/C++11 Memory Model

Compiler optimisations are usually described as program transformations over an intermediate representation of the source code; a typical compiler performs literally hundreds of optimisation passes. Although it is possible to prove the correctness of individual transformations, this presentation does not lend itself to a thorough characterisation of what program transformations are valid.

```
const unsigned int g3 = 0UL;
long long g4 = 0x1;
int g6 = 6L;
unsigned int g5 = 1UL;

void f(){
  int *l8 = &g6;
  int l36 = 0x5E9D070FL;
  unsigned int l107 = 0xAA37C3ACL;
  g4 &= g3;
  g5++;
  int *l102 = &l36;
  for (g6 = 4; g6 < (-3); g6 += 1);
  l102 = &g6;
  *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

This randomly generated function generates the following traces if compiled with `gcc -O0` and `gcc -O2`.



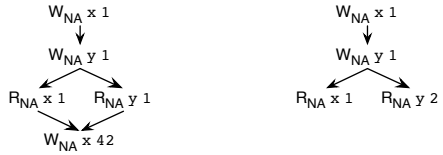
All the events in the optimised trace can be matched with events in the reference trace by performing valid eliminations and reorderings of events. Eliminated events are ~~struck-off~~ while the reorderings are represented by the arrows.

Figure 2: successful matching of reference and optimised traces

In a source program each thread consists of a sequence of *instructions*. During the execution of a program, any given static instruction may be iterated multiple times (for example due to looping) and display many behaviours. For example reads may read from different writes and conditional branches may be taken or not. We refer to each such instance of the dynamic execution of an instruction as an *instruction instance*. More precisely, each instruction instance performs zero, one, or more shared memory accesses, which we call *actions*. We account for all the differing ways a given program can execute by identifying a source program with the set of sets of all the actions (annotated with additional information as described below) it can perform when it executes in an arbitrary context. We call the set of the actions of a particular execution an *opsem* and the set of all the opsems of a program an *opsemset*. For instance, the snippet of code below:

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

has, among others, the two opsems below:



The opsem on the left corresponds to an execution where the reads read the last values written by the thread itself; the opsem on the right accounts for an arbitrary context that concurrently updated the value of y to 2. Nodes represent actions and black arrows show

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

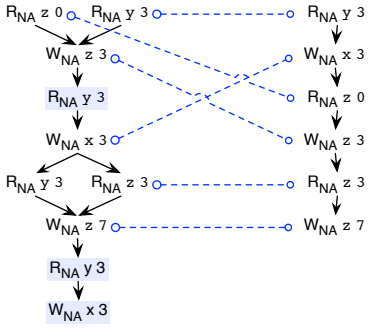


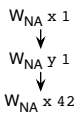
Figure 3: effect of loop invariant code motion (LIM) on an opsem

the *sequenced-before relation*, which orders actions (by the same thread) according to their program order. The sequenced-before relation is not total because the order of evaluation of the arguments of functions, or of the operands of most operators, is underspecified in C and C++. The memory accesses are non-atomic, as denoted by the NA label.

We can then characterise the effect of arbitrary optimisations of source code directly on opsemsets. On a given opsem, the effect of any transformation of the source code is to eliminate, reorder, or introduce actions. If the optimiser performs constant propagation, the previous code is rewritten as:

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

and its unique opsem is depicted here on the right. This opsem can be obtained from the one above on the left by eliminating the two read actions. A more complex example is shown in Figure 3, where the loop on the left is optimised by LIM. The figure shows opsems for the initial state z=0, y=3 assuming that the code is not run in parallel with an interfering context. Here the effect of the LIM optimisation is not only to remove some actions (shaded) but also to reorder the write to x.



An opsem captures a possible execution of the program, so by applying a transformation to an opsem we are actually optimising one particular execution. Lifting pointwise this definition of *semantic transformations* to opsemsets enables optimising all the execution paths of a program, one at a time, thus abstracting from actual source program transformation.

In the rest of this section we give a high-level overview of the C11/C++11 memory model as formalised by Batty et al. [4], defining opsems, opsemsets and executions, while in the next section we formalise program transformations and prove their correctness.

2.1 Background: overview of the C11/C++11 memory model

Let l range over locations (which are partitioned into non-atomic and atomic), v over values and $tid \in \{1..n\}$ over thread identifiers. We consider the following actions:

$\mathsf{mem\_ord}, \mu \;::=\; \mathsf{NA} \mid \mathsf{SC} \mid \mathsf{ACQ} \mid \mathsf{REL} \mid \mathsf{R/A} \mid \mathsf{RLX}$

$\phi \;::=\; \mathsf{R}_\mu\, l\, v \mid \mathsf{W}_\mu\, l\, v \mid \mathsf{Lock}\, l \mid \mathsf{Unlock}\, l \mid \mathsf{Fence}_\mu \mid \mathsf{RMW}_\mu\, l\, v_1\, v_2$

$\mathsf{actions} \;::=\; aid, tid{:}\phi$

The possible actions are loads from and stores to memory, lock and unlock of a mutex, fences, and read-modify-writes of memory locations. Each action is identified by an action identifier aid (ranged

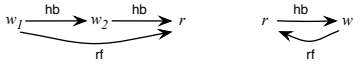
over by $r, w, \ldots$) and specifies its thread identifier tid, the location l it affects, the value read or written v (when applicable), and the memory-order μ (when applicable).[2] In the drawings we omit the action and thread identifiers.

The thread-local semantics identifies a program with a set of *opsems* (ranged over by O): triples $(A, \mathsf{sb}, \mathsf{asw})$ where $A \in \mathcal{P}(\mathsf{actions})$ and $\mathsf{sb}, \mathsf{asw} \subseteq A \times A$ are the *sequenced-before* and *additional-synchronised-with* relations. Sequenced-before (denoted sb) was introduced above; it is transitive and irreflexive and only relates actions by the same thread; *additional-synchronised-with* (denoted asw) contains additional edges from thread creation and thread join, and in particular orders initial writes to memory locations before all other actions in the execution.

The thread-local semantics assumes that all threads run in an arbitrary concurrent context which can update the shared memory at will. This is modelled by reads taking unconstrained values. We say that a set of opsems S is *receptive* if, for every opsem O, for every read action $r, t{:}\mathsf{R}_\mu\, l\, v$ in the opsem O, for all values v' there is an opsem O' in S which only differs from O because the read r returns v' rather than v, and for the actions that are sequenced-after r. Intuitively a set of opsems is receptive if it defines a behaviour for each possible value returned by each read.

We call the set of all the opsems of a program an *opsemset*, ranged over by P. The thread local semantics ensures that opsemsets are receptive. Opsems and opsemsets are subject to several well-formedness conditions, e.g. atomic accesses must access only atomic locations, which we omit here and can be found in [4]. We additionally require opsemsets to be sb-prefix closed, assuming that a program can halt at any time. Formally, we say that an opsem O' is an sb-prefix of an opsem O if there is an injection of the actions of O' into the actions of O that behaves as the identity on actions, preserves sb and asw, and, for each action $x \in O'$, whenever $x \in O$ and $y <_{\mathsf{sb}} x$, it holds that $y \in O'$.

Executions The denotation of each thread in an opsem is agnostic to the behaviour of the other threads of the program: the thread-local semantics takes into account only the structure of every thread's statements, not the semantics of memory operations. In particular, the values of reads are chosen arbitrarily, without regard for writes that have taken place. The memory model filters inconsistent opsems by constructing additional relations and checking the resulting candidate executions against the axioms of the model. For this an *execution witness* (denoted by W) for an opsem specifies an interrelationship between memory actions of different threads via three relations: *reads-from* (rf) relates a write to all the reads that read from it; the *sequential consistent order* (sc) is a total order over all SC actions; and *modification order* (mo) – or *coherence* – is the union of a per-location total order over writes to each atomic location. From these, the model infers the relations *synchronises-with* (denoted sw), which defines synchronisation and is described in detail below, and *happens-before* (denoted hb), showing the precedence of actions in the execution. Key constraints on executions depend on the happens-before relation, in particular a *non-atomic read* must not read any write related to it in hb other than its immediate predecessor. This property is called *consistent non-atomic read values*, and for writes w_1 and w_2 and a read r accessing the same non-atomic location, the following shapes are forbidden:



[2] we omit *consume* atomics: their semantics is intricate (e.g. happens-before is not transitive in the full model) and at the time of writing no major compiler profits from the their weaker semantics, treating consume as acquire. By general theorems [5], our results remain valid in the full model.

For atomic accesses the situation is more complex, and atomic reads can read from hb-unrelated writes.

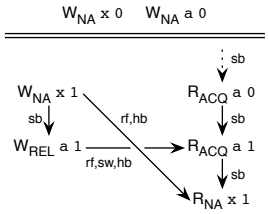
Happens-before is a partial relation defined as the transitive closure of sb, asw and sw: $\mathsf{hb} = (\mathsf{sb} \cup \mathsf{asw} \cup \mathsf{sw})^+$.

We refer to a pair of an opsem and witness (O, W) as a *candidate execution*. A pair (O, W) that satisfies a list of *consistency predicates* on these relations (including consistent non-atomic read values) is called a *pre-execution*. The model finally checks if none of the pre-executions contain an *undefined behaviour*. Undefined behaviours arise from *unsequenced races* (two conflicting accesses performed by the same thread not related by sb), *indeterminate reads* (an access that does not read a written value), or *data races* (two conflicting accesses not related by hb), where two accesses are *conflicting* if they are to the same address and at least one is a non-atomic write. Programs that exhibit an undefined behaviour (e.g. a data-race) in one of their pre-executions are undefined; programs that do not exhibit any undefined behaviour are called *well-defined*, and their semantics is given by the set of their pre-executions.

Synchronisation Synchronisation between threads is captured by the sw relation. The language provides two mechanisms for establishing synchronisation between threads and enabling race-free concurrent programming: *mutex locks* and *low-level atomics*. The semantics of mutexes is intuitive: the sc relation, part of the witness, imposes a total order over all lock and unlock accesses, and a synchronised-with (sw) edge is added between every unlock action, and every lock of the same mutex that follows it in sc-order. Low-level atomics are specific to C/C++ and designed as an escape hatch to implement high-performance racy algorithms. Atomic operations do not race with each other, by definition, and their semantics is specified by a memory-order attribute. *Sequentially consistent* atomics have the strongest semantics: all SC reads and writes are part of the total order sc (acyclic with hb). An SC read can only read from the closest sc-preceding write to the same location. A subtlety of the model is that, although sc and hb must be compatible, sc is not included in hb. Sequentially consistent atomics, as well as release-acquire atomics, generate synchronisation edges, which are included in hb. This is best explained for a classic message-passing idiom. Imagine that one thread writes some (perhaps multi-word) data x and then an atomic flag a, while another waits to see that flag write before reading the data:

```
                    x=0; atomic a=0
x = 1;              || while (0 == a.load(acq)) {};
a.store(1, rel);    || int r = x;
```

The synchronisation between the release and acquire ensures that the sender's write of x will be seen by the receiver. Below we depict a typical opsem for this program; we represent the asw relation with the double horizontal line: the init actions are asw-before all other events. The witness has an rf arrow between the write-release and read-acquire on a to justify that the read returns 1. A read between a write-release and a read-acquire generates an sw edge. Since hb includes sb and sw, the write of 1 to x is the last write in the hb order before the read of the second thread, which is then forced to return 1. *Relaxed atomics* instead do not generate synchronisation edges sw; they are only forbidden to read from the future, i.e. from writes later in hb.



Observable behaviour The C11/C++11 memory model does not explicitly define the *observable behaviour* of an execution. We extend the model with a special atomic location called world, and model the observable side-effects of the program (e.g., writes on

stdout) by relaxed writes to that location. The relaxed attribute guarantees that these accesses are totally ordered with each other, as captured by the mo relation. As a result, the *observable behaviour* of a pre-execution is the restriction of the mo relation to the distinguished `world` location. If none of its pre-executions exhibit an undefined behaviour, then the observable behaviour of a program is the set of all observable behaviours of its executions.

3. Sound Optimisations in the C Memory Model

C and C++ are shared-memory-concurrency languages with explicit thread creation and implicit sharing: any location might be read or written by any thread for which it is reachable from variables in scope. It is the programmer's responsibility to ensure that such accesses are race-free. This implies that compilers can perform optimisations that are not sound for racy programs and common thread-local optimisations can still be done without the compiler needing to determine which accesses might be shared.

Ševčík showed that a large class of elimination and reordering transformations are correct (that is, do not introduce any new behaviour when the optimised code is put in an arbitrary data-race free context) in an idealised DRF model [17, 19]. In this section we adapt and extend his results to optimise non-atomic accesses in the C11/C++11 memory model. As we have discussed, we classify program transformations as *eliminations*, *reorderings*, and *introductions* over opsemsets.

3.1 Eliminations of actions

The semantic elimination transformation is general enough to cover optimisations that eliminate memory accesses based on data-flow analysis, such as common subexpression elimination, induction variable elimination, and global value numbering, including the cases when these are combined with loop unrolling.

We define semantic elimination and discuss informally its soundness criterion; we then state the soundness theorems and briefly describe the proof structure.

Definition 3.1. An action is a *release* if it is an unlock action, an atomic write with memory-order REL or SC, a fence or read-modify-write with memory-order REL, R/A or SC.

Semantically, release actions can be seen as potential sources of sw edges. The intuition is that they "release" permissions to access shared memory to other threads.

Definition 3.2. An action is an *acquire* if it is a lock action, or an atomic read with memory-order ACQ or SC, or a fence or read-modify-write with memory order ACQ, R/A or SC.

Acquire actions can be seen as potential targets of sw edges; the intuition is that they "acquire" permissions to access shared memory from other threads.

To simplify the presentation we omit dynamic thread creation. This is easily taken into account by stating that spawning a thread has release semantics, while the first accesses in sb-order of the spawned function have acquire semantics. Reciprocally, the last actions of a thread have release semantics and a thread-join has acquire semantics.

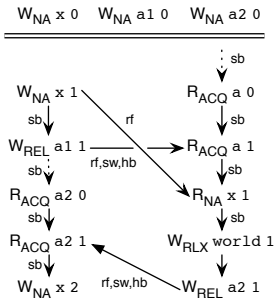
A key concept is that of a *same-thread release-acquire pair*:

Definition 3.3. A *same-thread release-acquire pair* (shortened *st-release-acquire pair*) is a pair of actions (r, a) such that r is a release, a is an acquire, and $r <_{\mathsf{sb}} a$.

Note that these may be to different locations and never synchronise together. To understand the role they play in optimisation soundness, consider the code on the left, running in the concurrent context on the right:

```
                                    x=0; atomic a1,a2=0
x = 1;                          ||  while(0==a1.load(acq)) {};
a1.store(1,rel);                ||  printf("%i",x);
while(0==a2.load(acq)) {};      ||  a2.store(1,rel);
x = 2;                          ||
```

All executions have similar opsems and witnesses, depicted below (we omitted rf arrows from initialisation writes). No consistent execution has a race and the only observable behaviour is printing 1.



Eliminating the first store to x (which might appear redundant as x is later overwritten by the same thread) would preserve DRF but would introduce a new behaviour where 0 is printed. However, if either the release or the acquire were not in between the two stores, then this context would be racy (respectively between the load performed by the print and the first store, or between the load and the second store) and it would be correct to optimise away the first write. More generally, the proof of the Theorem 3.1 below clearly shows that the presence of an intervening same-thread release-acquire is a necessary condition to allow a discriminating context to interact with a thread without introducing data races.

Definition 3.4. A read action $a, t{:}\mathsf{R}_{\mathsf{NA}}\, l\, v$ is *eliminable* in an opsem O of the opsemset P if one of the following applies:

- *Read after Read* (RaR): there exists another action $r, t{:}\mathsf{R}_{\mathsf{NA}}\, l\, v$ such that $r <_{\mathsf{sb}} a$, and there does not exist a memory access to location l or a st-release-acquire pair sb-between r and a;
- *Read after Write* (RaW): there exists an action $w, t{:}\mathsf{W}_{\mathsf{NA}}\, l\, v$ such that $w <_{\mathsf{sb}} a$, and there does not exist a memory access to location l or a st-release-acquire pair sb-between w and a;
- *Irrelevant Read* (IR): for all values v' there exists an opsem $O' \in P$ and a bijection f between actions in O and actions in O', such that $f(a) = a', t{:}\mathsf{R}_{\mathsf{NA}}\, l\, v'$, for all actions $u \in O$ different from a, $f(u) = u$, and f preserves sb and asw.

A write action $a, t{:}\mathsf{W}_{\mathsf{NA}}\, l\, v$ is *eliminable* in an opsem O of the opsemset P if one of the following applies:

- *Write after Read* (WaR): there exists an action $r, t{:}\mathsf{R}_{l}\, v$ such that $r <_{\mathsf{sb}} a$, and there does not exist a memory access to location l or a st-release-acquire pair sb-between r and a;
- *Overwritten Write* (OW): there exists another action $w, t{:}\mathsf{W}_{\mathsf{NA}}\, l\, v'$ such that $a <_{\mathsf{sb}} w$, and there does not exist a memory access to location l or a st-release-acquire pair sb-between a and w;
- *Write after Write* (WaW): there exists another action $w, t{:}\mathsf{W}_{\mathsf{NA}}\, l\, v$ such that $w <_{\mathsf{sb}} a$, and there does not exist a memory access to location l or a st-release-acquire pair sb-between w and a.

Note that the OW rule is the only rule where the value can differ between the action eliminated and the action that justifies the elimination. The IR rule can be rephrased as "a read in an execution is irrelevant if the program admits other executions (one for each value) that only differ for the value returned by the irrelevant read".

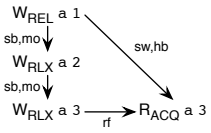
Definition 3.5. An opsem O' is an *elimination* of an opsem O if there exists a injection $f : O' \to O$ that preserves actions, sb, and asw, and such that the set $O \backslash f(O')$ contains exactly one eliminable action. The function f is called an *unelimination*.

To simplify the proof of Theorem 3.1 the definition above allows only one elimination at a time (this avoids a critical pair between the rules OW and WaW whenever we have two writes of

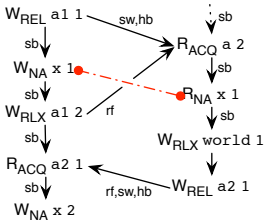
the same value to the same location), but, as the theorem shows, this definition can be iterated to eliminate several actions from one opsem while retaining soundness. The definition of eliminations lifts pointwise to opsemsets:

Definition 3.6. An opsemset P' is an *elimination* of an opsemset P if for all opsem $O' \in P'$ there exists an opsem $O \in P$ such that O' is an elimination of O.

In the previous section we did not describe all the intricacies of the C11/C++11 model but our theory takes all of them into account. For example, a release fence followed by an atomic write behaves as if the write had the REL attribute, except that the sw edge starts from the fence action. Another example is given by *release sequences*: if an atomic write with attribute `release` is followed immediately in mo-order by one or more `relaxed` writes in the same thread (to the same location), and an atomic load with attribute `acquire` reads-from one of these relaxed stores, an sw edge is created between the write release and the load acquire, analogously to the case where the acquire reads directly from the first write. These subtleties must be taken into account in the elimination correctness proof but do not invalidate the intervening same-thread release-acquire pair criterion. This follows from a property of the C11/C++11 design that makes every sw edge relate a release action to an acquire action. For instance, in the program below it is safe to remove the first write to `x` as OW, because all discriminating contexts will be necessarily racy:




```
a1.store(1,rel);
x = 1;
a1.store(2,rlx);
while(0==a2.load(acq)) {};
x = 2;
```



We establish that our semantic transformations have the following properties: any execution of the transformed opsemset has the same observable behaviour of some execution of the original opsemset, and the transformation preserves data-race freedom. As C11 and C++11 do not provide any guarantee for racy programs we cannot prove any result about out-of-thin-air value introduction.

Theorem 3.1. *Let the opsemset P' be an elimination of the opsemset P. If P is well-defined, then so is P', and any execution of P' has the same observable behaviour of some execution of P.*

We sketch the structure of the proof; details can be found online [1]. Let the opsemset P' be an elimination of the opsemset P, and let (O', W') be an execution of P' (that is, a pair of an opsem and a witness). Since P' is an elimination of P, there is at least one opsem $O \in P$ such that O' is an elimination of O, and an unelimination function f that injects the events of the optimised opsem into the events of the unoptimised opsem. We build the mo and sc relations of the witness of O by lifting the mo' and sc' relations of the witness W': this is always possible because our program transformations do not alter any atomic access. Analogously it is possible to build the rf relation on atomic accesses by lifting the rf' one. To complete the construction of the witness we lift the rf' relation on non-atomic events as well, and complete the rf relation following the case analysis below on the eliminated events in O:

- RaR: if i is a read action eliminated with the RaR rule because of a preceding read action r, and $w <_{\mathsf{rf}} r$, then add $w <_{\mathsf{rf}} i$;
- RaW: if i is a read action eliminated with the RaW rule because of a preceding write action w, then add $w <_{\mathsf{rf}} i$;
- IR: if i is an irrelevant read, and there is a write event to the same location that happens before it, then let w be a write event to the same location maximal with respect to hb and add $w <_{\mathsf{rf}} i$;
- OW: rf is unchanged;
- WaW: if i is a write event eliminated by the WaW rule because of the preceding write event w, then for all actions r such that $w <_{\mathsf{rf}'} r$ and $i <_{\mathsf{hb}} r$, replace $w <_{\mathsf{rf}} r$ by $i <_{\mathsf{rf}} r$;
- WaR: if i is a read event eliminated by the WaR rule, then every read of the same value at the same location, that happens-after i and that either read from a write $w <_{\mathsf{hb}} i$ or does not read from any write, now reads from i.

This completes the construction of the witness W and in turn of the candidate execution (O, W) of P. We must now prove that (O, W) is consistent, in particular that it satisfies *consistent non-atomic read values*, for which the construction has been tailored. This proceeds by a long case disjunction that relies on the following constructions:

- the absence of a release-acquire pair between two accesses a and b in the same thread guarantees the absence of an access c in another thread with $a <_{\mathsf{hb}} c <_{\mathsf{hb}} b$.
- in some cases the candidate execution (O, W) turns out to have conflicting accesses a and b that are not ordered by hb. We use the fact that opsemsets are receptive and closed under sb-prefix to build another candidate pre-execution of P where a and b are still hb-unordered, but for which we can prove it is a pre-execution (not necessarily with the same observable behaviour). From this we deduce that P is not data-race free and ignore these cases.

By construction the pre-execution (O, W) has the same observable behaviour as (O', W'); we conclude by showing that (O', W') can not have undefined behaviours that (O, W) does not have.

3.2 Reorderings of actions

Most compiler optimisations do not limit themselves to eliminating memory accesses, but also reorder some of them. In this category fall all the code motion optimisations. Of course not all accesses are reorderable without introducing new behaviours. In this section we state and prove the correctness of the class of reorderings we have observed being performed by `gcc` and `clang`, ignoring more complex reordering schemes. In particular we omit reorderings across synchronisation actions: neither `gcc` or `clang` perform them and a more complex statement taking into account partial reorderings on prefix-closures of an opsemset would be needed.

Definition 3.7. Two actions a and b are *reorderable* if they access different memory locations and neither is a synchronisation action (that is a lock, unlock, atomic access, rmw, or fence action).

Definition 3.8. An opsem O' is a *reordering* of an opsem O if: (i) the set of actions in O and O' are equal; (ii) O and O' define the same asw; (iii) for all actions $a, b \in O$, if $a <_{\mathsf{sb}} b$ then either $a <_{\mathsf{sb}'} b$ or $b <_{\mathsf{sb}'} a$ and a is reorderable with b.

Like for eliminations, the definition of reorderings lifts pointwise to opsemsets:

Definition 3.9. An opsemset P' is a *reordering* of an opsemset P if for all opsems $O' \in P'$ there exists an opsem $O \in P$ such that O' is a reordering of O.

The soundness theorem for reorderings can be stated analogously to the one for eliminations and, although the details are different, the proofs follow the same structure. In particular the proof shows that, given a witness for an execution of a reordered opsem,

it is possible to build the happens-before relation for the witness for the corresponding source opsem by lifting the synchronise-with relation, which is unchanged by the reorderings and relates the same release/acquire events in the two witnesses.

Theorem 3.2. *Let the opsemset P' be a reordering of the opsemset P. If P is well-defined, then so is P', and any execution of P' has the same observable behaviour of some execution of P.*

As the sb relation is partial, reordering instructions in the source code can occasionally introduce sb arrows between reordered actions. For instance, the optimised trace of Figure 3 not only reorders the $\mathsf{W_{NA}\,x\,3}$ with the $\mathsf{R_{NA}\,z\,0}$ and $\mathsf{W_{NA}\,z\,3}$ actions but also introduces an sb arrow between the first two events $\mathsf{R_{NA}\,y\,3}$ and $\mathsf{R_{NA}\,z\,0}$. Unsurprisingly, adding sb arrows to an opsem reduces the non-determinism of the candidate executions and does not introduce new behaviours on well-defined programs.

Definition 3.10. We say that an opsem O' is a *linearisation* of an opsem O if they contain the same actions and asw relation, and whenever $x <_{\mathsf{sb}} y \in O'$ it holds that $x <_{\mathsf{sb}} y \in O$. This definition lifts pointwise to opsemsets.

Theorem 3.3. *Let the opsemset P' be a linearisation of the opsemset P. If P is well-defined then so is P', and any execution of P' has the same observable behaviour of some execution of P.*

3.3 Introductions of actions

Even if it seems counterintuitive, compilers tend to introduce loads when optimising code (introducing writes is incorrect in DRF models most of the time [7], and always dubious — see the final example in Section 5). Usually the introduced loads are irrelevant, that is their value is never used. This program transformation sounds harmless but it can introduce data races and does not lend itself to a theorem analogous to those for eliminations and reorderings. Worse than that, if combined with DRF-friendly optimisations it can introduce unexpected behaviours [19]. We conjecture that a soundness property can be stated relating the source semantics to the actual hardware behaviour, along the lines of: if an opsem O' is obtained from an opsem O by introducing irrelevant reads, and it is then compiled naively following the standard compilation schemes for a given architecture [15, 23], then all the behaviours observable on the architecture are allowed by the original opsem. In some cases the introduced loads are not irrelevant, but are RaR-eliminable or RaW-eliminable. We proved that RaR-eliminable and RaW-eliminable introductions preserve DRF and do not introduce new behaviours under the hypothesis that there is no *release* action in sb order between the introduced read and the action that justifies it. Details are available online [1].

4. Compiler Testing

Building on the theory of the previous section, we designed and implemented a bug-hunting tool called `cmmtest`. The tool performs random testing of C and C++ compilers, implementing a variant of Eide and Regehr's *access summary testing* [10]. A test case is any well-defined, sequential C program; for each test case, `cmmtest`:

1. compiles the program using the compiler and compiler optimisations that are being tested;
2. runs the compiled program in an instrumented execution environment that logs all memory accesses to global variables and synchronisations;
3. compares the recorded trace with a reference trace for the same program, checking if the recorded trace can be obtained from the reference trace by valid eliminations, reorderings and introductions.

Test-case generation We rely on a small extension of Csmith [24] to generate random programs that cover a large subset of C while avoiding undefined and unspecified behaviours. We added mutex variables (as defined in `pthread.h`) and system calls to `pthread_mutex_lock` and `pthread_mutex_unlock`. Enforcing balancing of calls to lock and unlock along all possible execution paths of the generated test-cases is difficult, so mutex variables are declared with the attribute `PTHREAD_MUTEX_RECURSIVE`. We also added atomic variables and atomic accesses to atomic variables, labelled with a memory order attribute. For atomic variables we support both the C and the C++ syntax, the former not yet being widely supported by today's compilers. Due to limitations of our tracing infrastructure, for now we instruct Csmith to not generate programs with unions or consts.

Compilation Compilation is straightforward provided that the compiler does not attempt to perform whole-program optimisations. With whole-program optimisation, the compiler can deduce that some functions will never run in a concurrent context and perform more aggressive optimisations, such as eliminating all memory accesses after the last I/O or synchronisation. It is very hard to determine precise conditions under which these aggressive whole-program optimisations remain sound; for instance Ševčík's criteria are overly conservative and result in false positives when testing Intel's `icc -O3`. Neither `gcc` nor `clang` perform whole-program optimisations at the typical optimisation levels `-O2` or `-O3`.

Test case execution and tracing The goal of this phase is to record an execution trace for the compiled object file, that is the linearly-ordered sets of actions it performs. For the x86_64 architecture we implemented a tracing framework by binary instrumentation using the Pin tool [13]. Our Pin application intercepts all memory reads and writes performed by the program and for each records the address accessed, the value read or written, and the size. In addition it also traces calls to `pthread_mutex_lock` and `pthread_mutex_unlock`, recording the address of the mutex. With the exception of access size information, entries in the execution trace correspond to the actions of Section 2. Access size information is needed to analyse optimisations that merge adjacent accesses, discussed below.

The "raw" trace depends on the actual addresses where the global variables have been allocated, which is impractical for our purposes. The trace is thus processed with additional informations obtained from the ELF object file (directly via `objdump` or by parsing the debugging informations). The addresses of the accesses are mapped to variable names and similarly for values that refer to addresses of variables; in doing so we also recover array and struct access information. For example, if the global variable `int g[10]` is allocated at `0x1000`, the raw action `Store 0x1008 0x1004 4` is mapped to the action `Store g[2] &g[1] 4`. After this analysis execution traces are independent of the actual allocation addresses of the global variables. Finally, information about the initialisation of the global variables is added to the trace with the `Init` tag.

Irrelevant reads The Pin application also computes some data-flow information about the execution, recording all the store actions which *depend* on each read. This is done by tracking the flow of each read value across registers and thread-local memory. A dependency is reported if either the value read is used to compute the value written or used to determine the control that leads to a later write or synchronisation. With this information we reconstruct an approximation of the set of *irrelevant reads* of the execution: all reads whose returned value is never used to perform a write (or synchronisation) are labelled as irrelevant.

In addition, we use a second algorithm to identify irrelevant reads following their characterisation on the opsems: the program is replayed and its trace recorded again but binary instrumentation

injects a different value for the read action being tested. The read is irrelevant if, despite the different value read, the rest of the trace is unchanged. This approach is slower but accurately identifies irrelevant reads inserted by the optimiser (for instance when reorganising chains of conditions on global variables).

Reference trace In the absence of a reference interpreter for C11, we reuse our tracing infrastructure to generate a trace of the program compiled at `-O0` and use this as the reference trace. By doing so our tool might miss potential front-end concurrency bugs but there is no obvious alternative.

Trace matching Trace matching is performed in two phases. Initially, eliminable actions in the reference trace are identified and labelled as such. The labelling algorithm linearly scans the trace, recording the last action performed at any location and whether release/acquire actions have been encountered since; using this information each action is analysed and labelled following the definition of eliminable actions of Section 3.1. Irrelevant reads reported by the tracing infrastructure are labelled as such. To get an intuition for the occurrences of eliminable actions in program traces, out of 200 functions generated by Csmith with `-expr_complexity 3`, the average trace has 384 actions (max 15511) of which 280 (max 14096) are eliminable, distributed as follows:

IR	RaW	RaR	OW	WaR	WaW
8 (94)	94 (1965)	95 (10340)	75 (1232)	5 (305)	1 (310)

Additional complexity in labelling eliminable actions is due to the fact that a compiler performs many passes over the program and some actions may become eliminable only once other actions have been eliminated. Consider for instance this sequence of optimisations from the example in Figure 2:

```
Store g6 4                                          Store g6 4
Load  g6 4                                          Store g6 1
Load  g6 4   --RaW-->   Store g6 4   --OW-->   Store g6 1
Load  g6 4                                          Store g6 1
Store g6 1
```

In the original trace, the first store cannot be labelled as OW due to the intervening loads. However, if the optimiser first removes these loads (which is correct since they are RaW), it can subsequently remove the first store (which becomes an OW). Our analysis thus keeps track of the critical pairs between eliminations and can label actions eliminable under the assumption that other actions are themselves eliminated. Irrelevant and eliminable reads are also labelled in the optimised trace, to account for potential introductions.

Before describing the matching algorithm, we must account for one extra optimisation family we omitted in the previous section: *merging of adjacent accesses*. Consider the following loop updating the global array `char g[10]`:

```
for (int l=0; l<10; l++) g[l] = 1;
```

The reference trace performs ten writes of one byte to the array `g`. The object code generated by `gcc -O3` performs only two memory accesses: `Store g 101010101010101 8` and `Store g[8] 101 2`. This optimisation is sound under hypothesis similar to those of eliminations: there must not be intervined accesses or release/acquire pairs between the merged accesses; additionally the size of the merged access must be a power of two and the merged store must be aligned. Since the C11/C++11 memory model, as formalised by Batty et al., is agnostic to the size of accesses, we could not formally prove the soundness of this optimisation.

The matching algorithm takes two annotated traces and scans them linearly, comparing one annotated action of the reference trace and one of the optimised trace at a time. It performs a depth-first search exploring at each step the following options:

- if the reference and optimised actions are equal, then consider them as matched and move to the next actions in the traces;
- if the reference action is eliminable, delete it from the reference trace and match the next reference action; similarly if the optimised action is eliminable;
- if the reference action merges with later reference actions to match the the optimised action, then consider the merged actions as matched and move to the next actions in the traces; and
- if the optimised action can be matched by reordering actions in the reference trace, reorder the reference trace and match again.

The algorithm succeeds if it reaches a state where all the actions in the two input traces are either deleted or matched.

Some of these options are very expensive to explore (for instance when combinations of reordering, merging and eliminations must be considered), and we guide the depth-first search with a critical heuristic to decide the order in which the tree must be explored. The heuristic is dependent on the compiler being tested. The current implementation can match in a few minutes `gcc` traces of up to a few hundreds of actions on commodity hardware; these traces are well beyond manual analysis.

Outcome During tracing, `cmmtest` records one "execution" of a C or C++ program; using the terminology of the previous section, it should observe a witness for an opsem of the program. Since `cmmtest` traces sequential deterministic code in an empty concurrent environment, all reads of the opsem always return the last (in sb order) value written by the same thread to the same location. The witness is thus trivial: reads-from derives from sb while mo and sc are both included in sb. Note that the tool traces an execution of the generated assembler and as such it records a linearisation of the opsem's sb. Theorem 3.3 guarantees that this cannot introduce false positives due to extra sb arrows added between non-atomic accesses. Overestimating the sb relation between atomic accesses might induce false positives because `cmmtest` cannot reconstruct the original sb relation and considers all atomic accesses as unreorderable. To prevent this our version of Csmith never generates programs with atomic accesses not related by sb.

The `cmmtest` tool compares two opsems and returns true if the optimised opsem can be obtained from the reference opsem by a sequence of sound eliminations/reorderings/introductions, and false otherwise. If `cmmtest` returns *true* then we deduce that this opsem (that is, this execution path of the program) has been compiled correctly, even if we cannot conclude that the whole program has been compiled correctly (which would require exploring all the opsems of the opsemset, or, equivalently, all the execution paths of the program). More interestingly, whenever `cmmtest` returns *false*, then either the code has been miscompiled, or an exotic optimisation has been applied (since our theory of sound optimisations is not complete). In this case we perform test-case reduction using CReduce [16], and manually inspect the assembler. Reduced test-cases have short traces (typically less than 20 events) and it is immediate to build discriminating contexts and produce bug-reports. Currently the tool reports no false positives for `gcc` on any of the many thousands of test-cases we have tried without structs and atomics; we are getting closer to a similar result for arbitrary programs (with the restriction of a single atomic access per expression).

Stability against the HW memory model Compiled programs are executed on shared-memory multiprocessors which may expose behaviours that arise from hardware optimisations. Reference mappings of atomics instructions to x86 and ARM/POWER architectures have been proved correct [4, 5]: these mappings insert all the necessary assembly synchronisation instructions to prevent hardware optimisations from breaking the C11/C++11 semantics. On x86_64, `cmmtest` additionally traces memory fences and locked instructions, and under the hypothesis that the reference trace is obtained by applying a correct mapping (e.g., by putting fence instructions after all SCatomic writes), then `cmmtest` additionally ensures

that the optimiser does not make hardware behaviours observable (for instance by moving a write after a fence).

5. Impact on compiler development

Concurrency compiler bugs While developing `cmmtest` and tuning the matching heuristic, we tested the latest svn version of the 4.7 and 4.8 branches of the `gcc` compiler. During these months we reported several concurrency bugs (including bugs no. 52558, 54149, 54900, and 54906 in the `gcc` bugzilla), which have all been promptly fixed by the `gcc` developers. In one case the bug report highlights an obscure corner case of the `gcc` optimiser, as shown by a discussion on the `gcc-patches` mailing list;[3] even though the reported test-case has been fixed, the bug has been left open until a general solution is found. In all cases the bugs were wrongly introduced writes, speculated by the LIM or IFCVT (if-conversion) phases, similar to the example in Figure 1. These bugs do not only break the C11/C++11 memory model, but also the Posix DRF-guarantee which is assumed by most concurrent software written in C and C++. The corresponding patches are activated via the `---param allow-store-data-races=0` flag, which will eventually become default standard for `-std=c11` or `-std=c++11` flags. All these are *silent wrong-code bugs* for which the compiler issues no warning.

Unexpected behaviours Each compiler has its own set of internal invariants. If we tune the matching algorithm of `cmmtest` to check for compiler invariants rather than for the most permissive sound optimisations, it is possible to catch unexpected compiler behaviours. For instance, in the current phase of development, `gcc` forbids all reorderings of a memory access with an atomic one. We baked this invariant into `cmmtest` and in less than two hours of testing on an 8-core machine we found the following testcase:

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

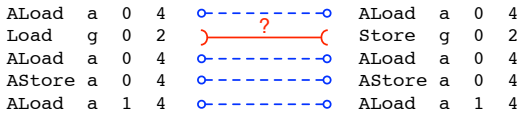
whose traces for the function `main` compiled with `gcc 4.8.0 20120627 (experimental)` at optimisation levels `-O0` and `-O2` (or `-O3`) are:

As we can observe, the optimiser moved the load of `g1` before the two atomic `SC` loads. Even though we conjecture that this reordering is not observable by a non-racy context, this unexpected compiler behaviour was fixed nevertheless (`rr190941`).

Interestingly, `cmmtest` found one unexpected compiler behaviour whose legitimacy is arguable. Consider the program below:

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

Traces for the `func_1` function, compiled with `gcc 4.8.0 20120911 (experimental)` at optimisation levels `-O0` and `-O2` (or `-O3`), are:



[3] http://gcc.gnu.org/ml/gcc-patches/2012-10/msg01411.html

The optimiser here replaced the inner loop with a single write:

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

thus substituting a read access with a write access in the execution path where `g` already contains `0`. The introduced store is *idempotent*, as it rewrites the value that is already stored in memory. Since in the unoptimised trace there is already a load at the same location, this extra write cannot be observed by a non-racy context. Strictly speaking, this is not a compiler bug. However, whether this should be allowed or not is subject to debate. Although we believe no architecture can detect this introduced write, in a world with hardware or software race detection it might fire a false positive. Also, possibly a bigger issue, the introduced write can introduce cache contention where there should not be any, potentially resulting in an unexpected performance loss.

6. Related work

Randomised techniques for testing compilers have been popular since the 60's and have been applied to a variety of languages, ranging from Cobol [22] and Fortran [9] to C [12, 14, 21] and C++ [25]. A survey (up to 1997) can be found in Boujarwah and Saleh [8], while today the state of art is represented by Yang et al.'s work on Csmith [24]. None of these works addresses concurrency compiler bugs and the techniques presented are unable to detect any of our bug reports.

The notable exception is Eide and Regehr's work [10] on hunting miscompilation of the `volatile` qualifier in production-quality C compilers. Eide and Regehr generate random well-defined *deterministic, sequential* programs with volatile memory accesses: miscompilations are detected by counting how many accesses to volatile variables are performed during the execution of an unoptimised and an optimised binary of the program. The semantics of the volatile attribute requires that accesses to volatile variables are never optimised away, so comparing the number of runtime accesses is enough to detect bugs in the optimiser. We were inspired by Eide and Regehr's approach to reduce hunting concurrency compilation bugs to analysis to differential testing of sequential code, but the complexity of the C11/C+11 memory model requires us to build a theory of sound optimisations and makes the analysis phase far more complicated.

As we discussed in Section 3, soundness of optimisations in an idealised DRF model was studied by Ševčík [17, 19]. We reuse Ševčík's classification of optimisations as eliminations, reorderings and introductions, but moving from an idealised DRF model to the full C11/C++11 memory model brings new challenges:

- we cannot identify a program with the set of linear orders of actions it can realise because in C and C++ the sequenced-before order is not total; although reasoning about opsems seems unintuitive, partial orders turn out be be easier to work with than the explicit manipulation of trace indices that Ševčík performs;
- the semantics of low-level atomic accesses and fences must be taken into account when computing synchronisations; in particular the weaker consistency and coherency conditions of the release/acquire/relaxed attributes made the soundness proof much more complex.

There are other minor differences, for instance Ševčík assumes that every variable has a default value, while C/C++ forbids accessing an uninitialised variable. Initially Ševčík gave a more restrictive condition for soundness of eliminations [17], namely eliminations are forbidden if there is an intervening release *or* acquire operation rather than a release/acquire pair. This simpler condition appears to be too strong as we have observed compilers eliminate accesses across a release or acquire access. All our analysis was driven by what optimisations we could actually observe: this led to identify-

ing WaW eliminations and RaR/RaW introductions, and motivated us to omit roach-motel reorderings.

More generally, reasoning about the C11/C++11 memory model is in its infancy. We follow the formalisation of the C11/C++11 memory model given in [4]; the original presentation of the model design [7] mentions (Sec. 2.1) that the soundness of reorderings under some conditions follows from previous work [2], with no mention of eliminations or introductions. Although the C11/C++11 model is based on the DRF design, we do believe that its complexity deserves careful proofs taking into account its whole design.

There is a long tradition of research on compiler optimisations that preserve sequential consistency dating back to Shasha and Snir [20], mostly focusing on whole program analyses. While these works show that a restricted class of optimisations can maintain the illusion of sequential consistency for all programs, we show that common compiler transformations maintain the illusion of sequential consistency for correctly synchronised programs.

7. Conclusion

This paper validates the belief that common compiler optimisations are sound in C11/C++11 memory model, contrary to the situation for the Java Memory Model [18]. With the exception of optimisations that change the size of memory accesses (which cannot be expressed in the current formalisation of the memory model) and irrelevant read introduction (which does not preserve DRF), we proved correct all the classes of optimisations performed by widely used optimising compilers (under the hypothesis that a program can halt at any time). We presented a general strategy to perform differential testing of real compilers against memory models, and designed and implemented a bug-hunting tool, `cmmtest`, building on our theory of optimisations in the C11/C++11 memory model. Subtle concurrency bugs and unexpected behaviours of the latest `gcc` optimiser have been discovered using `cmmtest`. None of these could have been found using the existing compiler testing methods.

Future work Bug-hunting via random testing is a slow process: each reported bug must be fixed before continuing testing, otherwise with high probability the tool keeps on rediscovering the same bug. Our priority now is to complete the tracing infrastructure to support all the features of the C language, most notably bit-fields, and to put `cmmtest` to work conducting extensive testing of `gcc` and `clang`. Since our testing strategy considers the compiler as a black box, it is easy to extend the tool to test other compilers. A preliminary trial run with `clang` suggests that only minor changes to the matching heuristic are required; contrary to `gcc`, `clang` systematically reorders accesses around relaxed atomic accesses and performs simultaneous merge and reorder of memory actions. Although this article focuses on the x86_64 architecture and x86_64 compiler backends, only the tracer module (and IR data-flow analysis) of the tool are dependent on the binary architecture. Building on Fox's ARM emulator [11], we are implementing a tracing infrastructure for ARMv7 binaries that will allow testing the ARM backends of compilers. By instrumenting an executable semantics for the LLVM IR (e.g., [26]), it would even be possible to run our analysis internally within the LLVM optimisers, comparing the traces before and after each optimisation pass.

Last but not least, the `gcc` developers expressed a keen interest in adopting our `cmmtest` tool as part of their testing infrastructure.

Acknowledgments We are grateful to Jaroslav Ševčík, Kayvan Memarian, and Peter Sewell for enlightening discussions, to John Regehr and Xuejun Yang for help with Csmith, to Aldy Hernandez, Andrew MacLeod and Torvald Riegel for promptly addressing our bug-reports. This work was partially supported by IRILL and ANR grant WMC (ANR-11-JS02-011).

References

[1] The `cmmtest` tool. http://www.di.ens.fr/~zappa/projects/cmmtest/.

[2] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.

[3] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *TACAS*, 2011.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[5] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, 2012.

[6] P. Becker. *Standard for Programming Language C++ - ISO/IEC 14882*, 2011.

[7] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[8] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology*, 39(9): 617–625, 1997.

[9] C. J. Burgess and M. Saidi. The automatic generation of test cases for optimizing fortran compilers. *Information & Software Technology*, 38 (2):111–119, 1996.

[10] E. Eide and J. Regehr. Volatiles are miscompiled and what to do about it. *EMSOFT*, 2008.

[11] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *ITP*, 2010.

[12] C. Lindig. Random testing of C calling conventions. In *AADEBUG*, 2005.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[14] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[15] P. E. McKenney and R. Silvera, 2011. http://www.rdrop.com/users/paulmck/scalability/!paper/N2745r.2011.03.04a.html.

[16] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI*, 2012.

[17] J. Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.

[18] J. Ševčík. The Sun Hotspot JVM does not conform with the Java memory model. Technical Report EDI-INF-RR-1252, School of Informatics, University of Edinburgh, 2008.

[19] J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[20] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2), 1988.

[21] F. Sheridan. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475–1488, 2007.

[22] R. L. Solder. A general test data generator for COBOL. In *AFIPS Joint Computer Conferences*, 1962.

[23] A. Terekhov. Brief tentative example x86 implementation for C/C++ memory model, 2008. http://www.decadent.org.uk/pipermail/~cpp-threads/2008-December/001933.html.

[24] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[25] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated test program generation for an industrial optimizing compiler. In *AST*, 2009.

[26] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.