The Thin-Air Problem — Introduction

Peter Sewell

University of Cambridge

MMmeet workshop, September 2014

Cambridge

Problem

We don’t have a credible semantics for any high-performance shared-memory concurrent language.

JMM (JSR-133): not sound w.r.t. implementations

C/C++11: sound but unreasonably weak, due to thin-air problem

Linux C: not specified

The Question

What semantics can we adopt for PL shared-memory accesses that:

1. are allowed to race (to be used in concurrent algorithms;
2. should be implementable with (high-performance) simple machine loads and stores, without extra barriers; and
3. are compatible with reasonable compiler optimisations, without excessively constraining existing compilers?

In particular, for C/C++ *relaxed atomics*.

Current Status

There are

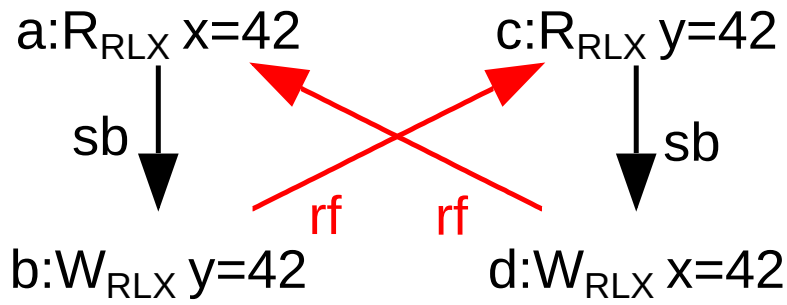
- example executions the semantics must allow (because reasonable compiler optimisation + current h/w exhibits them), and
- example executions we want the semantics to forbid (“thin-air 1 and 2”)

but we don’t yet have a good way to discriminate between them.

Example LB (language must allow)

```
r1=load_rlx(x) //reads 42
store_rlx(y,42)
```

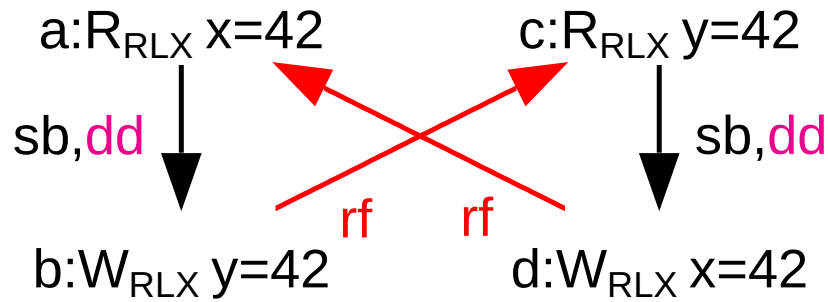
```
r2=load_rlx(y) //reads 42
store_rlx(x,42)
```



- permitted by the ARM and IBM POWER architectures (with obvious compilation)
- experimentally observable on current ARM multiprocessors
- hence the language semantics must allow it for relaxed atomics.

LB+datas (language can and should forbid)

```
r1=load_rlx(x) //reads 42
store_rlx(y,r1)
-----------------------
r2=load_rlx(y) //reads 42
store_rlx(x,r2)
```



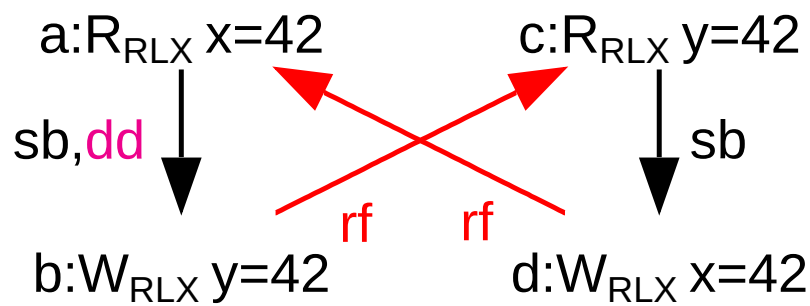
Thin-air 1: a *thin-air read value* execution

- architecturally forbidden on current hardware (x86, ARM, and IBM POWER)
- would need load-value prediction in future h/w
- AFAWK cannot be exhibited by any reasonable current compiler optimisation combined with current hardware
- Hence, the language semantics could forbid it.
- Moreover (cf BD2014) desirable to forbid it.
- forbid cycles in rf ∪ dep? no...

Example LB+ctrldata+po (language must allow)

```
r1=load_rlx(x) //reads 42
if (r1 == 42)
  store_rlx(y,r1)
```

```
r2=load_rlx(y) //reads 42
store_rlx(x,42)
```

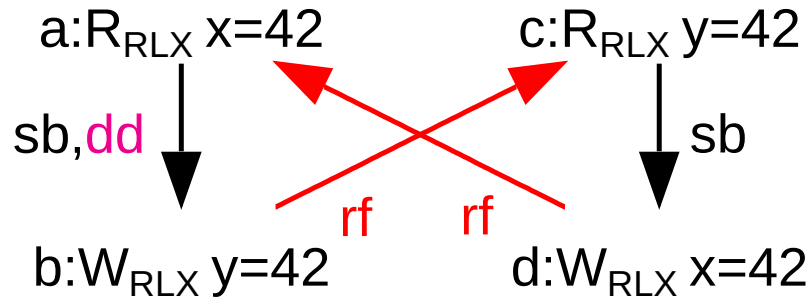


- as LB, architecturally allowed on ARM and Power, and observable on ARM
- hence the language must allow it

LB+ctrldata+ctrl-double (language must allow)

```
r1=load_rlx(x) //reads 42
if (r1 == 42)
  store_rlx(y,r1)
```

```
r2=load_rlx(y) //reads 42
if (r2 == 42)
  store_rlx(x,42)
else
  store_rlx(x,42)
```

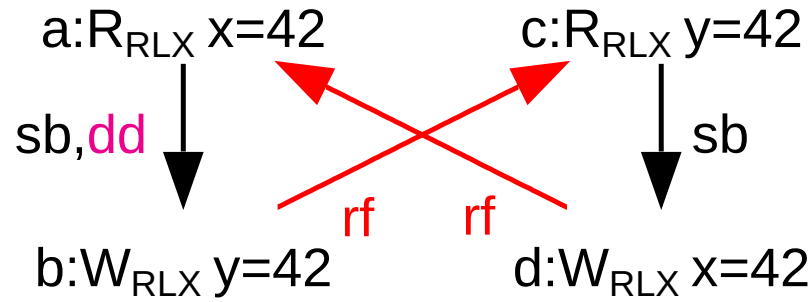


- forbidden on hardware if compiled naively (R-W ctrl)
- but optimisation will collapse to LB+ctrldata+po
- hence, the language must also allow this execution.
- but this execution has a cycle in rf ∪ dep, so we cannot simply exclude all those.

Then one might hope for some other adaptation of the C/C++11 model, but the following example shows at least that there is no per-candidate-execution solution.

LB+ctrldata+ctrl-single (can and should forbid)

```
r1=load_rlx(x) //reads 42
if (r1 == 42)
  store_rlx(y,r1)
---------------------------
r2=load_rlx(y) //reads 42
if (r2 == 42)
  store_rlx(x,42)
```



- Thin-air 2: a “self-satisfying conditional” execution
- forbidden on hardware if compiled naively
- unaffected by reasonable thread-local optimisation
- Hence, the language could forbid it.
- Moreover, desirable to forbid
- But this candidate execution is identical to the previous one, that we have to allow...

So we cannot do both simultaneously with any adaptation of the C/C++11 per-candidate-execution definitions that uses the same notion of candidate execution.

The basic point here is that compiler optimisations (such as the collapse of the LB+ctrldata+ctrl-double conditional) are operating over a representation of the *program*, covering all its executions, while the C/C++11 definition of candidate execution and consistency for those considers each candidate *execution* independently (it ignores the set of all executions); it is not able to capture the fact that the conditional is unnecessary because the two candidate executions corresponding to taking the two branches are equivalent.

Not just a problem for relaxed atomics

In C can mix atomics and nonatomics, e.g. reusing malloc'd region. Similar problems arise (see examples in "Challenges").

Possible approaches

1. ban all rf $\cup$ dep cycles after all – but costly
2. ban cycles involving relaxed – but via other compilation units?
3. declare programs with non-annotated dep cycles to be bad – unworkable?
4. prevent all load-to-store reordering – costly?
5. develop explicit out-of-order semantics – but ....
6. transitive closure of allowable abstract compiler optimisations – but...