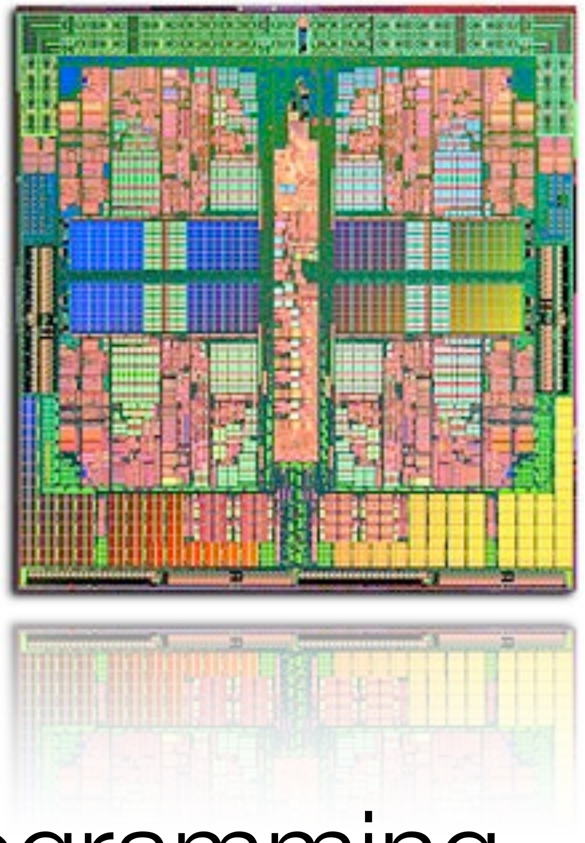
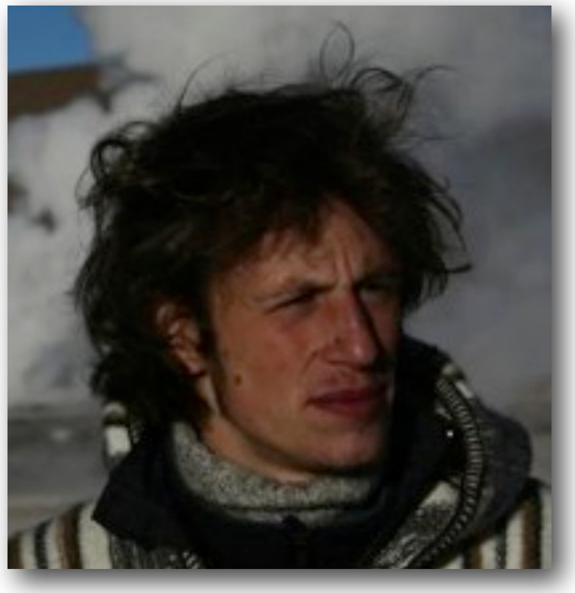


Semantics, languages and algorithms for multicore programming

Albert Cohen

Luc Maranget

Francesco Zappa Nardelli



Vote: topics for my this lecture

1. The lwarx and stwcx Power instructions [3]

2. Hunting compiler concurrency bugs [12]

3. Operational and axiomatic formalisation of x86-TSO [4]

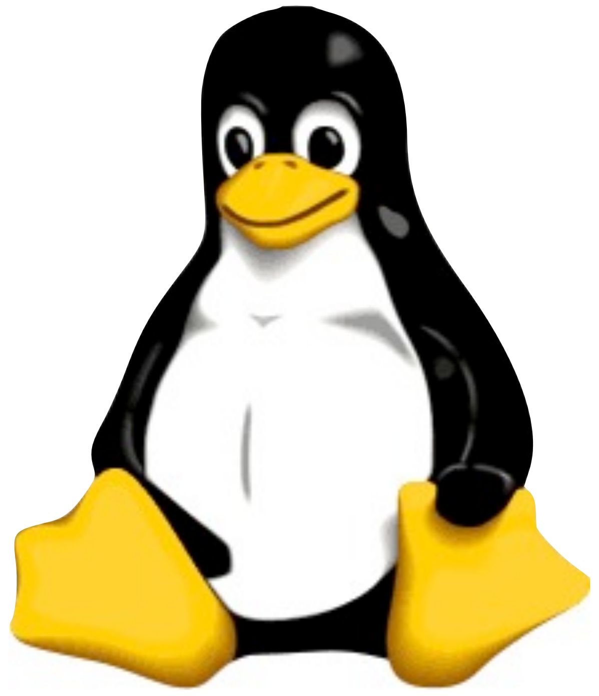
4. Fence optimisations for x86-TSO [4]

5. The Java memory model [4]

6. The C11/C++11 memory model [17]

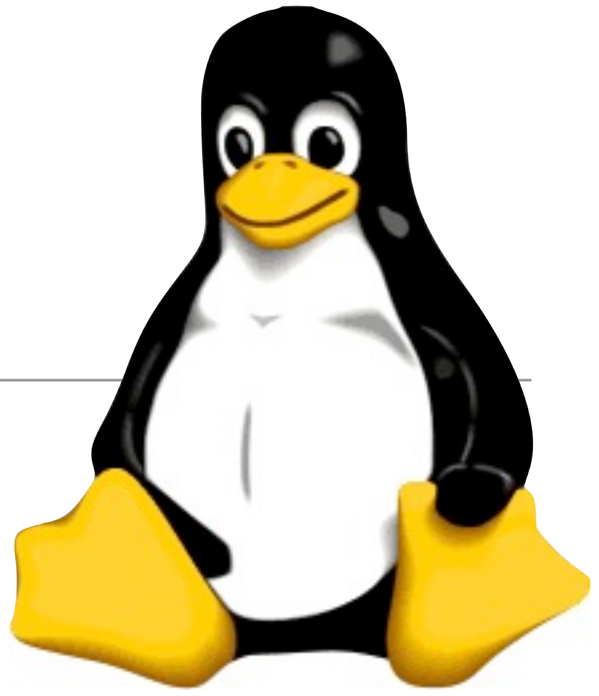
7. Static and dynamic techniques for data-race detection [7]

8. The Linux memory model (?!) [18]



1. The Linux memory model (ahem, kinda)

The Linux memory model



Facts:

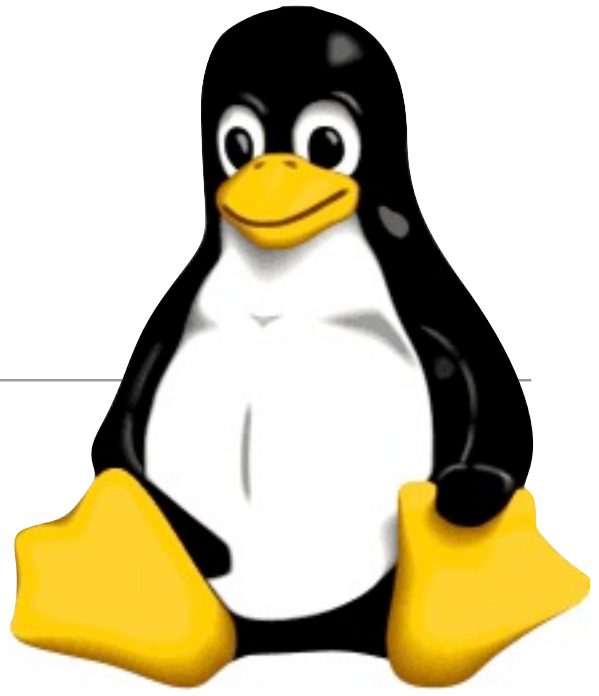
- abstraction layer over hardware and compilers

- relied upon by kernel developers to write "portable kernel code"

- documented by a text file:

`http://www.kernel.org/doc/Documentation/memory-barriers.txt`

The Linux memory model



Facts:

- abstraction layer over hardware and compilers

- relied upon by kernel developers to write "portable kernel code"

- documented by a text file:

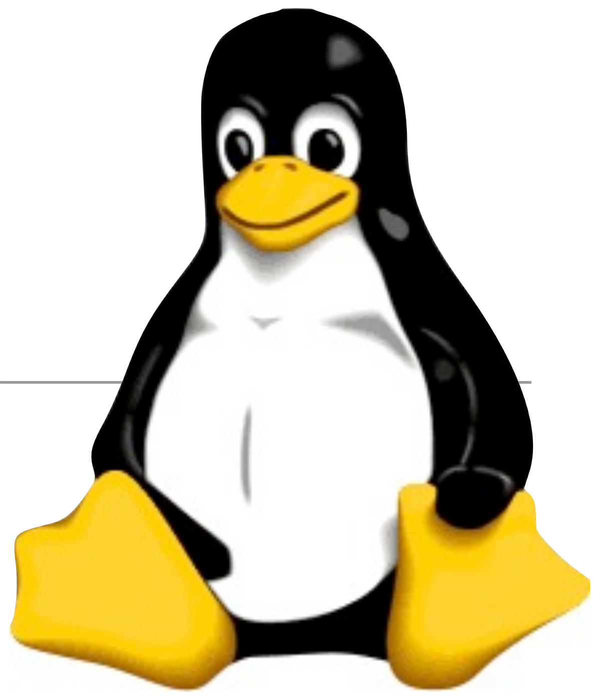
`http://www.kernel.org/doc/Documentation/memory-barriers.txt`

More facts: ...some time ago...

I attempted to understand the doc, and exchanged a few email with Paul Mc Kenney. However I don't understand much…

In the next hour, let's go over the documentation together and see if we can make sense of it...

The Linux memory model



Expected to account for all supported combinations of compiler and hardware memory model...

$$\begin{tikzcd}
\text{Linux kernel} \arrow[d, leftrightarrow, "\textit{compiler memory-model (gcc)}"] \arrow[dd, leftrightarrow, bend right, "\textit{Linux memory model}"'] \\
\text{Compiler} \arrow[d, leftrightarrow, "\textit{hardware memory-model}"] \\
\text{Hardware}
\end{tikzcd}$$

alpha: Weak ordering. No dependency ordering. "Time does not go backwards" gives guarantees similar to Power/ARM A-cumulativity. Possibly B-cumulativity as well. I am not aware of formalization of this architecture's memory ordering other than Gharachorloo's PhD.

arm: You know at least as much as I do about this one.

avr32: Uniprocessor-only, kernel build failure for SMP.

blackfin: Uniprocessor-only to the best of my knowledge. There are rumored to be some experimental SMP systems that lack cache coherence, and are thus outside of the Linux kernel's remit. See for example: https://docs.blackfin.uclinux.org/doku.php?id=linux-kernel:smp-like The system.h file flushes cache when a memory barrier is encountered, which is consistent with an attempt to run the Linux kernel on a non-cache-coherent system…

cris: Uniprocessor-only to the best of my knowledge. Though there appears to be recent addition of some SMP support. Its system.h file is consistent with full sequential consistency. Or extreme optimism on the part of the cris developers.

frv: Uniprocessor-only to the best of my knowledge.

h8300: Uniprocessor-only to the best of my knowledge. There is code in system.h that appears to be intended for SMP, but it looks to me like a (harmless) copy-paste error. Either that or SMP h8300 systems are sequentially consistent.

ia64: Total order of all release operations, which include the "mf" (memory fence) instruction. Memory fences cannot restore sequential consistency.

m32r: Uniprocessor-only to the best of my knowledge. However, there does appear to be some recent multiprocessor support. This is quite strange -- atomic instructions flush cache, but memory barriers are no-ops. Looks quite experimental.

m68k: Uniprocessor-only to the best of my knowledge.

microblaze: Uniprocessor-only to the best of my knowledge. At least one SMP attempt: http://microblazesmp.blogspot.com/ Its system.h file looks uniprocessor-only.

mips: Multiprocessor. Old SGI MIPS systems were sequentially consistent. Newer systems used for network infrastructure are rumored to have weak memory models similar to Power and ARM. And its system.h file is consistent with a weak memory model.

mn10300: Recent SMP support which I know little about. The system.h file looks uniprocessor only, and contains comments on Intel, so copy-pasted from x86.

parisc: TSO, similar to x86.

powerpc: You know at least as much about this as I do.

s390: TSO, but with self-snooping of store buffer prohibited.

score: Uniprocessor-only to the best of my knowledge.

sh: Recent SMP support which I know little about. Its system.h file is consistent with weak memory ordering.

sparc: TSO, similar to x86. There is documentation about weaker memory models (PSO and RMO), but in practice the hardware is TSO.

tile: Recent SMP CPU which I know little about. Seems to be weakly ordered based on its system.h file.

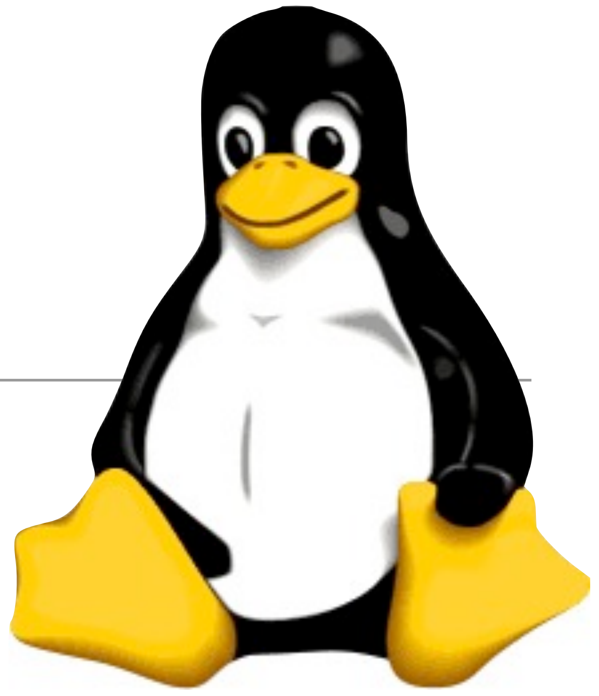
um: Looks like an x86 knockoff judging by the system.h file.

unicore32: Uniprocessor-only to the best of my knowledge.

x86: You know this one at least as well as do I.

xtensa: Uniprocessor-only -- kernel build failure otherwise.

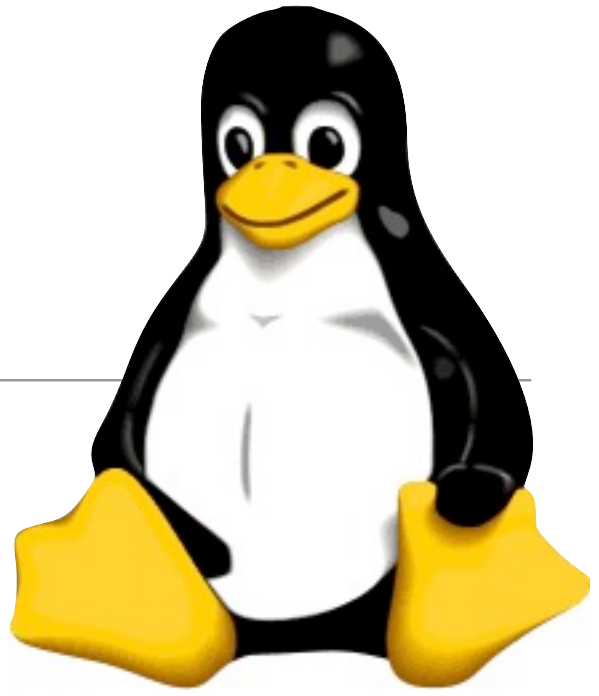
The Linux memory model



My intuition:

Annoying facts:

The Linux memory model



My intuition:

kinda of lowest common denominator between all hardware memory models of architectures Linux can be compiled to, taking into account also some common gcc optimisations, with some weirdnesses.

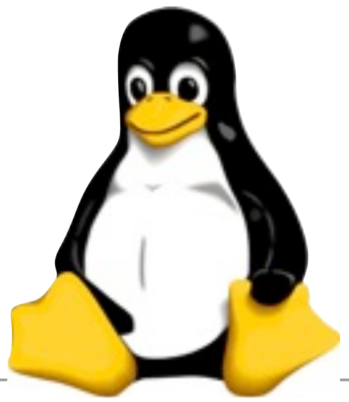
Annoying facts:

semantics of "read barriers" really weak, unclear how to formalise it

compilation of barriers on Itanium looks broken -- hardware might exhibit behaviours prohibited by the MM.

...let's read the doc...

The Linux memory model: macros



on x86:

```
#define mb()   asm volatile("mfence":::"memory")
#define rmb()  asm volatile("lfence":::"memory")
#define wmb()  asm volatile("sfence" ::: "memory")
```

in x86TSO lfence is a noop and sfence is like mfence, but things are different in kernel land, eg when performing dma accesses.

on Power:

```
#define mb()   __asm__ __volatile__ ("sync" : : : "memory")
#define rmb()  __asm__ __volatile__ ("sync" : : : "memory")
#define wmb()  __asm__ __volatile__ ("sync" : : : "memory")
#define read_barrier_depends()  do { } while(0)
```

So I still stick with my earlier statements:

o smp_mb() provides transitivity, but is not guaranteed to restore

o smp

o smp

Question:

Initially x=

Thread 0:

Thread 1:

Thread 2:

to be forbi

And a few emails exchanged last year...

Not forbidden. If thread 2 did smp_mb() instead of smp_rmb(), then it would be forbidden.

So I still stick with my earlier statements:

- o smp_mb() provides transitivity, but is not guaranteed to restore sequential consistency.
- o smp_rmb() simply orders reads. It does not provide transitivity.
- o smp_wmb() simply orders writes. It does not provide transitivity.

Question: is WRC+smp_mb+smp_rmb, i.e.

Initially x=0, y=0
Thread 0: Wx1
Thread 1: Rx1; smp_mb(); Wy1
Thread 2: Ry1; smp_rmb();Rx0

to be forbidden or not?

Not forbidden. If thread 2 did smp_mb() instead of smp_rmb(), then it would be forbidden.

On Itanium, both `rmb` and `mb` are compiled to Itanium's `mf`, so there should be no difference in outcome.

However, looking at the Itanium memory model, I do not see how Itanium would forbid the bad outcome in WRC+mb+mb because mf only imposes thread-local ordering, so thread 2 can see Wx1 much later (in particular, after the read of x).

To make the example work, the Wx1 would have to be `st.rel`, no?

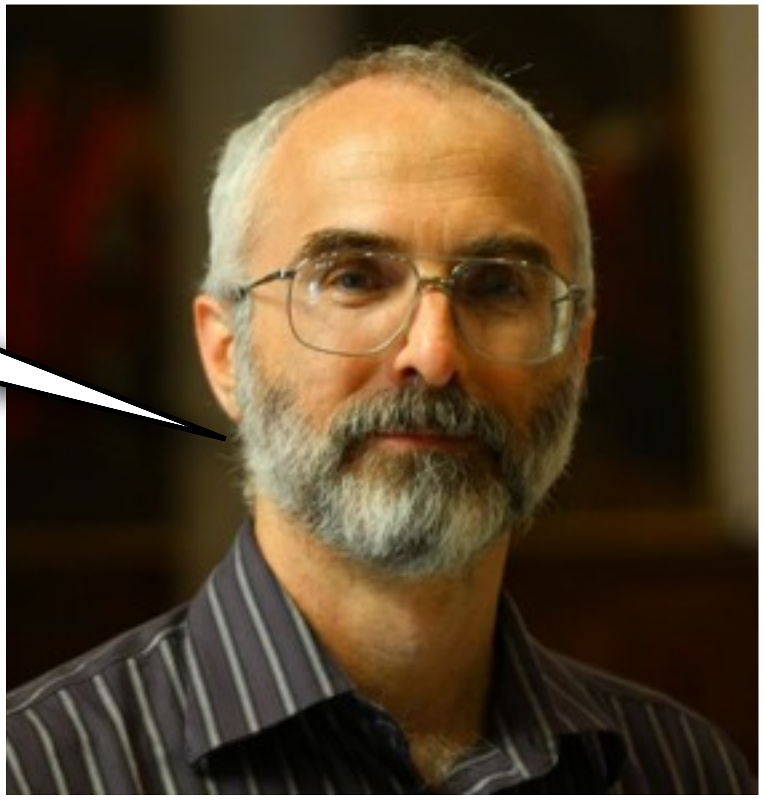
If this is the case, I suspect that the Linux kernel has a few possible failure modes when running on Itanium hardware. Which it might well have...

...it looks like you might need a significantly different linux mm to the one you've sketched, with weaker barriers and with release/acquire primitives, and to rewrite any WRC-like code using them, no?

Challenging research direction:

Sort out what the REAL Linux memory model is

Yes. Of course, if people come up with lots of situations where the more-complex programming model would help significantly, then it might be worth revisiting this.



Actually: how to design a high-level programming language memory model that does not assign undefined behaviour to racy programs?

2. The C++11 memory model

a good example of an axiomatic memory model

The C++11 memory model

1300 page prose specification defined by the ISO.

The design is a detailed compromise:

- hardware/compiler implementability
- useful abstractions
- broad spectrum of programmers

Welcome to the official home of

JTC1/SC22/WG21 - The C++ Standards Committee

2011-09-15: standards | projects | papers | mailings | internals | meetings | contacts

News 2011-09-11: The new C++ standard - C++11 - is published!

The syntactic divide

```
// for regular programmers:
atomic_int x = 0;
x.store(1);
y = x.load();

// for experts:
x.store(2, memory_order);
y = x.load(memory_order);
atomic_thread_fence(memory_order);
```

where *`memory_order`* is one of the following:

```
  mo_seq_cst  mo_release  mo_acquire
  mo_acq_rel  mo_consume  mo_relaxed
```

How may a program execute?

Two layer semantics:

1) a denotational semantics processes programs, identifying memory actions, and constructs candidate executions (E_{opsem});

$$P \longrightarrow E_1, \dots , E_n$$

2) an axiomatic memory model judges E_{opsem} paired with a memory ordering $X_{witness}$

$$E_i \longrightarrow X_{i1},\ldots,X_{im}$$

3) searches the consistent executions for races and uncostrained reads

is there an X_{ij} with a race?

Relations

An E_{opsem} part containing:

sb	sequenced before, program order
asw	additional synchronizes with, inter-thread ordering

An X_{witness} part containing:

rf	relates a write to any reads that take its value
sc	a total order over mo_seq_cst and mutex actions
mo	modification order, per location total order of writes

From these, compute synchronise-with (*sw*) and happens-before (*hb*).

We ignore *consume* atomics, which enables us to live in a simplified model.

Full details in Batty et al., POPL 11.

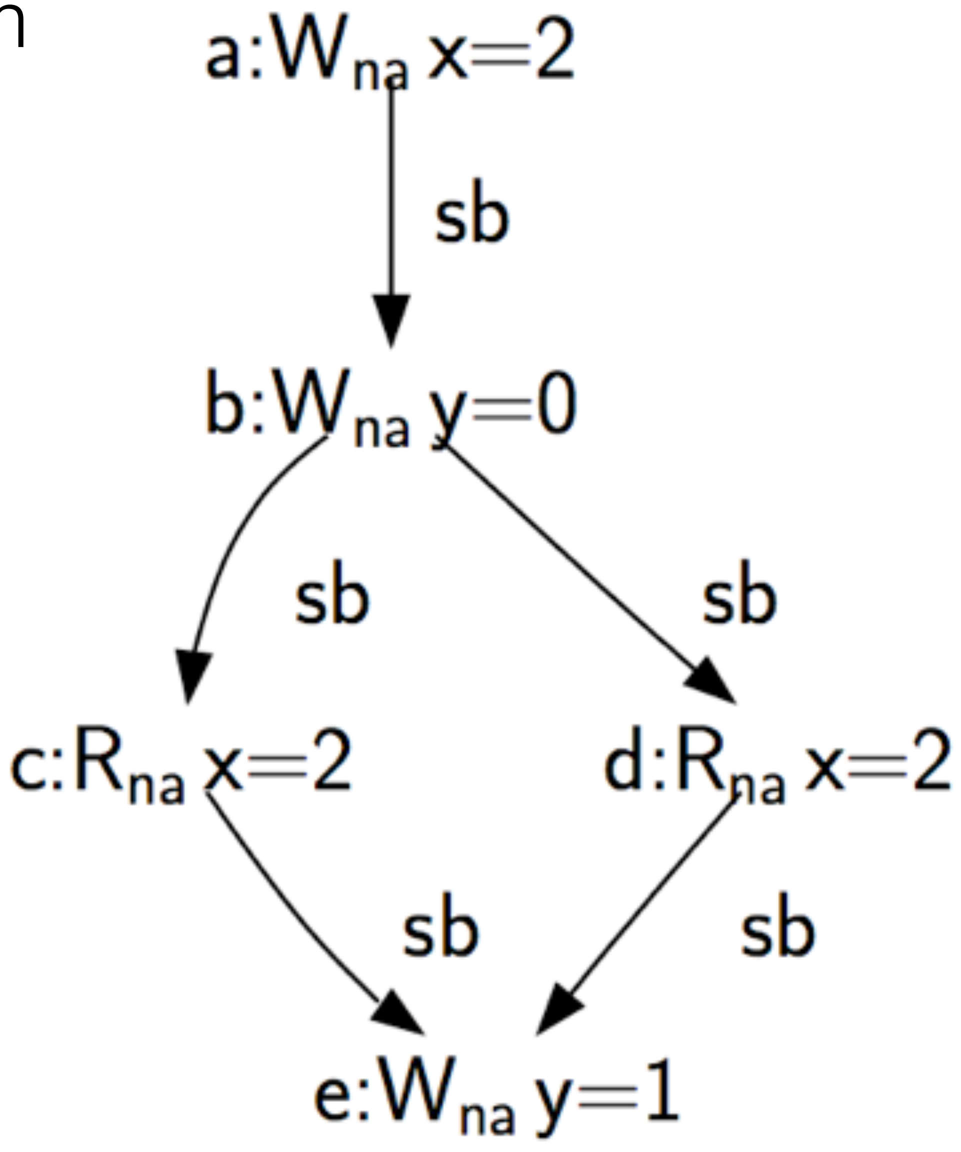
Formally

```
cpp_memory_model_opsem (p : program) =
let pre_executions =
  {(E_opsem,X_witness).  opsem p E_opsem ∧
    consistent execution (E_opsem, X_witness)}
in
if ∃X ∈ pre executions.
   (indeterminate reads X = {}) ∨
   (unsequenced races X = {}) ∨
   (data races X = {})
then NONE
else SOME pre_executions
```

A single-threaded example

1. sequenced before (sb) - given by opsem

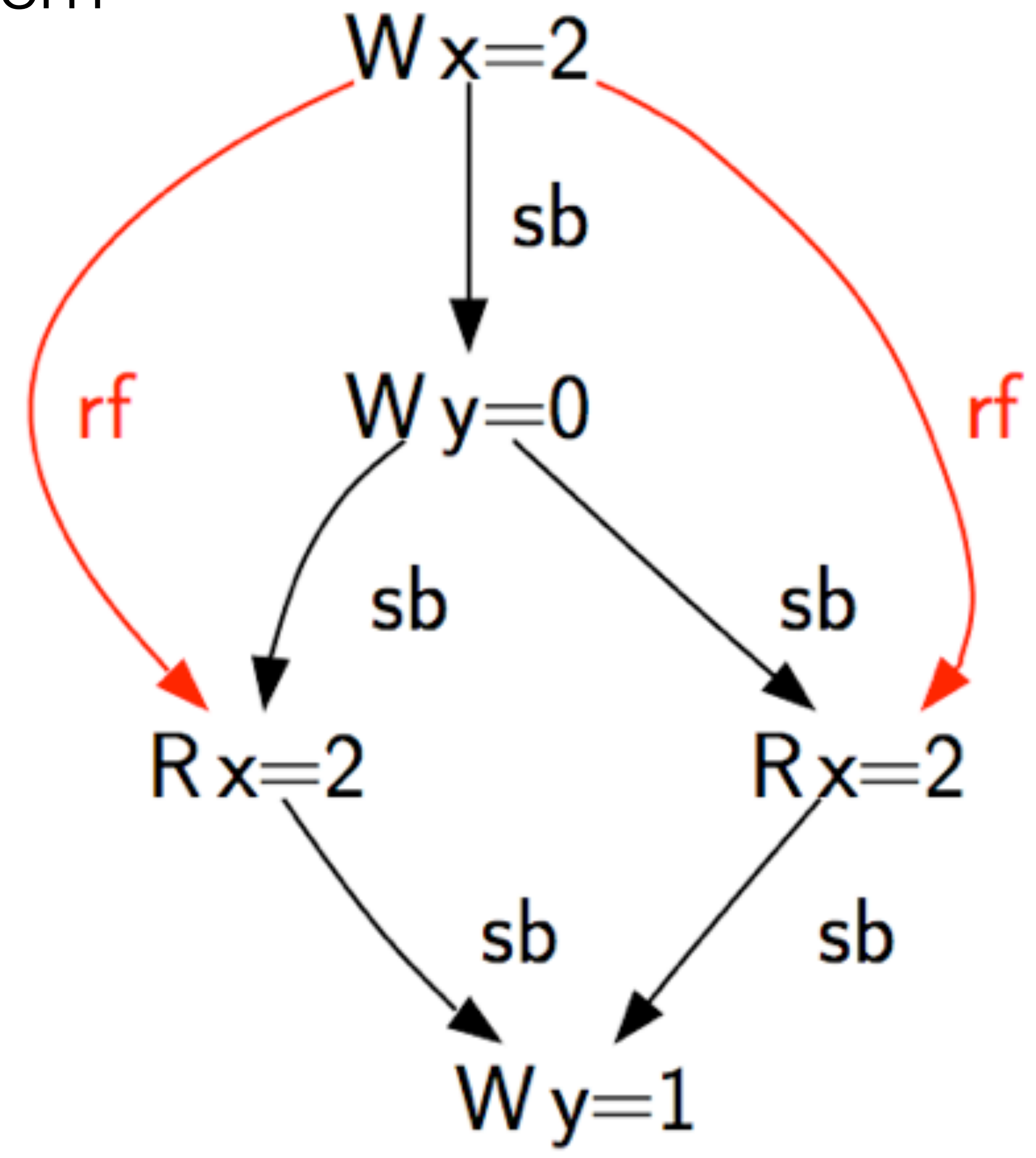
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



A single-threaded example

1. sequenced before (sb) - given by opsem
2. read-from (rf) - part of the witness

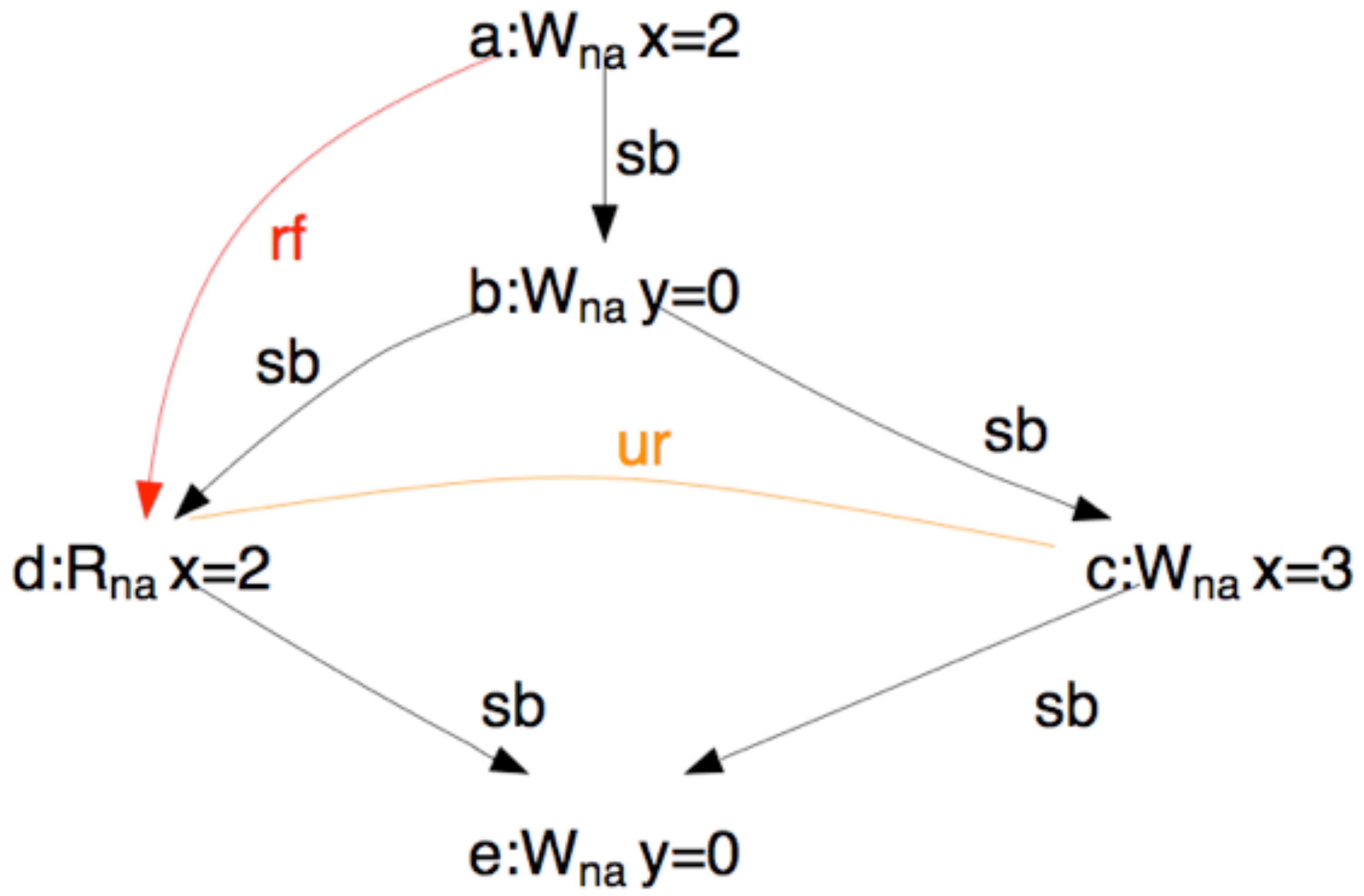
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



A single-threaded ex. with undefined behaviour

An unsequenced race.

```
int main() {
  int x = 2;
  int y = 0;
  y = (x==(x=3));
  return 0;
}
```



A simple concurrent program

```
int y, x = 2;
x = 3;                | y = (x==3);
```

$$\begin{tikzcd}
a{:}W_{na}\ x{=}2 \arrow[dr, "asw"'] \arrow[drr, "{asw,rf}"] & & \\
 & b{:}W_{na}\ x{=}3 & c{:}R_{na}\ x{=}2 \arrow[d, "sb"'] \\
 & & d{:}W_{na}\ y{=}0
\end{tikzcd}$$

We will omit `asw` arrows whenever we are not interested in the initialisation.

Locks and unlocks

```
int x, r;
mutex m;
m.lock();          | m.lock();
x = ...            | r = x;
m.unlock();        |
```

1. the operational semantics defines the sb arrows

$$\begin{tikzcd}
\text{c:L mutex} \arrow[d, "\text{sb}"'] & \text{h:L mutex} \arrow[d, "\text{sb}"'] \\
\text{d:}W_{na}\ x=1 \arrow[d, "\text{sb}"'] & \text{i:}R_{na}\ x=1 \\
\text{f:U mutex} &
\end{tikzcd}$$

Locks and unlocks

```
int x, r;
mutex m;
m.lock();          | m.lock();
x = ...            | r = x;
m.unlock();        |
```

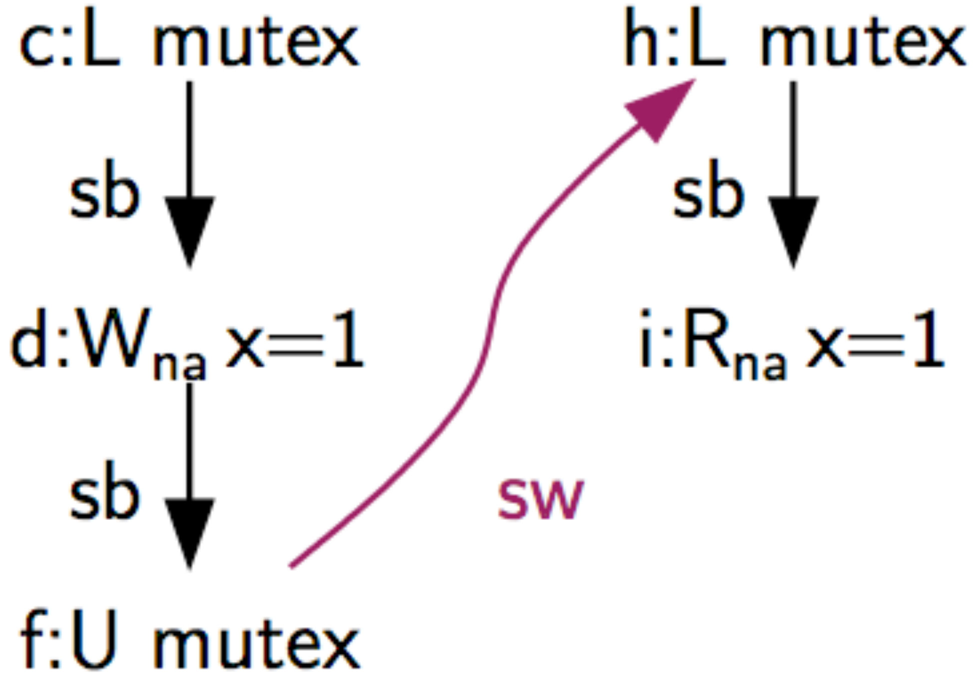
1. the operational semantics defines the sb arrows
2. guess an sc order on Unlock/Lock actions (part of the witness)

$$\begin{tikzcd}
c{:}L\ \mathsf{mutex} \arrow[d, "sb"'] & h{:}L\ \mathsf{mutex} \arrow[d, "sb"'] \\
d{:}W_{na}\ x=1 \arrow[d, "sb"'] & i{:}R_{na}\ x=1 \\
f{:}U\ \mathsf{mutex} \arrow[uur, bend right, "sc"'] &
\end{tikzcd}$$

Locks and unlocks

```
int x, r;
mutex m;
m.lock();          | m.lock();
x = ...            | r = x;
m.unlock();        |
```

1. the operational semantics defines the sb arrows
2. guess an sc order on Unlock/Lock actions (part of the witness)
3. the sc order is included in the syncronised-with relation


$$\begin{tikzcd}
\text{c:L mutex} \arrow[d, "sb"'] & \text{h:L mutex} \arrow[d, "sb"'] \\
\text{d:}W_{na}\ x{=}1 \arrow[d, "sb"'] & \text{i:}R_{na}\ x{=}1 \\
\text{f:U mutex} \arrow[uur, bend right, "sw"'] &
\end{tikzcd}$$

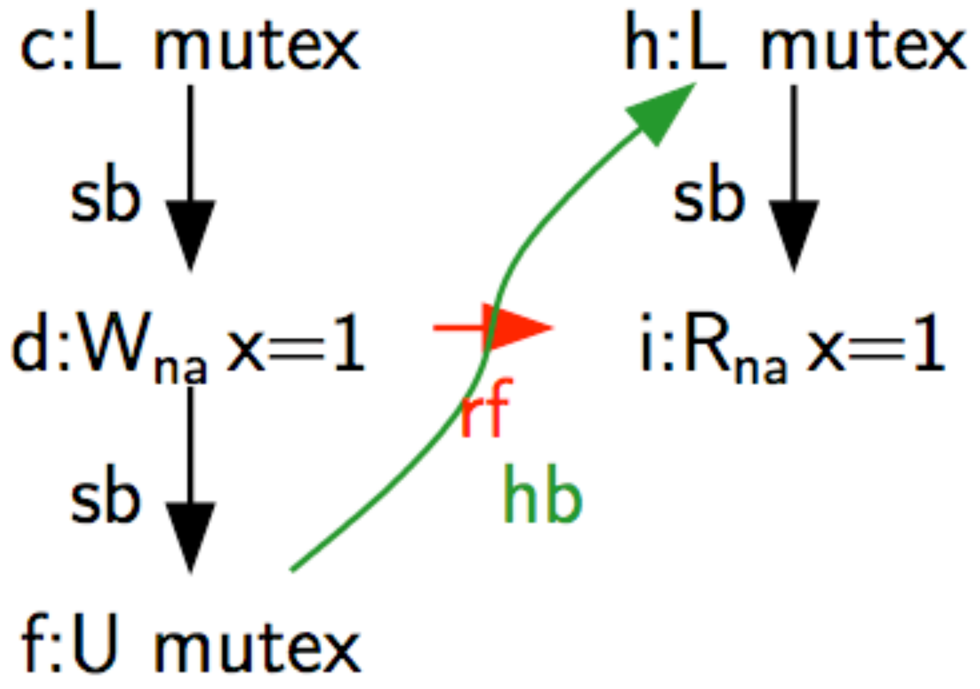
Locks and unlocks

$$\xrightarrow{\textit{simple-happens-before}} = \left(\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}}\right)^+$$

```
int x, r;
mutex m;
```

```
m.lock();          | m.lock();
x = ...            | r = x;
m.unlock();        |
```

1. the operational semantics defines the sb arrows
2. guess an sc order on Unlock/Lock actions (part of the witness)
3. the sc order is included in the syncronised-with relation
4. which in turn defines the happens-before relation...



Happens before

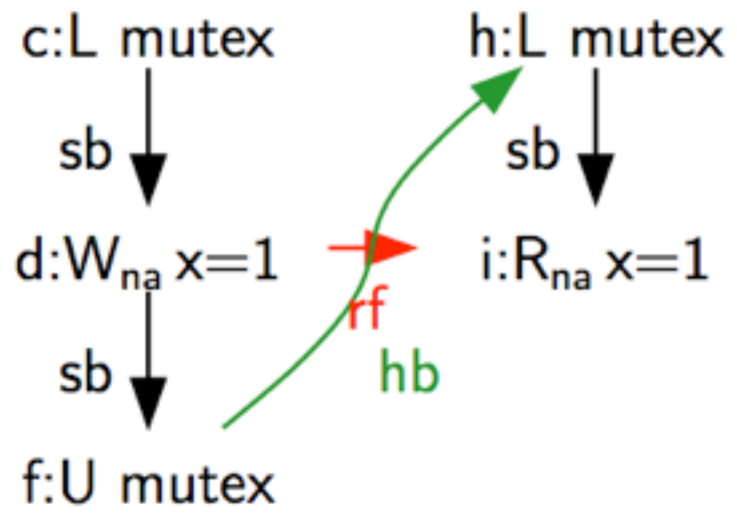
The *happens before* relation is key to the model:

1. non-atomic loads read the most recent write in happens before.
   (This is unique in DRF programs)

2. the story is more complex for atomics, as we shall see.

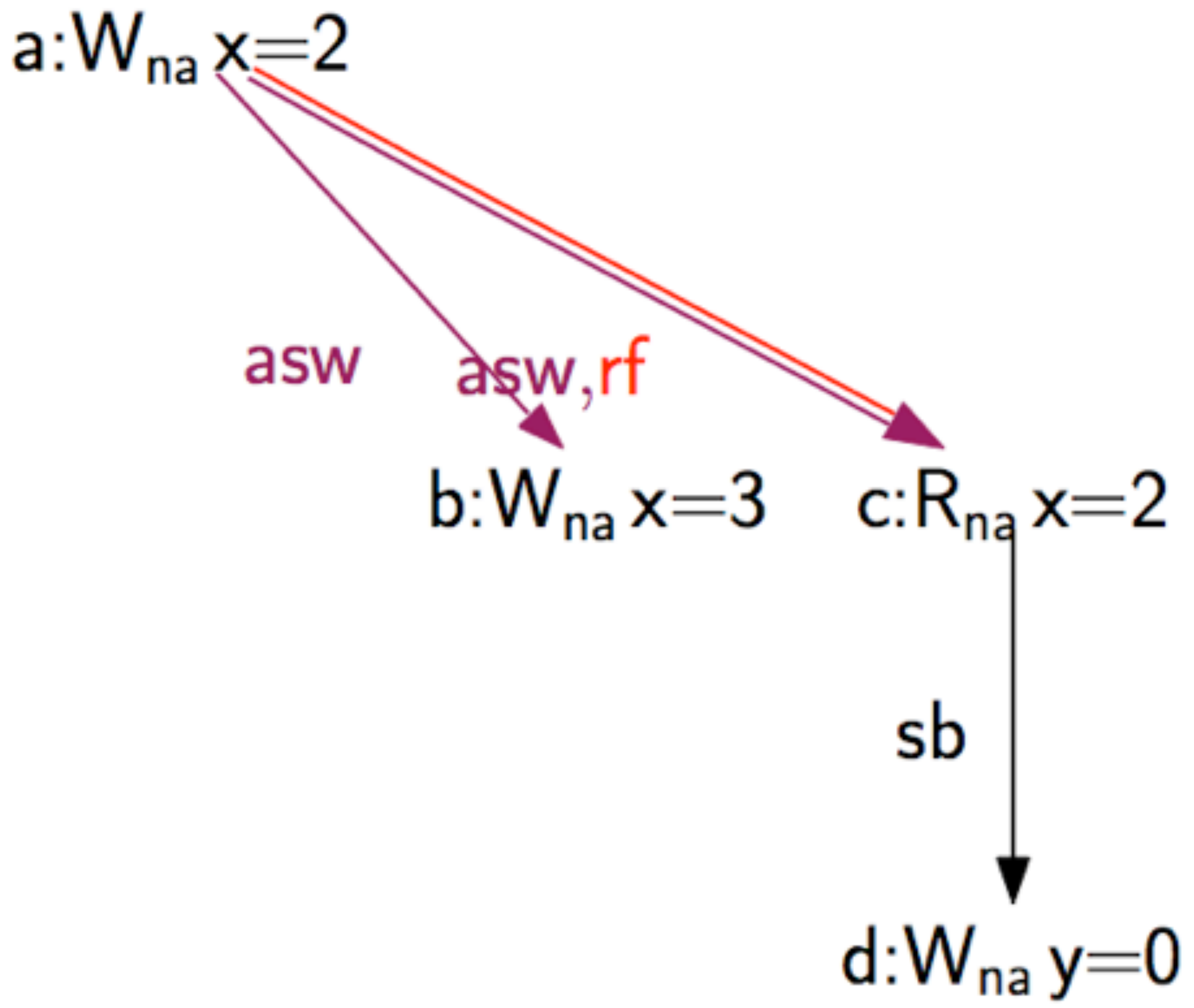
3. data races are defined as an absence of happens before
   between conflicting actions.

$$\xrightarrow{\textit{simple-happens-before}} = \left(\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}}\right)^{+}$$



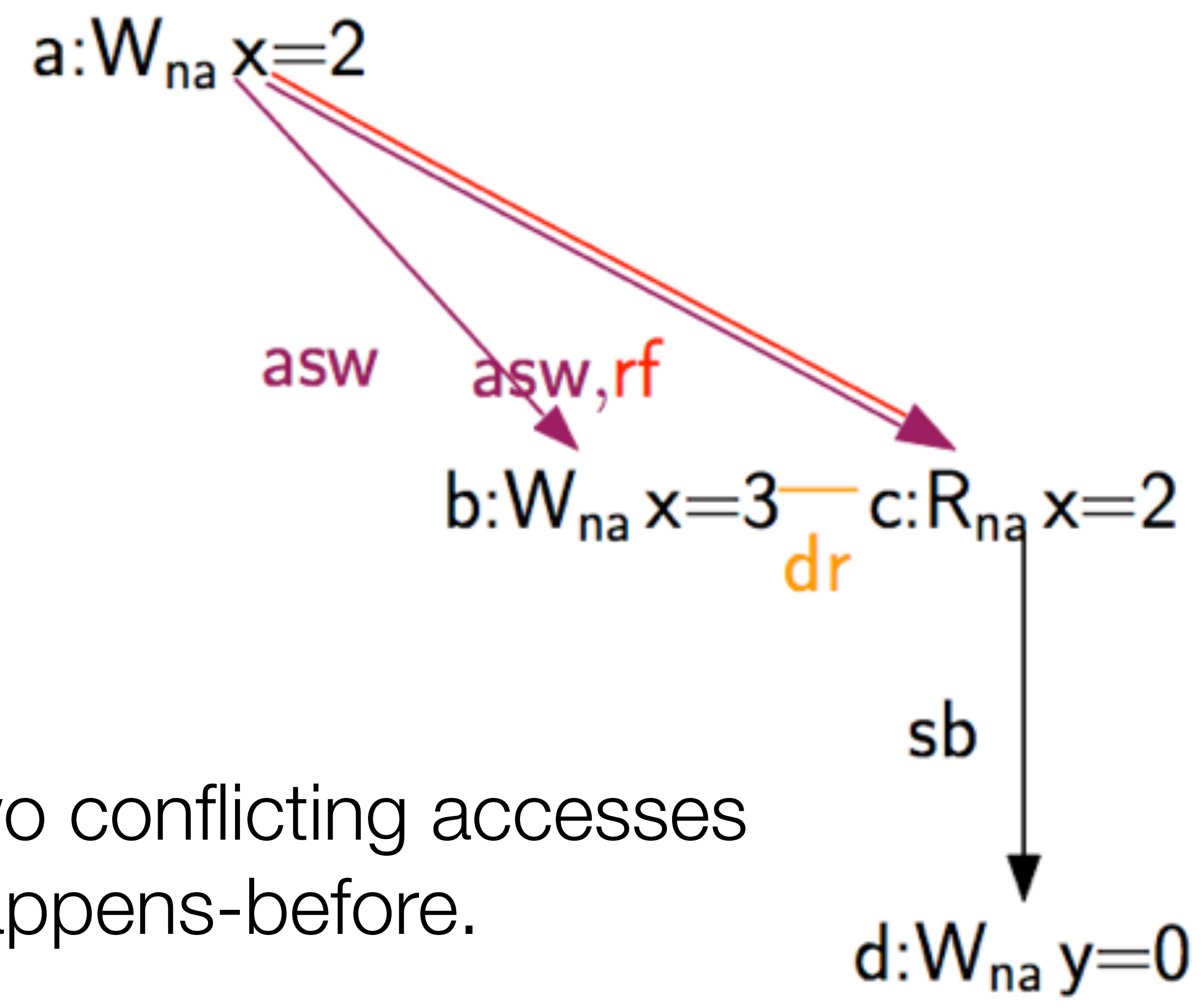
A data race

```
int y, x = 2;
x = 3;                 | y = (x==3);
```



A data race

```
int y, x = 2;
x = 3;                    | y = (x==3);
```



Here we have two conflicting accesses not related by happens-before.

Data race definition

```
let $data\_races\ actions\ hb =$
    $\{\ (a,\ b)\ \mid\ \forall\ a \in actions\ b \in actions\ \mid$
        $\neg\ (a = b)\ \wedge$
        $\mathrm{same\_location}\ a\ b\ \wedge$
        $(\mathrm{is\_write}\ a\ \vee\ \mathrm{is\_write}\ b)\ \wedge$
        $\neg\ (\mathrm{same\_thread}\ a\ b)\ \wedge$
        $\neg\ (\mathrm{is\_atomic\_action}\ a\ \wedge\ \mathrm{is\_atomic\_action}\ b)\ \wedge$
        $\neg\ ((a,\ b) \in hb\ \vee\ (b,\ a) \in hb)\ \}$
```

Programs with a data race have undefined behaviour (DRF model).

Simple concurrency: Dekker's example and SC

```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  | y.store(1, seq_cst);
y.load(seq_cst);      | x.load(seq_cst);
```

$$\begin{tikzcd}
c{:}W_{sc}\ y=1 \arrow[d, "sb"'] & \textcolor{red}{\text{FORBIDDEN}} & e{:}W_{sc}\ x=1 \arrow[d, "sb"'] \\
d{:}R_{sc}\ x=0 & & f{:}R_{sc}\ y=0
\end{tikzcd}$$

Why is this behaviour forbidden?

Simple concurrency, Dekker's example and SC

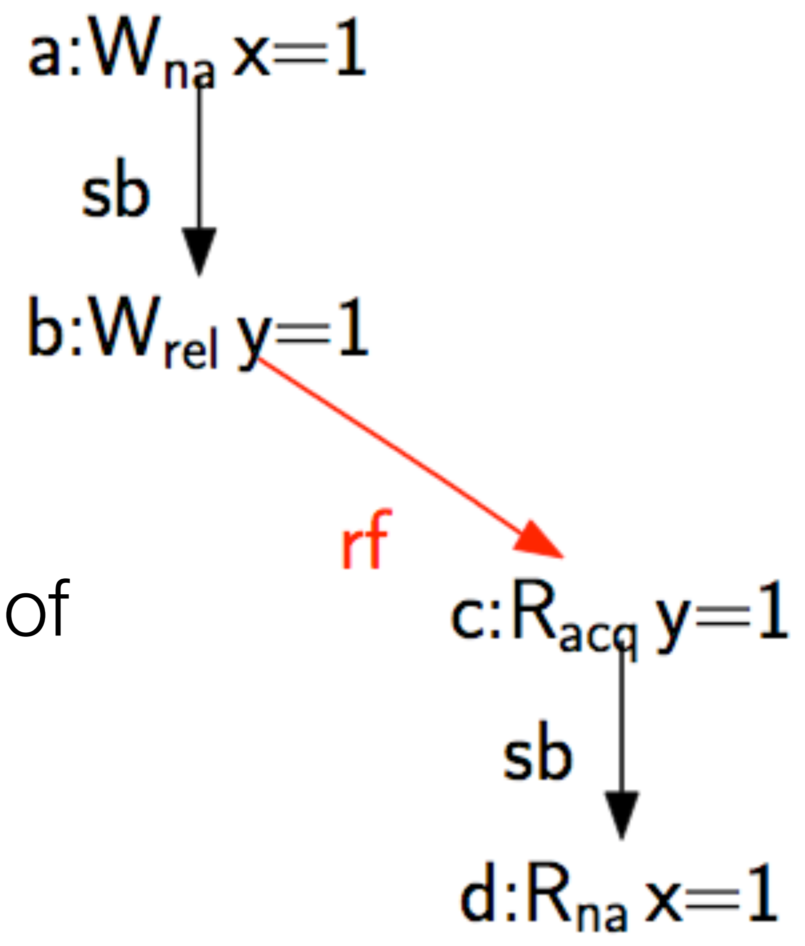
```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  | y.store(1, seq_cst);
y.load(seq_cst);      | x.load(seq_cst);
```

$$\begin{tikzcd}
c{:}W_{sc}\ y=1 \arrow[d, "sc"'] & e{:}W_{sc}\ x=1 \arrow[d, "sc"'] \\
d{:}R_{sc}\ x=0 \arrow[ur, "sc"'] & f{:}R_{sc}\ y=1
\end{tikzcd}$$

The `sc` relation must define a total order over unlocks/locks and `seq_cst` accesses... `sc` is included in `hb`, an `rf` must respect `hb`.

Expert concurrency: the release-acquire idiom

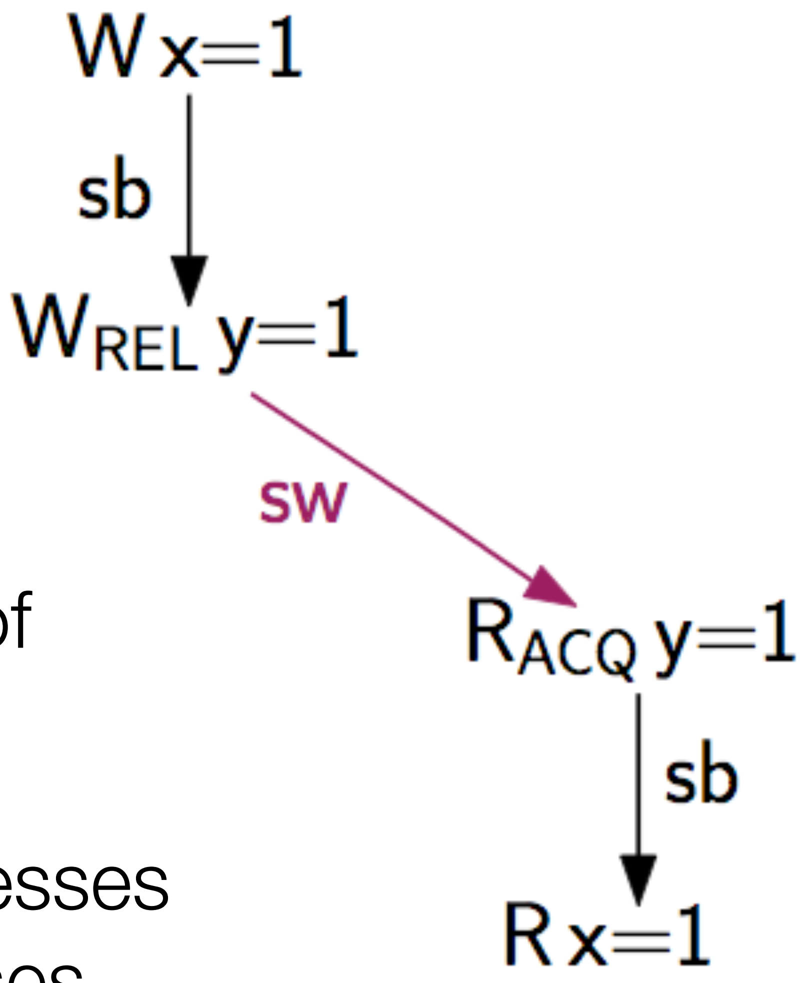
```
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
```



Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

Expert concurrency: the release-acquire idiom

```
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
```

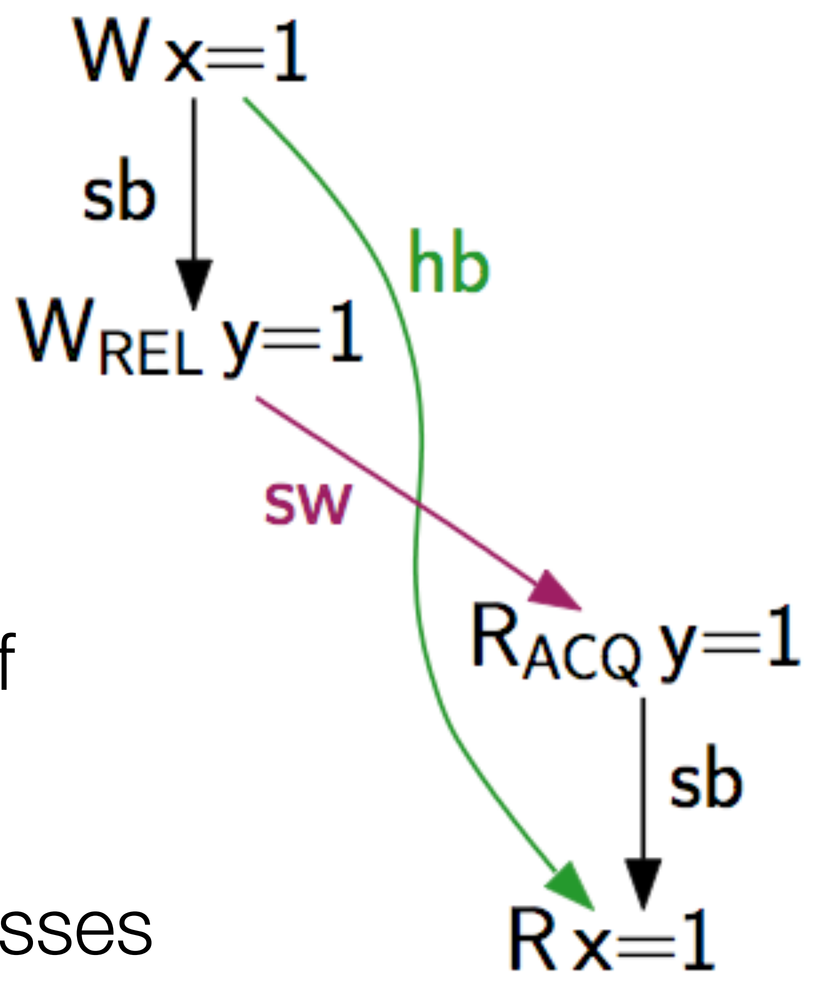


Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

The `rf` arrow beetwen release/acquire accesses induces an `sw` arrow between those accesses.

Expert concurrency: the release-acquire idiom

```
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
```



Here we have an **rf** arrow beetwen a pair of release/acquire accesses.

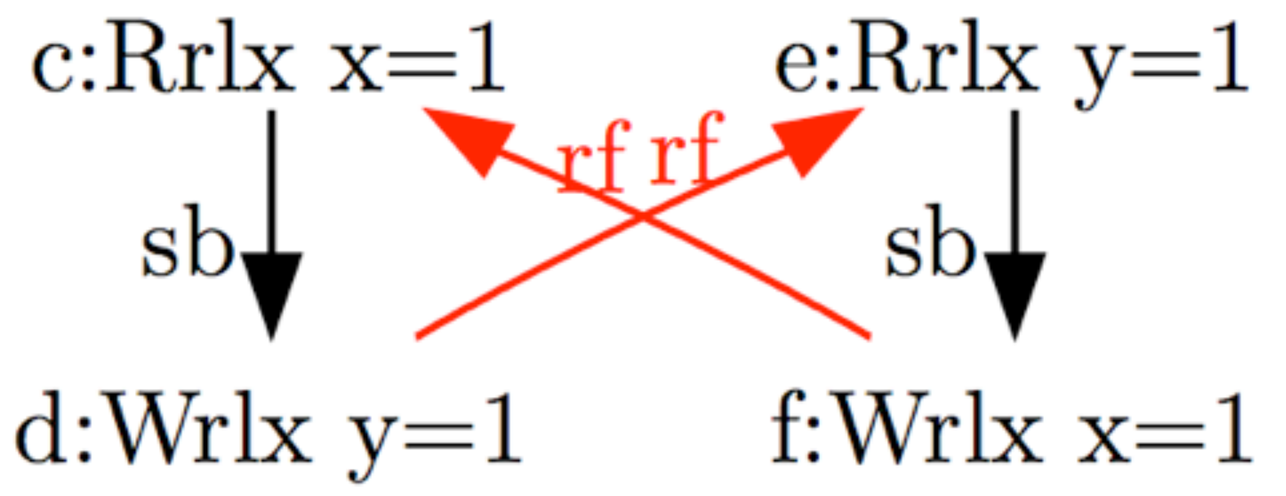
The **rf** arrow beetwen release/acquire accesses induces an **sw** arrow between those accesses.

And in turn defines an **hb** constraint.

$$\xrightarrow{\textit{simple-happens-before}} = \left(\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}}\right)^+$$

Relaxed writes

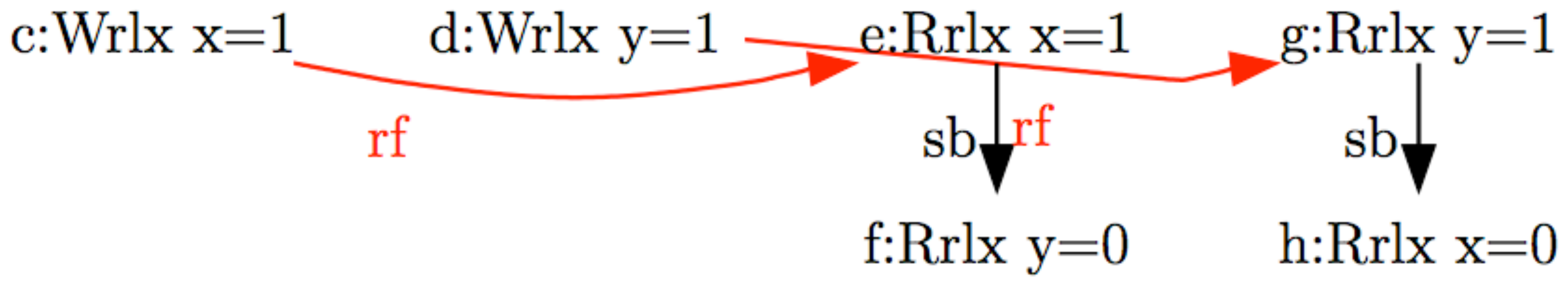
```
x.load(relaxed);          | y.load(relaxed);
y.store(1, relaxed);      | x.store(1, relaxed);
```


$$\begin{tikzcd} \text{c:Rrlx x=1} \arrow[d, "\text{sb}"'] & \text{e:Rrlx y=1} \arrow[d, "\text{sb}"'] \\ \text{d:Wrlx y=1} \arrow[ur, "\text{rf}"] & \text{f:Wrlx x=1} \arrow[ul, "\text{rf}"'] \end{tikzcd}$$

No data-races, no synchronisation cost, but weakly ordered.

Relaxed writes, ctd.

```
atomic_int x = 0;
atomic_int y = 0;
x.store(1, relaxed); | y.store(2, relaxed); | x.load(relaxed); | y.load(relaxed);
                     |                      | y.load(relaxed); | x.load(relaxed);
```



Again, no data-races, no synchronisation cost, but weakly ordered (IRIW).

Expert concurrency: fences avoid excess sync.

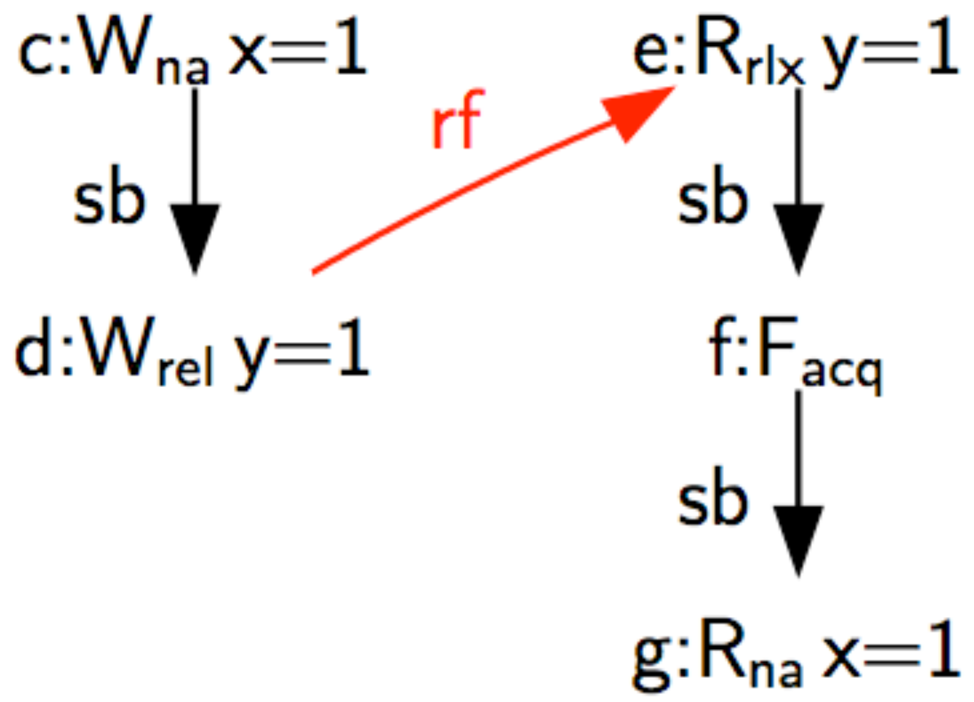
```
// sender                    | // receiver
x = ...                      | while (0 == y.load(acquire));
y.store(1, release);         | r = x;
```

```
// sender                    | // receiver
x = ...                      | while (0 == y.load(relaxed));
y.store(1, release);         | fence(acquire);
                             | r = x;
```

Expert concurrency: fences avoid excess sync.

```
// sender                    | // receiver
x = ...                      | while (0 == y.load(relaxed));
y.store(1, release);         | fence(acquire);
                             | r = x;
```

Here we have an `rf` arrow beetwen a release write and a relaxed write.

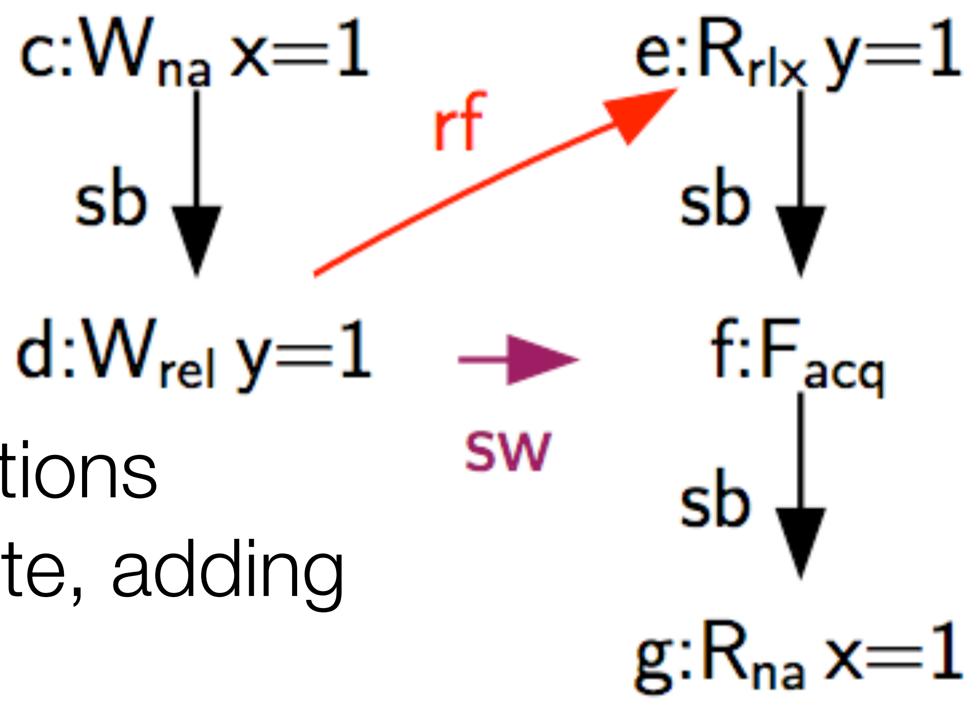

$$\begin{tikzcd}
c{:}W_{na}\ x{=}1 \arrow[d, "sb"'] & e{:}R_{rlx}\ y{=}1 \arrow[d, "sb"'] \\
d{:}W_{rel}\ y{=}1 \arrow[ur, "rf"] & f{:}F_{acq} \arrow[d, "sb"'] \\
 & g{:}R_{na}\ x{=}1
\end{tikzcd}$$

Expert concurrency: fences avoid excess sync.

```
// sender
x = ...
y.store(1, release);
```

```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```

Here we have an `rf` arrow beetwen a release write and a relaxed write.

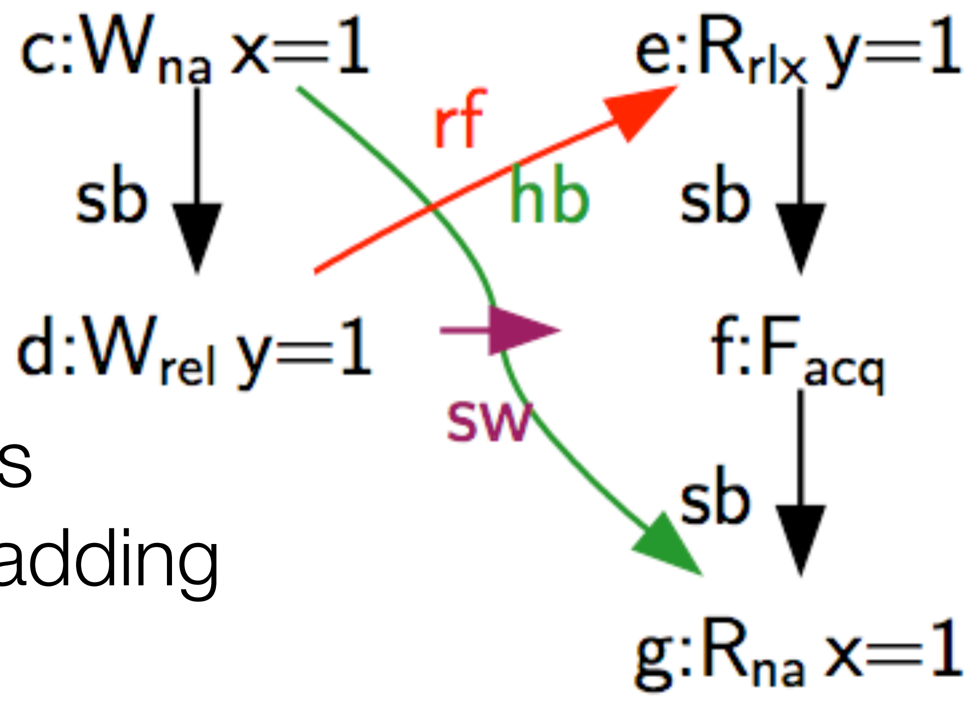

$$\begin{tikzcd}
c{:}W_{na}\ x{=}1 \arrow[d, "sb"'] & e{:}R_{rlx}\ y{=}1 \arrow[d, "sb"] \\
d{:}W_{rel}\ y{=}1 \arrow[ur, "rf"] \arrow[r, "sw"'] & f{:}F_{acq} \arrow[d, "sb"] \\
& g{:}R_{na}\ x{=}1
\end{tikzcd}$$

The acquire fence follows the `sb`/`rf` relations looking for the corresponding release write, adding a `sw` arrow.

Expert concurrency: fences avoid excess sync.

```
// sender                    // receiver
x = ...                      while (0 == y.load(relaxed));
y.store(1, release);         fence(acquire);
                             r = x;
```

Here we have an `rf` arrow beetwen a release write and a relaxed write.



The acquire fence follows the `sb`/`rf` relations looking for the corresponding release write, adding a `sw` arrow.

Happens-before follows as usual...

Modification order

```
   atomic_int x = 0;
x.store(1, relaxed); ││  x.load(relaxed);
x.store(2, relaxed); ││  x.load(relaxed);
```

$$\begin{tikzcd}
W_{RLX}\ x=1 \arrow[r, "rf"'] \arrow[d, "mo"'] & R_{RLX}\ x=1 \arrow[d, "sb"'] \\
W_{RLX}\ x=2 \arrow[r, "rf"'] & R_{RLX}\ x=2
\end{tikzcd}$$

Modification order is a total order over atomic writes of any memory order.

Coherence and atomic reads

All forbidden:

$$\begin{tikzcd}
\text{a:W x=1} \arrow[r, "\text{rf}"'] & \text{c:R x=1} \arrow[d, "\text{hb}"'] \\
\text{b:W x=2} \arrow[u, "\text{mo}"'] \arrow[r, "\text{rf}"'] & \text{d:R x=2}
\end{tikzcd}$$

CoRR

$$\begin{tikzcd}
\text{b:W x=2} \arrow[r, "\text{mo}"'] \arrow[dr, "\text{rf}"'] & \text{c:W x=1} \arrow[d, "\text{hb}"'] \\
 & \text{d:R x=2}
\end{tikzcd}$$

CoWR

$$\begin{tikzcd}
\text{a:W x=1} \arrow[d, shift right, "\text{hb}"'] \\
\text{b:W x=2} \arrow[u, shift right, "\text{mo}"']
\end{tikzcd}$$

CoWW

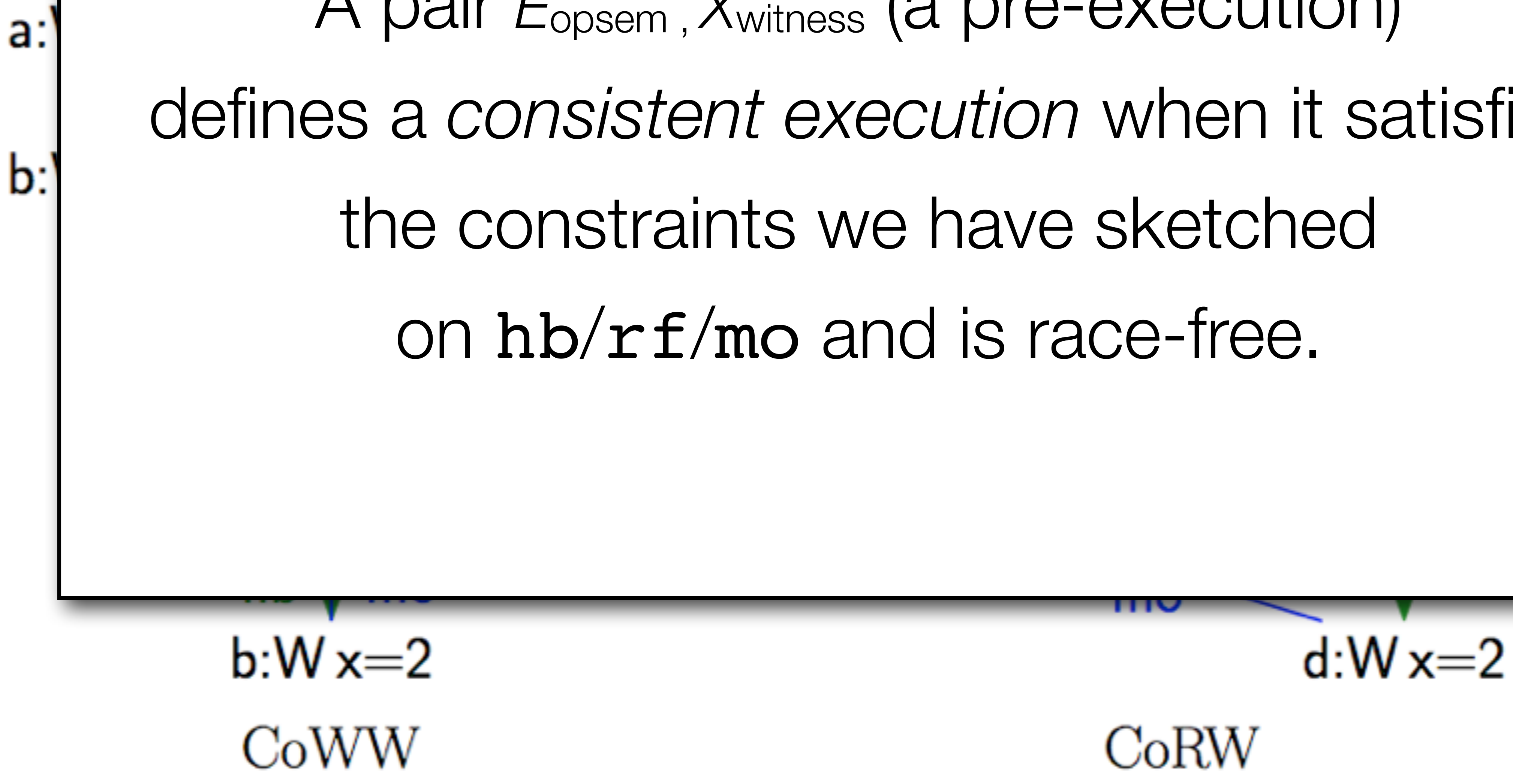
$$\begin{tikzcd}
\text{a:W x=1} \arrow[r, "\text{rf}"'] & \text{c:R x=1} \arrow[d, "\text{hb}"'] \\
 & \text{d:W x=2} \arrow[ul, "\text{mo}"']
\end{tikzcd}$$

CoRW

Idea: atomics cannot read from later writes in happens-before.

Coherence and atomic reads

All forb



A pair $E_{\text{opsem}}, X_{\text{witness}}$ (a pre-execution) defines a *consistent execution* when it satisfies the constraints we have sketched on `hb`/`rf`/`mo` and is race-free.

Idea: atomics cannot read from later writes in happens-before.

The full model

```
$a \xrightarrow{r} b = (a,b) \in r$
```

```
$a\ r\ b = (a,b) \in r$
```

```
$a \not\xrightarrow{r} b = (a,b) \notin r$
```

```
$\xrightarrow{r} = r$
```

```
$a \xrightarrow{r} b \xrightarrow{s} c = a \xrightarrow{r} b \wedge b \xrightarrow{s} c$
```

```
relation_over $s$ $rel$ = domain $rel \subseteq s \wedge$ range $rel \subseteq s$
```

```
$\xrightarrow{rel}|_s = rel \cap (s \times s)$
```

```
$rel|_s = rel \cap (s \times s)$
```

```
$\xrightarrow{rel}|_s = rel \cap (s \times s)$
```

```
$rel|_s = rel \cap (s \times s)$
```

```
strict_preorder $ord$ = irreflexive $ord$ $\wedge$ trans $ord$
```

```
total_over $s$ $ord$ =
  relation_over $s$ $ord$ $\wedge$
  $(\forall x \in s.\ \forall y \in s.\ x \xrightarrow{ord} y \vee y \xrightarrow{ord} x \vee (x = y))$
```

```
strict_total_order_over $s$ $ord$ =
  strict_preorder $ord$ $\wedge$ total_over $s$ $ord$
```

```
$x \xmapsto{ord}_{pred} y =$
  $pred\ x \wedge x \xrightarrow{ord} y \wedge \neg(\exists z.\ pred\ z \wedge x \xrightarrow{ord} z \xrightarrow{ord} y)$
```

```
$x \xmapsto{ord} y =$
  $x \xrightarrow{ord} y \wedge \neg(\exists z.\ x \xrightarrow{ord} z \xrightarrow{ord} y)$
```

```
well_founded $r$ = $wf$ $r$
```

```
type_abbrev action_id : string
```

```
type_abbrev thread_id : string
```

```
type_abbrev location : string
```

```
type_abbrev val : string
```

```
memory_order_enum =
    MO_SEQ_CST
  | MO_RELAXED
  | MO_RELEASE
  | MO_ACQUIRE
  | MO_CONSUME
  | MO_ACQ_REL
```

```
action =
    LOCK of action_id thread_id location
  | UNLOCK of action_id thread_id location
  | ATOMIC_LOAD of action_id thread_id memory_order_enum location val
  | ATOMIC_STORE of action_id thread_id memory_order_enum location val
  | ATOMIC_RMW of action_id thread_id memory_order_enum location val val
  | LOAD of action_id thread_id location val
  | STORE of action_id thread_id location val
  | FENCE of action_id thread_id memory_order_enum
```

```
(action_id_of (LOCK $aid$ _ _) = $aid$) $\wedge$
(action_id_of (UNLOCK $aid$ _ _) = $aid$) $\wedge$
(action_id_of (ATOMIC_LOAD $aid$ _ _ _ _) = $aid$) $\wedge$
(action_id_of (ATOMIC_STORE $aid$ _ _ _ _) = $aid$) $\wedge$
(action_id_of (ATOMIC_RMW $aid$ _ _ _ _ _) = $aid$) $\wedge$
(action_id_of (LOAD $aid$ _ _ _) = $aid$) $\wedge$
(action_id_of (STORE $aid$ _ _ _) = $aid$) $\wedge$
(action_id_of (FENCE $aid$ _ _) = $aid$)
```

```
(thread_id_of (LOCK _ $tid$ _) = $tid$) $\wedge$
(thread_id_of (UNLOCK _ $tid$ _) = $tid$) $\wedge$
(thread_id_of (ATOMIC_LOAD _ $tid$ _ _ _) = $tid$) $\wedge$
(thread_id_of (ATOMIC_STORE _ $tid$ _ _ _) = $tid$) $\wedge$
(thread_id_of (ATOMIC_RMW _ $tid$ _ _ _ _) = $tid$) $\wedge$
(thread_id_of (LOAD _ $tid$ _ _) = $tid$) $\wedge$
(thread_id_of (STORE _ $tid$ _ _) = $tid$) $\wedge$
(thread_id_of (FENCE _ $tid$ _) = $tid$)
```

```
(memory_order (ATOMIC_LOAD _ _ $mem\_ord$ _ _) =
  SOME $mem\_ord$) $\wedge$
(memory_order (ATOMIC_STORE _ _ $mem\_ord$ _ _) =
  SOME $mem\_ord$) $\wedge$
(memory_order (ATOMIC_RMW _ _ $mem\_ord$ _ _ _) =
  SOME $mem\_ord$) $\wedge$
(memory_order (FENCE _ _ $mem\_ord$) =
  SOME $mem\_ord$) $\wedge$
(memory_order _ =
  NONE)
```

```
(location (LOCK _ _ $l$) = SOME $l$) $\wedge$
(location (UNLOCK _ _ $l$) = SOME $l$) $\wedge$
(location (ATOMIC_LOAD _ _ _ $l$ _) = SOME $l$) $\wedge$
(location (ATOMIC_STORE _ _ _ $l$ _) = SOME $l$) $\wedge$
(location (ATOMIC_RMW _ _ _ $l$ _ _) = SOME $l$) $\wedge$
(location (LOAD _ _ $l$ _) = SOME $l$) $\wedge$
(location (STORE _ _ $l$ _) = SOME $l$) $\wedge$
(location (FENCE _ _ _) = NONE)
```

```
(value_read (ATOMIC_LOAD _ _ _ _ $v$) = SOME $v$) $\wedge$
(value_read (ATOMIC_RMW _ _ _ _ $v$ _) = SOME $v$) $\wedge$
(value_read (LOAD _ _ _ $v$) = SOME $v$) $\wedge$
(value_read _ = NONE)
```

```
(value_written (ATOMIC_STORE _ _ _ _ $v$) = SOME $v$) $\wedge$
(value_written (ATOMIC_RMW _ _ _ _ _ $v$) = SOME $v$) $\wedge$
(value_written (STORE _ _ _ $v$) = SOME $v$) $\wedge$
(value_written _ = NONE)
```

```
is_lock $a$ =
  case $a$ of LOCK _ _ _ $\to$ T || _ $\to$ F
```

```
is_unlock $a$ =
  case $a$ of UNLOCK _ _ _ $\to$ T || _ $\to$ F
```

```
is_atomic_load $a$ =
  case $a$ of ATOMIC_LOAD _ _ _ _ _ $\to$ T || _ $\to$ F
```

```
is_atomic_store $a$ =
  case $a$ of ATOMIC_STORE _ _ _ _ _ $\to$ T || _ $\to$ F
```

```
is_atomic_rmw $a$ =
  case $a$ of ATOMIC_RMW _ _ _ _ _ _ $\to$ T || _ $\to$ F
```

```
is_load $a$ = case $a$ of LOAD _ _ _ _ $\to$ T || _ $\to$ F
```

```
is_store $a$ = case $a$ of STORE _ _ _ _ $\to$ T || _ $\to$ F
```

```
is_fence $a$ = case $a$ of FENCE _ _ _ $\to$ T || _ $\to$ F
```

```
is_lock_or_unlock $a$ = is_lock $a$ $\vee$ is_unlock $a$
```

```
is_atomic_action $a$ =
  is_atomic_load $a$ $\vee$ is_atomic_store $a$ $\vee$ is_atomic_rmw $a$
```

```
is_load_or_store $a$ = is_load $a$ $\vee$ is_store $a$
```

```
is_read $a$ =
  is_atomic_load $a$ $\vee$ is_atomic_rmw $a$ $\vee$ is_load $a$
```

```
is_write $a$ =
  is_atomic_store $a$ $\vee$ is_atomic_rmw $a$ $\vee$ is_store $a$
```

```
is_acquire $a$ =
  (case memory_order $a$ of
    SOME $mem\_ord$ $\to$
      ($mem\_ord$ $\in$
        {MO_ACQUIRE, MO_ACQ_REL, MO_SEQ_CST} $\wedge$
      (is_read $a$ $\vee$ is_fence $a$)) $\vee$
      (* 29.8:5 states that consume fences are acquire fences. *)
      (($mem\_ord$ = MO_CONSUME) $\wedge$ is_fence $a$)
  || NONE $\to$ is_lock $a$)
```

```
is_consume $a$ =
  is_read $a$ $\wedge$ (memory_order $a$ = SOME MO_CONSUME)
```

```
is_release $a$ =
  (case memory_order $a$ of
    SOME $mem\_ord$ $\to$
      $mem\_ord$ $\in$ {MO_RELEASE, MO_ACQ_REL, MO_SEQ_CST} $\wedge$
     (is_write $a$ $\vee$ is_fence $a$)
  || NONE $\to$ is_unlock $a$)
```

```
is_seq_cst $a$ = (memory_order $a$ = SOME MO_SEQ_CST)
```

```
location_kind =
    MUTEX
  | NON_ATOMIC
  | ATOMIC
```

```
actions_respect_location_kinds = actions_respect_location_kinds =
  $\forall a.$
    case location $a$ of SOME $l$ $\to$
      (case $location\text{-}kind\ l$ of
        MUTEX $\to$ is_lock_or_unlock $a$
       || NON_ATOMIC $\to$ is_load_or_store $a$
       || ATOMIC $\to$ is_load_or_store $a$ $\vee$ is_atomic_action $a$)
    || NONE $\to$ T
```

```
is_at_location_kind = is_at_location_kind =
  case location $a$ of
    SOME $l$ $\to$ ($location\text{-}kind\ l = lk0$)
  || NONE $\to$ F
```

```
is_at_mutex_location $a$ =
  is_at_location_kind $a$ MUTEX
```

```
is_at_non_atomic_location $a$ =
  is_at_location_kind $a$ NON_ATOMIC
```

```
is_at_atomic_location $a$ =
  is_at_location_kind $a$ ATOMIC
```

```
same_thread $a$ $b$ = (thread_id_of $a$ = thread_id_of $b$)
```

```
threadwise_relation_over $s$ $rel$ =
  relation_over $s$ $rel$ $\wedge$ ($\forall(a,b) \in rel.$ same_thread $a$ $b$)
```

```
same_location $a$ $b$ = (location $a$ = location $b$)
```

```
locations_of $actions$ = {$l$. $\exists a.$ (location $a$ = SOME $l$)}
```

```
well_formed_action $a$ =
  case $a$ of
    ATOMIC_LOAD _ _ $mem\_ord$ _ _ $\to$ $mem\_ord$ $\in$
      {MO_RELAXED, MO_ACQUIRE, MO_SEQ_CST, MO_CONSUME}
  || ATOMIC_STORE _ _ $mem\_ord$ _ _ $\to$ $mem\_ord$ $\in$
      {MO_RELAXED, MO_RELEASE, MO_SEQ_CST}
  || ATOMIC_RMW _ _ $mem\_ord$ _ _ _ $\to$ $mem\_ord$ $\in$
      {MO_RELAXED, MO_RELEASE, MO_ACQUIRE, MO_ACQ_REL, MO_SEQ_CST, MO_CONSUME}
  || _ $\to$ T
```

```
well_formed_threads = well_formed_threads =
  inj_on action_id_of ($actions$) $\wedge$
  ($\forall a.$ well_formed_action $a$) $\wedge$
  threadwise_relation_over $actions$ $sequenced\text{-}before$ $\wedge$
  threadwise_relation_over $actions$ $data\text{-}dependency$ $\wedge$
  threadwise_relation_over $actions$ $control\text{-}dependency$ $\wedge$
  strict_preorder $sequenced\text{-}before$ $\wedge$
  strict_preorder $data\text{-}dependency$ $\wedge$
  strict_preorder $control\text{-}dependency$ $\wedge$
  relation_over $actions$ $additional\text{-}synchronized\text{-}with$ $\wedge$
  ($\forall a.$ thread_id_of $a$ $\in$ threads) $\wedge$
  actions_respect_location_kinds $\wedge$
  $data\text{-}dependency \subseteq sequenced\text{-}before$
```

```
well_formed_reads_from_mapping = well_formed_reads_from_mapping =
  relation_over $actions$ ($\xrightarrow{rf}$) $\wedge$
  $(\forall a.\ \forall a'.\ \forall b.\ a \xrightarrow{rf} b \wedge a' \xrightarrow{rf} b \implies (a = a'))$ $\wedge$
  $(\forall (a,b) \in \xrightarrow{rf}.$
    same_location $a$ $b$ $\wedge$
    (value_read $b$ = value_written $a$) $\wedge$
    $(a \neq b)$ $\wedge$
    (is_at_mutex_location $a$ $\implies$
      is_unlock $a$ $\wedge$ is_lock $b$) $\wedge$
    (is_at_non_atomic_location $a$ $\implies$
      is_store $a$ $\wedge$ is_load $b$) $\wedge$
    (is_at_atomic_location $a$ $\implies$
      (is_atomic_store $a$ $\vee$ is_atomic_rmw $a$ $\vee$ is_store $a$)
      $\wedge$ (is_atomic_load $b$ $\vee$ is_atomic_rmw $b$ $\vee$ is_load $b$)))
```

```
all_lock_or_unlock_actions_at $lopt$ $as$ =
  {$a \in as$. is_lock_or_unlock $a$ $\wedge$ (location $a$ = $lopt$)}
```

```
consistent_locks = consistent_locks =
$\forall l \in$ locations_of $actions$. ($location\text{-}kind\ l$ = MUTEX) $\implies$ (
  let $lock\_unlock\_actions$ =
    all_lock_or_unlock_actions_at (SOME $l$)$actions$ in
  let $lock\_order = \xrightarrow{sc}|_{lock\_unlock\_actions}$ in
  (* 30.4.1:5 - The implementation shall serialize those (lock and unlock) operations. *)
  strict_total_order_over $lock\_unlock\_actions$ $lock\_order$ $\wedge$

  (* 30.4.1:1 A thread owns a mutex from the time it successfully calls one of the lock functions until
  it calls unlock.*)
  (* 30.4.1:20 Requires: The calling thread shall own the mutex. *)
  (* 30.4.1:21 Effects: Releases the calling threads ownership of the mutex.*)
  ($\forall a_u \in lock\_unlock\_actions.$ is_unlock $a_u$ $\implies$
    ($\exists a_l \in lock\_unlock\_actions.$
      $a_l \xmapsto{lock\_order} a_u$ $\wedge$ same_thread $a_l$ $a_u$ $\wedge$ is_lock $a_l$)) $\wedge$

  (* 30.4.1:7 Effects: Blocks the calling thread until ownership of the mutex can be obtained for the
  calling thread.*)
  (* 30.4.1:8 Postcondition: The calling thread owns the mutex. *)
  ($\forall a_l \in lock\_unlock\_actions.$ is_lock $a_l$ $\implies$
    ($\forall a_u \in lock\_unlock\_actions.$
      $a_u \xmapsto{lock\_order} a_l \implies$ is_unlock $a_u$)))
```

```
rs_element $rs\_head$ $a$ =
  same_thread $a$ $rs\_head$ $\vee$ is_atomic_rmw $a$
```

```
release_sequence = $a_{rel} \xrightarrow{release\text{-}sequence} b$ =
  is_at_atomic_location $b$ $\wedge$
  is_release $a_{rel}$ $\wedge$ (
    $(b = a_{rel})$ $\vee$
    (rs_element $a_{rel}$ $b$ $\wedge$ $a_{rel} \xrightarrow{modification\text{-}order} b$ $\wedge$
      ($\forall c.\ a_{rel} \xrightarrow{modification\text{-}order} c \xrightarrow{modification\text{-}order} b \implies$
        rs_element $a_{rel}$ $c$)))
```

```
$release\_sequence\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ =

release_sequence $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ $a$ $b$}
```

```
$hypothetical\_release\_sequence = a \xrightarrow{hypothetical\text{-}release\text{-}sequence} b$ =
  is_at_atomic_location $b$ $\wedge$ (
  $(b = a)$ $\vee$
  (rs_element $a$ $b$ $\wedge$ $a \xrightarrow{modification\text{-}order} b$ $\wedge$
    ($\forall c.\ a \xrightarrow{modification\text{-}order} c \xrightarrow{modification\text{-}order} b \implies$
      rs_element $a$ $c$)))
```

```
$hypothetical\_release\_sequence\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ =

$hypothetical\_release\_sequence$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ $a$ $b$}
```

```
synchronizes_with = $a \xrightarrow{synchronizes\text{-}with} b$ =
  (* – additional synchronization, from thread create etc. – *)
  $a \xrightarrow{additional\text{-}synchronized\text{-}with} b$ $\vee$

  (same_location $a$ $b$ $\wedge$ $a \in actions$ $\wedge$ $b \in actions$ $\wedge$ (
    (* – mutex synchronization – *)
    (is_unlock $a$ $\wedge$ is_lock $b$ $\wedge$ $a \xrightarrow{sc} b$) $\vee$

    (* – release/acquire synchronization – *)
    (is_release $a$ $\wedge$ is_acquire $b$ $\wedge$ $\neg$same_thread $a$ $b$ $\wedge$
    ($\exists c.\ a \xrightarrow{release\text{-}sequence} c \xrightarrow{rf} b$)) $\vee$

    (* – fence synchronization – *)
    (is_fence $a$ $\wedge$ is_release $a$ $\wedge$ is_fence $b$ $\wedge$ is_acquire $b$ $\wedge$
    ($\exists x.\ \exists y.$ same_location $x$ $y$ $\wedge$
      is_atomic_action $x$ $\wedge$ is_atomic_action $y$ $\wedge$ is_write $x$ $\wedge$
      $a \xrightarrow{sequenced\text{-}before} x$ $\wedge$ $y \xrightarrow{sequenced\text{-}before} b$ $\wedge$
      ($\exists z.\ x \xrightarrow{hypothetical\text{-}release\text{-}sequence} z \xrightarrow{rf} y$))) $\vee$

    (is_fence $a$ $\wedge$ is_release $a$ $\wedge$
     is_atomic_action $b$ $\wedge$ is_acquire $b$ $\wedge$
    ($\exists x.$ same_location $x$ $b$ $\wedge$
      is_atomic_action $x$ $\wedge$ is_write $x$ $\wedge$
      $a \xrightarrow{sequenced\text{-}before} x$ $\wedge$
      ($\exists z.\ x \xrightarrow{hypothetical\text{-}release\text{-}sequence} z \xrightarrow{rf} b$))) $\vee$

    (is_atomic_action $a$ $\wedge$ is_release $a$ $\wedge$
     is_fence $b$ $\wedge$ is_acquire $b$ $\wedge$
    ($\exists x.$ same_location $a$ $x$ $\wedge$ is_atomic_action $x$ $\wedge$
      $x \xrightarrow{sequenced\text{-}before} b$ $\wedge$
      ($\exists z.\ a \xrightarrow{release\text{-}sequence} z \xrightarrow{rf} x$)))))
```

```
$synchronizes\_with\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $sc$ $release\text{-}sequence$ $hypothetical\text{-}release\text{-}sequence$ =

synchronizes_with $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $sc$ $release\text{-}sequence$ $hypothetical\text{-}release\text{-}sequence$ $a$ $b$}
```

```
carries_a_dependency_to = $a \xrightarrow{carries\text{-}a\text{-}dependency\text{-}to} b$ =
  $a\ ((\xrightarrow{rf} \cap \xrightarrow{sequenced\text{-}before}) \cup \xrightarrow{data\text{-}dependency})^+\ b$
```

```
$carries\_a\_dependency\_to\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ =

carries_a_dependency_to $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $a$ $b$}
```

```
dependency_ordered_before = $a \xrightarrow{dependency\text{-}ordered\text{-}before} d$ =
  $a \in actions$ $\wedge$ $d \in actions$ $\wedge$
  ($\exists b.$ is_release $a$ $\wedge$ is_consume $b$ $\wedge$
    ($\exists e.\ a \xrightarrow{release\text{-}sequence} e \xrightarrow{rf} b$) $\wedge$
    ($b \xrightarrow{carries\text{-}a\text{-}dependency\text{-}to} d \vee (b = d)$))
```

```
$dependency\_ordered\_before\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $release\text{-}sequence$ $carries\text{-}a\text{-}dependency\text{-}to$ =

dependency_ordered_before $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $release\text{-}sequence$ $carries\text{-}a\text{-}dependency\text{-}to$ $a$ $b$}
```

```
$simple\_happens\_before = \xrightarrow{simple\_happens\_before} =$
  $(\xrightarrow{sequenced\text{-}before} \cup \xrightarrow{synchronizes\text{-}with})^+$
```

```
$consistent\_simple\_happens\_before\ shb$ =
  irreflexive ($\xrightarrow{shb}$)
```

```
inter_thread_happens_before = $\xrightarrow{inter\text{-}thread\text{-}happens\text{-}before}$ =
  let $r = \xrightarrow{synchronizes\text{-}with} \cup$
          $\xrightarrow{dependency\text{-}ordered\text{-}before} \cup$
          $(\xrightarrow{synchronizes\text{-}with} \circ \xrightarrow{sequenced\text{-}before})$ in
  $(\xrightarrow{r} \cup (\xrightarrow{sequenced\text{-}before} \circ \xrightarrow{r}))^+$
```

```
consistent_inter_thread_happens_before = consistent_inter_thread_happens_before =
  irreflexive ($\xrightarrow{inter\text{-}thread\text{-}happens\text{-}before}$)
```

```
happens_before = $\xrightarrow{happens\text{-}before}$ =
  $\xrightarrow{sequenced\text{-}before} \cup \xrightarrow{inter\text{-}thread\text{-}happens\text{-}before}$
```

```
all_sc_actions = all_sc_actions =
  {$a$. (is_seq_cst $a$ $\vee$ is_lock $a$ $\vee$ is_unlock $a$)}
```

```
consistent_sc_order = consistent_sc_order =
  let sc_happens_before = $\xrightarrow{happens\text{-}before}|_{\text{all\_sc\_actions}}$ in
  let sc_mod_order = $\xrightarrow{modification\text{-}order}|_{\text{all\_sc\_actions}}$ in
  strict_total_order_over all_sc_actions ($\xrightarrow{sc}$) $\wedge$
  $\xrightarrow{\text{sc\_happens\_before}} \subseteq \xrightarrow{sc}$ $\wedge$
  $\xrightarrow{\text{sc\_mod\_order}} \subseteq \xrightarrow{sc}$
```

```
consistent_modification_order = consistent_modification_order =
  ($\forall a.\ \forall b.\ a \xrightarrow{modification\text{-}order} b \implies$ same_location $a$ $b$) $\wedge$
  ($\forall l \in$ locations_of $actions$. case $location\text{-}kind\ l$ of
      ATOMIC $\to$ (
        let $actions\_at\_l$ = {$a$. (location $a$ = SOME $l$)} in
        let $writes\_at\_l$ = {$a\_at\_l$. (is_store $a$ $\vee$
          is_atomic_store $a$ $\vee$ is_atomic_rmw $a$)} in
           strict_total_order_over $writes\_at\_l$
           ($\xrightarrow{modification\text{-}order}|_{actions\_at\_l}$) $\wedge$
           (* happens-before at the writes of $l$ is a subset of mo for $l$ *)
           $\xrightarrow{happens\text{-}before}|_{writes\_at\_l} \subseteq \xrightarrow{modification\text{-}order}$ $\wedge$
           (* MO_SEQ_CST fences impose modification order *)
           ($\xrightarrow{sequenced\text{-}before} \circ (\xrightarrow{sc}|_{\text{is\_fence}}) \circ \xrightarrow{sequenced\text{-}before}|_{writes\_at\_l}$
              $\subseteq \xrightarrow{modification\text{-}order}$)
    || _ $\to$ (
        let $actions\_at\_l$ = {$a$. (location $a$ = SOME $l$)} in
        ($\xrightarrow{modification\text{-}order}|_{actions\_at\_l}$) = {}))
```

```
visible_side_effect = $a \xrightarrow{visible\text{-}side\text{-}effect} b$ =
  $a \xrightarrow{happens\text{-}before} b$ $\wedge$
  is_write $a$ $\wedge$ is_read $b$ $\wedge$ same_location $a$ $b$ $\wedge$
  $\neg(\exists c.\ (c \neq a) \wedge (c \neq b)$ $\wedge$
     is_write $c$ $\wedge$ same_location $c$ $b$ $\wedge$
     $a \xrightarrow{happens\text{-}before} c \xrightarrow{happens\text{-}before} b$)
```

```
$visible\_side\_effect\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $happens\text{-}before$ =
{$ab \in happens\text{-}before$. let $(a,b) = ab$ in
visible_side_effect $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $happens\text{-}before$ $a$ $b$}
```

```
visible_sequence_of_side_effects_tail = visible_sequence_of_side_effects_tail $vsse\_head$ $b$ =
  {$c$. $vsse\_head \xrightarrow{modification\text{-}order} c$ $\wedge$
    $\neg(b \xrightarrow{happens\text{-}before} c)$ $\wedge$
    ($\forall a.\ vsse\_head \xrightarrow{modification\text{-}order} a \xrightarrow{modification\text{-}order} c$
        $\implies \neg(b \xrightarrow{happens\text{-}before} a)$)}
```

```
myimage $f$ $s$ = {$y$. $\exists x \in s.\ (y = f\ x)$}
```

```
visible_sequences_of_side_effects = visible_sequences_of_side_effects =
  $\lambda(vsse\_head, b).$
    ($b$, if is_at_atomic_location $b$ then
          {$vsse\_head$} $\cup$
          visible_sequence_of_side_effects_tail $vsse\_head$ $b$
        else
          {})
```

```
$visible\_sequences\_of\_side\_effects\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ $happens\text{-}before$ $visible\text{-}side\text{-}effect$ =
    myimage (visible_sequences_of_side_effects $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ $happens\text{-}before$ $visible\text{-}side\text{-}effect$)$visible\text{-}side\text{-}effect$
```

```
consistent_reads_from_mapping = consistent_reads_from_mapping =
  ($\forall b.$ (is_read $b$ $\wedge$ is_at_non_atomic_location $b$) $\implies$
      (if ($\exists a_{vse}.\ a_{vse} \xrightarrow{visible\text{-}side\text{-}effect} b$)
       then ($\exists a_{vse}.\ a_{vse} \xrightarrow{visible\text{-}side\text{-}effect} b \wedge a_{vse} \xrightarrow{rf} b$)
       else $\neg(\exists a.\ a \xrightarrow{rf} b)$)) $\wedge$

  ($\forall b.$ (is_read $b$ $\wedge$ is_at_atomic_location $b$) $\implies$
      (if ($\exists (b', vsse) \in visible\text{-}sequences\text{-}of\text{-}side\text{-}effects.\ (b' = b)$)
       then ($\exists (b', vsse) \in visible\text{-}sequences\text{-}of\text{-}side\text{-}effects.$
             $(b' = b) \wedge (\exists c \in vsse.\ c \xrightarrow{rf} b)$)
       else $\neg(\exists a.\ a \xrightarrow{rf} b)$)) $\wedge$

  ($\forall (x,a) \in \xrightarrow{rf}.$
    $\forall (y,b) \in \xrightarrow{rf}.$
      $a \xrightarrow{happens\text{-}before} b$ $\wedge$
        same_location $a$ $b$ $\wedge$ is_at_atomic_location $b$
        $\implies (x = y) \vee x \xrightarrow{modification\text{-}order} y$) $\wedge$
(* new CoWR *)
  ($\forall (a,b) \in \xrightarrow{happens\text{-}before}.$
    $\forall c.$
      $c \xrightarrow{rf} b$ $\wedge$
        is_write $a$ $\wedge$ same_location $a$ $b$ $\wedge$ is_at_atomic_location $b$
        $\implies (c = a) \vee a \xrightarrow{modification\text{-}order} c$) $\wedge$
(* new CoRW *)
  ($\forall (a,b) \in \xrightarrow{happens\text{-}before}.$
    $\forall c.$
      $c \xrightarrow{rf} a$ $\wedge$
        is_write $b$ $\wedge$ same_location $a$ $b$ $\wedge$ is_at_atomic_location $a$
        $\implies c \xrightarrow{modification\text{-}order} b$) $\wedge$

  ($\forall (a,b) \in \xrightarrow{rf}.$ is_atomic_rmw $b$
    $\implies a \xmapsto{modification\text{-}order} b$) $\wedge$

  ($\forall (a,b) \in \xrightarrow{rf}.$ is_seq_cst $b$
    $\implies (\neg$is_seq_cst $a \wedge (\forall x.\ x \xmapsto{sc}_{\lambda c.\ \text{is\_write}\ c \wedge \text{same\_location}\ b\ c}\ b \implies x \xrightarrow{modification\text{-}order} a)) \vee$
    $a \xmapsto{sc}_{\lambda c.\ \text{is\_write}\ c \wedge \text{same\_location}\ b\ c}\ b)$ $\wedge$

  (* -Fence restrictions- *)

  (* 29.3:3 *)
  ($\forall a.\ \forall (x,b) \in \xrightarrow{sequenced\text{-}before}.\ \forall y.$
    (is_fence $x$ $\wedge$ is_seq_cst $x$ $\wedge$ is_atomic_action $b$ $\wedge$
      is_write $a$ $\wedge$ same_location $a$ $b$ $\wedge$
      $a \xmapsto{sc} x \wedge y \xrightarrow{rf} b$)
    $\implies (y = a) \vee a \xrightarrow{modification\text{-}order} y$) $\wedge$

  (* 29.3:4 *)
  ($\forall (a,x) \in \xrightarrow{sequenced\text{-}before}.\ \forall (y,b) \in \xrightarrow{rf}.$
    (is_atomic_action $a$ $\wedge$ is_fence $x$ $\wedge$ is_seq_cst $x$ $\wedge$
      is_write $a$ $\wedge$ same_location $a$ $b$ $\wedge$
      $x \xrightarrow{sc} b$ $\wedge$ is_atomic_action $b$)
    $\implies (y = a) \vee a \xrightarrow{modification\text{-}order} y$) $\wedge$

  (* 29.3:5 *)
  ($\forall (a,x) \in \xrightarrow{sequenced\text{-}before}.\ \forall (y,b) \in \xrightarrow{sequenced\text{-}before}.\ \forall z.$
    (is_atomic_action $a$ $\wedge$ is_fence $x$ $\wedge$ is_seq_cst $x$ $\wedge$
      is_write $a$ $\wedge$ is_fence $y$ $\wedge$ is_seq_cst $y$ $\wedge$
      is_atomic_action $b$ $\wedge$ same_location $a$ $b$ $\wedge$
      $x \xrightarrow{sc} y \wedge z \xrightarrow{rf} b$)
    $\implies (z = a) \vee a \xrightarrow{modification\text{-}order} z$)
```

```
$all\_data\_dependency = \xrightarrow{all\_data\_dependency} =$
  $(\xrightarrow{rf} \cup \xrightarrow{carries\text{-}a\text{-}dependency\text{-}to})^+$
```

```
consistent_control_dependency = consistent_control_dependency =
  irreflexive $((\xrightarrow{control\text{-}dependency} \cup \xrightarrow{all\_data\_dependency})^+)$
```

```
$consistent\_execution$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $sc$ =
 well_formed_threads $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $\wedge$
 consistent_locks $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $sc$ $\wedge$ (
let $release\text{-}sequence$ = $release\_sequence\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ in
let $hypothetical\text{-}release\text{-}sequence$ = $hypothetical\_release\_sequence\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ in
let $synchronizes\text{-}with$ = $synchronizes\_with\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $sc$ $release\text{-}sequence$ $hypothetical\text{-}release\text{-}sequence$ in
let $carries\text{-}a\text{-}dependency\text{-}to$ = $carries\_a\_dependency\_to\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ in
let $dependency\text{-}ordered\text{-}before$ = $dependency\_ordered\_before\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $release\text{-}sequence$ $carries\text{-}a\text{-}dependency\text{-}to$ in
let $inter\text{-}thread\text{-}happens\text{-}before$ = inter_thread_happens_before $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $synchronizes\text{-}with$ $dependency\text{-}ordered\text{-}before$ in
let $happens\text{-}before$ = happens_before $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $inter\text{-}thread\text{-}happens\text{-}before$ in
let $visible\text{-}side\text{-}effect$ = $visible\_side\_effect\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $happens\text{-}before$ in
let $visible\text{-}sequences\text{-}of\text{-}side\text{-}effects$ = $visible\_sequences\_of\_side\_effects\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ $happens\text{-}before$ $visible\text{-}side\text{-}effect$ in
 consistent_inter_thread_happens_before $inter\text{-}thread\text{-}happens\text{-}before$ $\wedge$
 consistent_sc_order $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ $sc$ $happens\text{-}before$ $\wedge$
 consistent_modification_order $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $sc$ $modification\text{-}order$ $happens\text{-}before$ $\wedge$
 well_formed_reads_from_mapping $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $\wedge$
 consistent_reads_from_mapping $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $sc$ $modification\text{-}order$ $happens\text{-}before$ $visible\text{-}side\text{-}effect$ $visible\text{-}sequences\text{-}of\text{-}side\text{-}effects$)
```

```
indeterminate_reads $actions$ threads = indeterminate_reads =
  {$b$. is_read $b$ $\wedge$ $\neg(\exists a.\ a \xrightarrow{rf} b)$}
```

```
unsequenced_races = unsequenced_races = {$(a,b)$.
  $(a \neq b)$ $\wedge$ same_location $a$ $b$ $\wedge$ (is_write $a$ $\vee$ is_write $b$) $\wedge$
  same_thread $a$ $b$ $\wedge$
  $\neg(a \xrightarrow{sequenced\text{-}before} b \vee b \xrightarrow{sequenced\text{-}before} a)$}
```

```
data_races = data_races = {$(a,b)$.
  $(a \neq b)$ $\wedge$ same_location $a$ $b$ $\wedge$ (is_write $a$ $\vee$ is_write $b$) $\wedge$
  $\neg$same_thread $a$ $b$ $\wedge$
  $\neg$(is_atomic_action $a$ $\wedge$ is_atomic_action $b$) $\wedge$
  $\neg(a \xrightarrow{happens\text{-}before} b \vee b \xrightarrow{happens\text{-}before} a)$}
```

```
data_races' $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $sc$ =
let $release\text{-}sequence$ = $release\_sequence\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ in
let $hypothetical\text{-}release\text{-}sequence$ = $release\_sequence\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $modification\text{-}order$ in
let $synchronizes\text{-}with$ = $synchronizes\_with\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $sc$ $release\text{-}sequence$ $hypothetical\text{-}release\text{-}sequence$ in
let $carries\text{-}a\text{-}dependency\text{-}to$ = $carries\_a\_dependency\_to\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ in
let $dependency\text{-}ordered\text{-}before$ = $dependency\_ordered\_before\_set$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $release\text{-}sequence$ $carries\text{-}a\text{-}dependency\text{-}to$ in
let $inter\text{-}thread\text{-}happens\text{-}before$ = inter_thread_happens_before $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $synchronizes\text{-}with$ $dependency\text{-}ordered\text{-}before$ in
let $happens\text{-}before$ = happens_before $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $inter\text{-}thread\text{-}happens\text{-}before$ in
 data_races $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $happens\text{-}before$
```

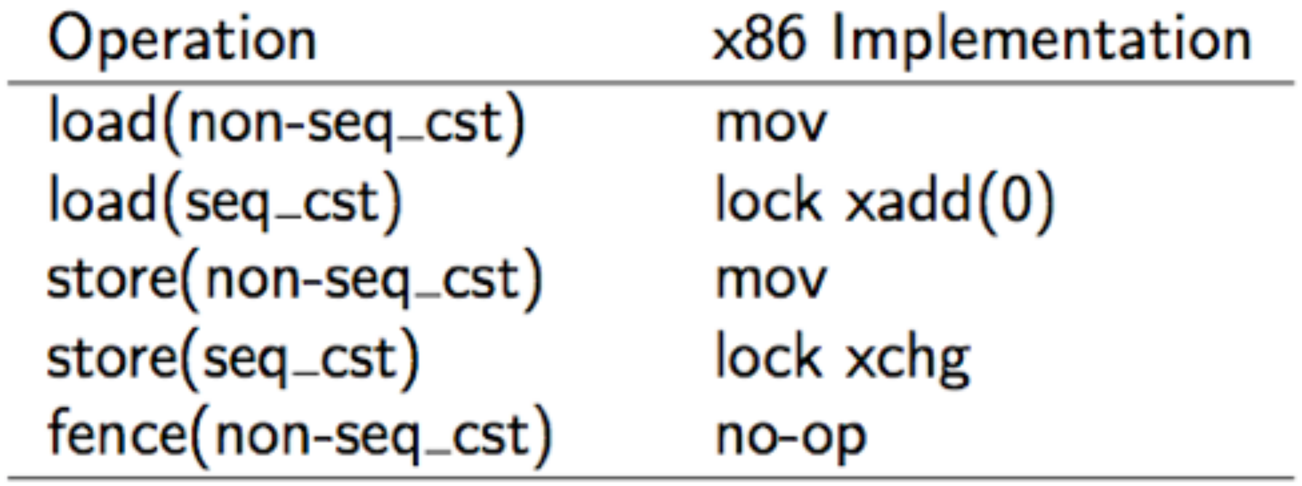
```
$cpp\_memory\_model\ opsem\ (p \in {}'program)$ =
  let executions = {($actions$, threads, $location\text{-}kind$, $sequenced\text{-}before$, $additional\text{-}synchronized\text{-}with$, $data\text{-}dependency$, $control\text{-}dependency$, $rf$, $modification\text{-}order$, $sc$).
      $opsem$ $p$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $\wedge$ $consistent\_execution$ $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $sc$} in
    if $\exists$($actions$, threads, $location\text{-}kind$, $sequenced\text{-}before$, $additional\text{-}synchronized\text{-}with$, $data\text{-}dependency$, $control\text{-}dependency$, $rf$, $modification\text{-}order$, $sc$) $\in$ executions .
        (indeterminate_reads $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf \neq$ {}) $\vee$
        (unsequenced_races $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency \neq$ {}) $\vee$
        (data_races' $actions$ threads $location\text{-}kind$ $sequenced\text{-}before$ $additional\text{-}synchronized\text{-}with$ $data\text{-}dependency$ $control\text{-}dependency$ $rf$ $modification\text{-}order$ $sc \neq$ {})
     then {}
     else executions
```

Is C++11 hopelessly complicated?

Programmers cannot be given this model.

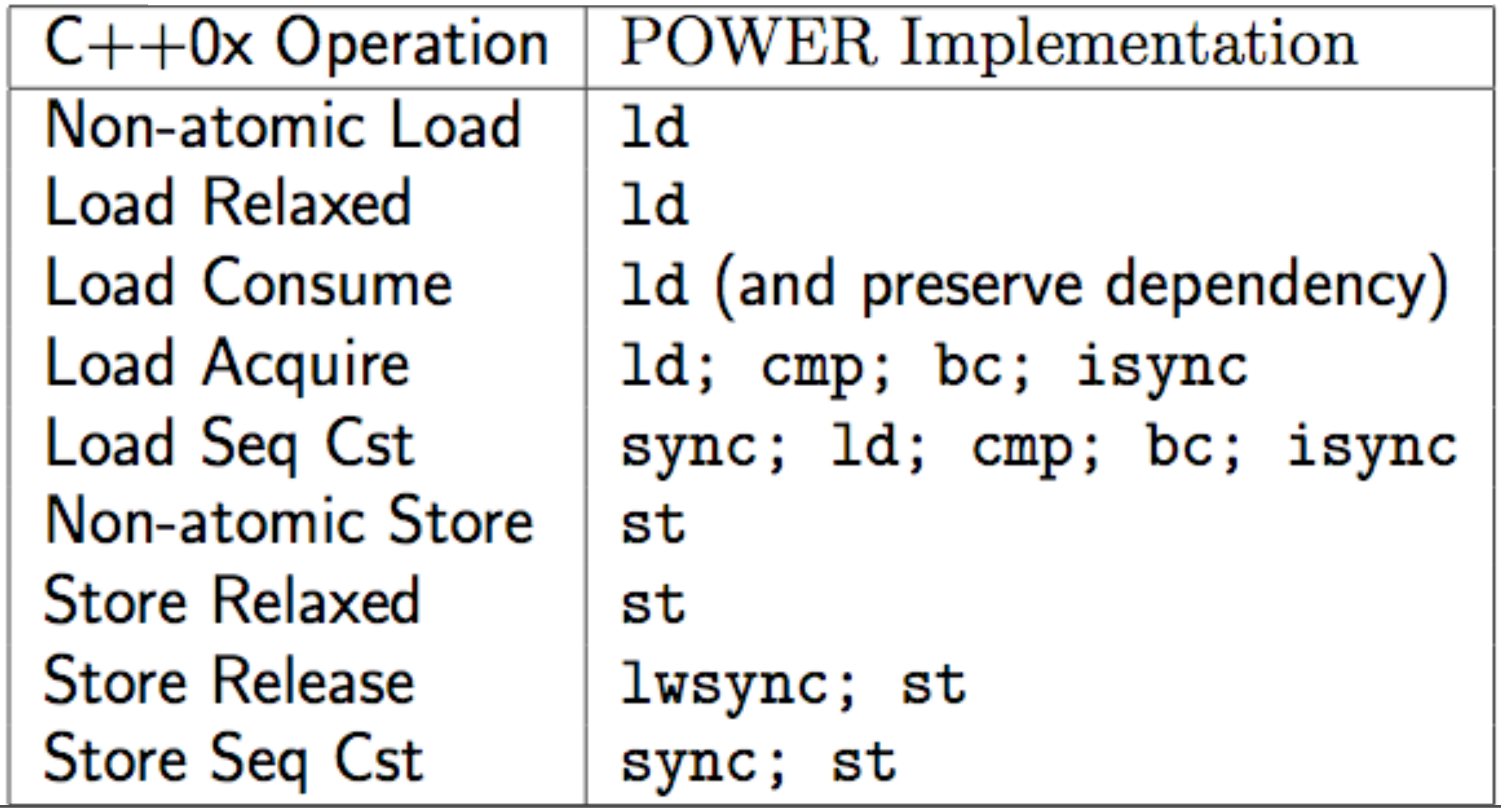
However, with a formal definition, we can do proofs!

- Can we compile to x86?



Operation	x86 Implementation
load(non-seq_cst)	mov
load(seq_cst)	lock xadd(0)
store(non-seq_cst)	mov
store(seq_cst)	lock xchg
fence(non-seq_cst)	no-op

- Can we compile to Power?



C++0x Operation	POWER Implementation
Non-atomic Load	`ld`
Load Relaxed	`ld`
Load Consume	`ld` (and preserve dependency)
Load Acquire	`ld; cmp; bc; isync`
Load Seq Cst	`sync; ld; cmp; bc; isync`
Non-atomic Store	`st`
Store Relaxed	`st`
Store Release	`lwsync; st`
Store Seq Cst	`sync; st`

Is C++11 hopelessly complicated?

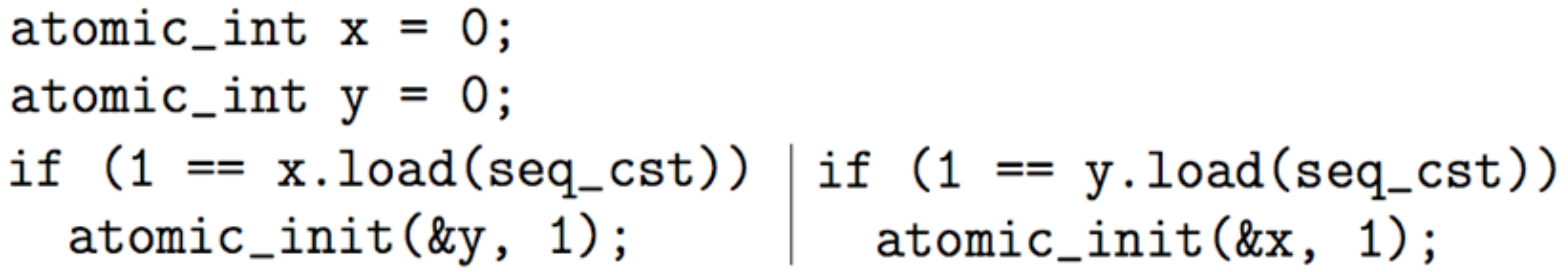
Simplifications:

Full model: *visible sequences of side effects* are unneeded (HOL4)

Derivative models:

- without consume, happens-before is transitive

- DRF programs using only **`seq_cst`** atomics are SC (false)



```
atomic_int x = 0;
atomic_int y = 0;
if (1 == x.load(seq_cst))  | if (1 == y.load(seq_cst))
  atomic_init(&y, 1);      |   atomic_init(&x, 1);
```

`atomic_init` is a non-atomic write, and in C++11 they race.

The current state of the standard

Fixed:

- in some cases, happens-before was cyclic
- coherence
- `seq_cst` atomics were more broken

Not fixed:

- self satisfying conditional

```
r1 = x.load(mo_relaxed);          | r2 = y.load(mo_relaxed);
if (r1 == 42)                     | if (r2 == 42)
  y.store(r1, mo_relaxed);        |   x.store(42, mo_relaxed);
```

$$\begin{tikzcd}
\text{c:Rrlx x=1} \arrow[d, "\text{sb}"'] & \text{e:Rrlx y=1} \arrow[d, "\text{sb}"'] \\
\text{d:Wrlx y=1} \arrow[ur, "\text{rf}"] & \text{f:Wrlx x=1} \arrow[ul, "\text{rf}"']
\end{tikzcd}$$

- `seq_cst` atomics are still not SC

3. A word on dynamic techniques for data-race detection

Data race detection

Modern high-performance dynamic race detectors are based either on:

happens-before ordering

reconstruct happens-before order in the current execution
report a race if intersection if two conflicting accesses are not related by hb

sound

drawback: misses races occurring on rare executions

lockset computation

records which locks protect every memory access
report a race if intersection of all locksets for a variable is empty

popularised by Eraser (Savage et al.) ’97

can detect races not observed in the execution being monitored

drawback: unsound (false positives)

Examples of lockset computation

```
lock(b)      |  lock(a)
lock(a)      |  x=2
x=1          |  unlock(a)
unlock(a)
```

```
        1:L(b);1:L(a);1:Wx1;1:U(a);2:L(a);2:Wx2;2:U(a)
```

```
locks held:  1:b   1:b,a                 2:a

C(x):                    x:a,b                     x:a
```

lockset for x non-empty at the end, no data-race

```
lock(b)      |  lock(c)
lock(a)      |  x=2
x=1          |  unlock(c)
unlock(a)
```

```
     1:L(b);1:L(a);1:Wx1;1:U(a);2:L(c);2:Wx2;2:U(c)

C(x):                x:a,b                   x:empty
```

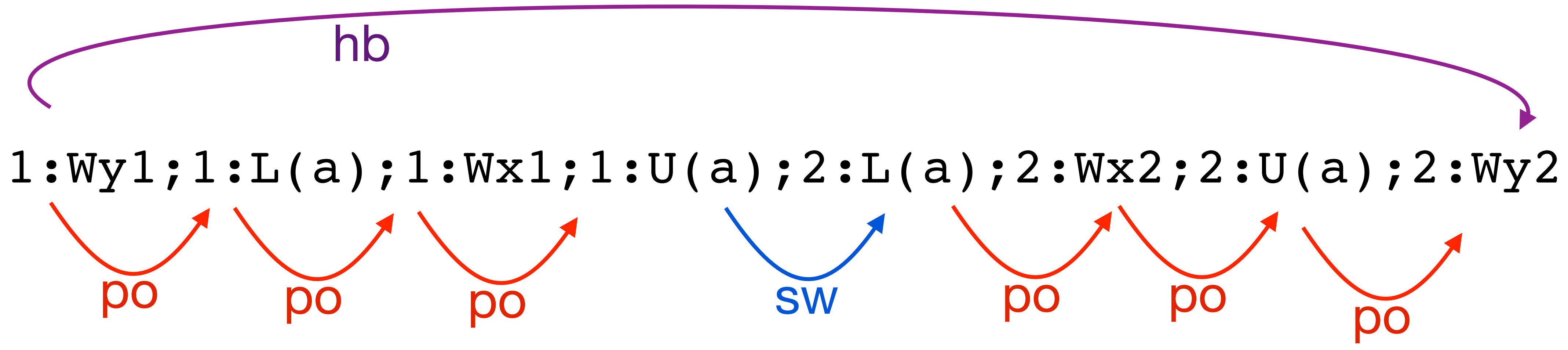
lockset for x empty at the end, possible data-race

lockset vs happens-before

```
y=1              lock(a)
lock(a)          x=2
x=1              unlock(a)
unlock(a)        y=2
```

This program has a race on y

If only the execution below is observed:



happens-before computation does not report a race.

Lockset computation detects instead that accesses to y are unprotected and reports a possible race.

lockset vs happens-before (2)

```
y=1             |  lock(a)
lock(a)         |  tmp=x
x=1             |  unlock(a)
unlock(a)          if tmp == 1
                     then print y
```

This program instead is DRF.

Happens-before computation will not report a race
(no matter which execution is observed)

Since accesses to y are unprotected, locksets computation reports a false positive.

Data race detection

Modern high-performance dynamic race detectors are based either on:

happens-before ordering

reconstruct happens-before order in the current execution
report a race if intersection if two conflicting accesses are not related by hb

sound

drawback: misses races occurring on rare executions

lockset computation

records which locks protect every memory access
report a race if intersection of all locksets for a variable is empty

popularised by Eraser (Savage et al.) ’97

can detect races not observed in the execution being monitored

drawback: unsound (false positives)

Data race detection

Mode er on:

hap

recons

in

report all

conflicti y

al.) ’97

d in

ed

dr

occu ves)

Current state of the art:

hybrid approaches combining locksets and happens-before ordering + other dynamic annotations

Helgrind, RaceFuzzer, ThreadSanitizer...

Impressive:

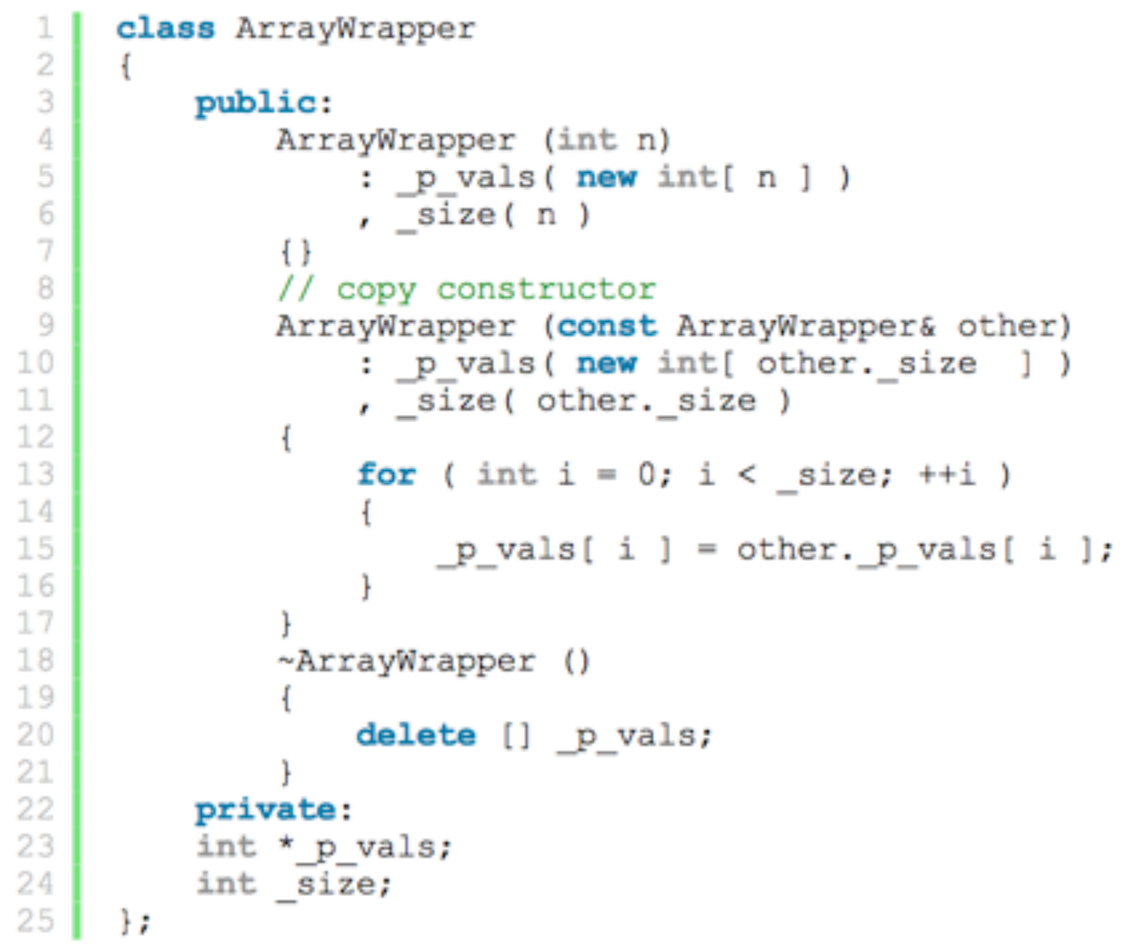
<10x slowdown on large applications
found thousands races

Still not as reliable as the tool we dream of...

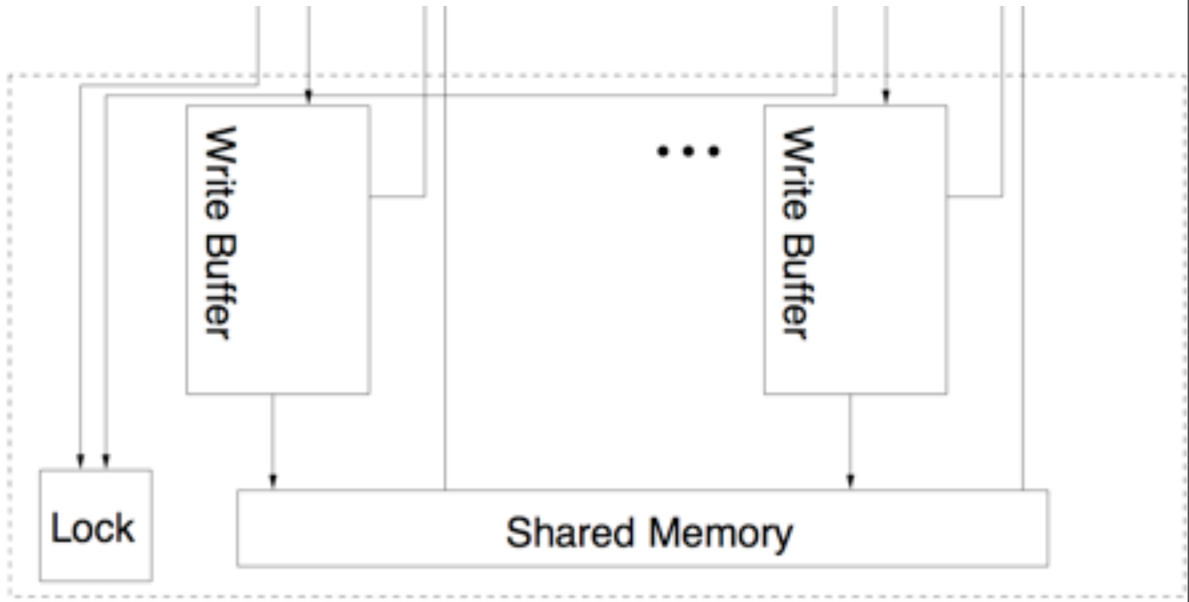
4. Sketch of an operational formalisation of x86-TSO

...starting with a formalisation of SC

Separate language and memory semantics



```
class ArrayWrapper
{
    public:
        ArrayWrapper (int n)
            : _p_vals( new int[ n ] )
            , _size( n )
        {}
        // copy constructor
        ArrayWrapper (const ArrayWrapper& other)
            : _p_vals( new int[ other._size  ] )
            , _size( other._size )
        {
            for ( int i = 0; i < _size; ++i )
            {
                _p_vals[ i ] = other._p_vals[ i ];
            }
        }
        ~ArrayWrapper ()
        {
            delete [] _p_vals;
        }
    private:
    int *_p_vals;
    int _size;
};
```



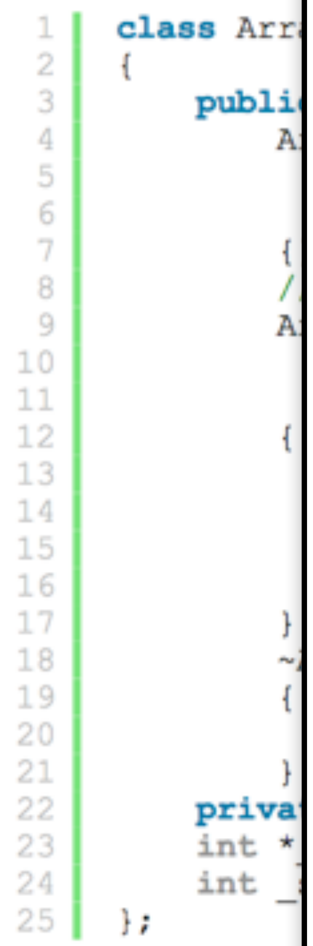
program
semantics defined via an LTS

memory
semantics defined via an LTS

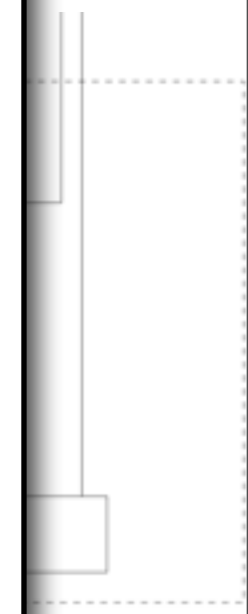
Labels for interaction:

$W_t[a]v$: a write of value **v** to address **a** by thread **t**
$R_t[a]v$: a read of **v** from **a** by **t** by thread **t**
+ other events for barriers and locked instructions

Separate language and memory semantics



```
class Arr
{
    publi
        A

        {
        /
        A

        {

        }
        ~
        {

        }
    priva
    int *
    int _
};
```



Separate language and state semantics

proved to be a very good choice

in many (unrelated) projects I worked on!

semantics defined via an LTS

semantics defined via an LTS

Labels for interaction:

W$_t$[a]v : a write of value **v** to address **a** by thread **t**

R$_t$[a]v : a read of **v** from **a** by **t** by thread **t**

+ other events for barriers and locked instructions

A tiny language

$location,\ x,\ m$	address (or pointer value)
$integer,\ n$	integer
$thread\_id,\ t$	thread id
$k,\ i,\ j$	

$expression,\ e$	$::=$		expression
	$\vert$	n	integer literal
	$\vert$	$*x$	read from pointer
	$\vert$	$*x = e$	write to pointer
	$\vert$	$e; e'$	sequential composition
	$\vert$	$e + e'$	plus
$process,\ p$	$::=$		process
	$\vert$	$t{:}e$	thread
	$\vert$	$p \vert p'$	parallel composition

What can a thread do in isolation?

$\boxed{e \xrightarrow{l} e'}$ e does l to become e'

$$\frac{}{*x \xrightarrow{\mathsf{R}\ x=n} n} \quad \text{READ}$$

$$\frac{}{*x = n \xrightarrow{\mathsf{W}\ x=n} n} \quad \text{WRITE}$$

$$\frac{e \xrightarrow{l} e'}{*x = e \xrightarrow{l} *x = e'} \quad \text{WRITE\_CONTEXT}$$

$$\frac{}{n; e \xrightarrow{\tau} e} \quad \text{SEQ}$$

$$\frac{e_1 \xrightarrow{l} e_1'}{e_1; e_2 \xrightarrow{l} e_1'; e_2} \quad \text{SEQ\_CONTEXT}$$

$$\frac{e_1 \xrightarrow{l} e_1'}{e_1 + e_2 \xrightarrow{l} e_1' + e_2} \quad \text{PLUS\_CONTEXT\_1}$$

$$\frac{e_2 \xrightarrow{l} e_2'}{n_1 + e_2 \xrightarrow{l} n_1 + e_2'} \quad \text{PLUS\_CONTEXT\_2}$$

$$\frac{n = n_1 + n_2}{n_1 + n_2 \xrightarrow{\tau} n} \quad \text{PLUS}$$

Observe that we can read an arbitrary value from the memory.

Example

Show that the expression:

$$(*x = *y); *x$$

can perform the following trace:

$$(*x = *y); *x \xrightarrow{\mathsf{R}\,y=7} \xrightarrow{\mathsf{W}\,x=7} \xrightarrow{\tau} \xrightarrow{\mathsf{R}\,x=9} 9$$

Lifting to processes

$\boxed{p \xrightarrow{l_t} p'}$ p does l_t to become p'

$$\frac{e \xrightarrow{l} e'}{t{:}e \xrightarrow{l_t} t{:}e'} \quad \text{THREAD}$$

Actions are labelled by the thread that performed the action.

$$\frac{p_1 \xrightarrow{l_t} p_1'}{p_1|p_2 \xrightarrow{l_t} p_1'|p_2} \quad \text{PAR\_CONTEXT\_LEFT}$$

Free interleaving.

$$\frac{p_2 \xrightarrow{l_t} p_2'}{p_1|p_2 \xrightarrow{l_t} p_1|p_2'} \quad \text{PAR\_CONTEXT\_RIGHT}$$

A sequentially consistent memory

Take M to be a function from addresses to integers.

$\boxed{M \xrightarrow{l} M'}$ M does l to become M'

$$\frac{M(x) = n}{M \xrightarrow{\mathsf{R}\,x=n} M} \quad \textsc{MRead}$$

$$\frac{}{M \xrightarrow{\mathsf{W}\,x=n} M \oplus (x \mapsto n)} \quad \textsc{MWrite}$$

SC semantics: whole system transitions

$\boxed{s \xrightarrow{l_t} s'}$ s does l_t to become s'

$$\frac{p \xrightarrow{\mathsf{R}_t\, x=n} p' \qquad M \xrightarrow{\mathsf{R}\, x=n} M'}{\langle p,\, M\rangle \xrightarrow{\mathsf{R}_t\, x=n} \langle p',\, M'\rangle}\ \textsc{SRead}$$

$$\frac{p \xrightarrow{\mathsf{W}_t\, x=n} p' \qquad M \xrightarrow{\mathsf{W}\, x=n} M'}{\langle p,\, M\rangle \xrightarrow{\mathsf{W}_t\, x=n} \langle p',\, M'\rangle}\ \textsc{SWrite}$$

$$\frac{p \xrightarrow{\tau_t} p'}{\langle p,\, M\rangle \xrightarrow{\tau_t} \langle p',\, M\rangle}\ \textsc{STau}$$

Synchronising between the processes and the memory.

SC semantics, example

All threads read and write the shared memory. Threads execute asynchronously,the semantics allows any interleaving of the thread transitions.

$$\begin{tikzcd}
 & \langle t_1{:}{*x} = 1 | t_2{:}{*x} = 2,\ \{x \mapsto 0\}\rangle \arrow[dl, "\mathsf{W}_{t_1}\ x{=}1"'] \arrow[dr, "\mathsf{W}_{t_2}\ x{=}2"] & \\
\langle t_1{:}1 | t_2{:}{*x} = 2,\ \{x \mapsto 1\}\rangle \arrow[d, "\mathsf{W}_{t_2}\ x{=}2"'] & & \langle t_1{:}{*x} = 1 | t_2{:}2,\ \{x \mapsto 2\}\rangle \arrow[d, "\mathsf{W}_{t_1}\ x{=}1"] \\
\langle t_1{:}1 | t_2{:}2,\ \{x \mapsto 2\}\rangle & & \langle t_1{:}1 | t_2{:}2,\ \{x \mapsto 1\}\rangle
\end{tikzcd}$$

Each interleaving has a linear order of reads and writes to memory.

...now we just have to define a TSO memory...

A sequentially consistent memory

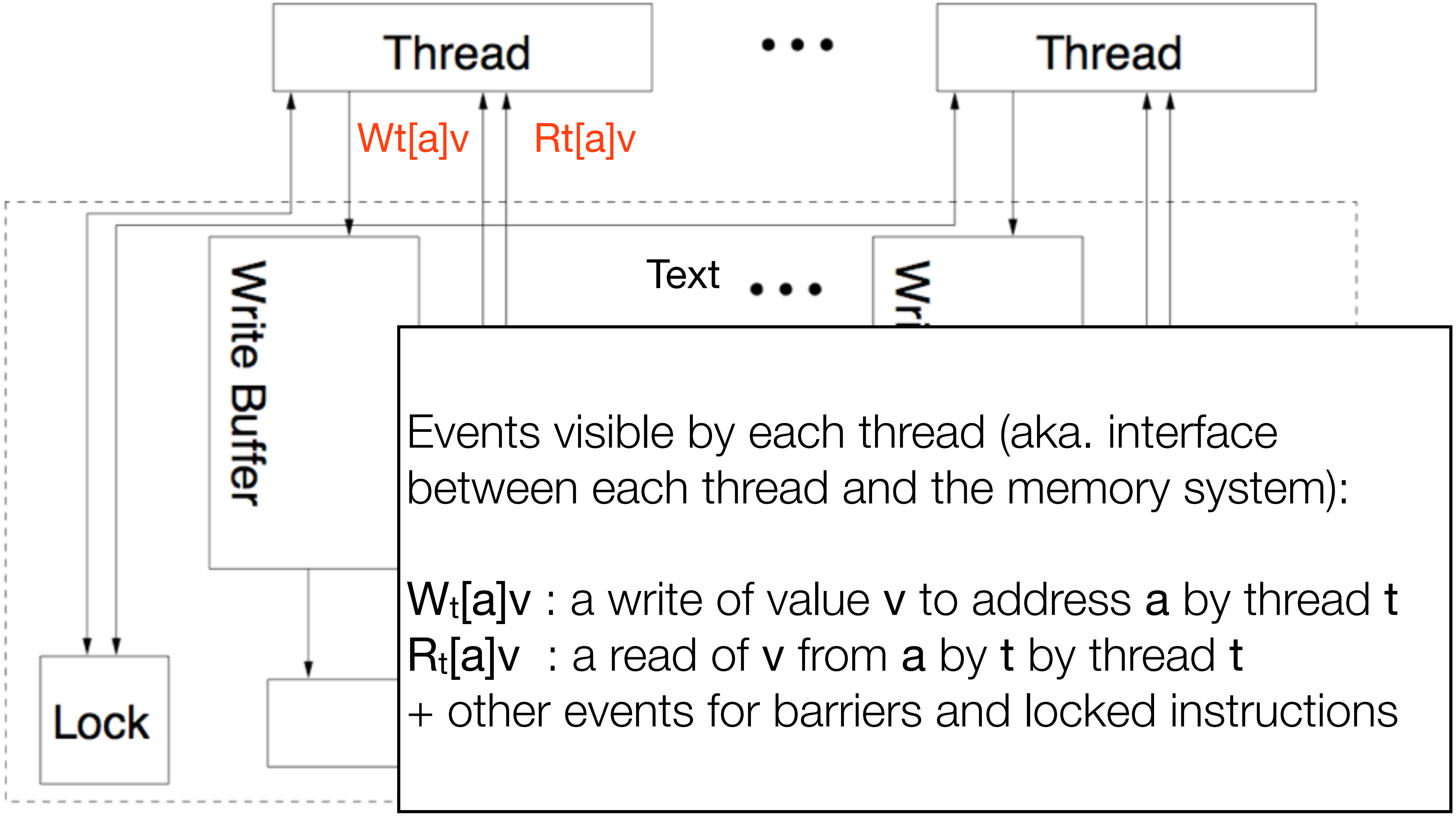
Take **M** to be a function from addresses to integers.

$\boxed{M \xrightarrow{l} M'}$ M **does** l **to become** M'

$$\frac{M(x) = n}{M \xrightarrow{\mathsf{R}\,x=n} M} \quad \textsc{MRead}$$

$$\frac{}{M \xrightarrow{\mathsf{W}\,x=n} M \oplus (x \mapsto n)} \quad \textsc{MWrite}$$

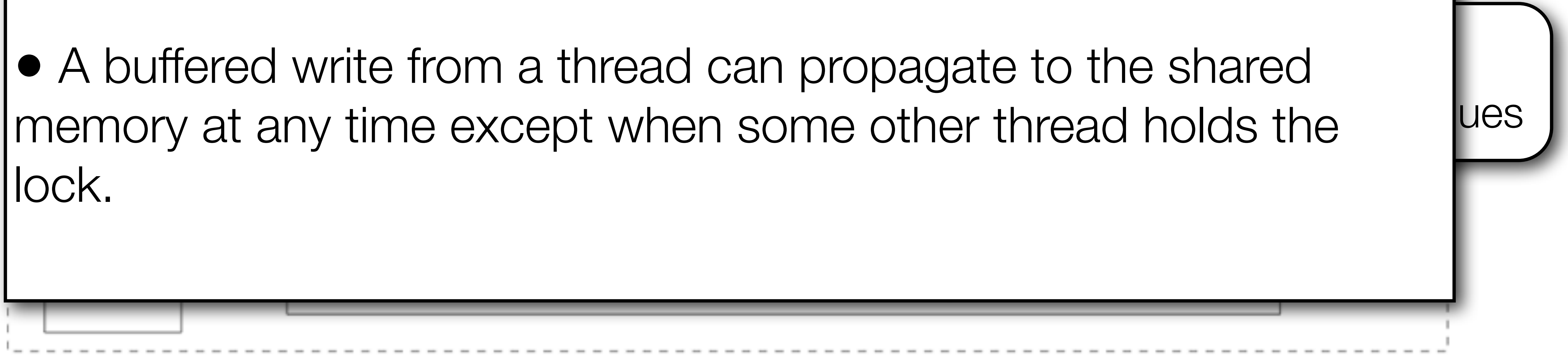
x86-TSO abstract machine



Events visible by each thread (aka. interface between each thread and the memory system):

$W_t[a]v$: a write of value **v** to address **a** by thread **t**
$R_t[a]v$: a read of **v** from **a** by **t** by thread **t**
+ other events for barriers and locked instructions

x86-TSO abstract machine



- The store buffers are FIFO. A reading thread must read its most recent buffered write, if there is one, to that address; otherwise reads are satisfied from shared memory.

- To execute a LOCK'd instruction, a thread must first obtain the global lock. At the end of the instruction, it flushes its store buffer and relinquishes the lock. While the lock is held by one thread, no other thread can read.

- A buffered write from a thread can propagate to the shared memory at any time except when some other thread holds the lock.

x86-tso: a formalisation using an LTS

The machine state `s` can be represented by a tuple `(M,B,L)`:

```
M : address -> value option
B : tid -> (address * value) list
L : tid option
```

where:

`M` is the shared memory, mapping addresses to values

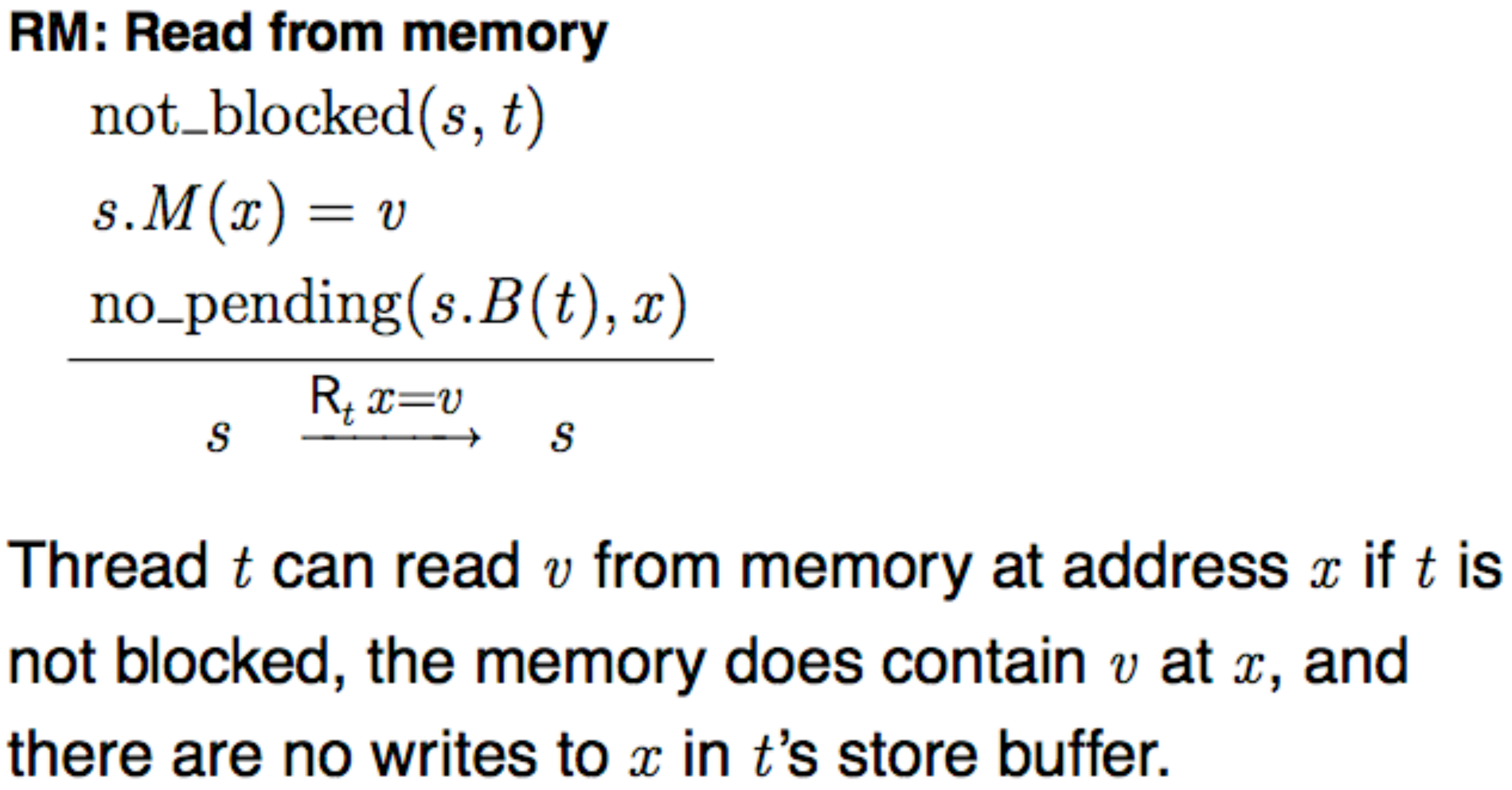
`B` gives the store buffer for each thread

`L` is the global machine lock indicating when a thread has exclusive access to memory (omitted in these slides)

x86-tso abstract machine: selected transition rules

t is *not blocked* in machine state s = (M,B,L) if [… or] the lock is not held.

In buffer B(t) there are *no pending writes* for address x if there are no (x,v) elements in B(t).



RM: Read from memory

$$\frac{\begin{array}{l}\text{not\_blocked}(s,t)\\ s.M(x)=v\\ \text{no\_pending}(s.B(t),x)\end{array}}{s \xrightarrow{\mathsf{R}_t\, x=v} s}$$

Thread t can read v from memory at address x if t is not blocked, the memory does contain v at x, and there are no writes to x in t's store buffer.

x86-tso abstract machine: selected transition rules

RB: Read from write buffer

$$\frac{\begin{array}{l}\text{not\_blocked}(s,t)\\ \exists b_1\, b_2.\ s.B(t) = b_1 \mathbin{++} [(x,v)] \mathbin{++} b_2\\ \text{no\_pending}(b_1, x)\end{array}}{s \xrightarrow{\mathsf{R}_t\, x=v} s}$$

Thread t can read v from its store buffer for address x if t is not blocked and has v as the newest write to x in its buffer;

x86-tso abstract machine: selected transition rules

WB: Write to write buffer

$$\frac{}{s \xrightarrow{\mathsf{W}_t\, x=v} s \oplus \langle\!| B := s.B \oplus (t \mapsto ([(x,v)] +\!\!+ s.B(t))) |\!\rangle}$$

Thread t can write v to its store buffer for address x at any time;

WM: Write from write buffer to memory

$$\frac{\begin{array}{l}\text{not\_blocked}(s,t)\\ s.B(t) = b +\!\!+ [(x,v)]\end{array}}{\begin{array}{l} s \xrightarrow{\tau_t\, x=v} \\ s \oplus \langle\!| M := s.M \oplus (x \mapsto v) |\!\rangle \oplus \langle\!| B := s.B \oplus (t \mapsto b) |\!\rangle \end{array}}$$

If t is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread

5. Hunting compiler concurrency bugs

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Thread 1 returns without modifying `b`.

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Thread 1 returns without modifying `b`.

Since Thread 1 does not update `b`, program is *data-race free (DRF)*

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Thread 1 returns without modifying `b`.

Since Thread 1 does not update `b`, program is *data-race free (DRF)*

DRF programs must only exhibit sequentially consistent behaviours

C11/C++11 standard

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Thread 1 returns without modifying `b`.

Since Thread 1 does not update `b`, program is *data-race free (DRF)*

DRF programs must only exhibit sequentially consistent behaviours

C11/C++11 standard

This program only prints `42`.

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

gcc 4.7 -O2

...sometimes we get `0` on the screen

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

gcc 4.7 -O2

```
 movl   a(%rip), %edx     # load a into edx
 movl   b(%rip), %eax     # load b into eax
 testl  %edx, %edx        # if a!=0
 jne    .L2               # jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %eax, b(%rip)     # store eax into b
 xorl   %eax, %eax        # store 0 into eax
 ret                      # return
```

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

gcc 4.7 -O2

The outer loop can be (and is) optimised away

```
 movl   a(%rip), %edx     # load a into edx
 movl   b(%rip), %eax     # load b into eax
 testl  %edx, %edx        # if a!=0
 jne    .L2               # jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %eax, b(%rip)     # store eax into b
 xorl   %eax, %eax        # store 0 into eax
 ret                      # return
```

```c
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

gcc 4.7 -O2

```
 movl  a(%rip), %edx     # load a into edx
 movl  b(%rip), %eax     # load b into eax
 testl %edx, %edx        # if a!=0
 jne   .L2               # jump to .L2
 movl  $0, b(%rip)
 ret
.L2:
 movl  %eax, b(%rip)     # store eax into b
 xorl  %eax, %eax        # store 0 into eax
 ret                     # return
```

```c
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

gcc 4.7 -O2

```
 movl   a(%rip), %edx     # load a into edx
 movl   b(%rip), %eax     # load b into eax
 testl  %edx, %edx        # if a!=0
 jne    .L2               # jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %eax, b(%rip)     # store eax into b
 xorl   %eax, %eax        # store 0 into eax
 ret                      # return
```

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

gcc 4.7 -O2

```
 movl   a(%rip), %edx    # load a into edx
 movl   b(%rip), %eax    # load b into eax
 testl  %edx, %edx       # if a!=0
 jne    .L2              # jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %eax, b(%rip)    # store eax into b
 xorl   %eax, %eax       # store 0 into eax
 ret                     # return
```

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

gcc 4.7 -O2

```
movl   a(%rip), %edx    # load a into edx
movl   b(%rip), %eax    # load b into eax
testl  %edx, %edx       # if a!=0
jne    .L2              # jump to .L2
movl   $0, b(%rip)
ret
.L2:
movl   %eax, b(%rip)    # store eax into b
xorl   %eax, %eax       # store 0 into eax
ret                     # return
```

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

gcc 4.7 -O2

```
 movl   a(%rip), %edx     # load a into edx
 movl   b(%rip), %eax     # load b into eax
 testl  %edx, %edx        # if a!=0
 jne    .L2               # jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %eax, b(%rip)     # store eax into b
 xorl   %eax, %eax        # store 0 into eax
 ret                      # return
```

The compiled code saves and restores b

Correct in a sequential setting, but...

```
 movl   a(%rip), %edx     # load a into edx
 movl   b(%rip), %eax     # load b into eax
 testl  %edx, %edx        # if a!=0
 jne    .L2               # jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %eax, b(%rip)     # store eax into b
 xorl   %eax, %eax        # store 0 into eax
 ret                      # return
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
  movl   a(%rip),%edx
  movl   b(%rip),%eax
  testl  %edx, %edx
  jne    .L2
  movl   $0, b(%rip)
  ret
.L2:
  movl   %eax, b(%rip)
  xorl   %eax, %eax
  ret
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
  movl    a(%rip),%edx
  movl    b(%rip),%eax
  testl   %edx, %edx
  jne     .L2
  movl    $0, b(%rip)
  ret
.L2:
  movl    %eax, b(%rip)
  xorl    %eax, %eax
  ret
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

- Read a (1) into edx

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
  movl    a(%rip),%edx
  movl    b(%rip),%eax
  testl   %edx, %edx
  jne     .L2
  movl    $0, b(%rip)
  ret
.L2:
  movl    %eax, b(%rip)
  xorl    %eax, %eax
  ret
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

- Read a (1) into edx
- Read b (0) into eax

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
  movl   a(%rip),%edx
  movl   b(%rip),%eax
  testl  %edx, %edx
  jne    .L2
  movl   $0, b(%rip)
  ret
.L2:
  movl   %eax, b(%rip)
  xorl   %eax, %eax
  ret
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

- Read a (1) into edx
- Read b (0) into eax
- Store 42 into b

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
  movl    a(%rip),%edx
  movl    b(%rip),%eax
  testl   %edx, %edx
  jne     .L2
  movl    $0, b(%rip)
  ret
.L2:
  movl    %eax, b(%rip)
  xorl    %eax, %eax
  ret
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

- Read a (1) into edx
- Read b (0) into eax
- Store 42 into b
- Store eax (0) into b

Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
  movl    a(%rip),%edx
  movl    b(%rip),%eax
  testl   %edx, %edx
  jne     .L2
  movl    $0, b(%rip)
  ret
.L2:
  movl    %eax, b(%rip)
  xorl    %eax, %eax
  ret
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

- Read a (1) into edx
- Read b (0) into eax
- Store 42 into b
- Store eax (0) into b
- Print b... 0 is printed

The compiled code saves and restores **b**

Correct in a sequential setting

Introduces unexpected behaviours in some concurrent context

```
  ret
.L2:
  movl    %eax, b(%rip)
  xorl    %eax, %eax
  ret
```

- Read a (1) into edx
- Read b (0) into eax
- Store 42 into b
- Store eax (0) into b
- Print b... 0 is printed

The compiled code saves and restores **b**

Correct in a sequential setting

Introduces unexpected behaviours in some concurrent context

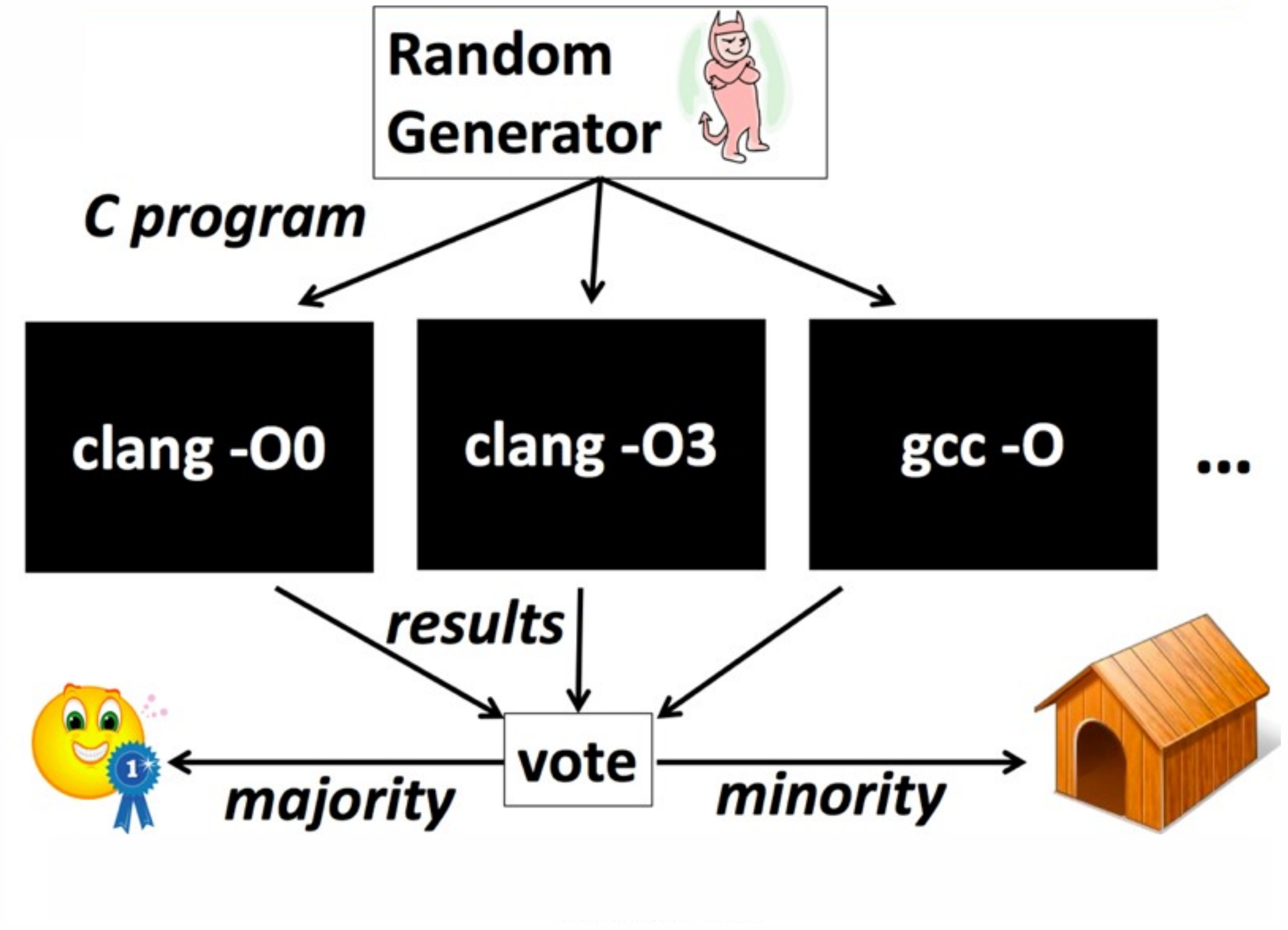
This is a *concurrency compiler bug*

```
movl    %eax, b(%rip)
xorl    %eax, %eax
ret
```

- Read b (0) into eax
- Store 42 into b
- Store eax (0) into b
- Print b... 0 is printed

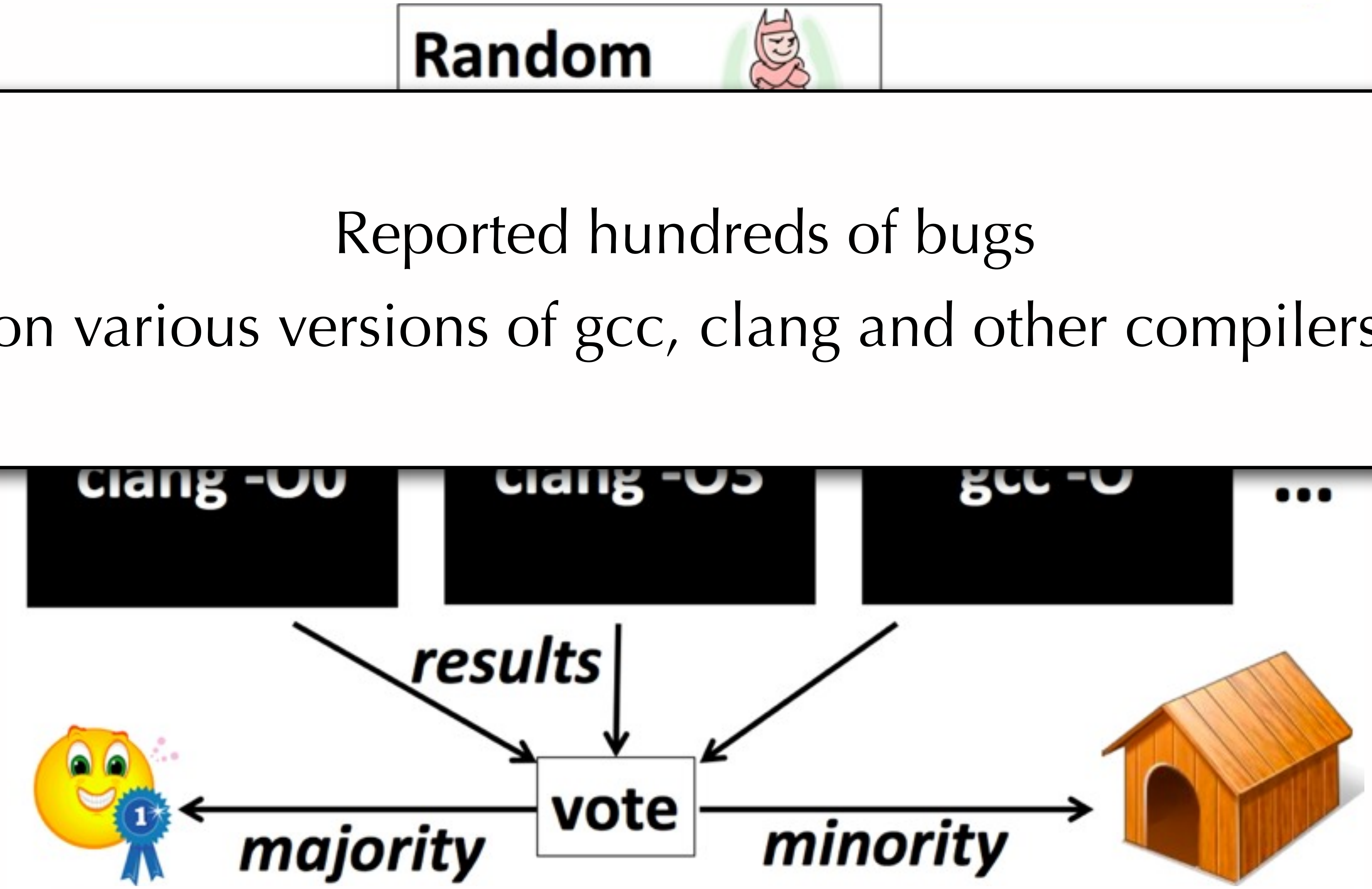
Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

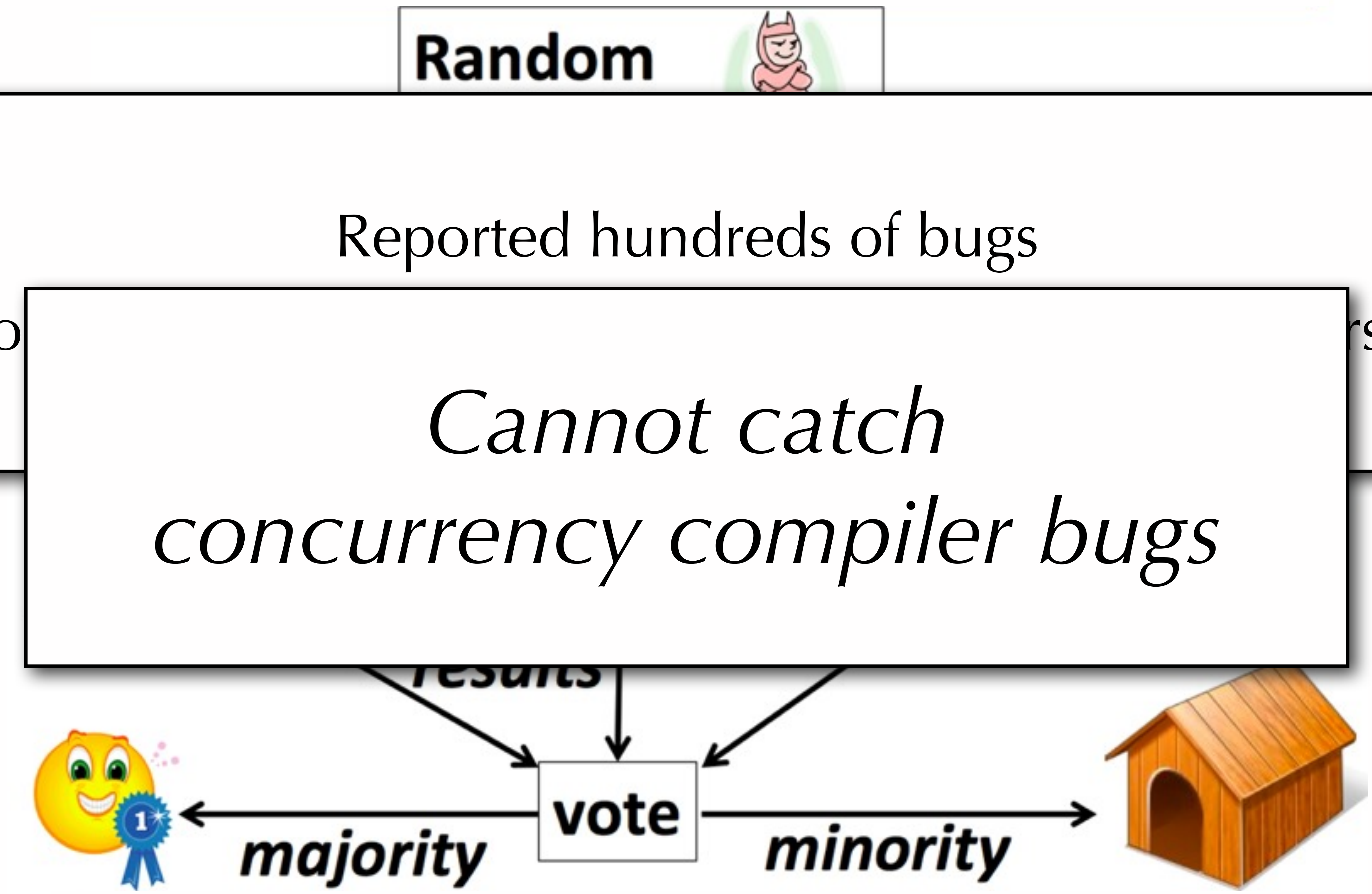


Reported hundreds of bugs

on various versions of gcc, clang and other compilers

Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



Reported hundreds of bugs

o rs

Cannot catch concurrency compiler bugs

Hunting *concurrency* compiler bugs?

How to deal with non-determinism?

How to generate non-racy interesting programs?

How to capture all the behaviours of concurrent programs?

A compiler can optimise away behaviours:

how to test for correctness?

limit case: two compilers generate correct code with disjoint final states

Idea

C/C++ compilers support separate compilation
Functions can be called in arbitrary non-racy concurrent contexts

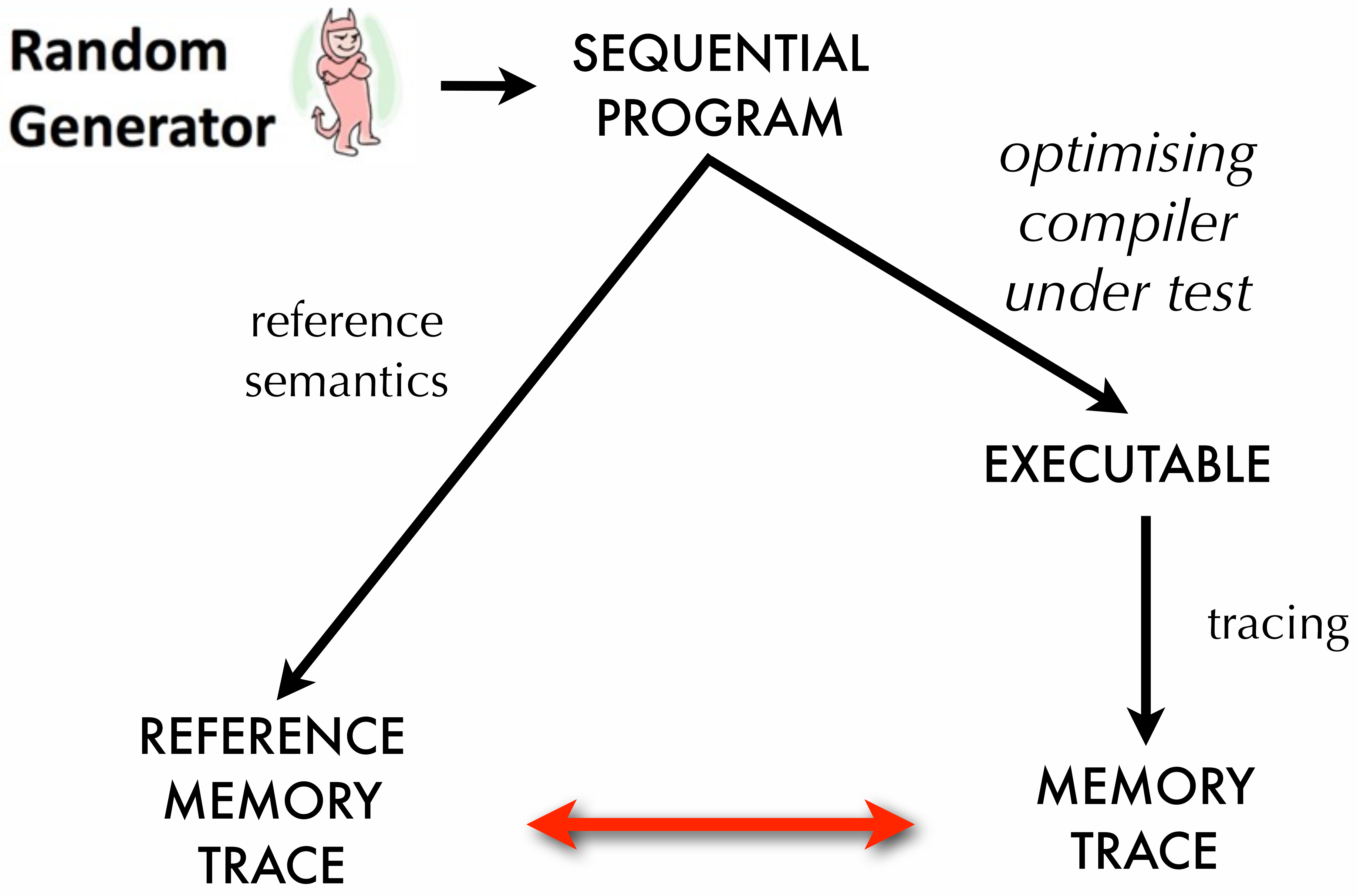
$\downarrow$

C/C++ compilers can only apply transformations sound with respect to an arbitrary non-racy concurrent context

Hunt concurrency compiler bugs

=

search for transformations of sequential code not sound in an arbitrary non-racy context



only transformations sound in any concurrent non-racy context?

```
int a = 1;
int b = 0;
```

```
int s;
for (s=0; s!=4; s++) {
  if (a==1)
    return NULL;
  for (b=0; b>=26; ++b)
    ;
}
```

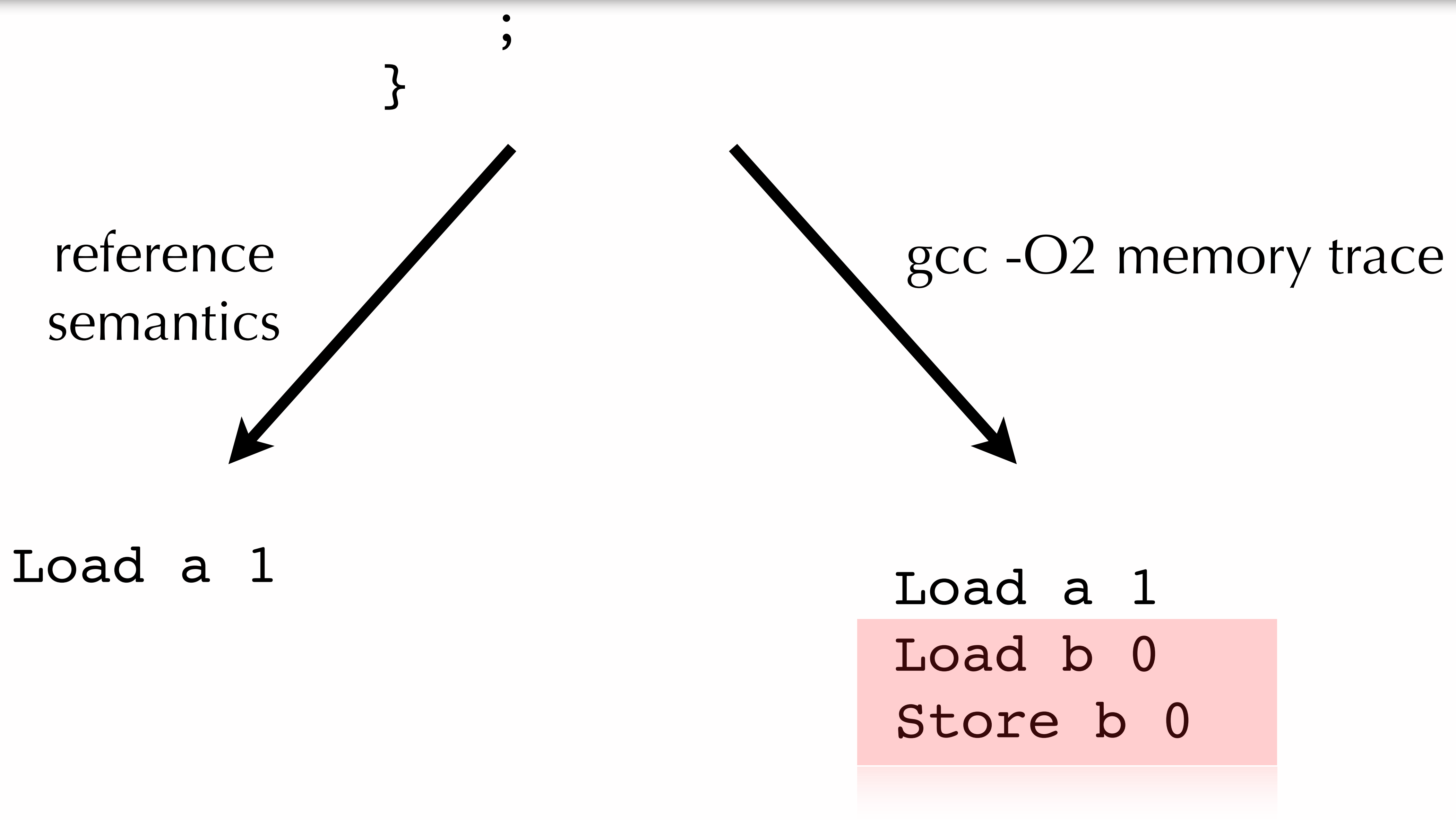
reference semantics

```
Load a 1
```

gcc -O2 memory trace

```
Load a 1
Load b 0
Store b 0
```

Cannot match some events $\longrightarrow$ detect compiler bug



Contributions

Sound optimisations in the C11/C++11 memory model

extending Sevcik's work on an idealised DRF model - PLDI 11

A tool to hunt concurrency bugs in C and C++ compilers

Interaction with GCC developers

Sound Optimisations in the C11/C++11 Memory Model

Example: loop invariant code motion

Compiler Writer

Semanticist

Example: loop invariant code motion

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

Example: loop invariant code motion

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

```
for (int i=0; i<2; i++) {
  z = i;
  x[i] += y+1 ;
}
```

Example: loop invariant code motion

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

```
tmp = y+1 ;
for (int i=0; i<2; i++) {
  z = i;
  x[i] += tmp ;
}
```

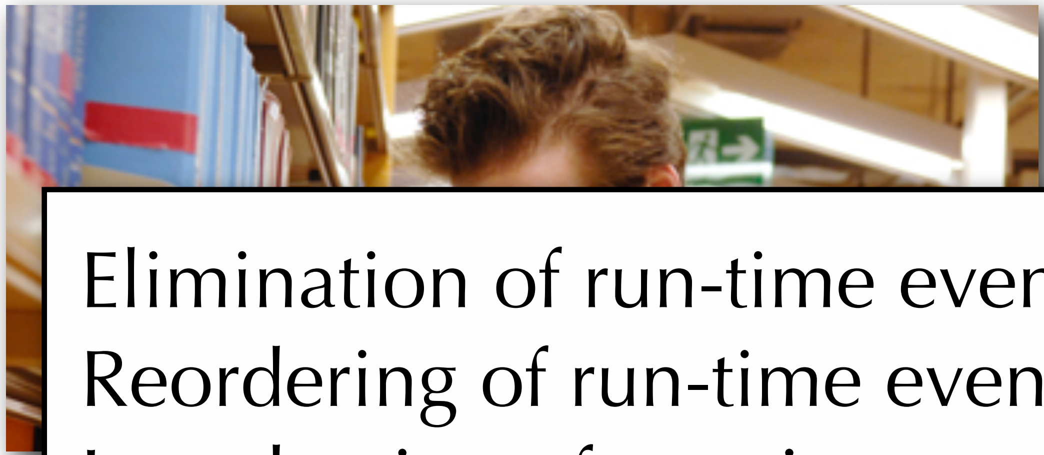
Example: loop invariant code motion

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist



Elimination of run-time events
Reordering of run-time events
Introduction of run-time events

Operations on sets of events

```
tmp = y+1 ;
for (int i=0; i<2; i++) {
  z = i;
  x[i] += tmp ;
}
```

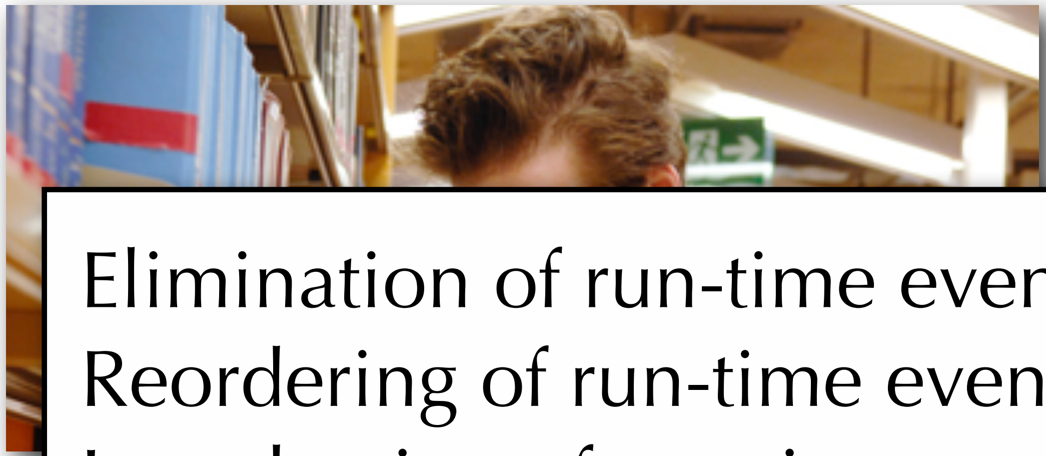
Example: loop invariant code motion

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist



Elimination of run-time events
Reordering of run-time events
Introduction of run-time events

Operations on sets of events

```
tmp = y+1 ;
for (int i=0; i<2; i++) {
  z = i;
  x[i] += tmp ;
}
```

```
Store z 0
Load y 42
Store x[0] 43
Store z 1
Load y 42
Store x[1] 43
```

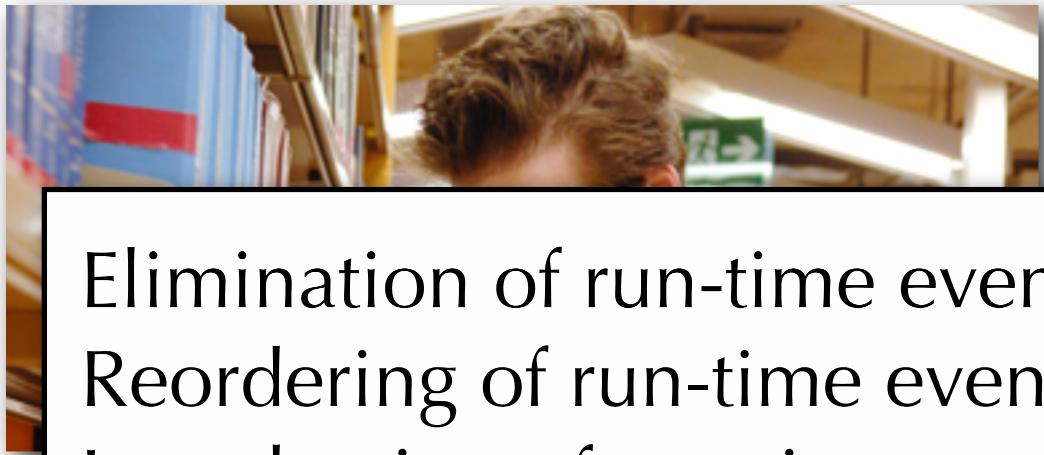
Example: loop invariant code motion

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist



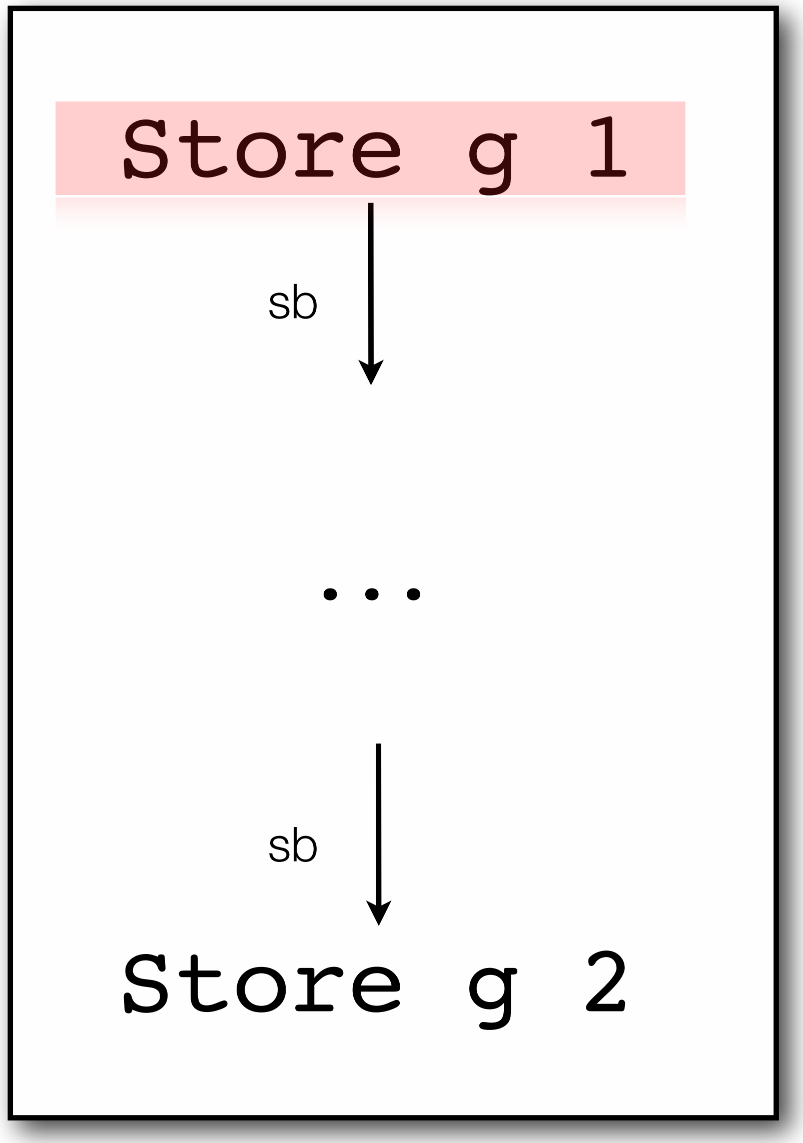
Elimination of run-time events
Reordering of run-time events
Introduction of run-time events

Operations on sets of events

```
tmp = y+1 ;
for (int i=0; i<2; i++) {
  z = i;
  x[i] += tmp ;
}
```

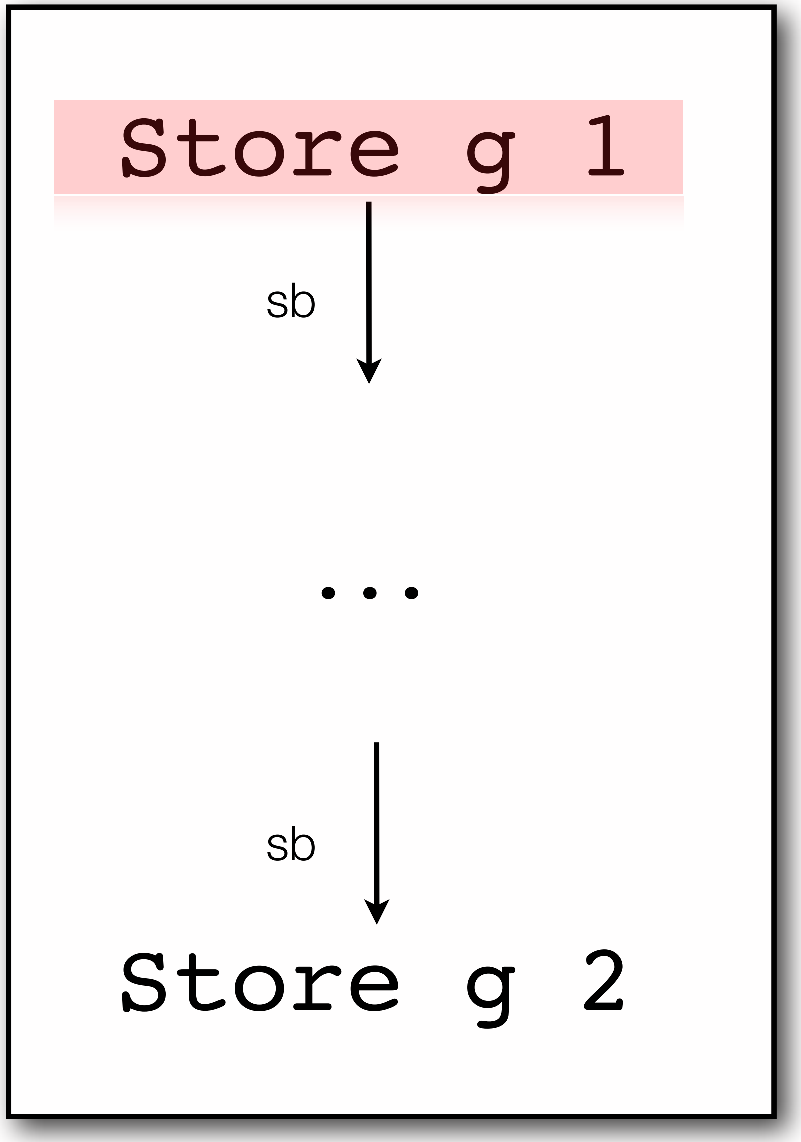
```
Load y 42
Store z 0

Store x[0] 43
Store z 1

Store x[1] 43
```

Elimination of *overwritten writes*



Under which conditions is it correct to eliminate the first store?

Elimination of *overwritten writes*



Under which conditions is it correct to eliminate the first store?

What is the semantics of C11/C++11 concurrent code?

The C11/C++11 memory model

C11/C++11 are based on the DRF approach:

— racy code is undefined
— race-free code must exhibit only sequentially consistent behaviours
— main synchronisation mechanism: lock/unlock

Escape mechanism for experts, *low-level atomics*:

— races allowed
— attributes on accesses specify their semantics:

```
MO_SEQ_CST    MO_RELEASE/MO_ACQUIRE    MO_RELAXED
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;
f.store(1,MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE)==0);
printf (“%d”,g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;
f.store(1,MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE)==0);
printf (“%d”,g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;
f.store(1,MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE)==0);
printf (“%d”,g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;
f.store(1,MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE)==0);
printf (“%d”,g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;
f.store(1,MO_RELEASE);
```

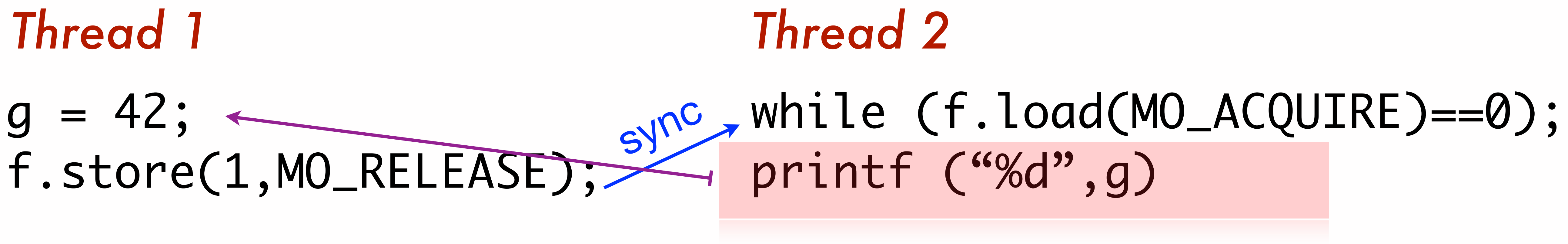
$\xrightarrow{\text{sync}}$

Thread 2

```
while (f.load(MO_ACQUIRE)==0);
printf (“%d”,g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```



Thread 1

```
g = 42;
f.store(1,MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE)==0);
printf ("%d",g)
```

The release/acquire synchronisation guarantees that:

— the program is DRF

— `42` is printed at the end of the execution

Remark: unlock ≃ release, lock ≃ acquire.

Same-thread release/acquire pairs

A *same-thread release-acquire pair* is a pair of a *release action* followed by an *acquire action* in program order.

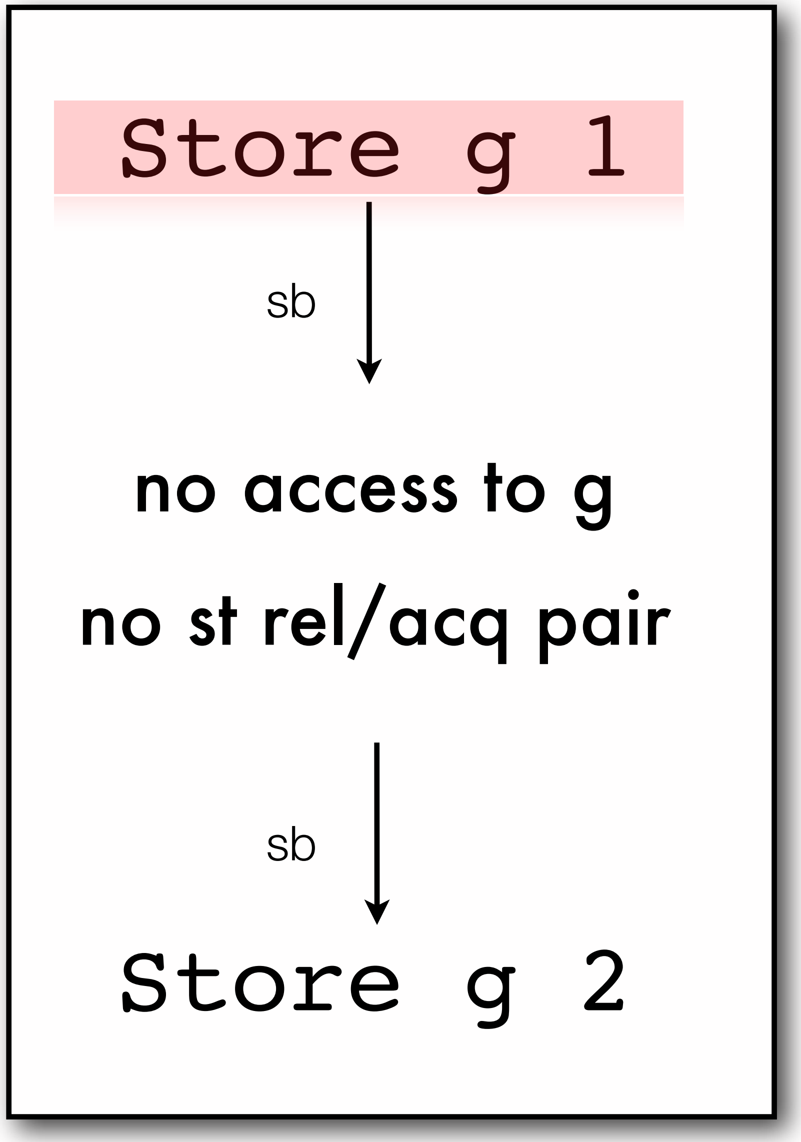
An action is a *release* if it is a possible source of a synchronisation

unlock mutex, release or seq_cst atomic write

An action is an *acquire* if it is a possible target of a synchronisation

lock mutex, acquire or seq_cst atomic read

Elimination of *overwritten writes*



It is safe to eliminate the first store
if there are:

1. no intervening accesses to g
2. no intervening
   same-thread release-acquire pairs

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

candidate overwritten write

```
g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

candidate overwritten write

```
g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

same-thread release-acquire pair

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf(“%d”, g);
f2.store(1,RELEASE);
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

sync

sync

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

Thread 2 is non-racy

The soundness condition

Shared memory

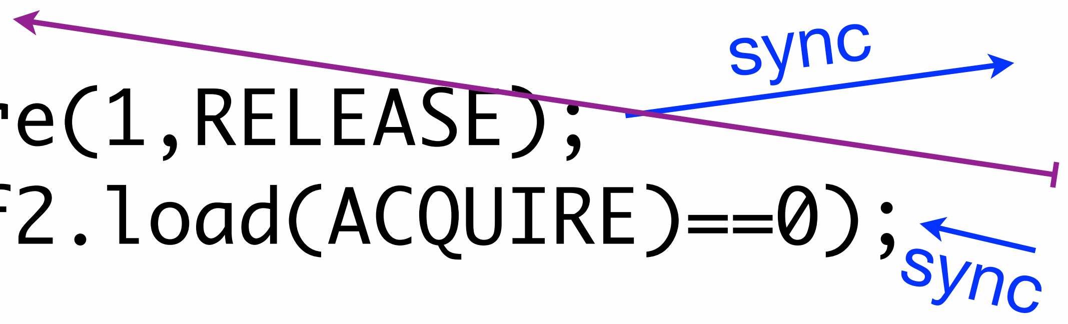
```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

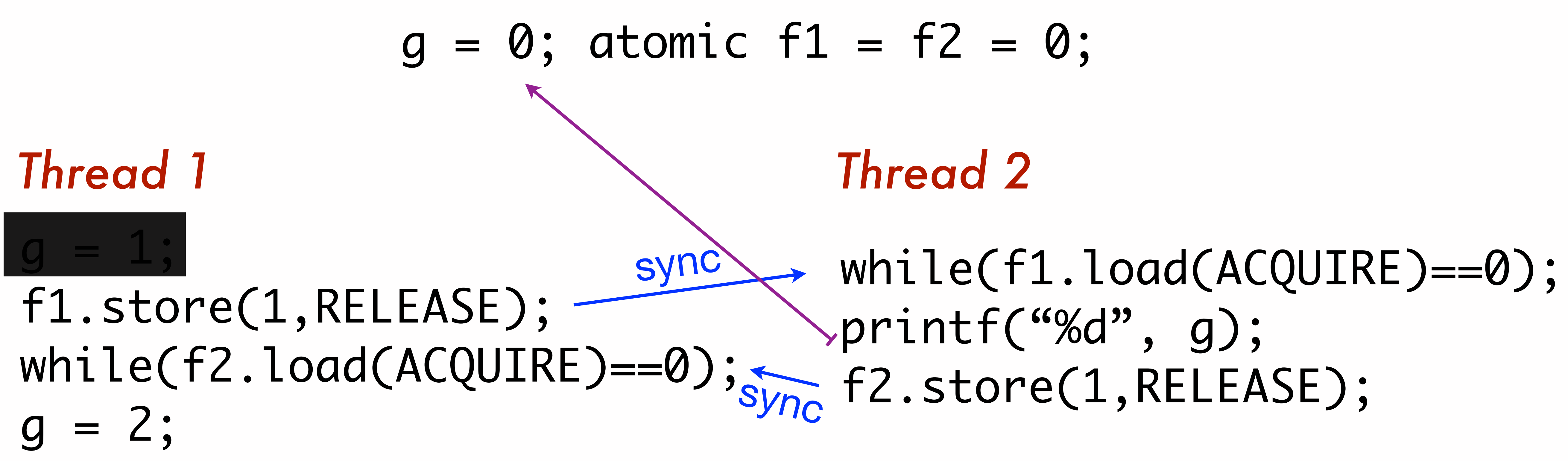


Thread 2 is non-racy

The program should only print `1`

The soundness condition

Shared memory



```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
[redacted]
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

Thread 2 is non-racy

The program should only print `1`

If we perform overwritten write elimination it prints `0`

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

$\xrightarrow{\text{sync}}$

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
f1.store(1,RELEASE);

g = 2;
```

sync

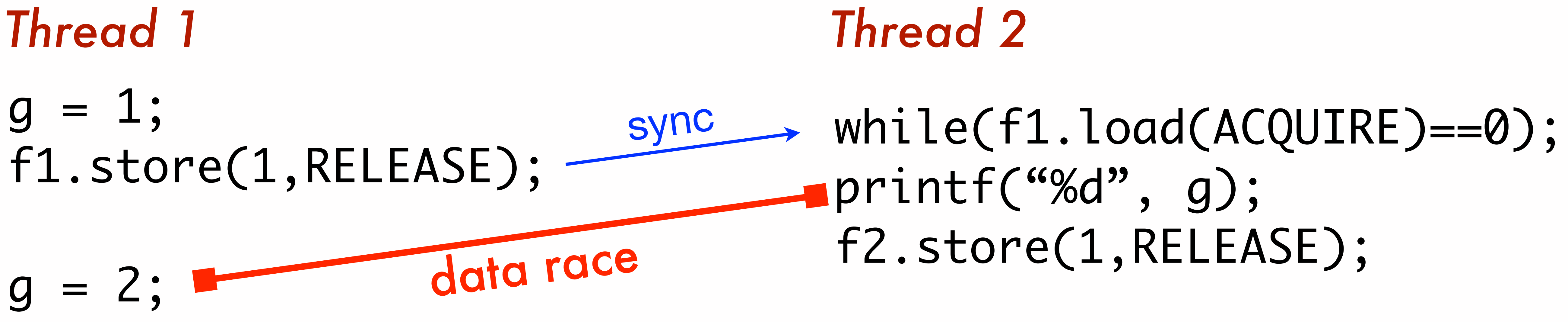
Thread 2

```
while(f1.load(ACQUIRE)==0);
printf(“%d”, g);
f2.store(1,RELEASE);
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```



Thread 1

```
g = 1;
f1.store(1,RELEASE);

g = 2;
```

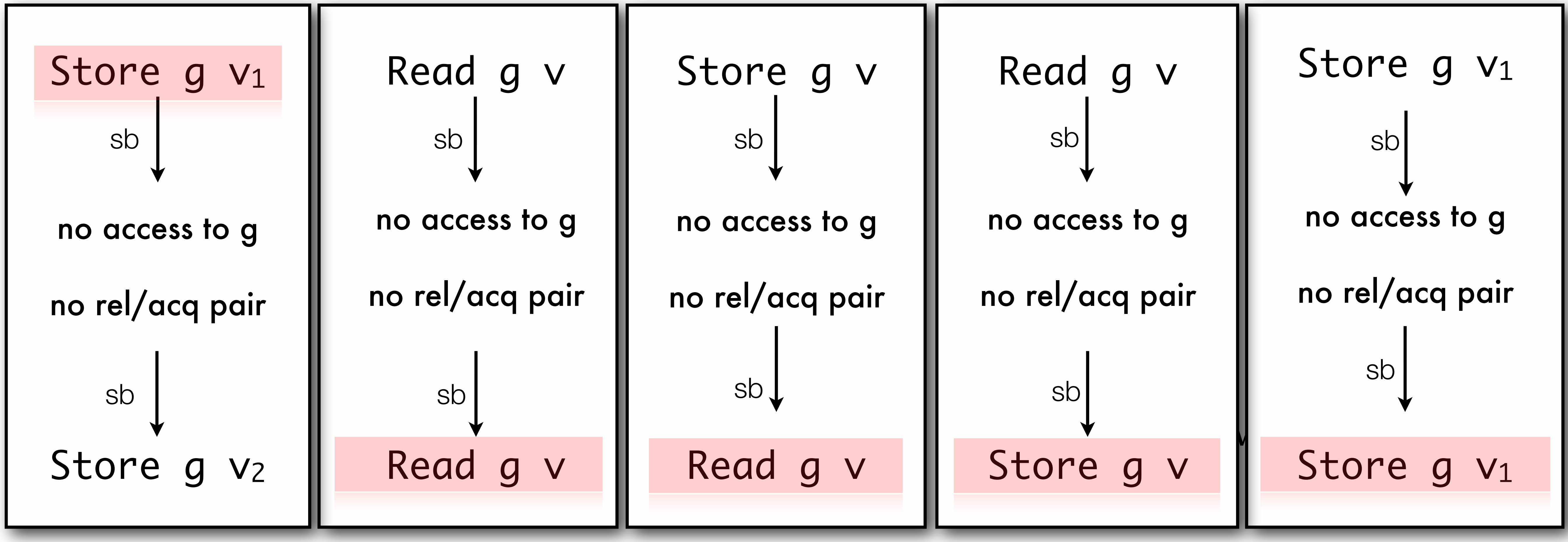
Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

If only a release (or acquire) is present, then all discriminating contexts *are racy*.

It is sound to optimise the overwritten write.

Eliminations: bestiary



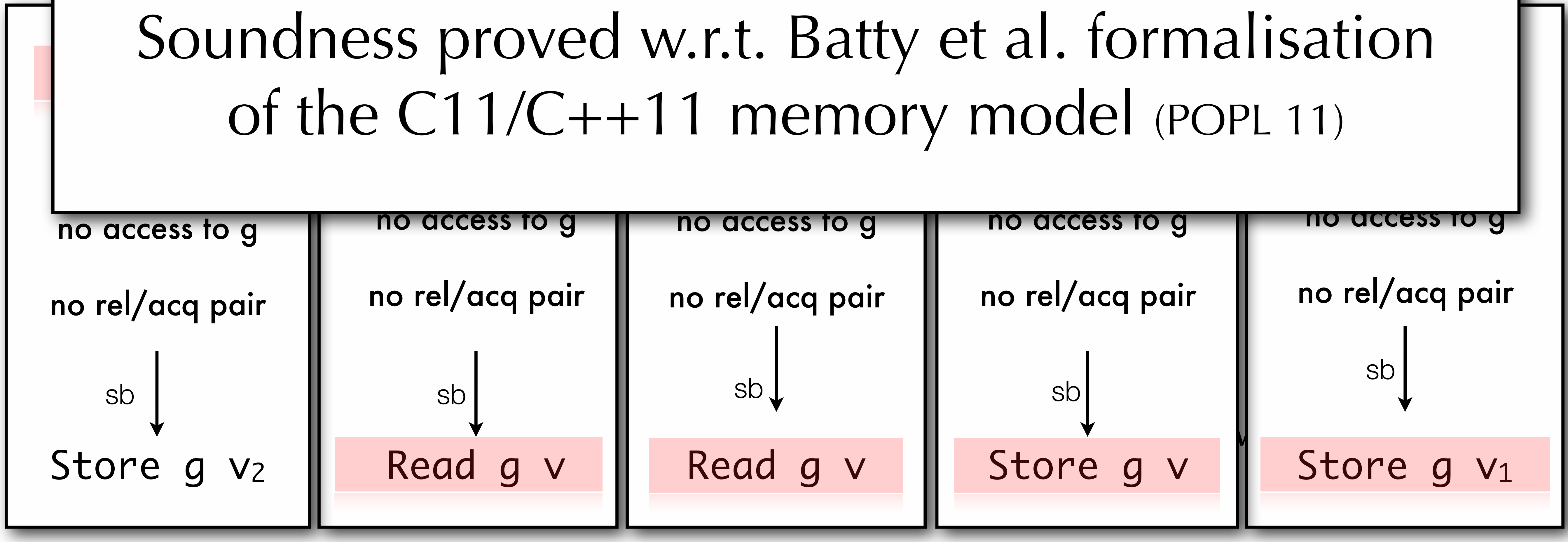
Overwritten-Write Read-after-Read Read-after-Write Write-after-Read Write-after-Write

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (*irrelevant reads*).

Eliminations: bestiary

Theorem

Soundness proved w.r.t. Batty et al. formalisation of the C11/C++11 memory model (POPL 11)



Overwritten-Write Read-after-Read Read-after-Write Write-after-Read Write-after-Write

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (*irrelevant reads*).

Reorderings and introductions

Correctness criterion for reordering events:

— different addresses

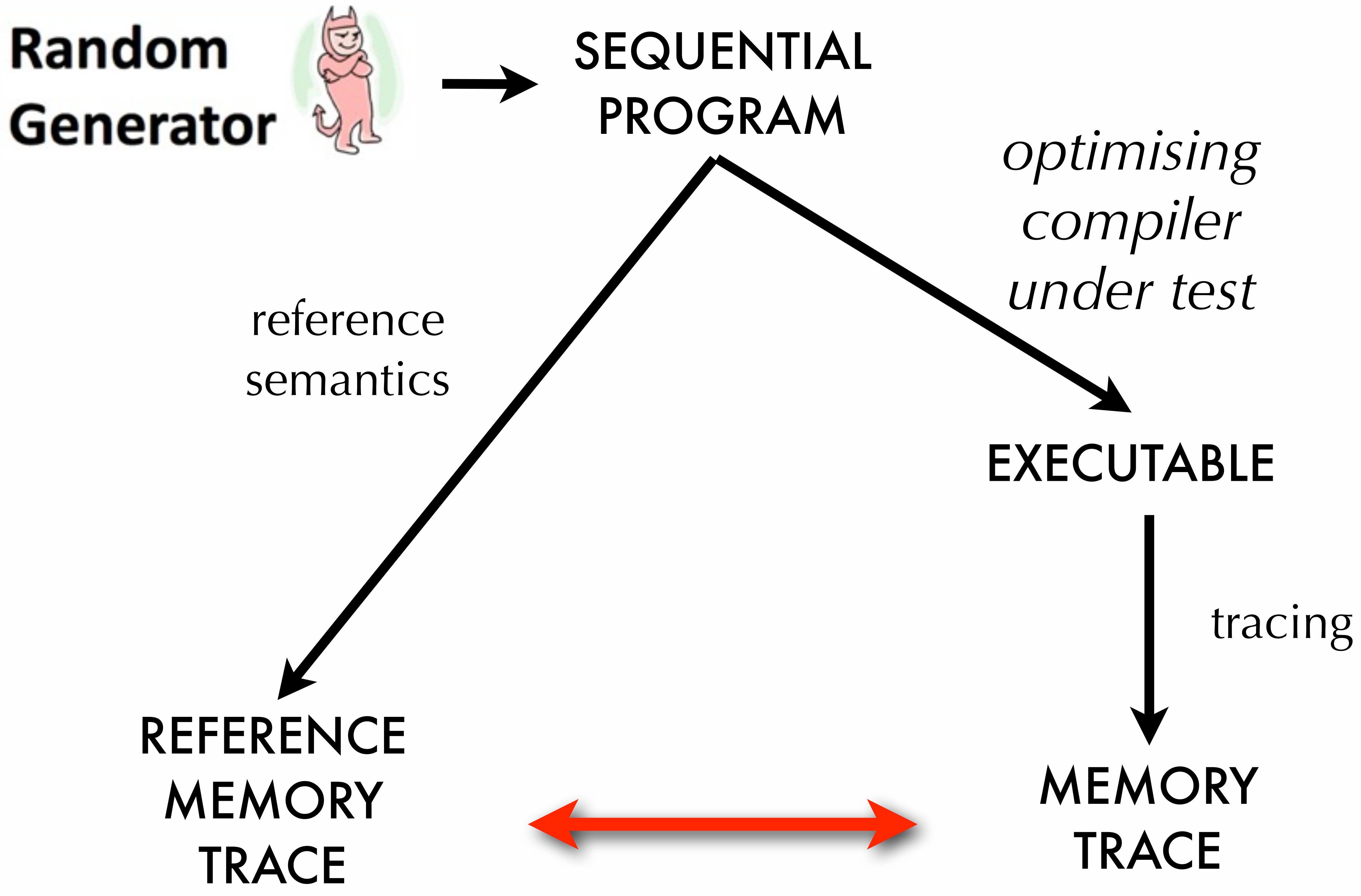
— no synchronisations in-between

Roach-motel reordering (reordering across locks) not observed in practice

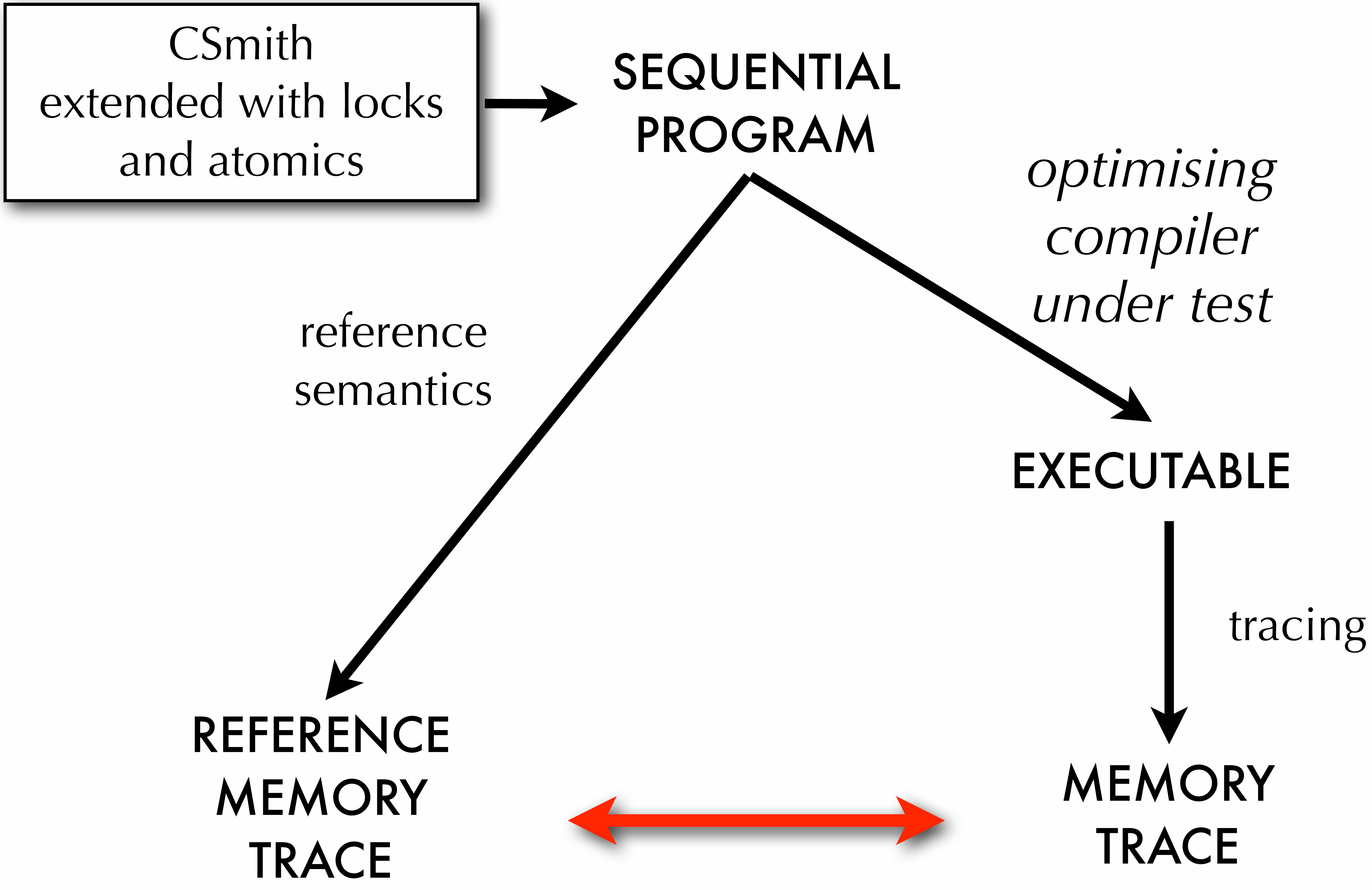
Read introductions observed in practice (gcc, clang).

Introduction of eliminable reads proved correct.
Introduction of irrelevant reads does not introduce new behaviours, but cannot be proved correct in a DRF model.

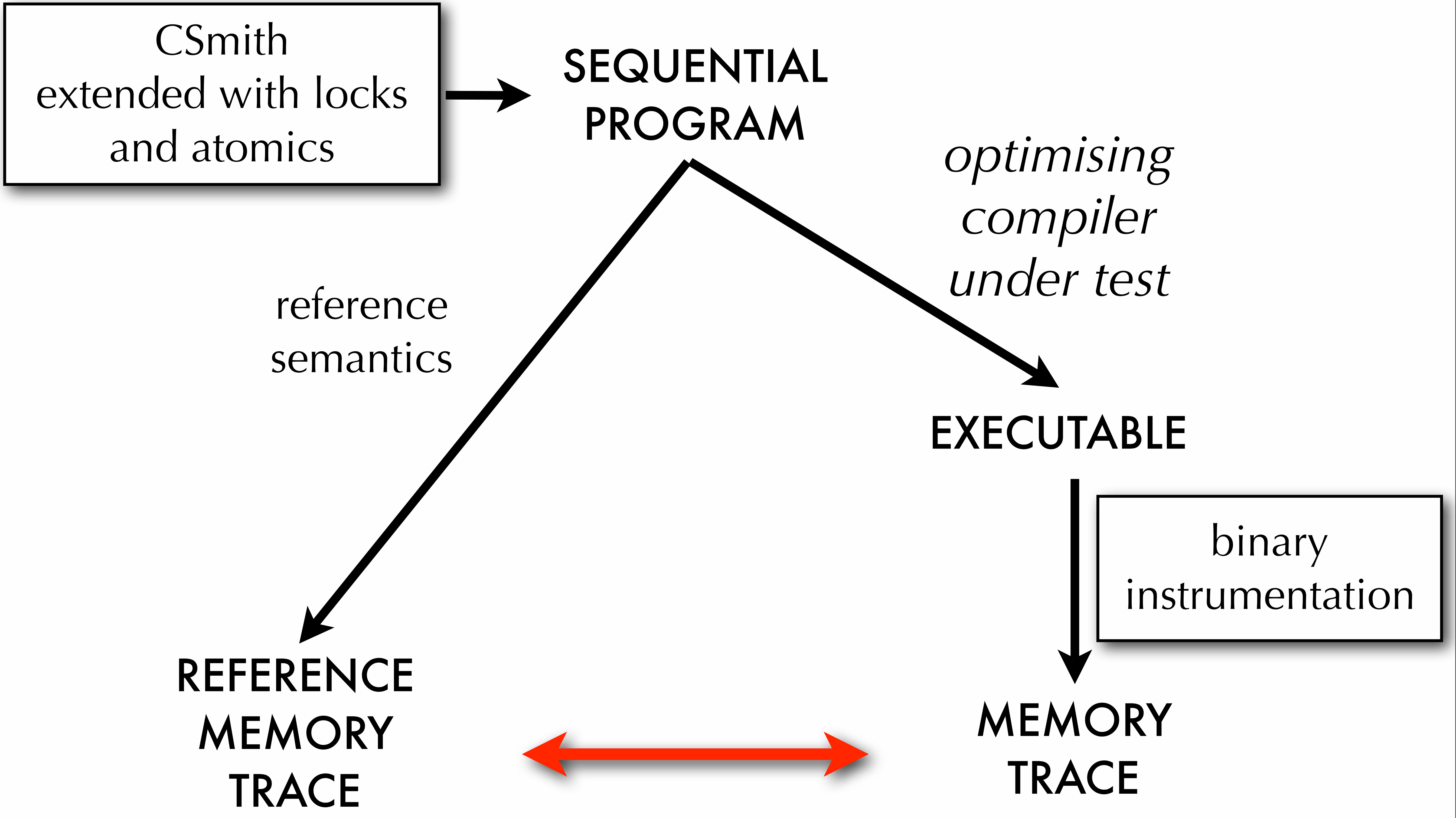
The CMMTEST Tool



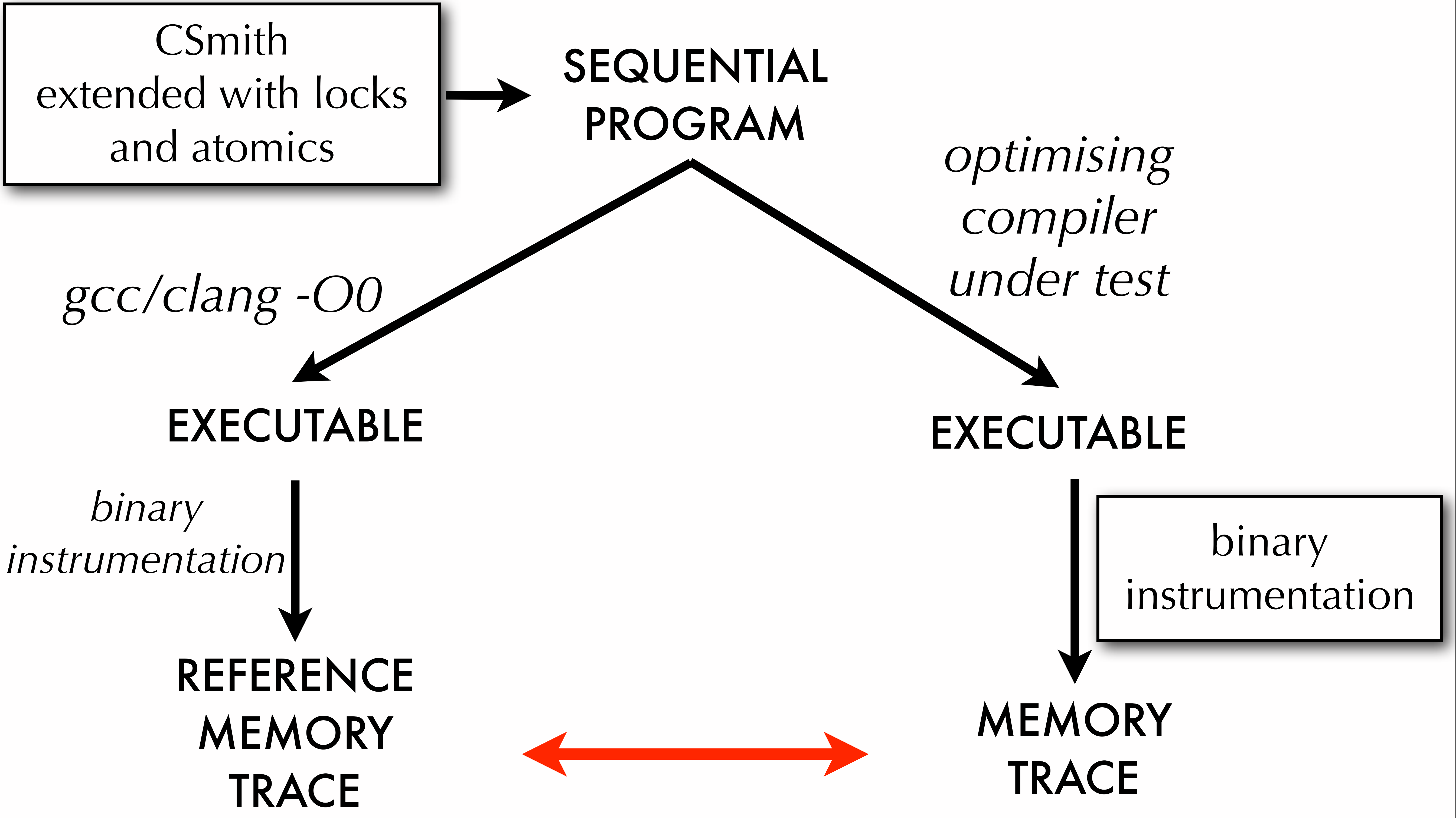
only transformations sound in any concurrent (non-racy) context?



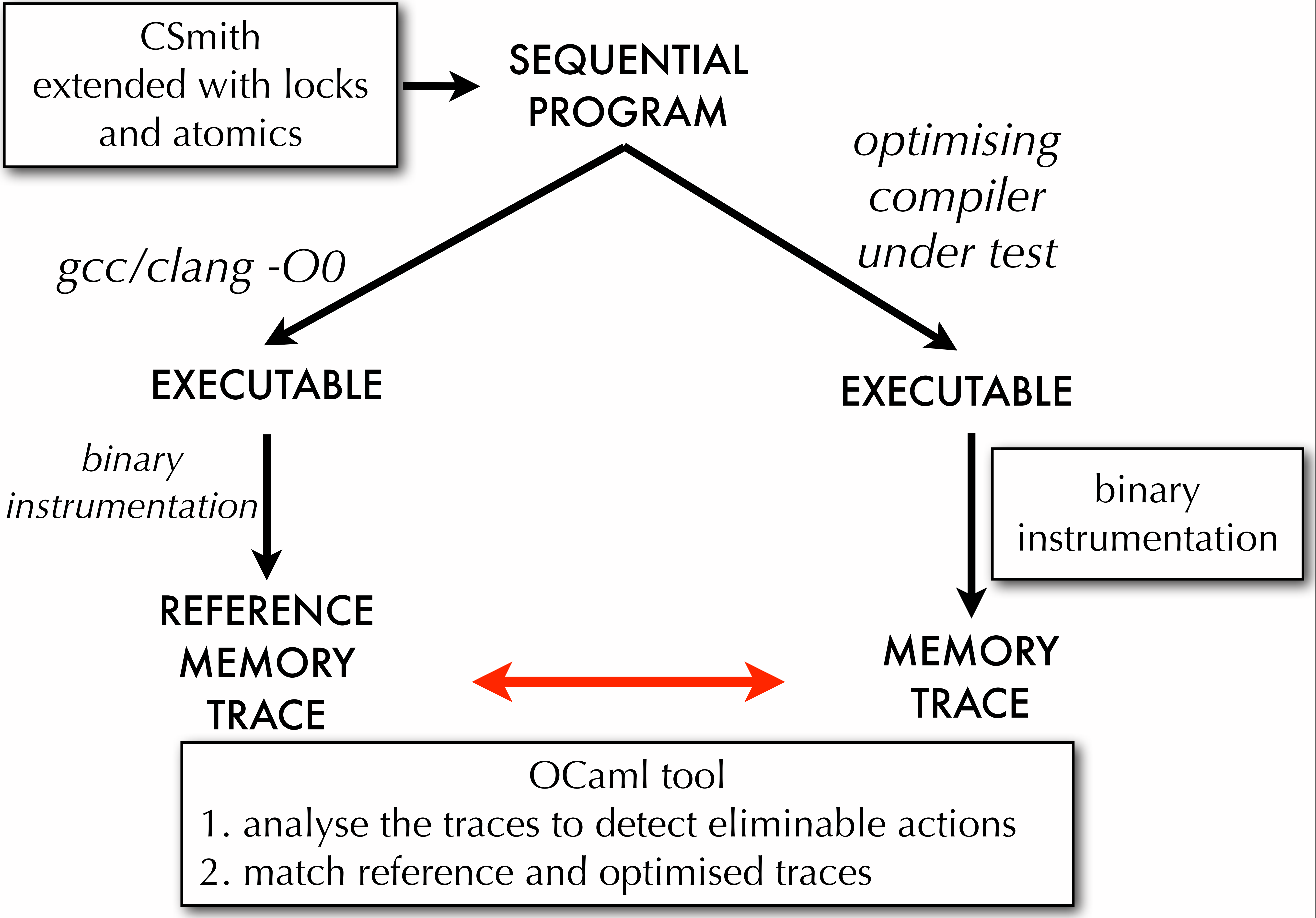
only transformations sound in any concurrent (non-racy) context?



only transformations sound in any concurrent (non-racy) context?



only transformations sound in any concurrent (non-racy) context?


CSmith extended with locks and atomics
SEQUENTIAL PROGRAM
gcc/clang -O0
optimising compiler under test
EXECUTABLE
EXECUTABLE
binary instrumentation
binary instrumentation
REFERENCE MEMORY TRACE
MEMORY TRACE
OCaml tool
1. analyse the traces to detect eliminable actions
2. match reference and optimised traces

CSmith
extended with locks
and atomics

→ **SEQUENTIAL PROGRAM**

optimising compiler

Subtleties:

- *dependencies between eliminable events*
- *some optimisations (e.g. merging of accesses) cannot be expressed in the C11/C++11 formalisation*
- *the tool also ensures that the compilation of atomic accesses is preserved by the optimiser*

ir

n

TRACE

OCaml tool
1. analyse the traces to detect eliminable actions
2. match reference and optimised traces

Interaction with GCC developers

1. Some GCC bugs

Some concurrency compiler bugs found

in the latest version of GCC.

Store introductions performed by loop invariant motion or if-conversion optimisations.

All promptly fixed.

Remark: these bugs break the Posix thread model too.

2. Checking compiler invariants

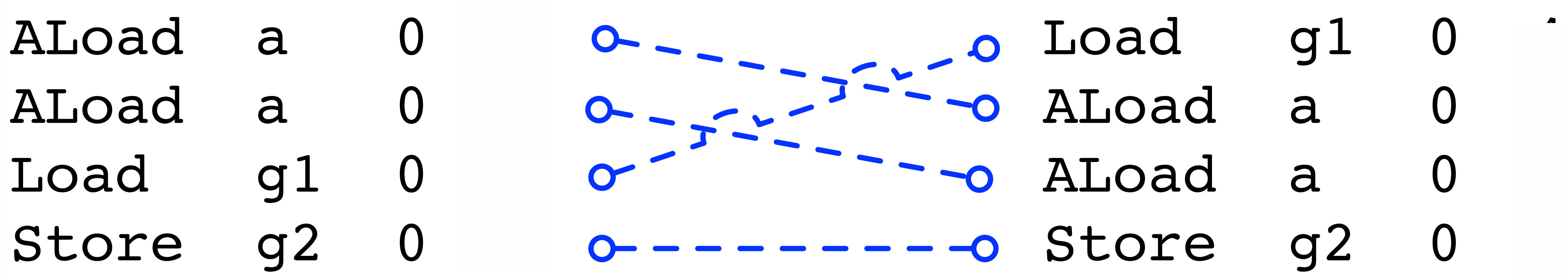
GCC internal invariant: never reorder with an atomic access

Baked this invariant into the tool and found a counterexample...

...not a bug, but fixed anyway

```
atomic_uint a;
int32_t g1, g2;
```

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



```
ALoad   a    0
ALoad   a    0
Load    g1   0
Store   g2   0
```

```
Load    g1   0
ALoad   a    0
ALoad   a    0
Store   g2   0
```

3. Detecting unexpected behaviours

`uint16_t g`

`for (; g==0; g--);` $\longrightarrow$ `g=0;`

If `g` is initialised with `0`, a load gets replaced by a store:

`Load g 0`

`Store g 0`

The introduced store cannot be observed by a non-racy context.

Still, *arguable if a compiler should do this or not.*

Conclusion

Syllabus

In these lectures we have covered the hardware models of two modern computer architectures (x86 and Power/ARM - at least for a large subset of their instruction set).

We have seen how compiler optimisations can also break concurrent programs and the importance of defining the memory model of high-level programming languages.

We have also introduced some proof methods to reason about concurrency.

After these lectures, you might have the feeling that multicore programming is a mess and things can't just work.

The memory models of modern hardware are better understood.

Programming languages attempt to specify and implement reasonable memory models.

Researchers and programmers are now interested in these problems.

The memory models of modern hardware are better understood.

attempt

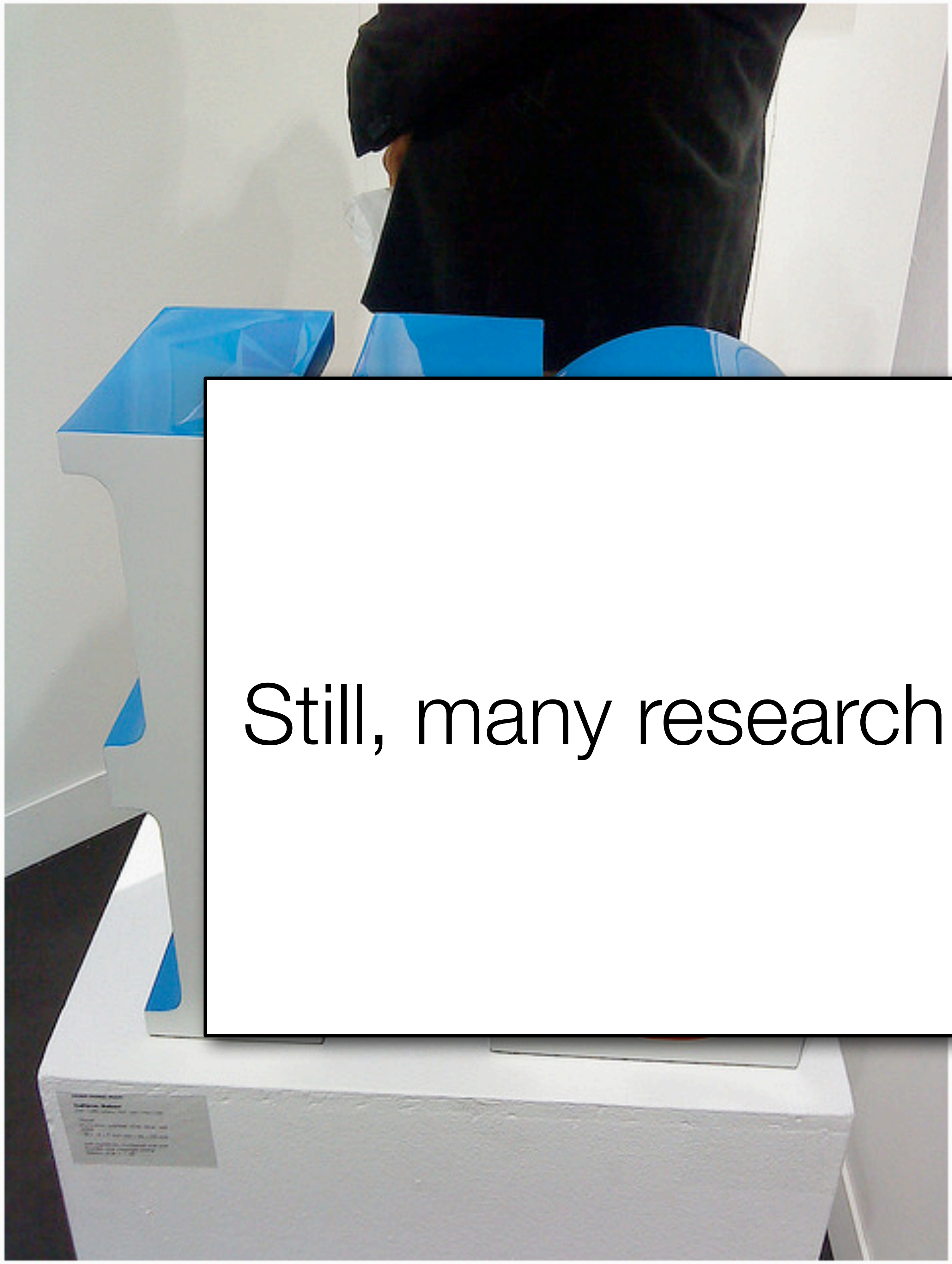
els.

mmers

se

problems.

Still, many open problems...



The memory models of modern hardware are better understood.

attempt

els.

mmers

se

problems.

Still, many research opportunities!

All these lectures are based on work done with/by my colleagues. Thank you!

And thank you all for attending these lectures!

Please, fill the course evaluation form, that's important to make a better course next year.