An out-of-order thread-local semantics for something like volatile relaxed atomics in C and the problems it highlights

Jean Pichon

24th of September 2014

Goal

How to avoid "out-of-thin-air" with C11's relaxed atomics?

Remark by Mark Batty: no *per-candidate-execution* semantics (like the C11 standard) can at the same time allow load buffering

```
r1 = x;  ||  r2 = y;
y = 42   ||  x = 42
```

$r1 = 42 \wedge r2 = 42$ OK

but forbid "out-of-thin-air" behaviour
such as load buffering plus data dependencies ("LB+datas")

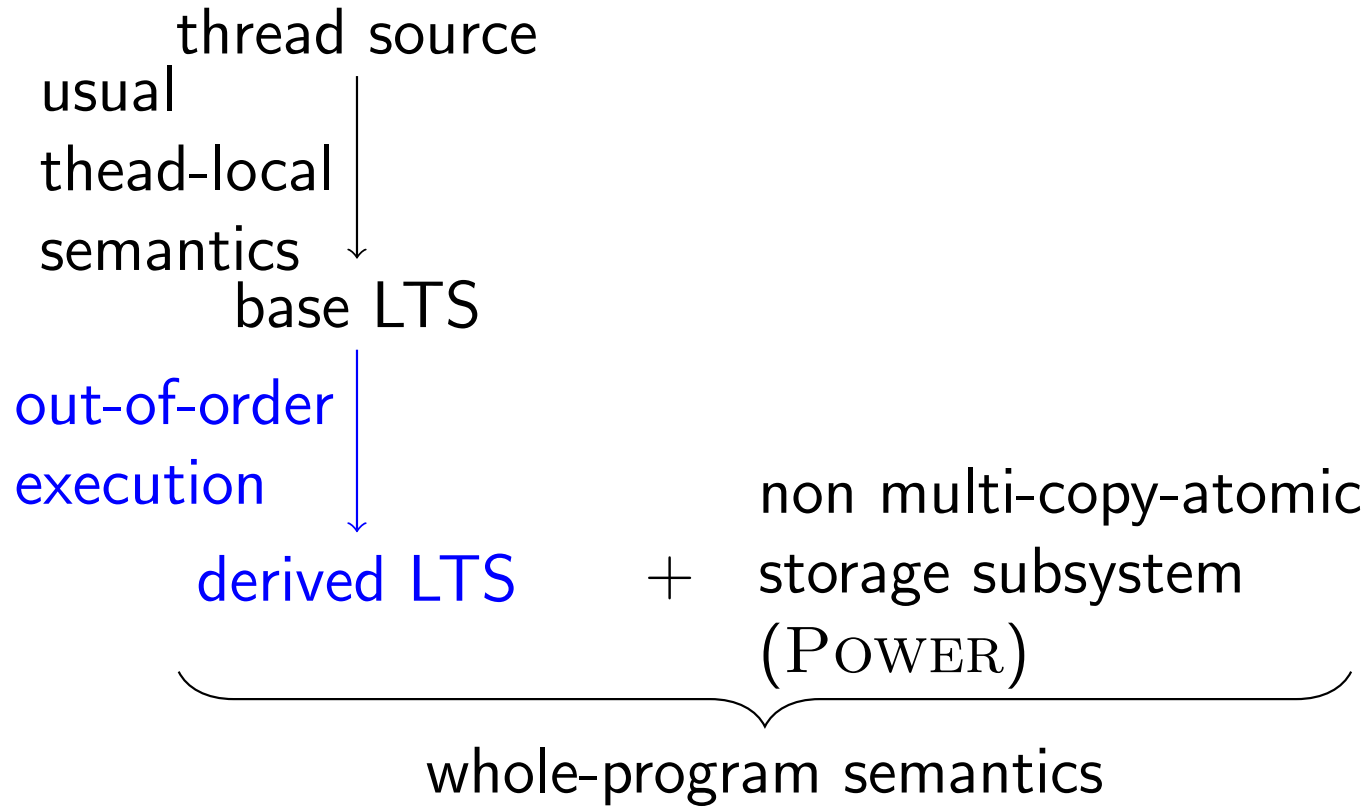
```
r1 = x;  ||  r2 = y;
y = r1   ||  x = r2
```

$r1 = 42 \wedge r2 = 42$ BAD

where the value 42 appears "out of thin air".

Contribution

1) A thread-local semantics with "the right amount" of out-of-order execution.



---

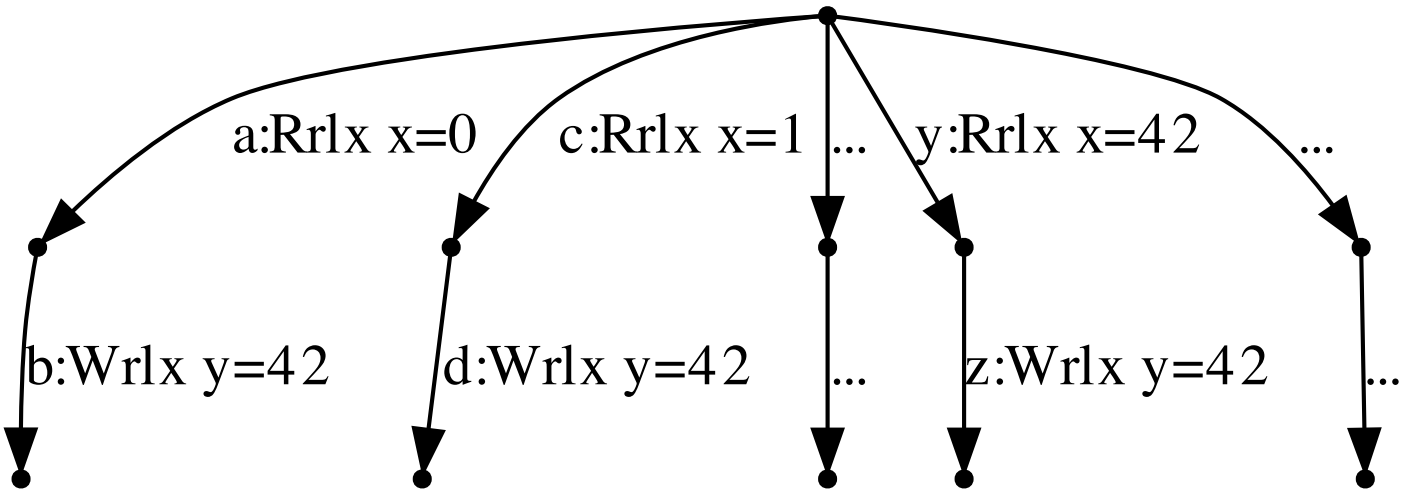
2) And its use to illustrate problems.

Observation 1

Starting from the program

```
r1 = x;
if (r1 == 42) {
  y = r1
} else {
  y = 42
}
```

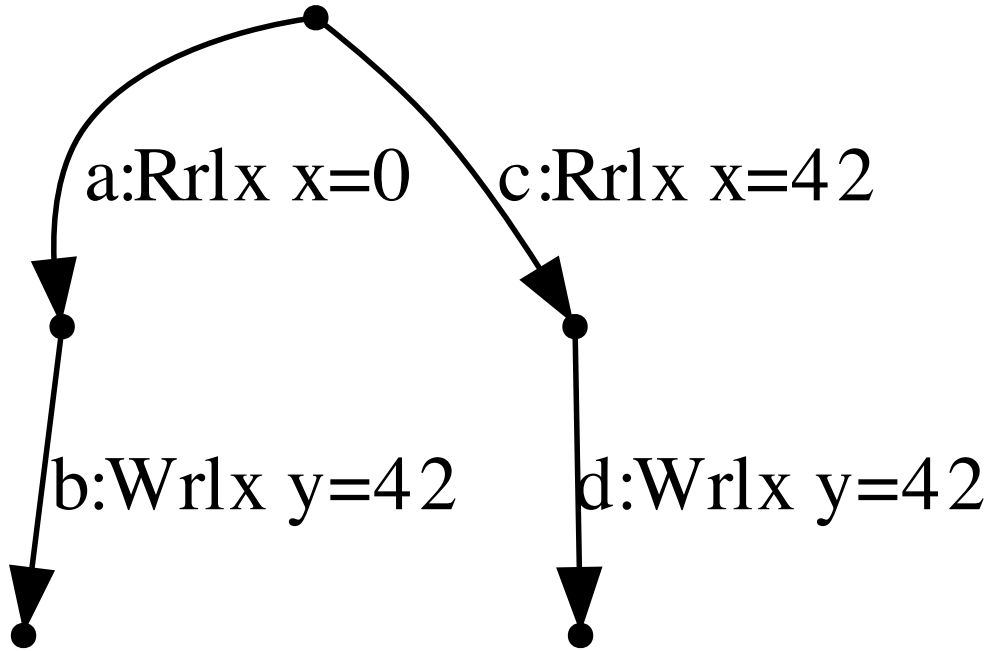
the base semantics gives the base LTS



The thread-local semantics does not specify what can be read ($\rightsquigarrow$ receptivity).

Observation 2

```
r1 = x;
if (r1 == 42) {
  y = r1
} else {
  y = 42
}
```



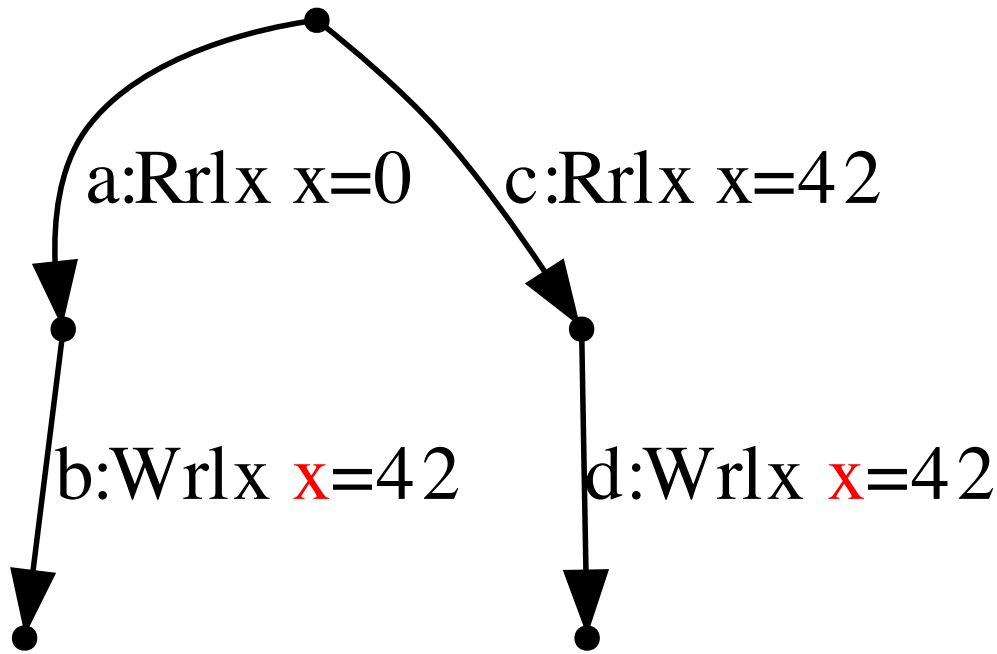
The write to `y` can be executed before the read from `x` as

- it happens in all the branches of the program;
- nothing (in particular not POWER "coherence") forces us to execute the read from `x` before.

Observation 3

On the other hand, if the write is to `x`, then it can't be executed before the read
(because of POWER "coherence"):

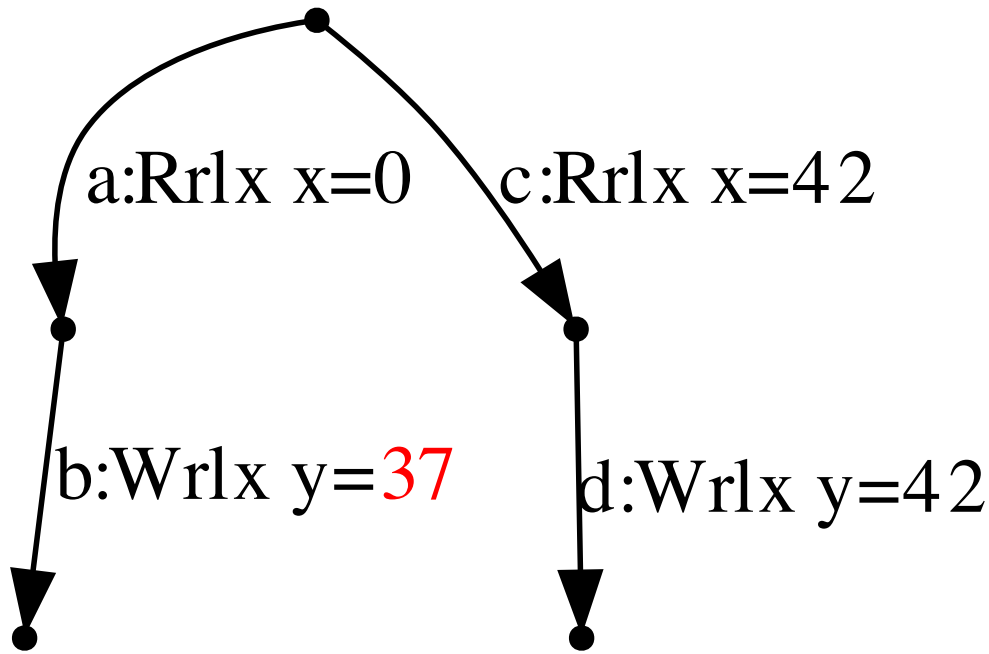
```
r1 = x;
if (r1 == 42) {
  x = r1
} else {
  x = 42
}
```



Observation 4

If the write is not available in all branches of the program, we can't execute the write before the read:

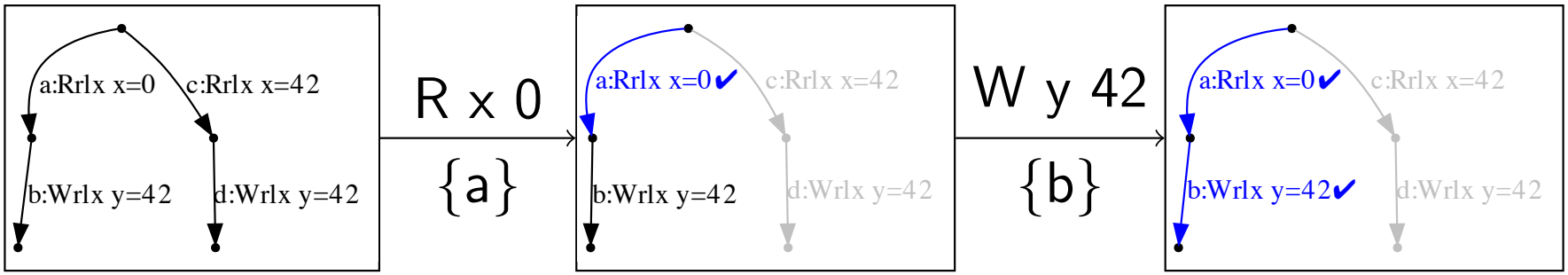
```
r1 = x;
if (r1 == 42) {
  y = r1
} else {
  y = 37
}
```



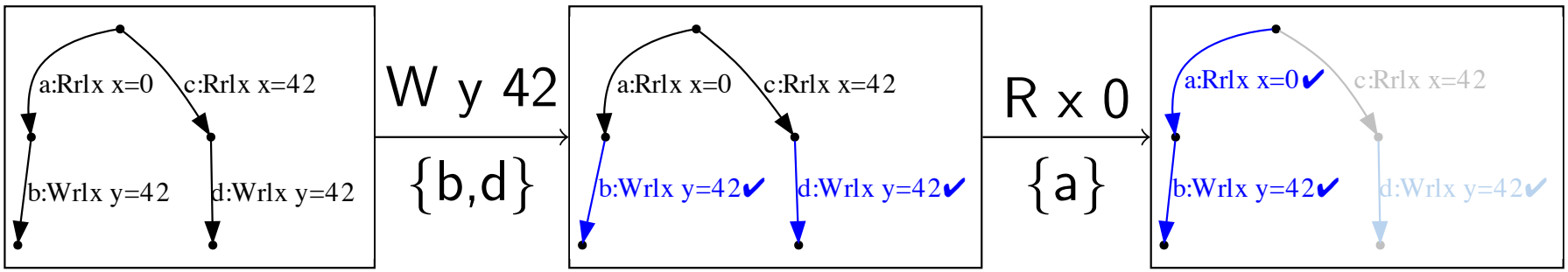
Idea: ticking

Executing the base LTS out-of-order, by ticking sets of edges.

Like in the base LTS, we can have

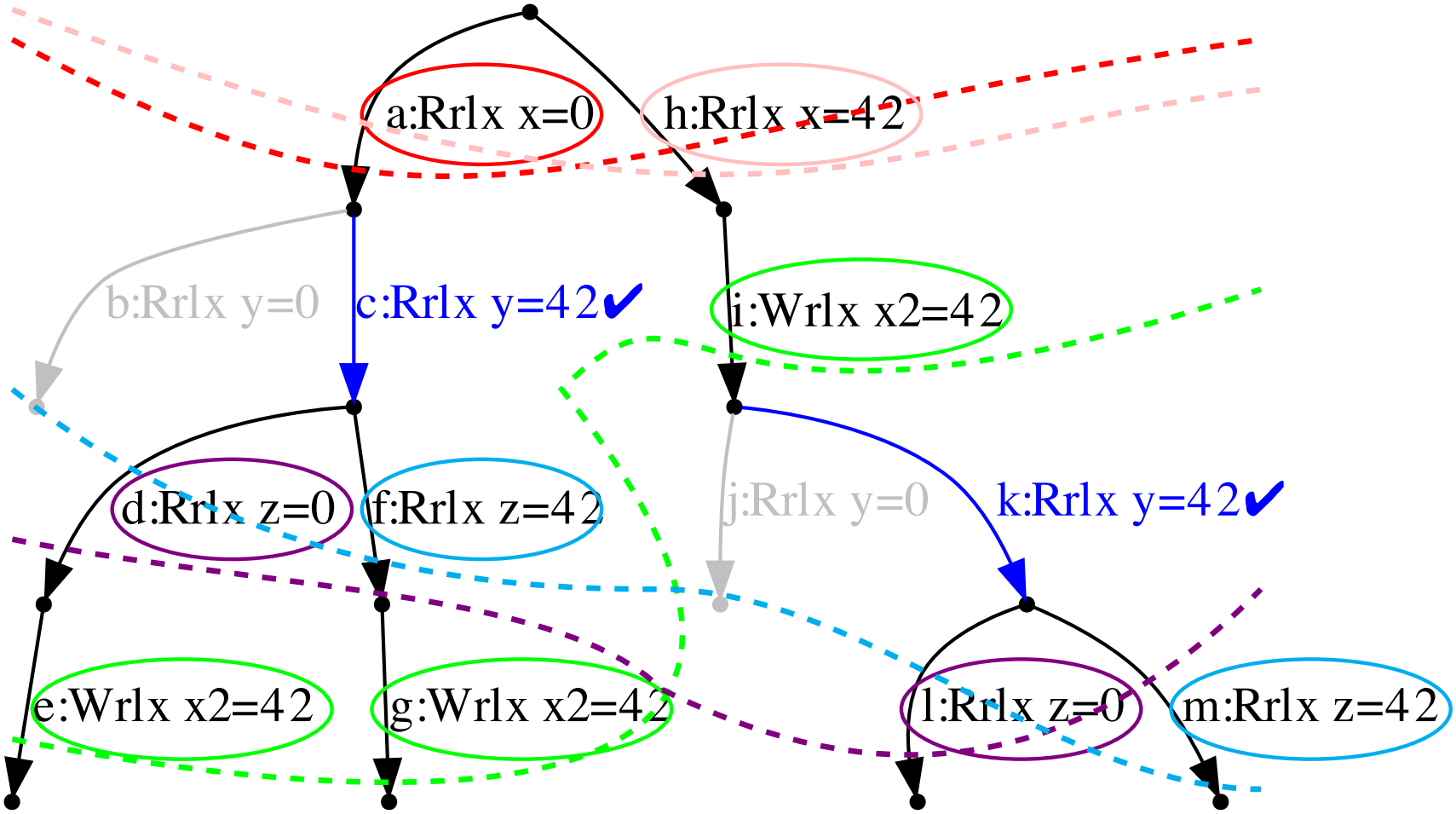


But we can also have



because the `Wrlx y=42` is available in all branches.

Frontier

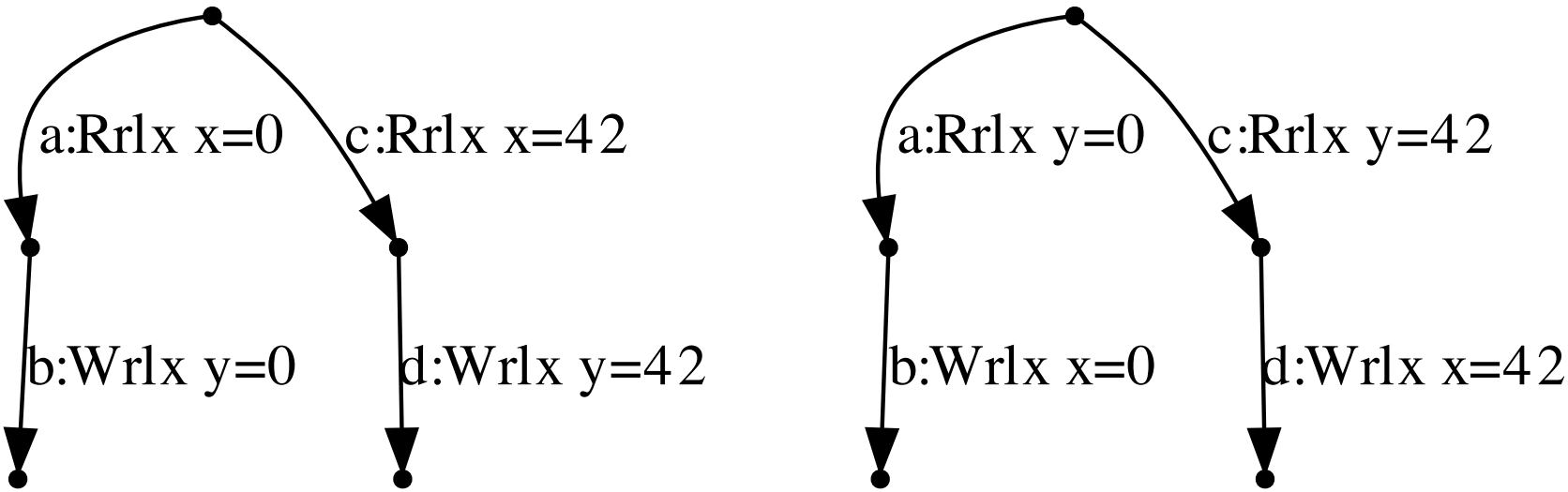

a:Rrlx x=0
h:Rrlx x=42
b:Rrlx y=0
c:Rrlx y=42✔
i:Wrlx x2=42
d:Rrlx z=0
f:Rrlx z=42
j:Rrlx y=0
k:Rrlx y=42✔
e:Wrlx x2=42
g:Wrlx x2=42
l:Rrlx z=0
m:Rrlx z=42

No more out-of-thin-air

LB+datas is not problematic anymore:

```
r1 = x;  || r2 = y;
y = r1   || x = r2
```

yields



$\Longrightarrow$ no out-of-order execution

$\Longrightarrow$ no out-of-thin-air behaviour

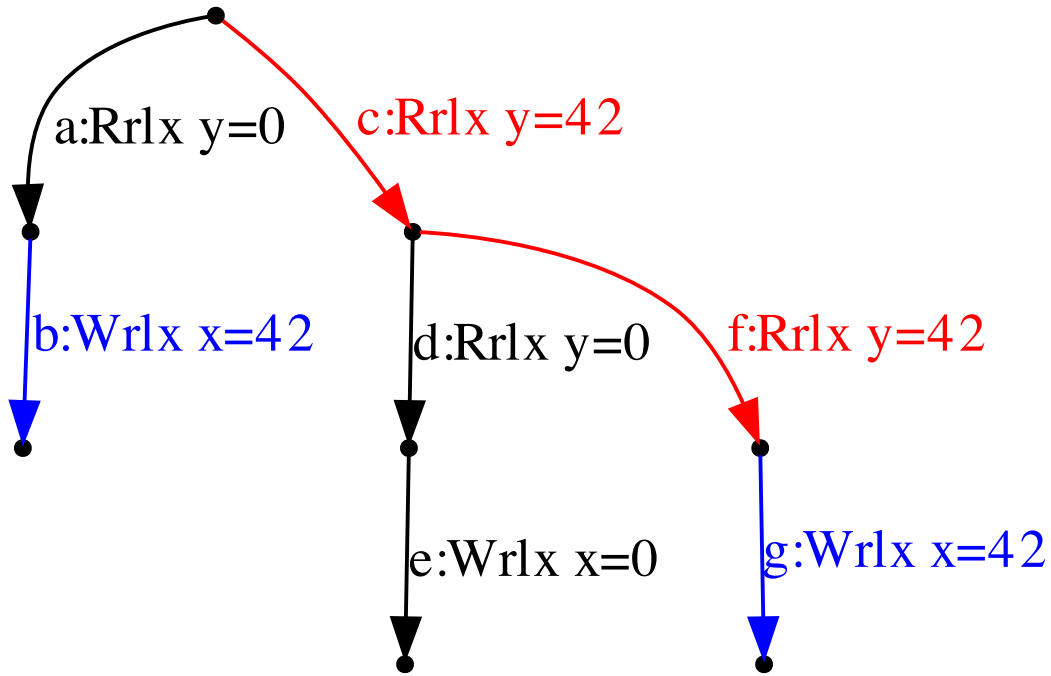
Problems

Problem with (thread-local) optimisations

each action is executed once (and only once)
$\Longrightarrow$ sort of `volatile`: no introduction or elimination

Jaroslav Ševčík's example:

```
r2 = y;
if (r2 == 42) {
  r3 = y;
  x = r3
} else {
  x = 42
}
```

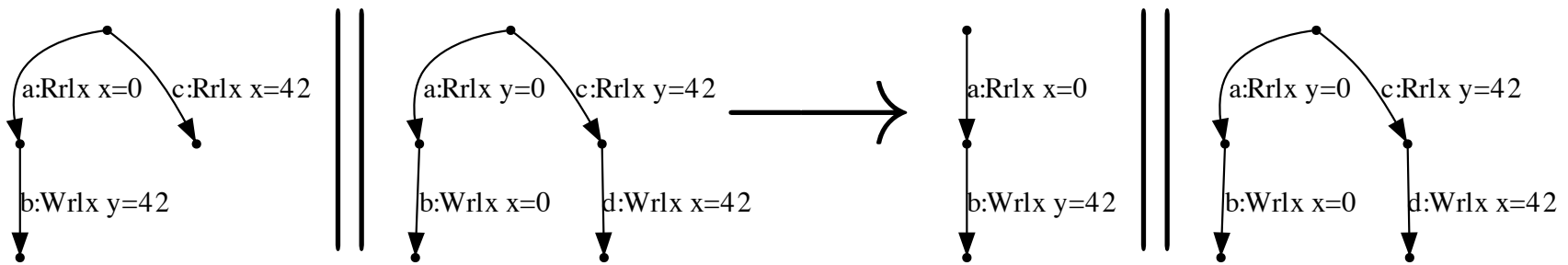


`r2 = y` and `r3 = y` should be mergeable,
so that `x = 42` is available in both branches.

Problem with inter-thread optimisations

```
r1 = x;                  ||  r2 = y;
if (r1 == 0) {           ||  x = r2
   y = 42                ||
}                        ||
```

Value-range analysis can determine `x` can only contain `0`:



$\Longrightarrow$ out-of-thin-air reappears!

Problem with thread-locality

Variables as representations of data-flow (register variables `r`) vs. variables as memory locations (shared variables `x`).

Escape analysis allows

```
int f(void) {
  int x = 42;
  e1; // no x
  g(x);
  e2; // no x
  return x;
}
```

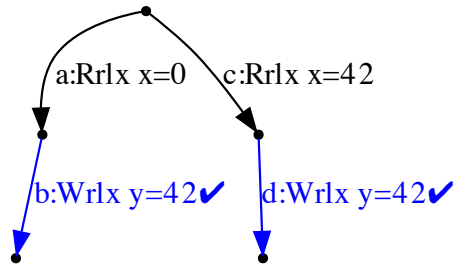
```
int f(void) {
  e1;
  g(42);
  e2;
  return 42;
}
```

Optimisations are “automatic” on register variables.

Interacts with the problem with intra-thread optimisations:
$\rightsquigarrow$ *how much* escape analysis?

Conclusion

Out-of-order execution by ticking frontiers



It covers relaxed reads and writes, fences, and non-atomic.

It gives the desired results on the “out-of-thin-air test suite”.

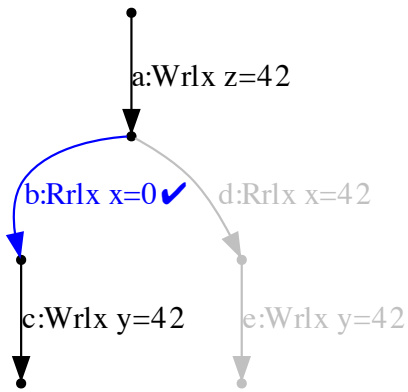
...but no optimisations (everything is `volatile`).

This page intentionally left blank.

Ticking

A set of edges can be ticked iff it forms a "frontier":

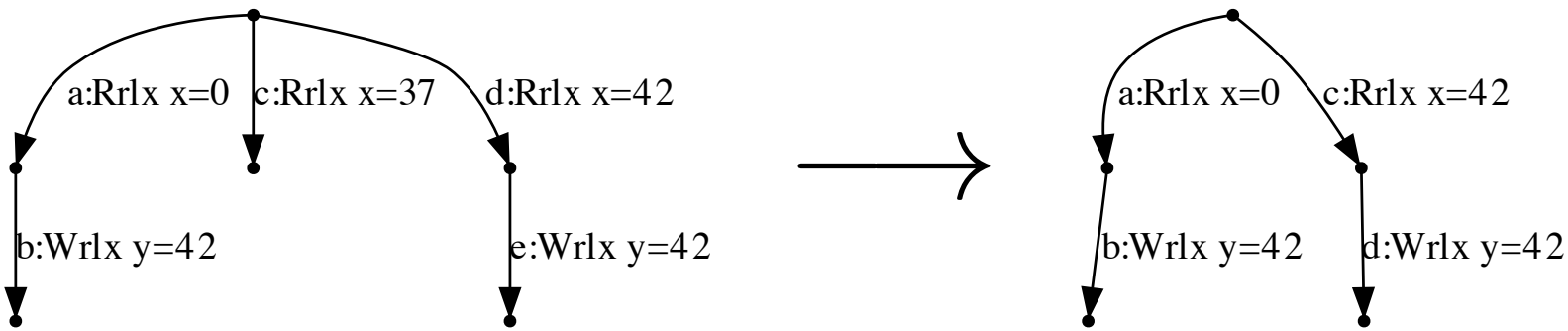
1. all the edges have the same label;
2. all the edges are unticked;
3. all the edges are "executable"
   (not blocked by coherence or a fence);
4. in each non-discarded path, there is one (and only one)
   edge from the set.



A path is discarded iff one of its edges
(necessarily labelled with a read)
has a ticked sibling edge.

Problem with inter-thread optimisations, part 2

```
r1 = x;                                  ||  r2 = y;
if (r1 == 0 || r1 == 42) {               ||  x = r2
   y = 42                                ||
}                                        ||
```



Is this out-of-thin-air?

For Java, no. For common sense, maybe...