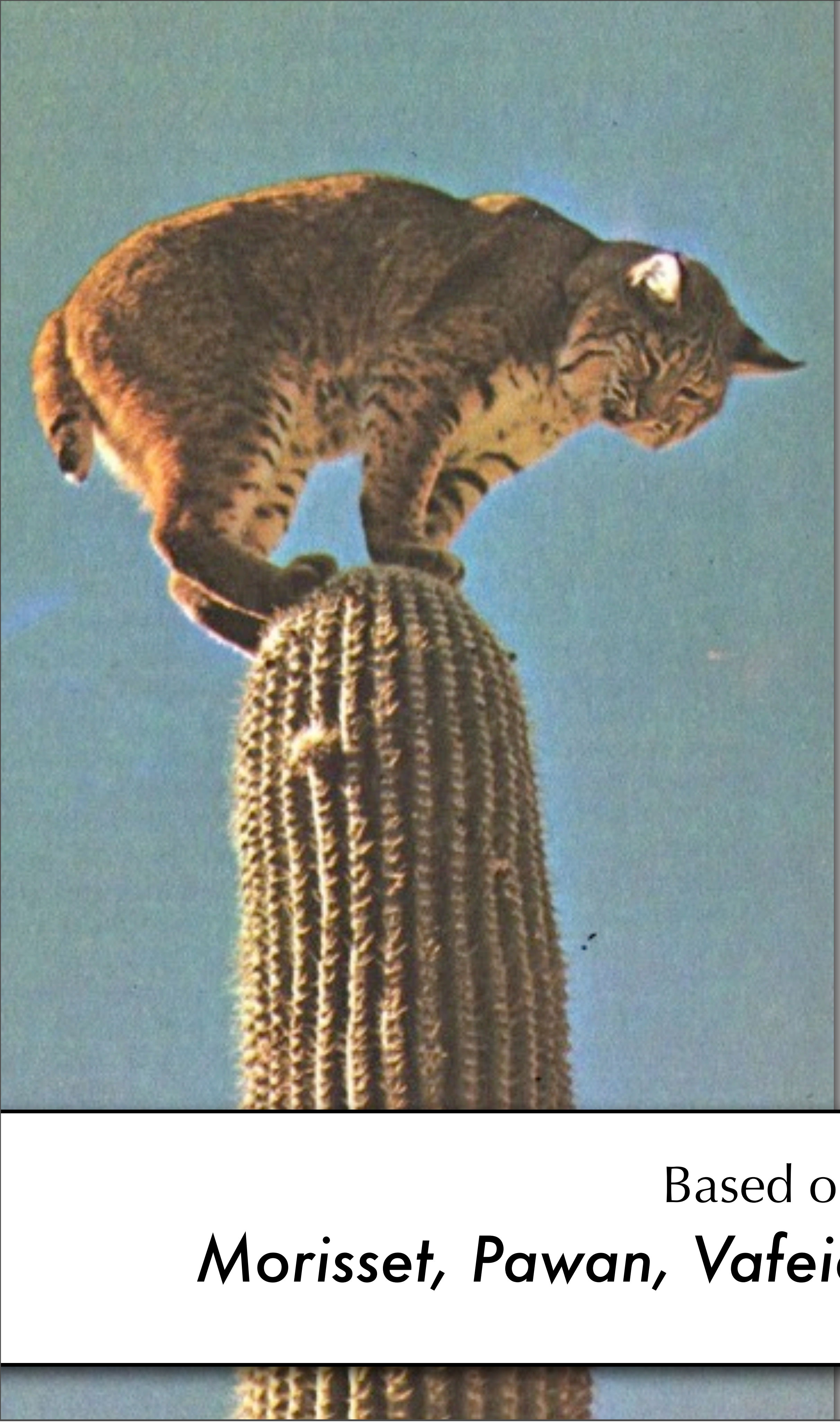


C Concurrency: Still Tricky

Francesco Zappa Nardelli

Inria, France

Based on work done with

Morisset, Pawan, Vafeiadis, Balabonsky, Chakraborty

MPI-SWS and Inria

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`

Thread 2 is not affected by Thread 1 and vice-versa

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`

Thread 2 is not affected by Thread 1 and vice-versa

I expect this program to print `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)
    ;
}
```

```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), %eax    # load a into eax
 movl  b(%rip), %ebx    # load b into ebx
 testl %eax, %eax       # if a==1
 jne   .L2              #   jump to .L2
 movl  $0, b(%rip)
 ret
.L2:
 movl  %ebx, b(%rip)    # store ebx 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), %eax    # load a into eax
 movl   b(%rip), %ebx    # load b into ebx
 testl %eax, %eax        # if a==1
 jne    .L2              #   jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %ebx, b(%rip)    # store ebx 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), %eax    # load a into eax
 movl  b(%rip), %ebx    # load b into ebx
 testl %eax, %eax       # if a==1
 jne   .L2              #   jump to .L2
 movl  $0, b(%rip)
 ret
.L2:
 movl  %ebx, b(%rip)    # store ebx 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), %eax     # load a into eax
 movl   b(%rip), %ebx     # load b into ebx
 testl  %eax, %eax        # if a==1
 jne    .L2               #   jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %ebx, b(%rip)     # store ebx 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), %eax    # load a into eax
 movl   b(%rip), %ebx    # load b into ebx
 testl  %eax, %eax       # if a==1
 jne    .L2              #   jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %ebx, b(%rip)    # store ebx 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), %eax     # load a into eax
movl   b(%rip), %ebx     # load b into ebx
testl  %eax, %eax        # if a==1
jne    .L2               #   jump to .L2
movl   $0, b(%rip)
ret
.L2:
movl   %ebx, b(%rip)     # store ebx 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), %eax    # load a into eax
 movl   b(%rip), %ebx    # load b into ebx
 testl  %eax, %eax       # if a==1
 jne    .L2              #   jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %ebx, b(%rip)    # store ebx into b
 xorl   %eax, %eax       # store 0 into eax
 ret                     # return
```

The compiled code saves and restores **b**

Correct result in a sequential setting

```
 movl   a(%rip), %eax    # load a into eax
 movl   b(%rip), %ebx    # load b into ebx
 testl  %eax, %eax       # if a==1
 jne    .L2              #   jump to .L2
 movl   $0, b(%rip)
 ret
.L2:
 movl   %ebx, b(%rip)    # store ebx into b
 xorl   %eax, %eax       # store 0 into eax
 ret                     # return
```

Shared memory

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

Thread 1

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

Thread 2

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

- Read `a` (1) into `eax`

Shared memory

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

Thread 1

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

Thread 2

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

- Read `a` (`1`) into `eax`
- Read `b` (`0`) into `ebx`

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

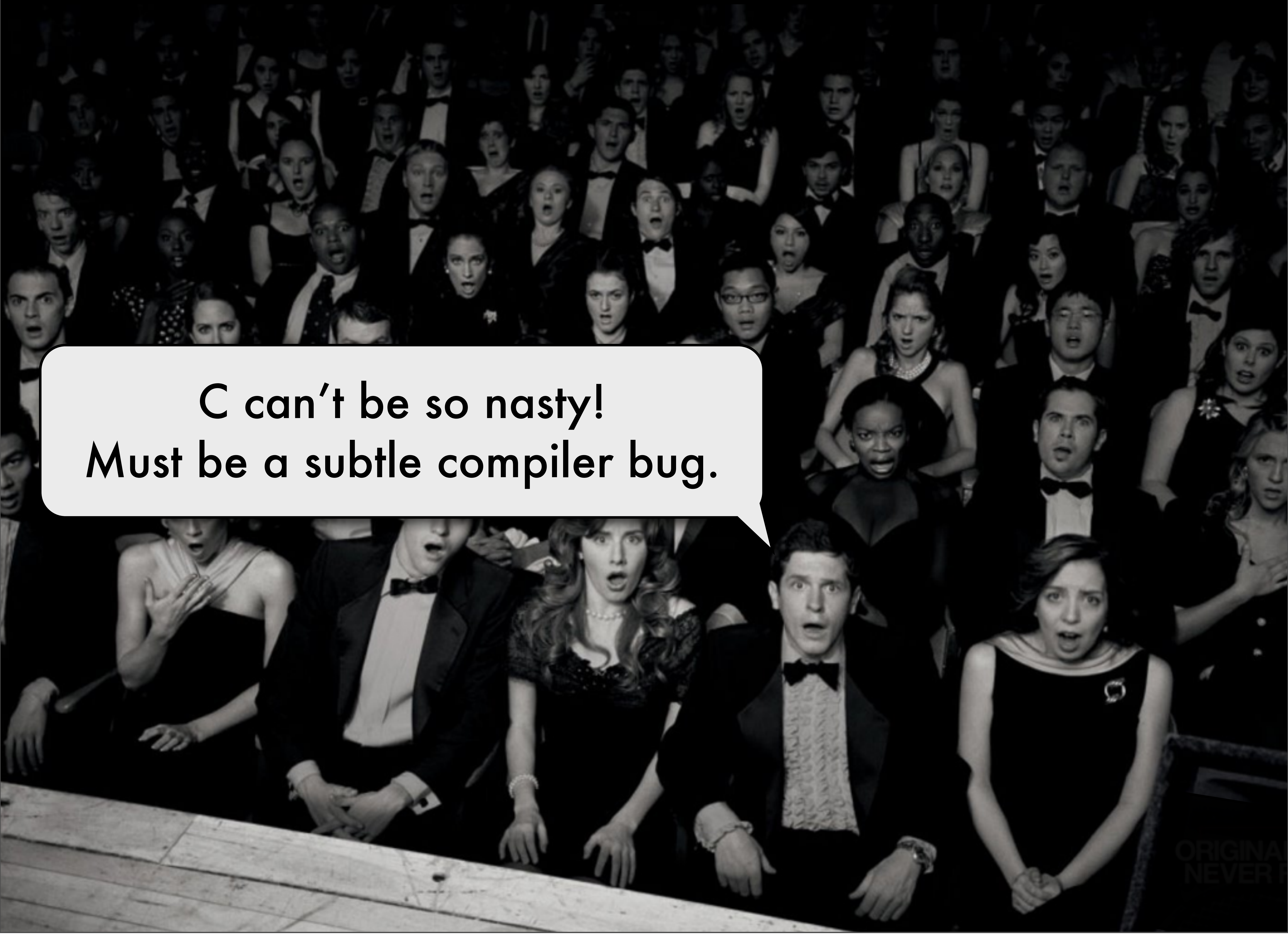
Thread 1

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

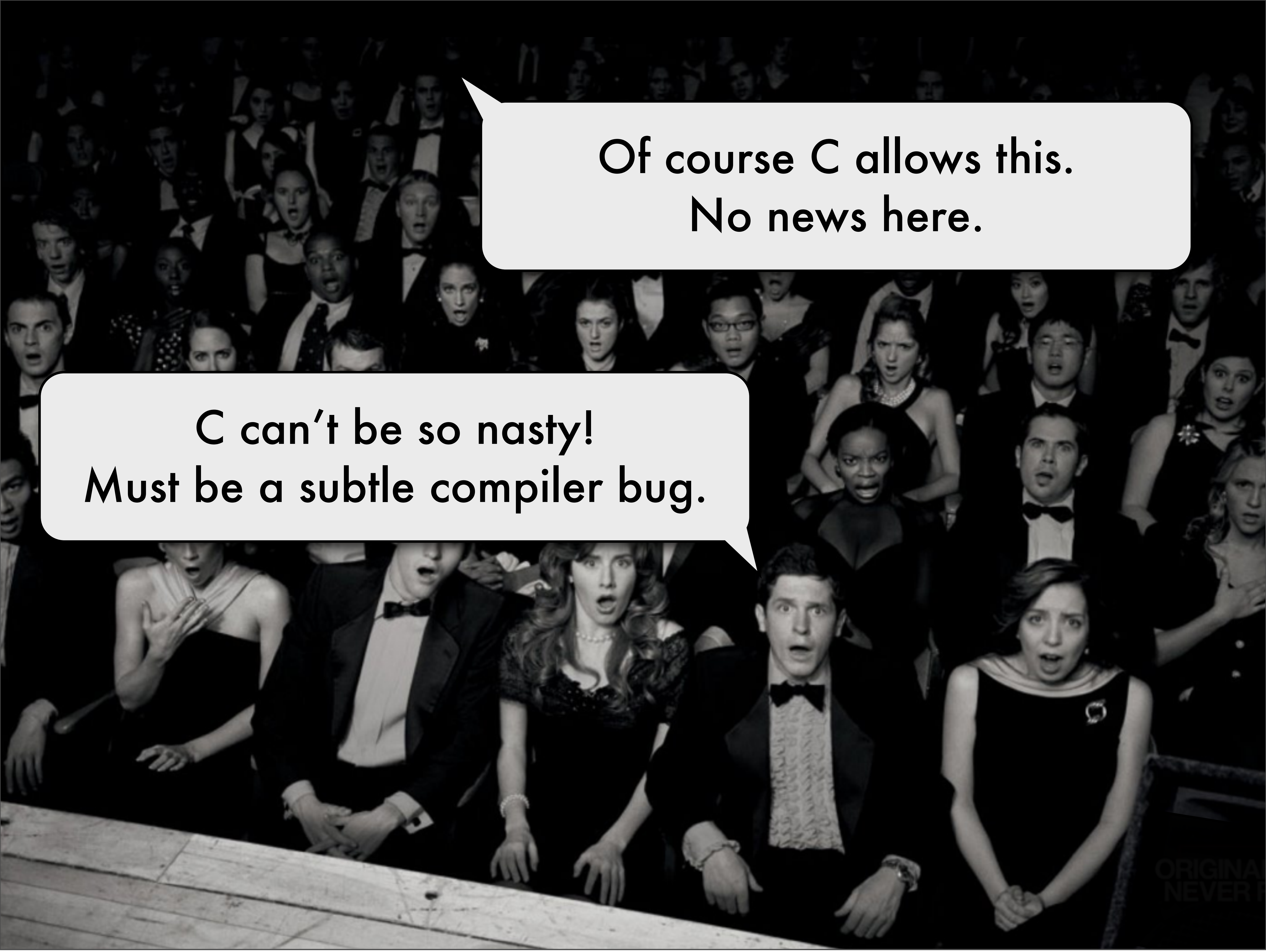
Thread 2

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

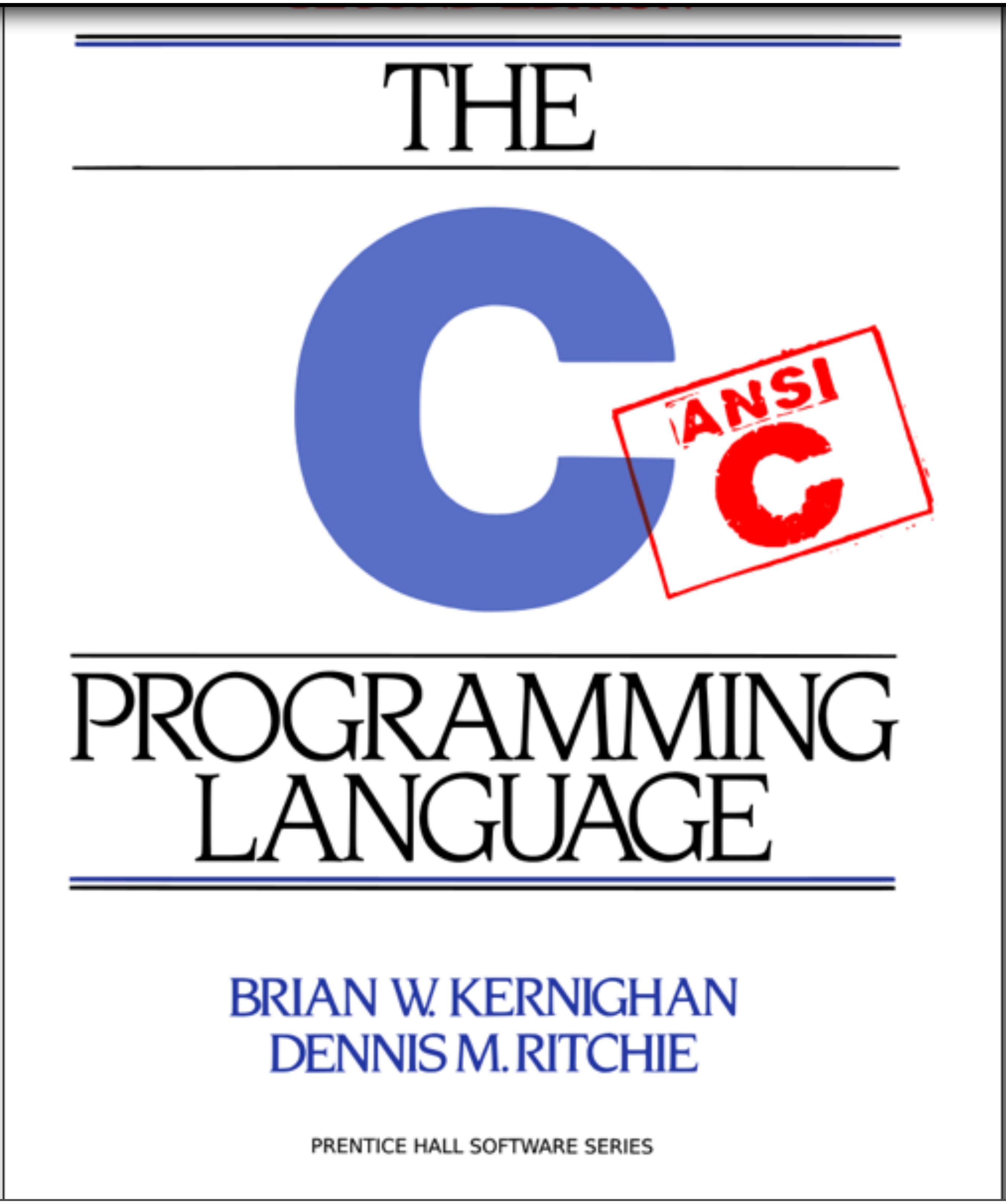
- Read `a` (`1`) into `eax`
- Read `b` (`0`) into `ebx`
- Store `42` into `b`
- Store `ebx` (`0`) into `b`
- Print `b`: `0` is printed



C can’t be so nasty!
Must be a subtle compiler bug.


Of course C allows this.
No news here.
C can’t be so nasty!
Must be a subtle compiler bug.

What is C?



THE

C

ANSI C

PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

K&R

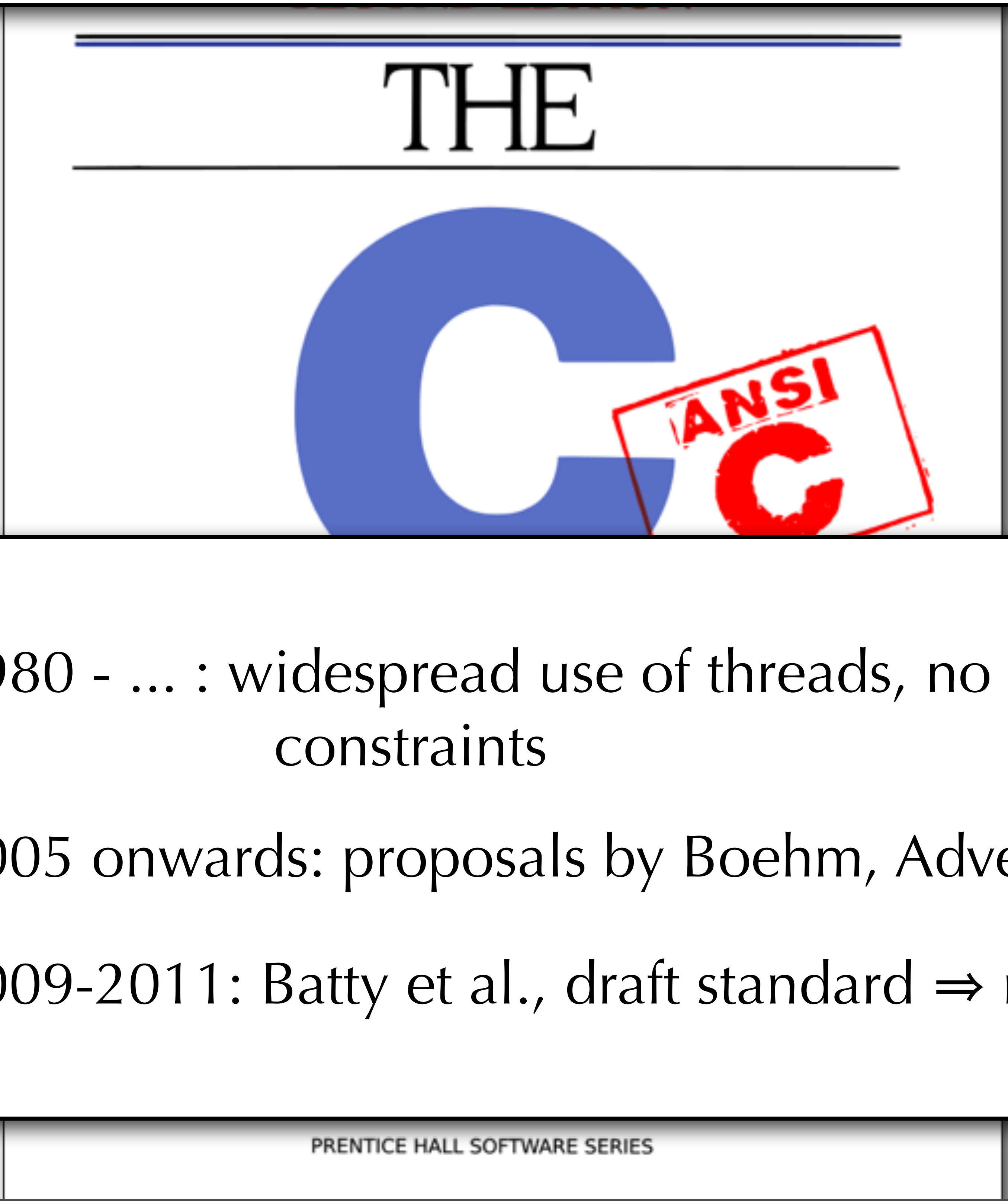
ANSI C

C99

C11

DeFacto C: whatever C compilers implement C programmers rely on

What is C?



K&R

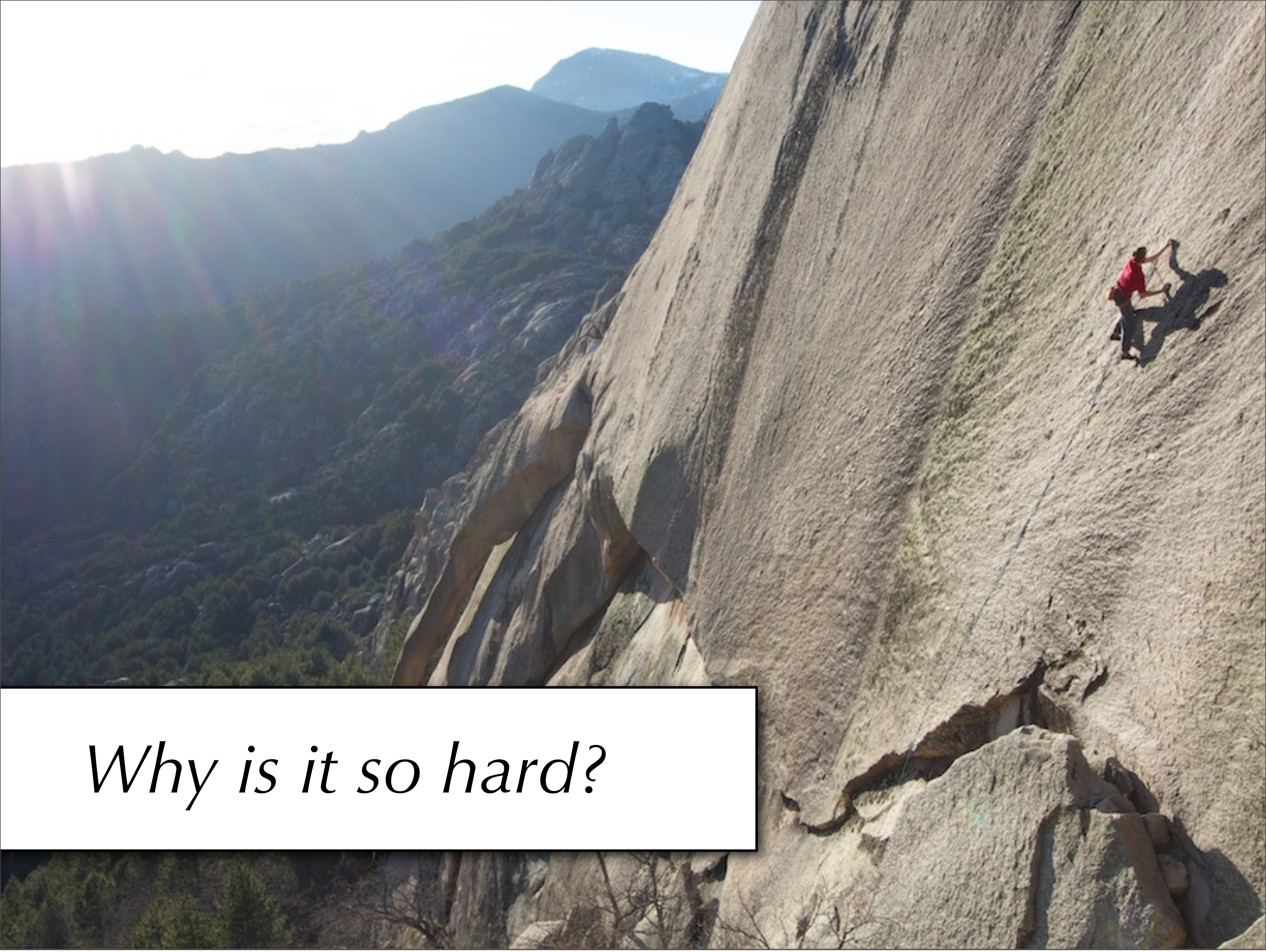
ANSI C

C99

1980 - ... : widespread use of threads, no spec, poor understanding of constraints

2005 onwards: proposals by Boehm, Adve, C++0x concurrency subgroup

2009-2011: Batty et al., draft standard ⇒ math ⇒ fixes ⇒ **C/C++11 standard**



Why is it so hard?

Constant propagation

A simple, and *innocuous*, optimisation:

Source code

```
x = 14
y = 7 - x / 2
```

Optimised code

```
x = 14
y = 7 - 14 / 2
```

```
x = 14
y = 0
```

Shared memory concurrency

Shared memory

```
x = y = 0
```

Thread 1

```
x = 1
if (y == 1)
  print x
```

Thread 2

```
if (x == 1) {
  x = 0
  y = 1 }
```

Shared memory concurrency

Shared memory

```
x = y = 0
```

Thread 1

```
x = 1
if (y == 1)
  print x
```

Thread 2

```
if (x == 1) {
  x = 0
  y = 1 }
```

Intuitively this program always prints 0

Shared memory concurrency

But if the compiler propagates the *constant* x = 1…

```
x = y = 0
```

Thread 1

```
x = 1
if (y == 1)
  print x
```

Shared memory concurrency

But if the compiler propagates the *constant* x = 1...

```
x = y = 0
```

Thread 1

```
x = 1
if (y == 1)
  ~~print x~~
    print 1
```

Thread 2

```
if (x == 1) {
  x = 0
  y = 1 }
```

...the program always writes 1 rather than 0.

This talk

0. Concurrency and optimisations, not so simple

1. The layman semantics
2. Escape lanes for the expert programmer

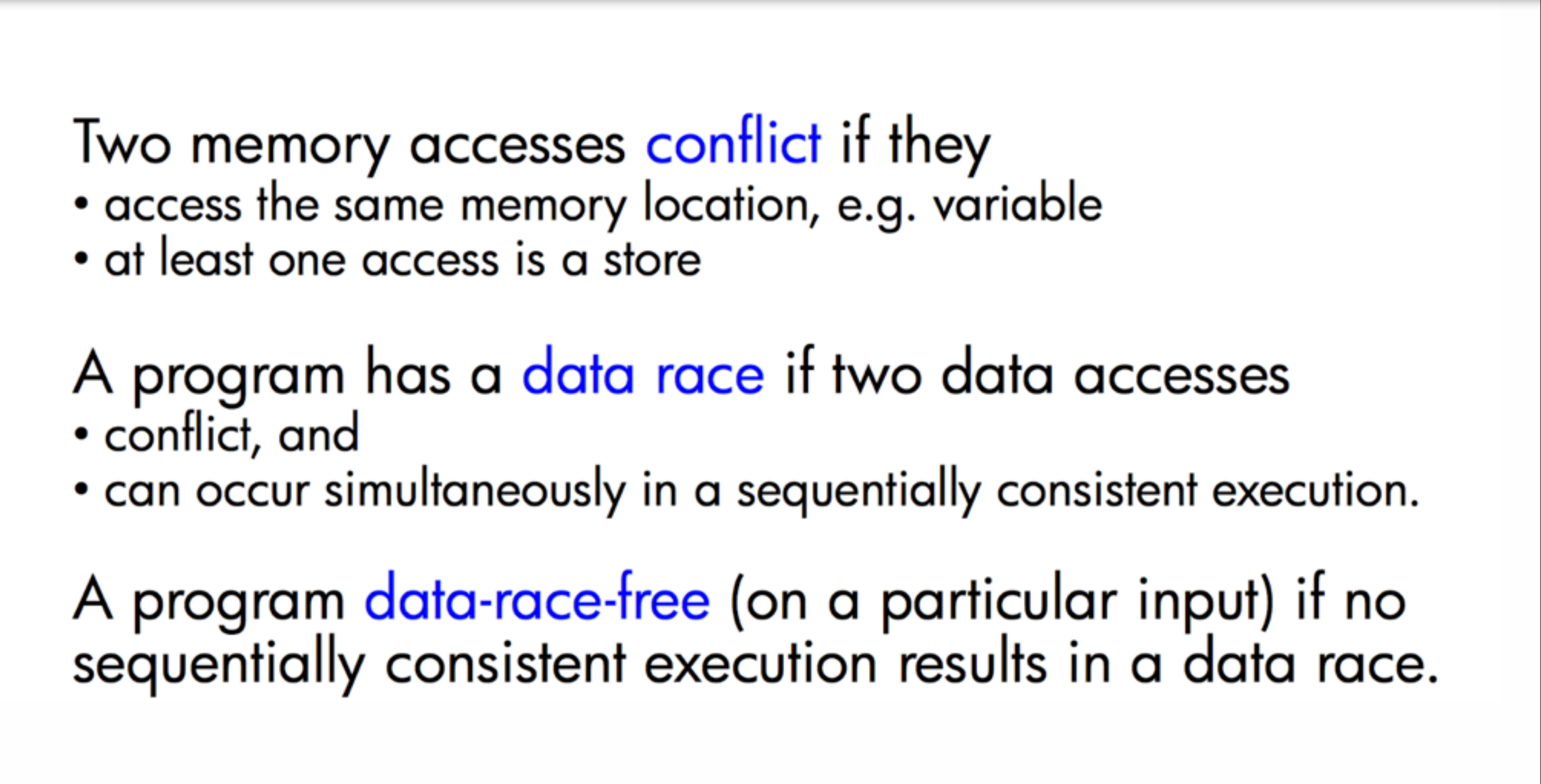
3. Compiler testing via a theory of sound optimisations

4. Escape lanes are a Pandora's box
5. The way forward...

The layman solution

forbid data-races

Standard way out: prohibit data races



Two memory accesses conflict if they

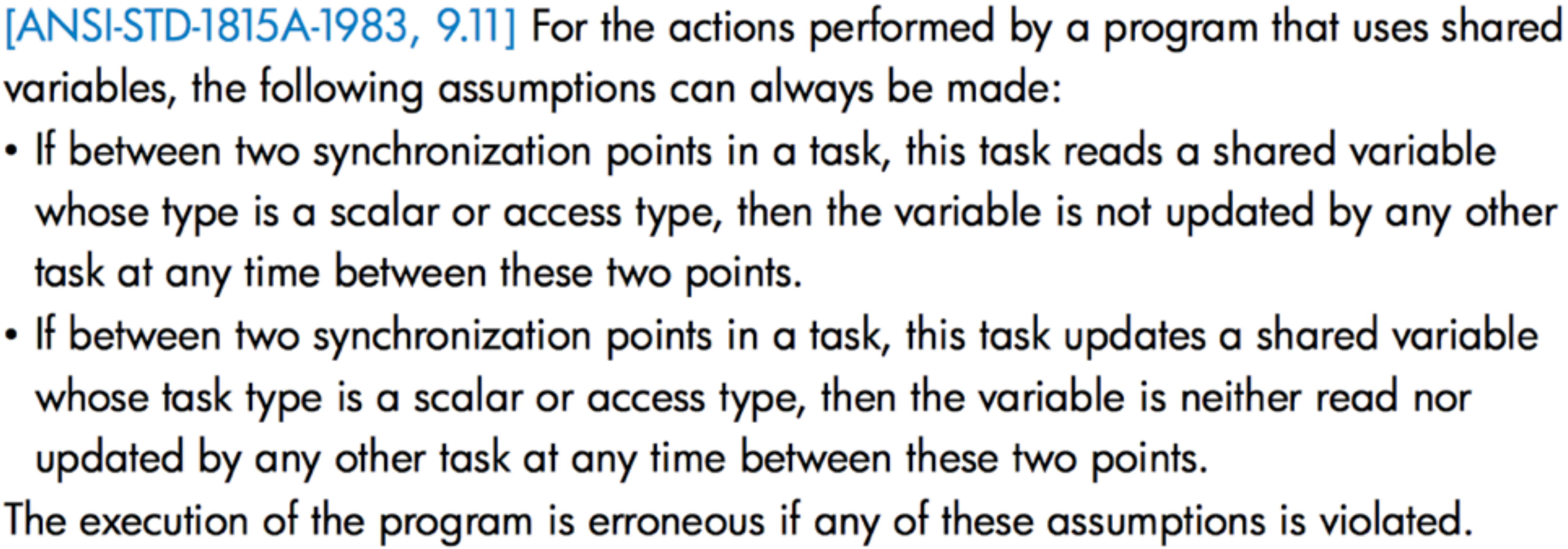
- access the same memory location, e.g. variable
- at least one access is a store

A program has a data race if two data accesses

- conflict, and
- can occur simultaneously in a sequentially consistent execution.

A program data-race-free (on a particular input) if no sequentially consistent execution results in a data race.

ADA 83



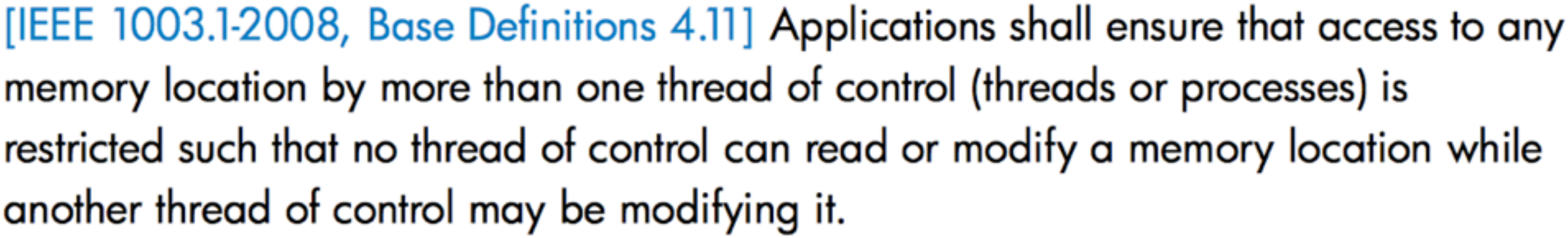
[ANSI-STD-1815A-1983, 9.11] For the actions performed by a program that uses shared variables, the following assumptions can always be made:

- If between two synchronization points in a task, this task reads a shared variable whose type is a scalar or access type, then the variable is not updated by any other task at any time between these two points.
- If between two synchronization points in a task, this task updates a shared variable whose task type is a scalar or access type, then the variable is neither read nor updated by any other task at any time between these two points.

The execution of the program is erroneous if any of these assumptions is violated.

Data-races are errors

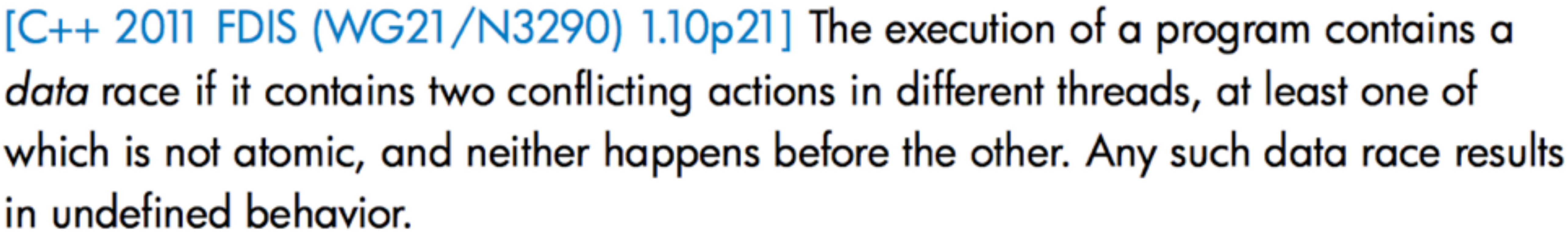
Posix Threads Specification



[IEEE 1003.1-2008, Base Definitions 4.11] Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it.

Data-races are errors

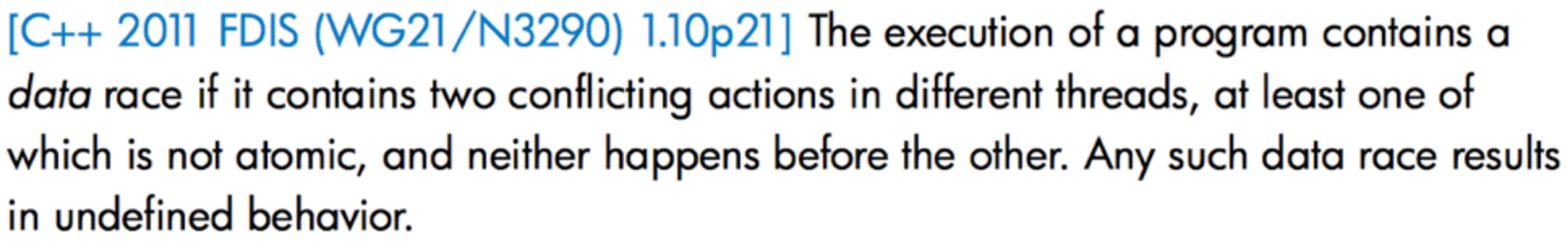
C++2011 / C11



[C++ 2011 FDIS (WG21/N3290) 1.10p21] The execution of a program contains a *data* race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

Data-races are errors

C++2011 / C11



[C++ 2011 FDIS (WG21/N3290) 1.10p21] The execution of a program contains a *data* race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

How to use C/C++ to implement low-level system code?

Data-races are errors

Escape lanes for expert programmers

Low-level atomics in C11/C++11

```
std::atomic<int> flag0(0),flag1(0),turn(0);

void lock(unsigned index) {
    if (0 == index) {
        flag0.store(1, std::memory_order_relaxed);
        turn.exchange(1, std::memory_order_acq_rel);

        while (flag1.load(std::memory_order_acquire)
            && 1 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    } else {
        flag1.store(1, std::memory_order_relaxed);
        turn.exchange(0, std::memory_order_acq_rel);

        while (flag0.load(std::memory_order_acquire)
            && 0 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    }
}

void unlock(unsigned index) {
    if (0 == index) {
        flag0.store(0, std::memory_order_release);
    } else {
        flag1.store(0, std::memory_order_release);
    }
}
```

Atomic variable declaration

New syntax
for memory accesses

Qualifier

The qualifiers

LESS RELAXED

`MO_SEQ_CST`

`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

`MO_RELAXED`

MORE RELAXED

The qualifiers

LESS RELAXED

Sequential consistent accesses

`MO_SEQ_CST`

`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

`MO_RELAXED`

MORE RELAXED

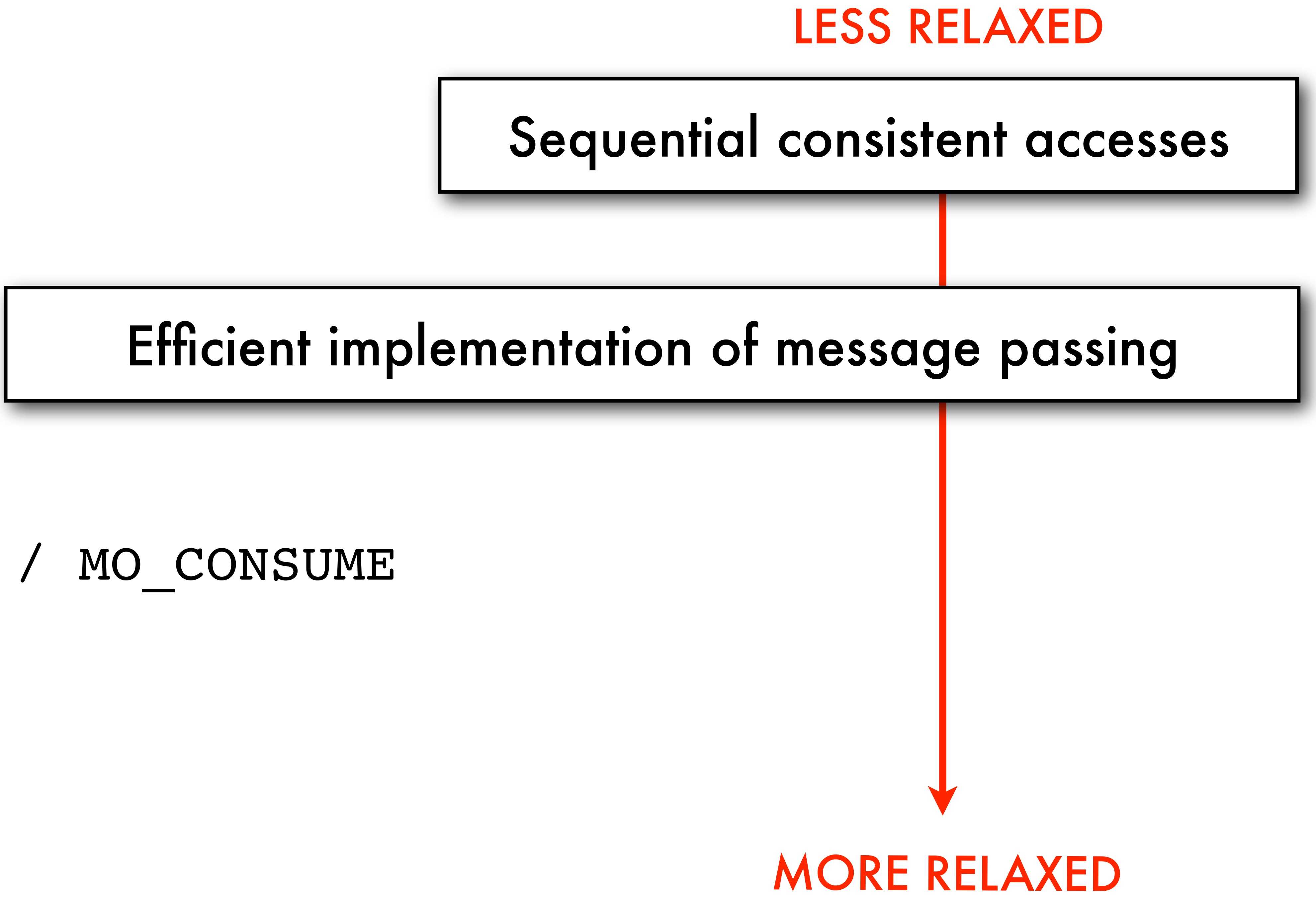
The qualifiers

`MO_SEQ_CST`

`MO_RELEASE`

`MO_RELEASE / MO_CONSUME`

`MO_RELAXED`



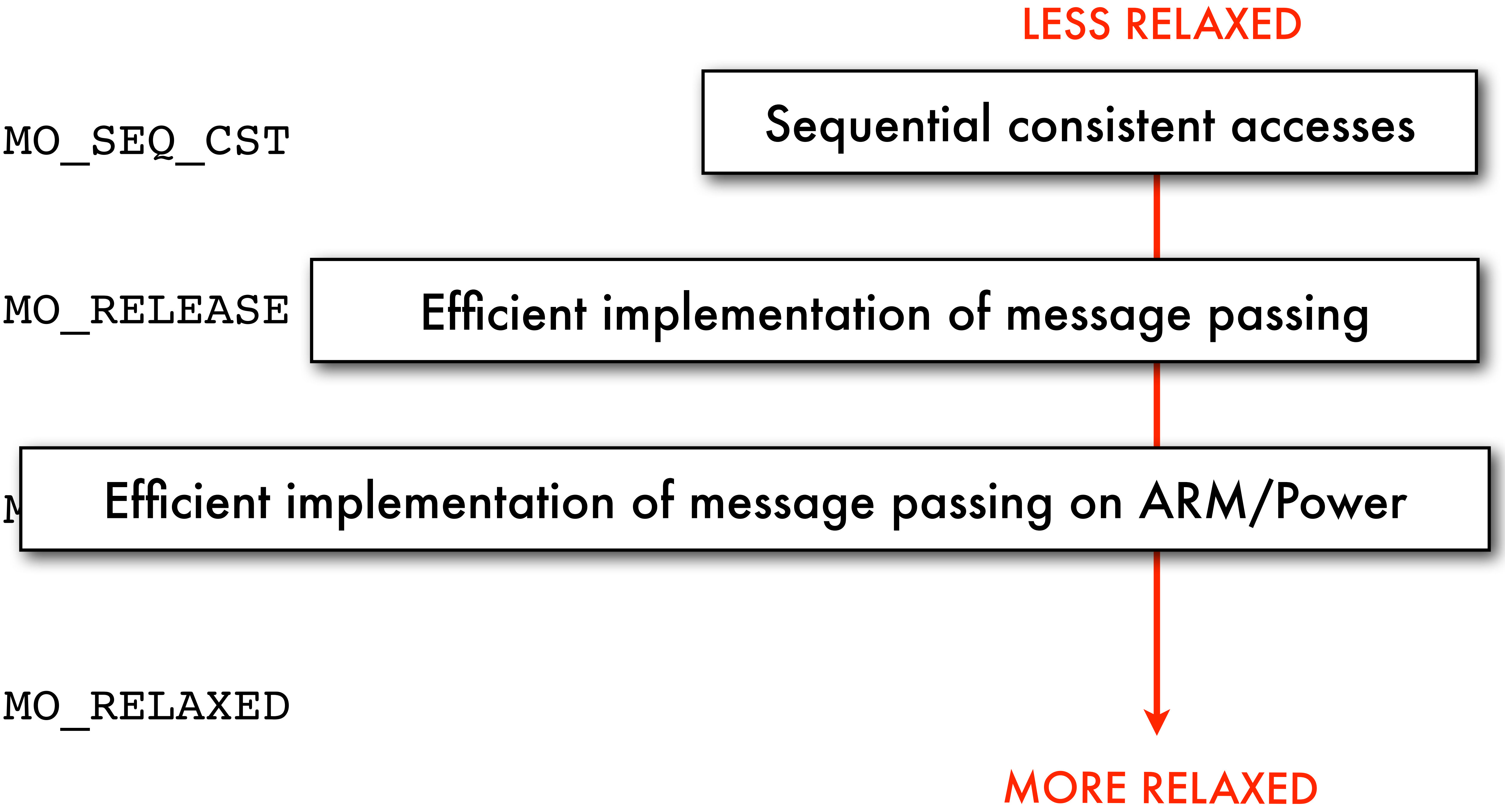
LESS RELAXED

Sequential consistent accesses

Efficient implementation of message passing

MORE RELAXED

The qualifiers



LESS RELAXED

MO_SEQ_CST

Sequential consistent accesses

MO_RELEASE

Efficient implementation of message passing

M

Efficient implementation of message passing on ARM/Power

MO_RELAXED

MORE RELAXED

The qualifiers

LESS RELAXED

`MO_SEQ_CST`

Sequential consistent accesses

`MO_RELEASE`

Efficient implementation of message passing

`M`

Efficient implementation of message passing on ARM/Power

`MO_RELAX`

No synchronisation; direct access to hardware

MORE RELAXED

Memory access synchronisation

```
x = y = 0
```

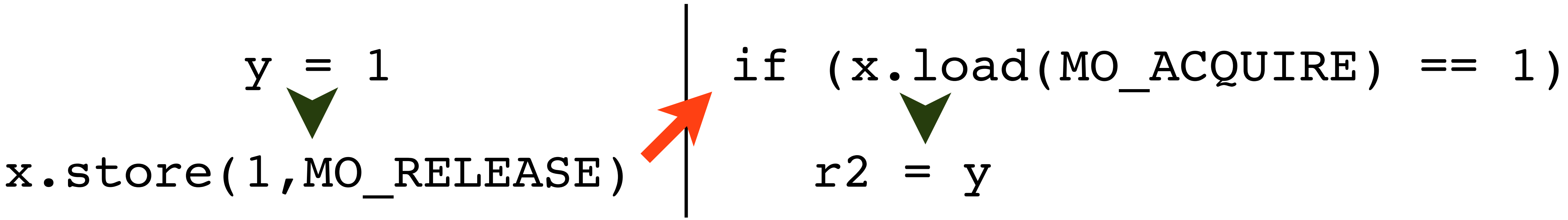
Thread 1

```
y = 1

x.store(1,MO_RELEASE)
```

Thread 2

```
if (x.load(MO_ACQUIRE) == 1)

   r2 = y
```

Memory access synchronisation



```
x = y = 0
```

Thread 1

```
y = 1
x.store(1,MO_RELEASE)
```

Thread 2

```
if (x.load(MO_ACQUIRE) == 1)
    r2 = y
```

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

Non-atomic loads must return the *most recent write* in the happens-before order (unique in a DRF program)

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1

```
y = 1
x.store(1,MO_RELAXED)
```

Thread 2

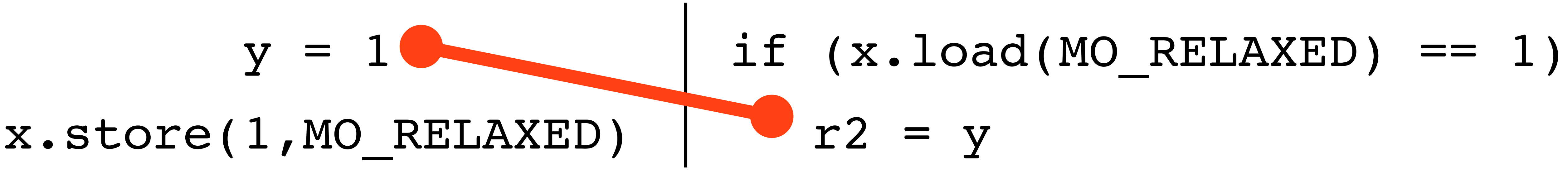
```
if (x.load(MO_RELAXED) == 1)
   r2 = y
```

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1

Thread 2



```
           y = 1             | if (x.load(MO_RELAXED) == 1)
x.store(1,MO_RELAXED)        |    r2 = y
```

DATA RACE

Two conflicting accesses not related by happens-before

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1

```
y.store(1,MO_RELAXED)
x.store(1,MO_RELAXED)
```

Thread 2

```
if (x.load(MO_RELAXED) == 1)
    r2 = y.load(MO_RELAXED)
```

WELL DEFINED

but r2 = 0 is possible

Intuition
the compiler (or hardware) can reorder independent accesses

```
x = y = 0
```

Thread 1

```
y.store(1,MO_RELAXED)
x.store(1,MO_RELAXED)
```

Thread 2

```
if (x.load(MO_RELAXED) == 1)
    r2 = y.load(MO_RELAXED)
```

WELL DEFINED

but r2 = 0 is possible

Intuition

the compiler (or hardware) can reorder independent accesses

```
x = y = 0
```

Thread 1	Thread 2
`y.store(1,MO_RELAXED)`	`if (x.load(MO_RELAXED) == 1)`
`x.store(1,MO_RELAXED)`	`r2 = y.load(MO_RELAXED)`

Allow a RELAXED load to see any store that:

- does not happens-after it
- is not hidden by an intervening store hb-ordered between them

The full model

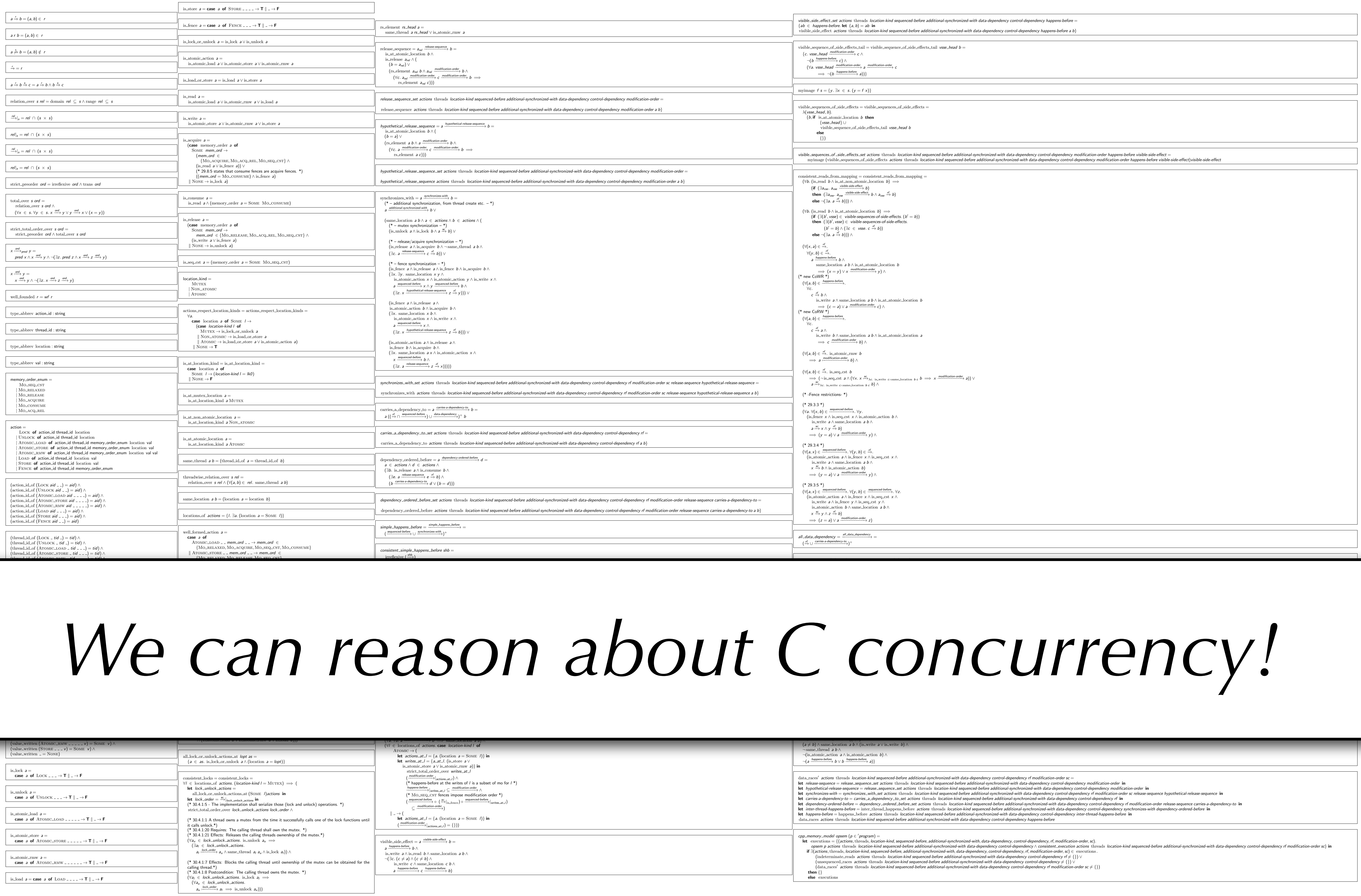
```
$a \xrightarrow{r} b = (a, b) \in r$

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

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

$\xrightarrow{r} = r$

$a \xrightarrow{r} b \xrightarrow{s} c = a \xrightarrow{r} b \land b \xrightarrow{s} c$

relation_over $s$ $rel$ = domain $rel \subseteq s \land$ 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 $ord$ $\land$ trans $ord$

total_over $s$ $ord$ =
  relation_over $s$ $ord$ $\land$
  $(\forall x \in s. \forall y \in s. x \xrightarrow{ord} y \lor y \xrightarrow{ord} x \lor (x = y))$

strict_total_order_over $s$ $ord$ =
  strict_preorder $ord$ $\land$ total_over $s$ $ord$

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

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

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) ∧
(action_id_of (UNLOCK aid _ _) = aid) ∧
(action_id_of (ATOMIC_LOAD aid _ _ _ _) = aid) ∧
(action_id_of (ATOMIC_STORE aid _ _ _ _) = aid) ∧
(action_id_of (ATOMIC_RMW aid _ _ _ _ _) = aid) ∧
(action_id_of (LOAD aid _ _ _) = aid) ∧
(action_id_of (STORE aid _ _ _) = aid) ∧
(action_id_of (FENCE aid _ _) = aid)

(thread_id_of (LOCK _ tid _) = tid) ∧
(thread_id_of (UNLOCK _ tid _) = tid) ∧
(thread_id_of (ATOMIC_LOAD _ tid _ _ _) = tid) ∧
(thread_id_of (ATOMIC_STORE _ tid _ _ _) = tid) ∧
(thread_id_of (ATOMIC_RMW _ tid _ _ _ _) = tid) ∧
(thread_id_of (LOAD _ tid _ _) = tid) ∧
(thread_id_of (STORE _ tid _ _) = tid) ∧
(thread_id_of (FENCE _ tid _) = tid)

(memory_order (ATOMIC_LOAD _ _ mem_ord _ _) =
  SOME mem_ord) ∧
(memory_order (ATOMIC_STORE _ _ mem_ord _ _) =
  SOME mem_ord) ∧
(memory_order (ATOMIC_RMW _ _ mem_ord _ _ _) =
  SOME mem_ord) ∧
(memory_order (FENCE _ _ mem_ord) =
  SOME mem_ord) ∧
(memory_order _ =
  NONE)

(location (LOCK _ _ l) = SOME l) ∧
(location (UNLOCK _ _ l) = SOME l) ∧
(location (ATOMIC_LOAD _ _ _ l _) = SOME l) ∧
(location (ATOMIC_STORE _ _ _ l _) = SOME l) ∧
(location (ATOMIC_RMW _ _ _ l _ _) = SOME l) ∧
(location (LOAD _ _ l _) = SOME l) ∧
(location (STORE _ _ l _) = SOME l) ∧
(location (FENCE _ _ _) = NONE)

(value_read (ATOMIC_LOAD _ _ _ _ v) = SOME v) ∧
(value_read (ATOMIC_RMW _ _ _ _ v _) = SOME v) ∧
(value_read (LOAD _ _ _ v) = SOME v) ∧
(value_read _ = NONE)

(value_written (ATOMIC_STORE _ _ _ _ v) = SOME v) ∧
(value_written (ATOMIC_RMW _ _ _ _ _ v) = SOME v) ∧
(value_written (STORE _ _ _ v) = SOME v) ∧
(value_written _ = NONE)

is_lock a =
  case a of LOCK _ _ _ → T ‖ _ → F

is_unlock a =
  case a of UNLOCK _ _ _ → T ‖ _ → F

is_atomic_load a =
  case a of ATOMIC_LOAD _ _ _ _ _ → T ‖ _ → F

is_atomic_store a =
  case a of ATOMIC_STORE _ _ _ _ _ → T ‖ _ → F

is_atomic_rmw a =
  case a of ATOMIC_RMW _ _ _ _ _ _ → T ‖ _ → F

is_load a = case a of LOAD _ _ _ _ → T ‖ _ → F
```

```
is_store a = case a of STORE _ _ _ _ → T ‖ _ → F

is_fence a = case a of FENCE _ _ _ → T ‖ _ → F

is_lock_or_unlock a = is_lock a ∨ is_unlock a

is_atomic_action a =
  is_atomic_load a ∨ is_atomic_store a ∨ is_atomic_rmw a

is_load_or_store a = is_load a ∨ is_store a

is_read a =
  is_atomic_load a ∨ is_atomic_rmw a ∨ is_load a

is_write a =
  is_atomic_store a ∨ is_atomic_rmw a ∨ is_store a

is_acquire a =
  (case memory_order a of
    SOME mem_ord →
      (mem_ord ∈
        {MO_ACQUIRE, MO_ACQ_REL, MO_SEQ_CST} ∧
      (is_read a ∨ is_fence a)) ∨
      (* 29.8:5 states that consume fences are acquire fences. *)
      ((mem_ord = MO_CONSUME) ∧ is_fence a)
  ‖ NONE → is_lock a)

is_consume a =
  is_read a ∧ (memory_order a = SOME MO_CONSUME)

is_release a =
  (case memory_order a of
    SOME mem_ord →
      mem_ord ∈ {MO_RELEASE, MO_ACQ_REL, MO_SEQ_CST} ∧
    (is_write a ∨ is_fence a)
  ‖ NONE → 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 =
  ∀a.
    case location a of SOME l →
      (case location-kind l of
        MUTEX → is_lock_or_unlock a
      ‖ NON_ATOMIC → is_load_or_store a
      ‖ ATOMIC → is_load_or_store a ∨ is_atomic_action a)
    ‖ NONE → T

is_at_location_kind = is_at_location_kind =
  case location a of
    SOME l → (location-kind l = lk0)
  ‖ NONE → 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 ∧ (∀(a, b) ∈ rel. same_thread a b)

same_location a b = (location a = location b)

locations_of actions = {l. ∃a. (location a = SOME l)}

well_formed_action a =
  case a of
    ATOMIC_LOAD _ _ mem_ord _ _ → mem_ord ∈
      {MO_RELAXED, MO_ACQUIRE, MO_SEQ_CST, MO_CONSUME}
  ‖ ATOMIC_STORE _ _ mem_ord _ _ → mem_ord ∈
      {MO_RELAXED, MO_RELEASE, MO_SEQ_CST}
  ‖ ATOMIC_RMW _ _ mem_ord _ _ _ → mem_ord ∈
      {MO_RELAXED, MO_RELEASE, MO_ACQUIRE, MO_ACQ_REL, MO_SEQ_CST, MO_CONSUME}
  ‖ _ → T

well_formed_threads = well_formed_threads =
  inj_on action_id_of (actions) ∧
  (∀a. well_formed_action a) ∧
  threadwise_relation_over actions sequenced-before ∧
  threadwise_relation_over actions data-dependency ∧
  threadwise_relation_over actions control-dependency ∧
  strict_preorder sequenced-before ∧
  strict_preorder data-dependency ∧
  strict_preorder control-dependency ∧
  relation_over actions additional-synchronized-with ∧
  (∀a. thread_id_of a ∈ threads) ∧
  actions_respect_location_kinds ∧
  data-dependency ⊆ sequenced-before

well_formed_reads_from_mapping = well_formed_reads_from_mapping =
  relation_over actions ($\xrightarrow{rf}$) ∧
  (∀a. ∀a'. ∀b. a $\xrightarrow{rf}$ b ∧ a' $\xrightarrow{rf}$ b ⟹ (a = a')) ∧
  (∀(a, b) ∈ $\xrightarrow{rf}$.
    same_location a b ∧
    (value_read b = value_written a) ∧
    (a ≠ b) ∧
    (is_at_mutex_location a ⟹
      is_unlock a ∧ is_lock b) ∧
    (is_at_non_atomic_location a ⟹
      is_store a ∧ is_load b) ∧
    (is_at_atomic_location a ⟹
      (is_atomic_store a ∨ is_atomic_rmw a ∨ is_store a)
      ∧ (is_atomic_load b ∨ is_atomic_rmw b ∨ is_load b)))

all_lock_or_unlock_actions_at lopt as =
  {a ∈ as. is_lock_or_unlock a ∧ (location a = lopt)}

consistent_locks = consistent_locks =
∀l ∈ locations_of actions. (location-kind l = MUTEX) ⟹ (
  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 ∧

  (* 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.*)
  (∀$a_u$ ∈ lock_unlock_actions. is_unlock $a_u$ ⟹
    (∃$a_l$ ∈ lock_unlock_actions.
      $a_l \xmapsto{lock\_order} a_u$ ∧ same_thread $a_l$ $a_u$ ∧ is_lock $a_l$)) ∧

  (* 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. *)
  (∀$a_l$ ∈ lock_unlock_actions. is_lock $a_l$ ⟹
    (∀$a_u$ ∈ lock_unlock_actions.
      $a_u \xmapsto{lock\_order} a_l$ ⟹ is_unlock $a_u$)))
```

```
rs_element rs_head a =
  same_thread a rs_head ∨ is_atomic_rmw a

release_sequence = $a_{rel} \xrightarrow{release\text{-}sequence} b$ =
  is_at_atomic_location b ∧
  is_release $a_{rel}$ ∧ (
    (b = $a_{rel}$) ∨
    (rs_element $a_{rel}$ b ∧ $a_{rel} \xrightarrow{modification\text{-}order} b$ ∧
      (∀c. $a_{rel} \xrightarrow{modification\text{-}order} c \xrightarrow{modification\text{-}order} b$ ⟹
        rs_element $a_{rel}$ c)))

release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order =

release_sequence actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order a b}

hypothetical_release_sequence = $a \xrightarrow{hypothetical\text{-}release\text{-}sequence} b$ =
  is_at_atomic_location b ∧ (
  (b = a) ∨
  (rs_element a b ∧ $a \xrightarrow{modification\text{-}order} b$ ∧
    (∀c. $a \xrightarrow{modification\text{-}order} c \xrightarrow{modification\text{-}order} b$ ⟹
      rs_element a c)))

hypothetical_release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order =

hypothetical_release_sequence actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order a b}

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

  (same_location a b ∧ a ∈ actions ∧ b ∈ actions ∧ (
    (* – mutex synchronization – *)
    (is_unlock a ∧ is_lock b ∧ $a \xrightarrow{sc} b$) ∨

    (* – release/acquire synchronization – *)
    (is_release a ∧ is_acquire b ∧ ¬same_thread a b ∧
    (∃c. $a \xrightarrow{release\text{-}sequence} c \xrightarrow{rf} b$)) ∨

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

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

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

synchronizes_with_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc release-sequence hypothetical-release-sequence =

synchronizes_with actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc release-sequence hypothetical-release-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-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf =

carries_a_dependency_to actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf a b}

dependency_ordered_before = $a \xrightarrow{dependency\text{-}ordered\text{-}before} d$ =
  a ∈ actions ∧ d ∈ actions ∧
  (∃b. is_release a ∧ is_consume b ∧
    (∃e. $a \xrightarrow{release\text{-}sequence} e \xrightarrow{rf} b$) ∧
    ($b \xrightarrow{carries\text{-}a\text{-}dependency\text{-}to} d$ ∨ (b = d)))

dependency_ordered_before_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order release-sequence carries-a-dependency-to =

dependency_ordered_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order release-sequence carries-a-dependency-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 = $\xrightarrow{synchronizes\text{-}with}$ ∪
          $\xrightarrow{dependency\text{-}ordered\text{-}before}$ ∪
          ($\xrightarrow{synchronizes\text{-}with} \circ \xrightarrow{sequenced\text{-}before}$) 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 ∨ is_lock a ∨ is_unlock a)}

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

consistent_modification_order = consistent_modification_order =
  (∀a. ∀b. $a \xrightarrow{modification\text{-}order} b$ ⟹ same_location a b) ∧
  (∀l ∈ locations_of actions. case location-kind l of
      ATOMIC → (
        let actions_at_l = {a. (location a = SOME l)} in
        let writes_at_l = {a_at_l. (is_store a ∨
          is_atomic_store a ∨ is_atomic_rmw a)} in
            strict_total_order_over writes_at_l
            ($\xrightarrow{modification\text{-}order}|_{actions\_at\_l}$) ∧
            (* happens-before at the writes of l is a subset of mo for l *)
            $\xrightarrow{happens\text{-}before}|_{writes\_at\_l} \subseteq \xrightarrow{modification\text{-}order}$ ∧
            (* MO_SEQ_CST fences impose modification order *)
            $(\xrightarrow{sequenced\text{-}before} \circ (\xrightarrow{sc}|_{is\_fence}) \circ \xrightarrow{sequenced\text{-}before}|_{writes\_at\_l})$
              $\subseteq \xrightarrow{modification\text{-}order}$)
    ‖ _ → (
        let actions_at_l = {a. (location a = SOME l)} in
        ($\xrightarrow{modification\text{-}order}|_{actions\_at\_l}$) = {}))

visible_side_effect = $a \xrightarrow{visible\text{-}side\text{-}effect} b$ =
  $a \xrightarrow{happens\text{-}before} b$ ∧
  is_write a ∧ is_read b ∧ same_location a b ∧
  ¬(∃c. (c ≠ a) ∧ (c ≠ b) ∧
    is_write c ∧ same_location c b ∧
    $a \xrightarrow{happens\text{-}before} c \xrightarrow{happens\text{-}before} b$)
```

```
visible_side_effect_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency happens-before =
{ab ∈ happens-before. let (a, b) = ab in
visible_side_effect actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency happens-before a b}

visible_sequence_of_side_effects_tail = visible_sequence_of_side_effects_tail vsse_head b =
  {c. $vsse\_head \xrightarrow{modification\text{-}order} c$ ∧
    ¬($b \xrightarrow{happens\text{-}before} c$) ∧
    (∀a. $vsse\_head \xrightarrow{modification\text{-}order} a \xrightarrow{modification\text{-}order} c$
        ⟹ ¬($b \xrightarrow{happens\text{-}before} a$))}

myimage f s = {y. ∃x ∈ s. (y = f x)}

visible_sequences_of_side_effects = visible_sequences_of_side_effects =
  λ(vsse_head, b).
    (b, if is_at_atomic_location b then
          {vsse_head} ∪
          visible_sequence_of_side_effects_tail vsse_head b
        else
          {})

visible_sequences_of_side_effects_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect =
    myimage (visible_sequences_of_side_effects actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect)visible-side-effect

consistent_reads_from_mapping = consistent_reads_from_mapping =
  (∀b. (is_read b ∧ is_at_non_atomic_location b) ⟹
      (if (∃$a_{vse}$. $a_{vse} \xrightarrow{visible\text{-}side\text{-}effect} b$)
       then (∃$a_{vse}$. $a_{vse} \xrightarrow{visible\text{-}side\text{-}effect} b$ ∧ $a_{vse} \xrightarrow{rf} b$)
       else ¬(∃a. $a \xrightarrow{rf} b$))) ∧

  (∀b. (is_read b ∧ is_at_atomic_location b) ⟹
      (if (∃(b', vsse) ∈ visible-sequences-of-side-effects. (b' = b))
       then (∃(b', vsse) ∈ visible-sequences-of-side-effects.
              (b' = b) ∧ (∃c ∈ vsse. $c \xrightarrow{rf} b$))
       else ¬(∃a. $a \xrightarrow{rf} b$))) ∧

  (∀(x, a) ∈ $\xrightarrow{rf}$.
    ∀(y, b) ∈ $\xrightarrow{rf}$.
      $a \xrightarrow{happens\text{-}before} b$ ∧
        same_location a b ∧ is_at_atomic_location b
        ⟹ (x = y) ∨ $x \xrightarrow{modification\text{-}order} y$) ∧
(* new CoWR *)
  (∀(a, b) ∈ $\xrightarrow{happens\text{-}before}$.
    ∀c.
      $c \xrightarrow{rf} b$ ∧
        is_write a ∧ same_location a b ∧ is_at_atomic_location b
        ⟹ (c = a) ∨ $a \xrightarrow{modification\text{-}order} c$) ∧
(* new CoRW *)
  (∀(a, b) ∈ $\xrightarrow{happens\text{-}before}$.
    ∀c.
      $c \xrightarrow{rf} a$ ∧
        is_write b ∧ same_location a b ∧ is_at_atomic_location a
        ⟹ $c \xrightarrow{modification\text{-}order} b$) ∧

  (∀(a, b) ∈ $\xrightarrow{rf}$. is_atomic_rmw b
    ⟹ $a \xmapsto{modification\text{-}order} b$) ∧

  (∀(a, b) ∈ $\xrightarrow{rf}$. is_seq_cst b
    ⟹ (¬is_seq_cst a ∧ (∀x. $x \xmapsto{sc}_{\lambda c.\ is\_write\ c \land same\_location\ b\ c}$ b ⟹ $x \xrightarrow{modification\text{-}order} a$)) ∨
    $a \xmapsto{sc}_{\lambda c.\ is\_write\ c \land same\_location\ b\ c}$ b) ∧

  (* -Fence restrictions- *)

  (* 29.3:3 *)
  (∀a. ∀(x, b) ∈ $\xrightarrow{sequenced\text{-}before}$. ∀y.
    (is_fence x ∧ is_seq_cst x ∧ is_atomic_action b ∧
      is_write a ∧ same_location a b ∧
      $a \xmapsto{sc} x$ ∧ $y \xrightarrow{rf} b$)
    ⟹ (y = a) ∨ $a \xrightarrow{modification\text{-}order} y$) ∧

  (* 29.3:4 *)
  (∀(a, x) ∈ $\xrightarrow{sequenced\text{-}before}$. ∀(y, b) ∈ $\xrightarrow{rf}$.
    (is_atomic_action a ∧ is_fence x ∧ is_seq_cst x ∧
      is_write a ∧ same_location a b ∧
      $x \xrightarrow{sc} b$ ∧ is_atomic_action b)
    ⟹ (y = a) ∨ $a \xrightarrow{modification\text{-}order} y$) ∧

  (* 29.3:5 *)
  (∀(a, x) ∈ $\xrightarrow{sequenced\text{-}before}$. ∀(y, b) ∈ $\xrightarrow{sequenced\text{-}before}$. ∀z.
    (is_atomic_action a ∧ is_fence x ∧ is_seq_cst x ∧
      is_write a ∧ is_fence y ∧ is_seq_cst y ∧
      is_atomic_action b ∧ same_location a b ∧
      $x \xrightarrow{sc} y$ ∧ $z \xrightarrow{rf} b$)
    ⟹ (z = a) ∨ $a \xrightarrow{modification\text{-}order} z$)

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-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc =
 well_formed_threads actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency ∧
 consistent_locks actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency sc ∧ (
let release-sequence = release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order in
let hypothetical-release-sequence = hypothetical_release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order in
let synchronizes-with = synchronizes_with_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc release-sequence hypothetical-release-sequence in
let carries-a-dependency-to = carries_a_dependency_to_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf in
let dependency-ordered-before = dependency_ordered_before_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order release-sequence carries-a-dependency-to in
let inter-thread-happens-before = inter_thread_happens_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency synchronizes-with dependency-ordered-before in
let happens-before = happens_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency inter-thread-happens-before in
let visible-side-effect = visible_side_effect_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency happens-before in
let visible-sequences-of-side-effects = visible_sequences_of_side_effects_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect in
 consistent_inter_thread_happens_before inter-thread-happens-before ∧
 consistent_sc_order actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order sc happens-before ∧
 consistent_modification_order actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency sc modification-order happens-before ∧
 well_formed_reads_from_mapping actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf ∧
 consistent_reads_from_mapping actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf sc modification-order happens-before visible-side-effect visible-sequences-of-side-effects)

indeterminate_reads actions threads = indeterminate_reads =
  {b. is_read b ∧ ¬(∃a. $a \xrightarrow{rf} b$)}

unsequenced_races = unsequenced_races = {(a, b).
  (a ≠ b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
  same_thread a b ∧
  ¬($a \xrightarrow{sequenced\text{-}before} b$ ∨ $b \xrightarrow{sequenced\text{-}before} a$)}

data_races = data_races = {(a, b).
  (a ≠ b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
  ¬same_thread a b ∧
  ¬(is_atomic_action a ∧ is_atomic_action b) ∧
  ¬($a \xrightarrow{happens\text{-}before} b$ ∨ $b \xrightarrow{happens\text{-}before} a$)}

data_races' actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc =
let release-sequence = release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order in
let hypothetical-release-sequence = release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order in
let synchronizes-with = synchronizes_with_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc release-sequence hypothetical-release-sequence in
let carries-a-dependency-to = carries_a_dependency_to_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf in
let dependency-ordered-before = dependency_ordered_before_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order release-sequence carries-a-dependency-to in
let inter-thread-happens-before = inter_thread_happens_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency synchronizes-with dependency-ordered-before in
let happens-before = happens_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency inter-thread-happens-before in
 data_races actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency happens-before

cpp_memory_model opsem (p ∈ 'program) =
  let executions = {(actions, threads, location-kind, sequenced-before, additional-synchronized-with, data-dependency, control-dependency, rf, modification-order, sc).
        opsem p actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency ∧ consistent_execution actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc} in
    if ∃(actions, threads, location-kind, sequenced-before, additional-synchronized-with, data-dependency, control-dependency, rf, modification-order, sc) ∈ executions .
          (indeterminate_reads actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf ≠ {}) ∨
          (unsequenced_races actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency ≠ {}) ∨
          (data_races' actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc ≠ {})
     then {}
     else executions
```

The full model



We can reason about C concurrency!

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 2 is not affected by Thread 1 and vice-versa

This program is data-race free

This program must print 42

Shared memory

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

This is a *compiler bug*

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

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

Thread 2 is not affected by Thread 1 and vice-versa

This program is data-race free

This program must print 42

Shared memory

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

This is a *concurrency compiler bug*

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

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

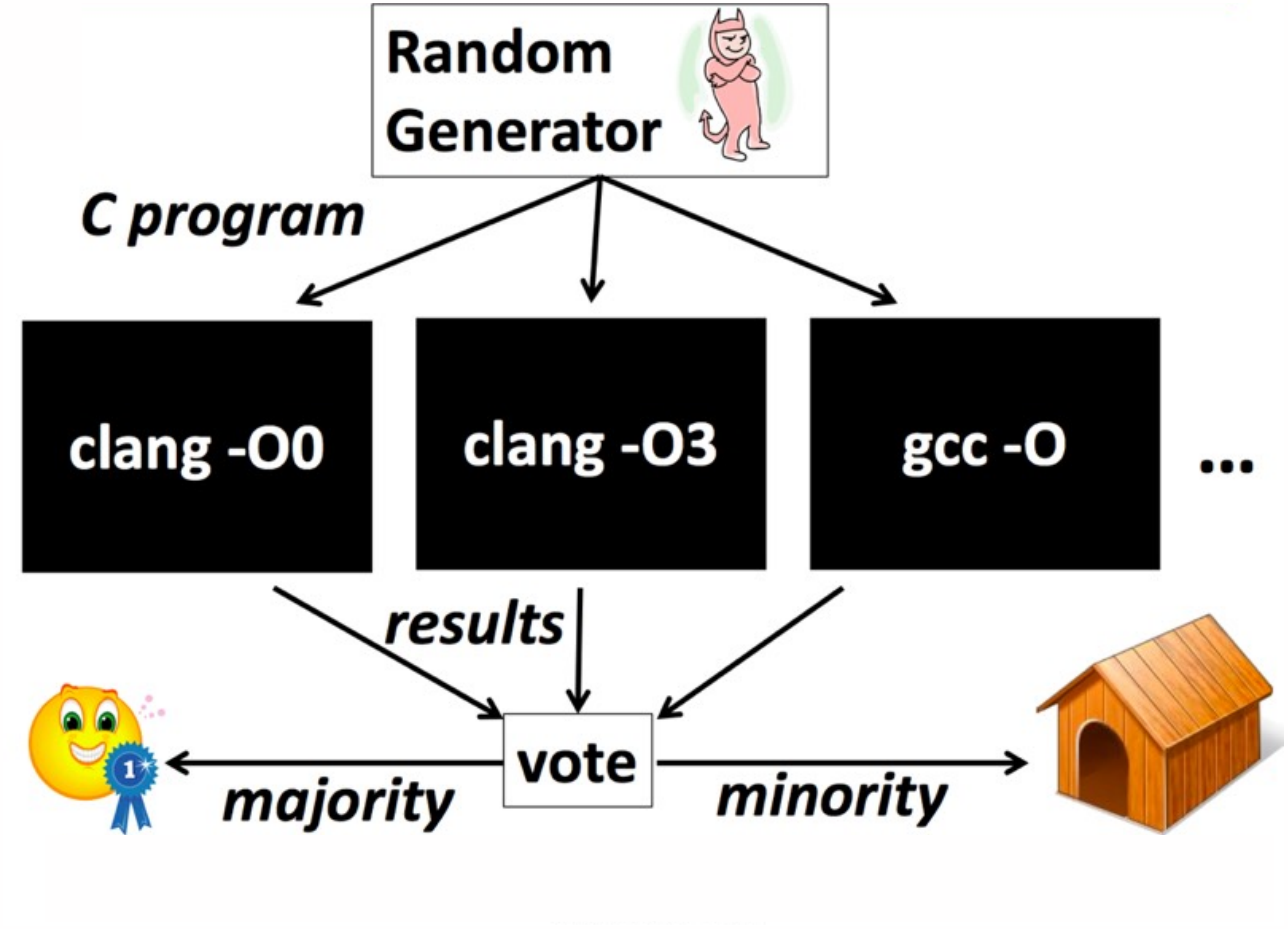
Thread 2 is not affected by Thread 1 and vice-versa

This program is data-race free

This program must print 42

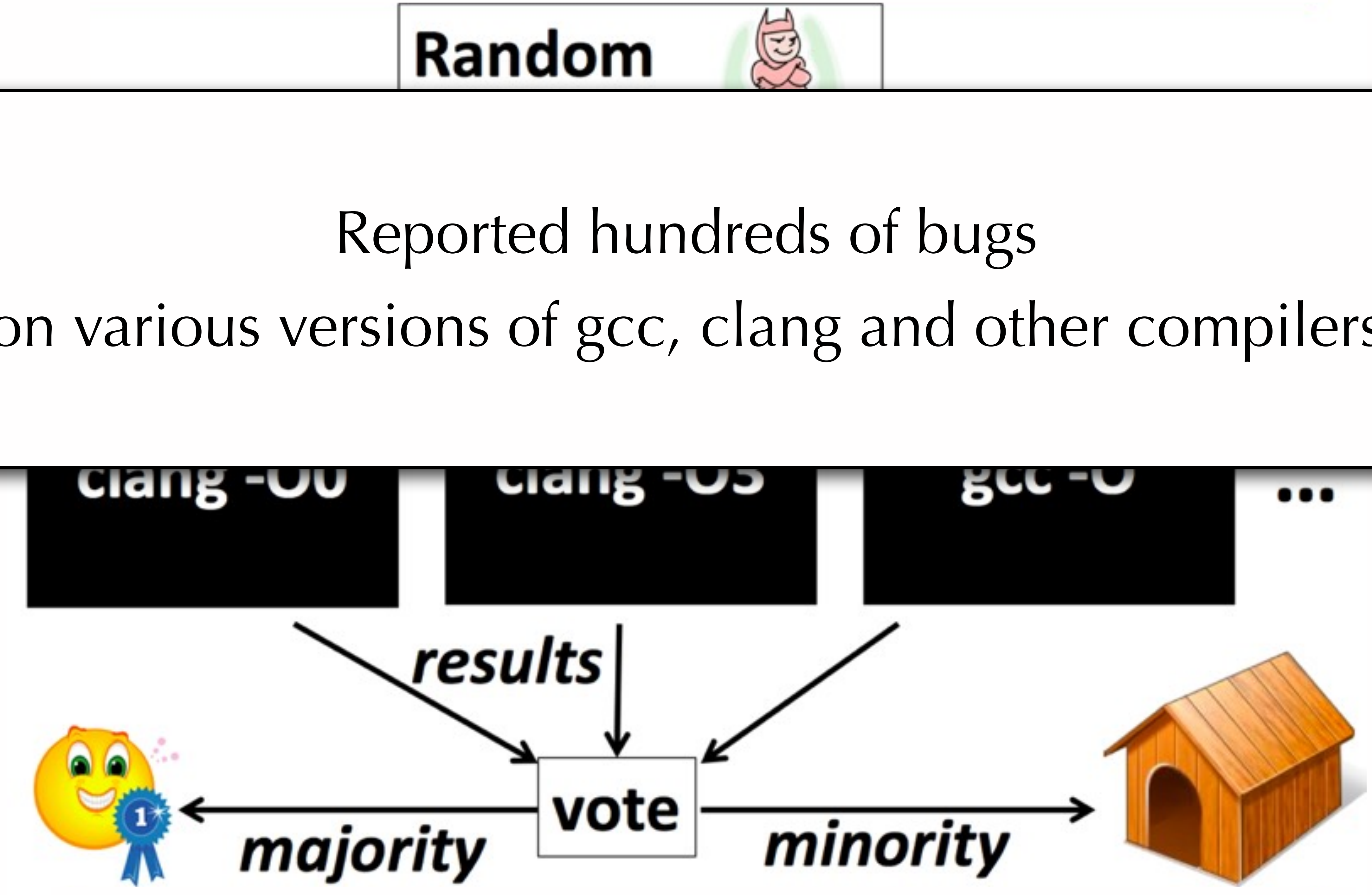
Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

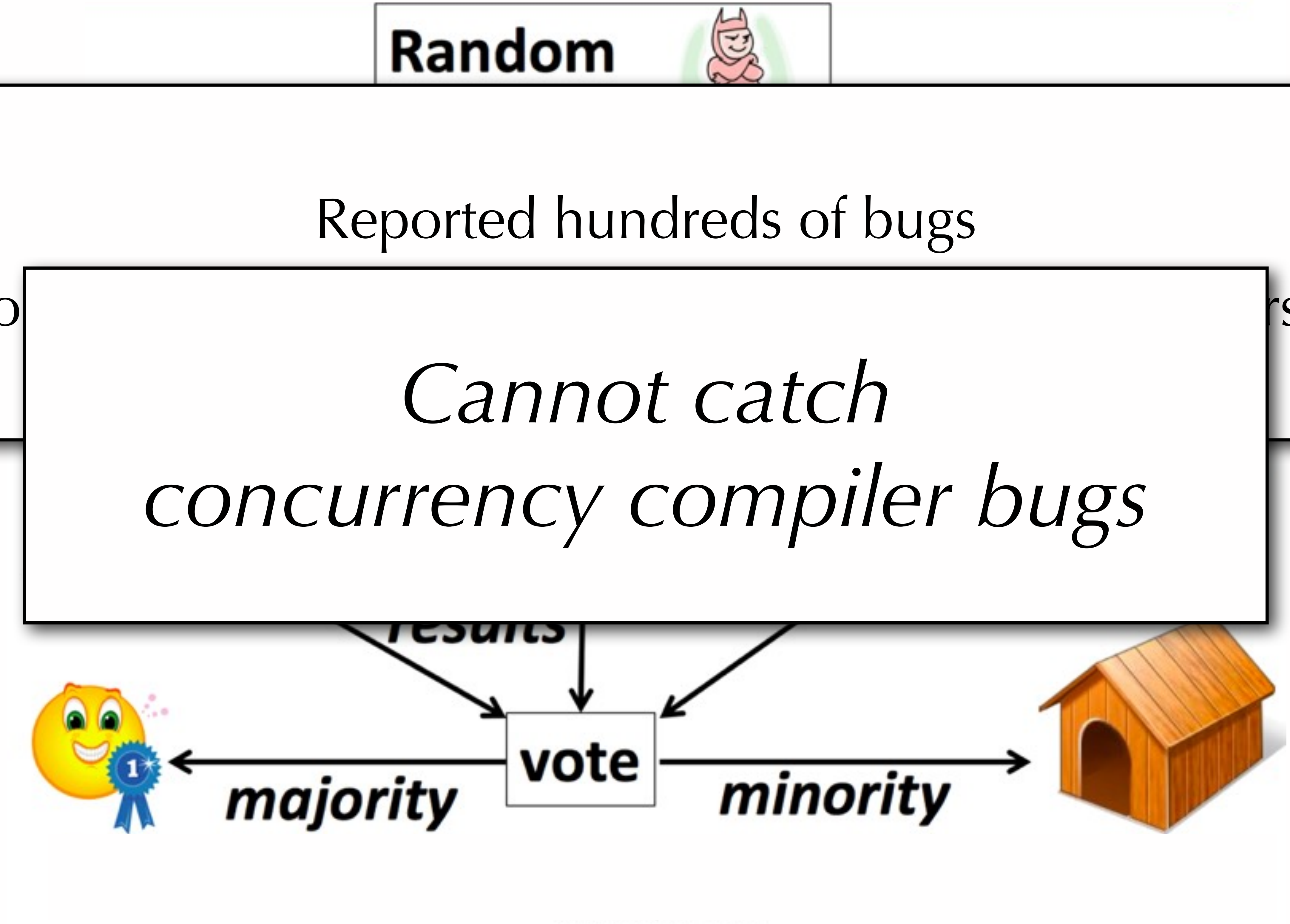


Reported hundreds of bugs

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

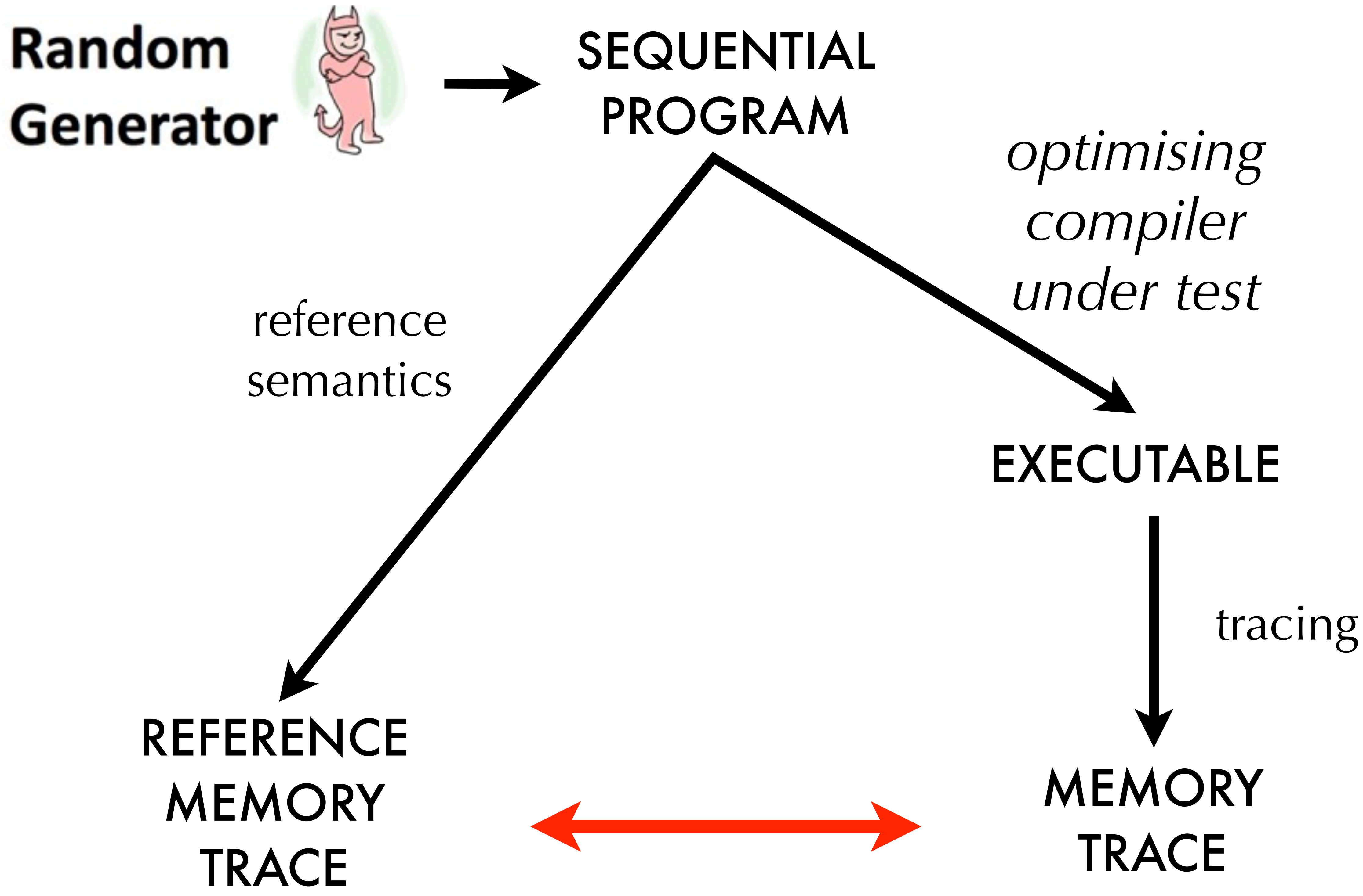
↓

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



Random Generator

SEQUENTIAL PROGRAM

optimising compiler under test

reference semantics

EXECUTABLE

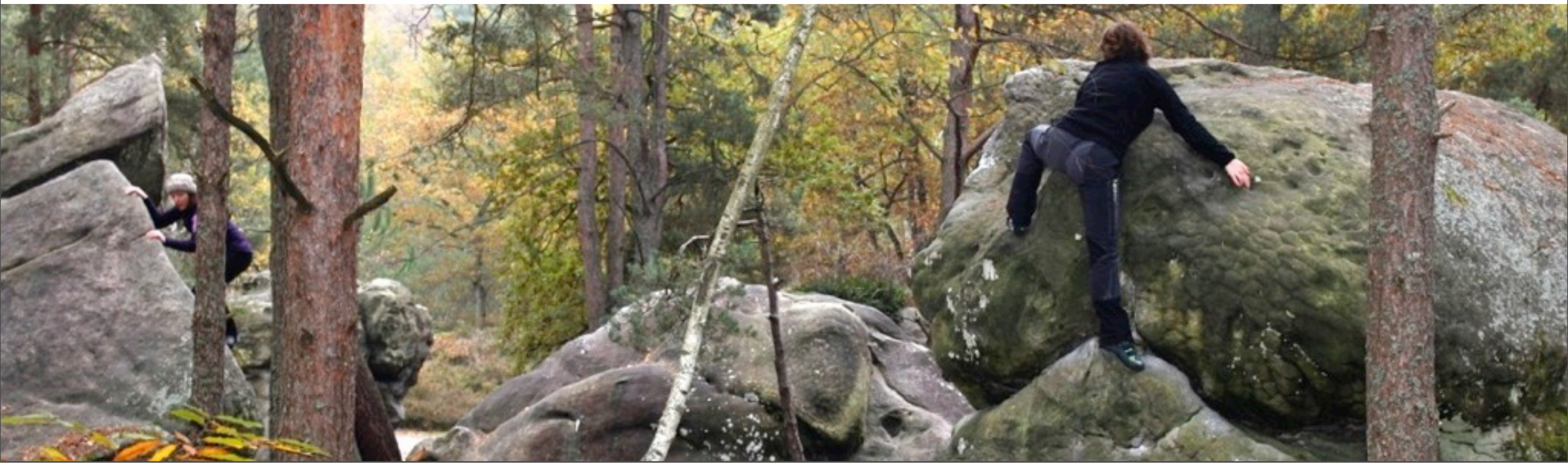
tracing

REFERENCE MEMORY TRACE

MEMORY TRACE

Check: only transformations sound in any concurrent non-racy context

Soundness of compiler optimisations in the C11/C++11 memory model



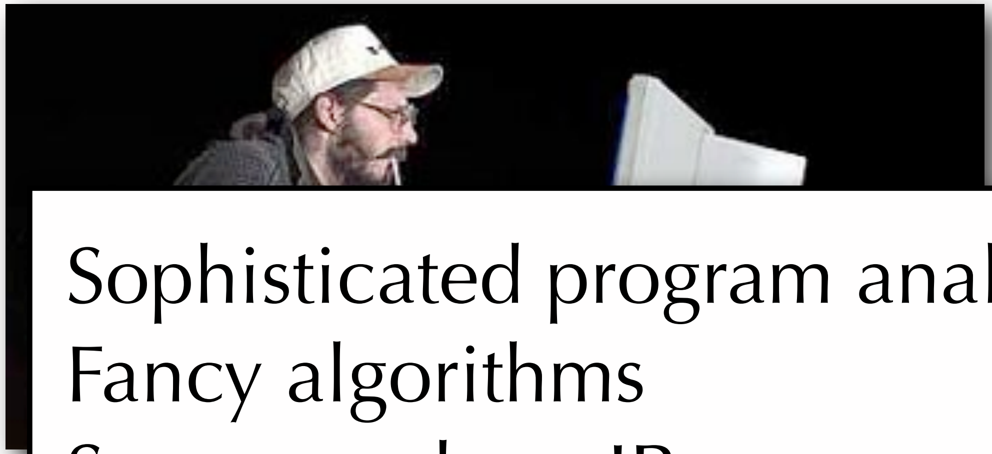
What is an optimisation?

Compiler Writer

Semanticist

What is an optimisation?

Compiler Writer



Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

What is an optimisation?

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 ;
}
```

What is an optimisation?

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 ;
}
```

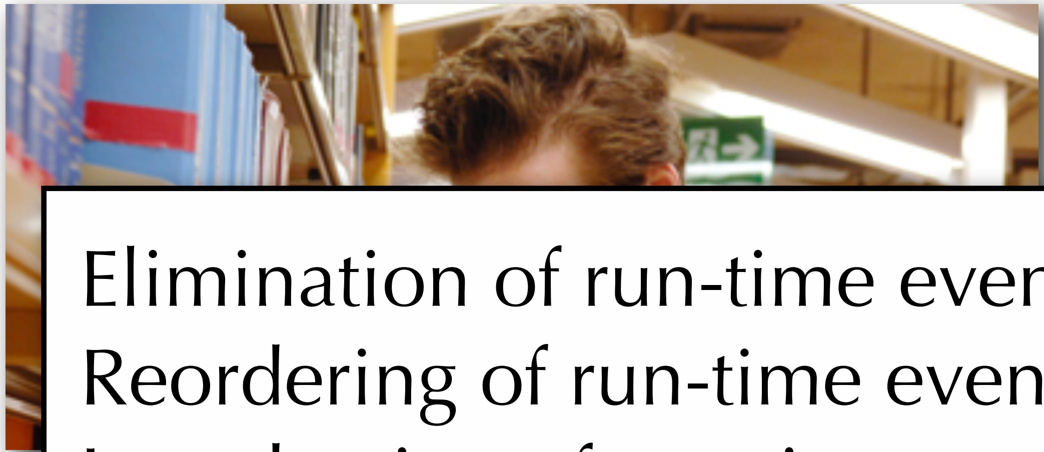
What is an optimisation?

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 ;
}
```

What is an optimisation?

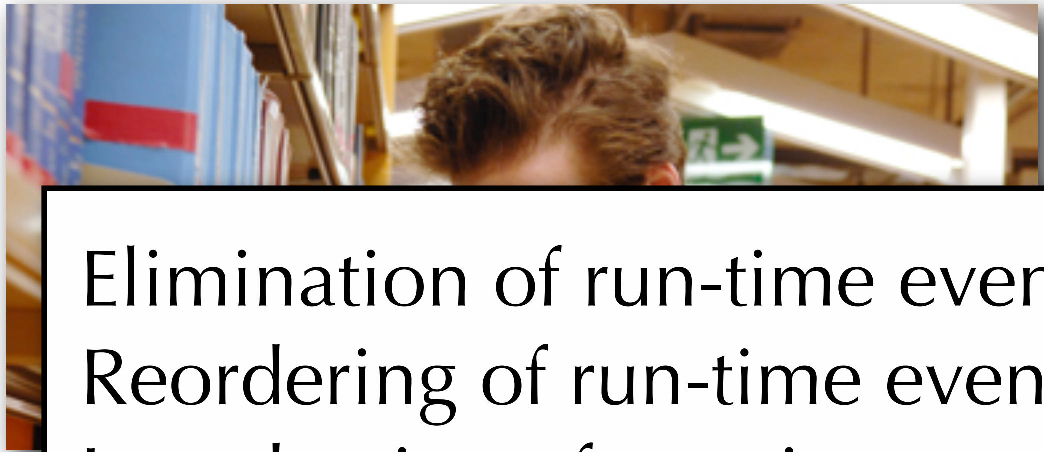
Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

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

Semanticist



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

Operations on sets of events

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

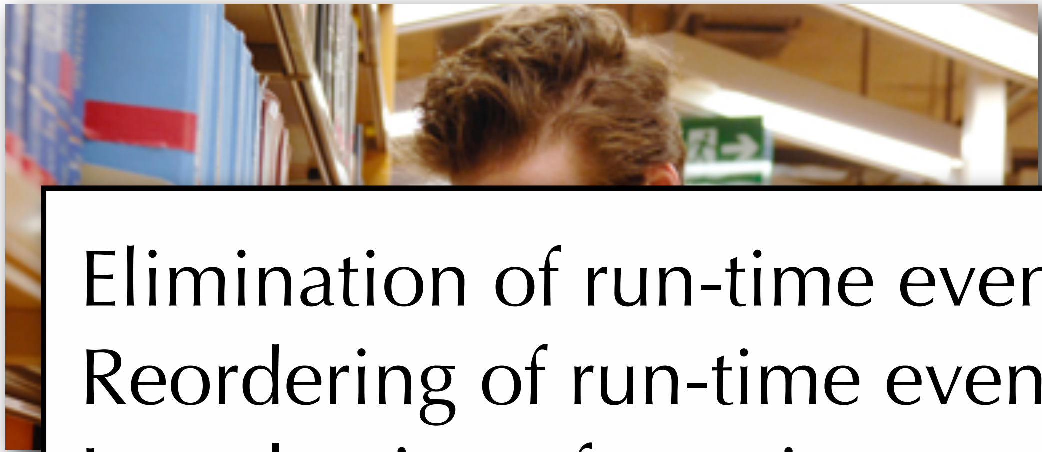
What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist



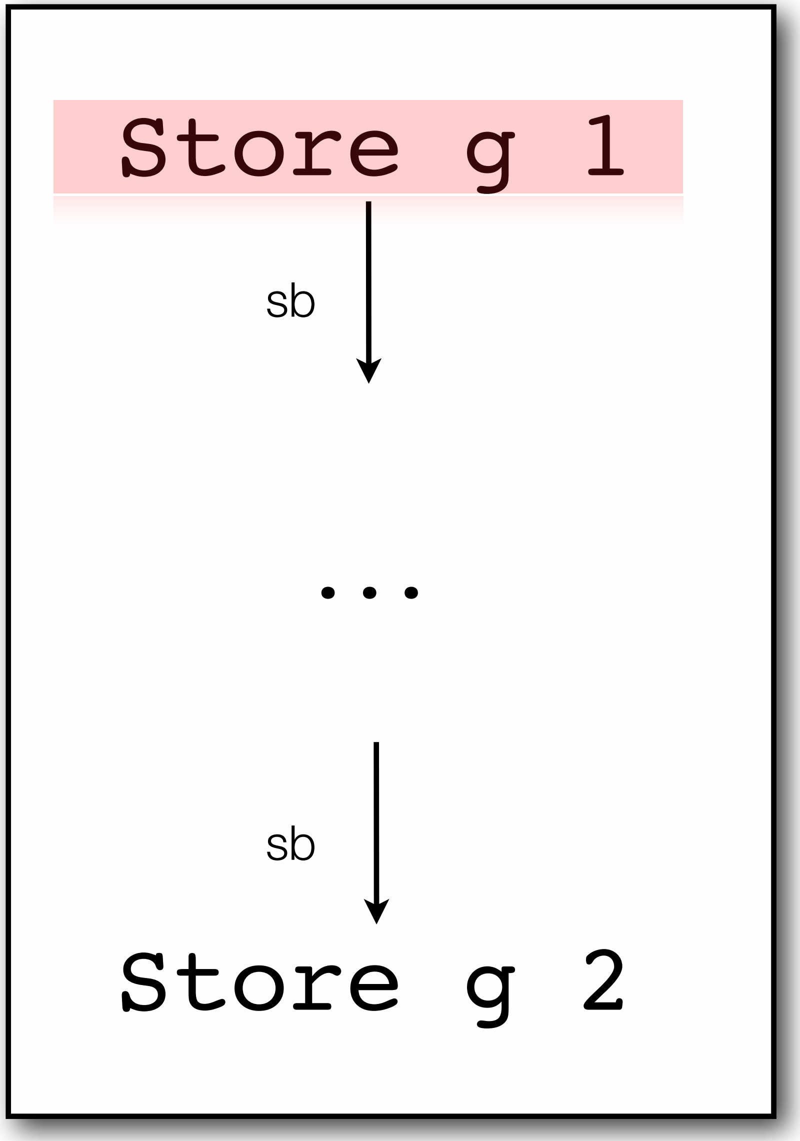
Elimination of run-time events
Reordering of run-time events
Introduction of run-time events

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?

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

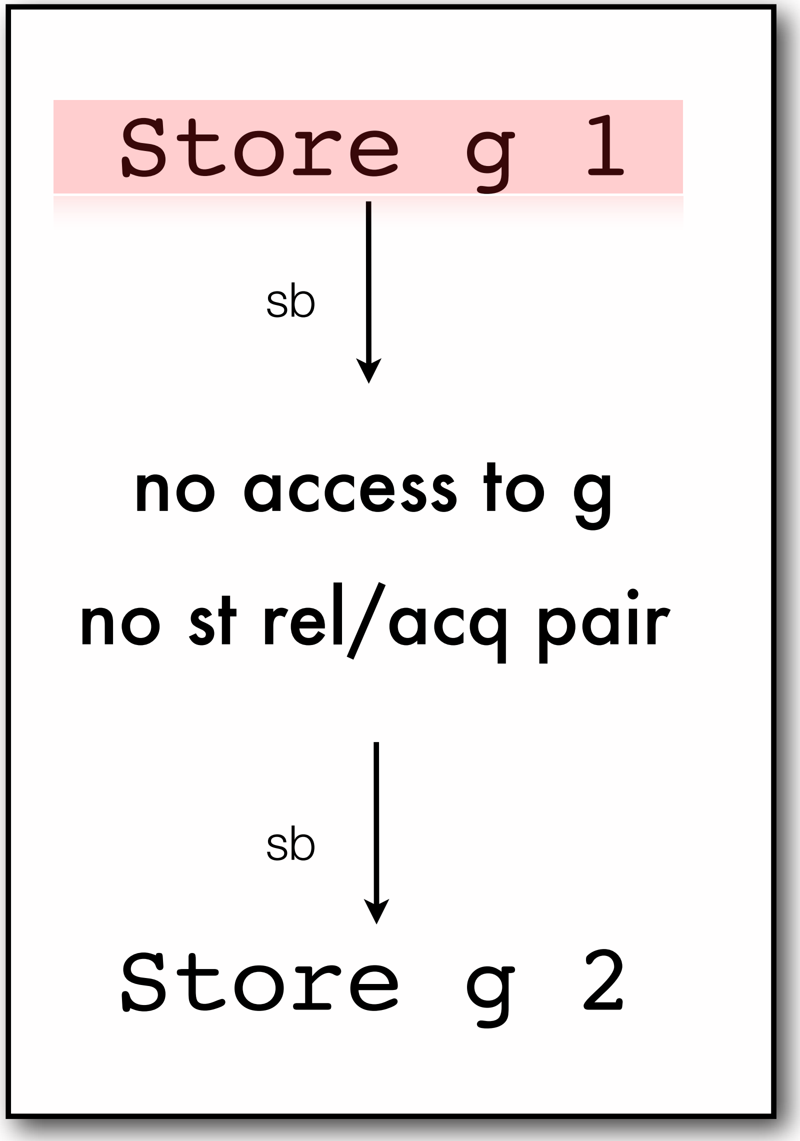
An action is a *release* if it is a possible source of a synchronisation

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*



`Store g 1`

sb

no access to g

no st rel/acq pair

sb

`Store g 2`

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

1. no intervening accesses to g
2. no intervening
   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;
```

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;
```

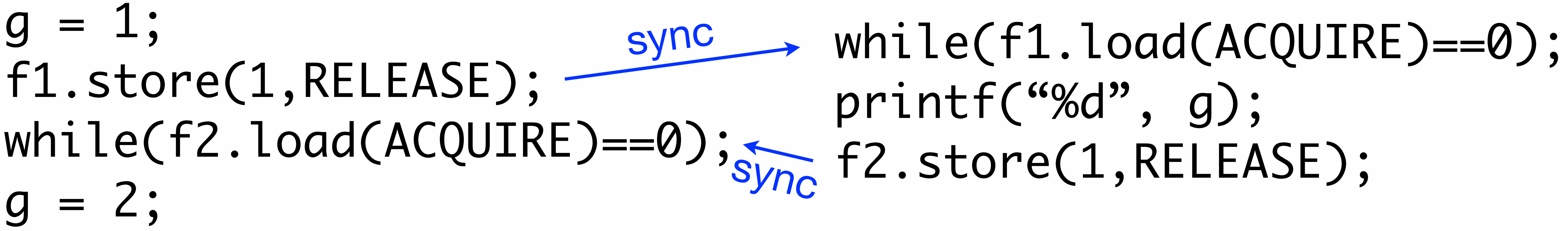
Thread 2

```
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;
```

Thread 2

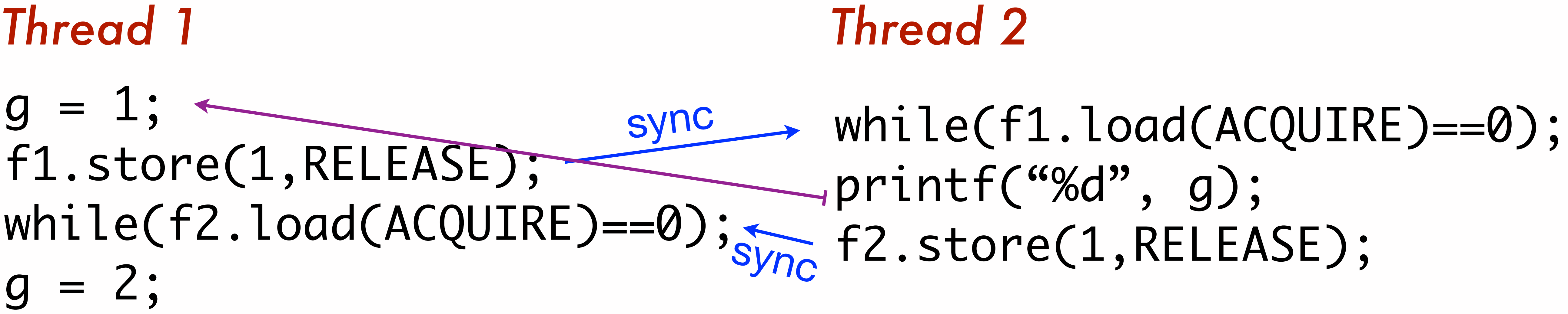
```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

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

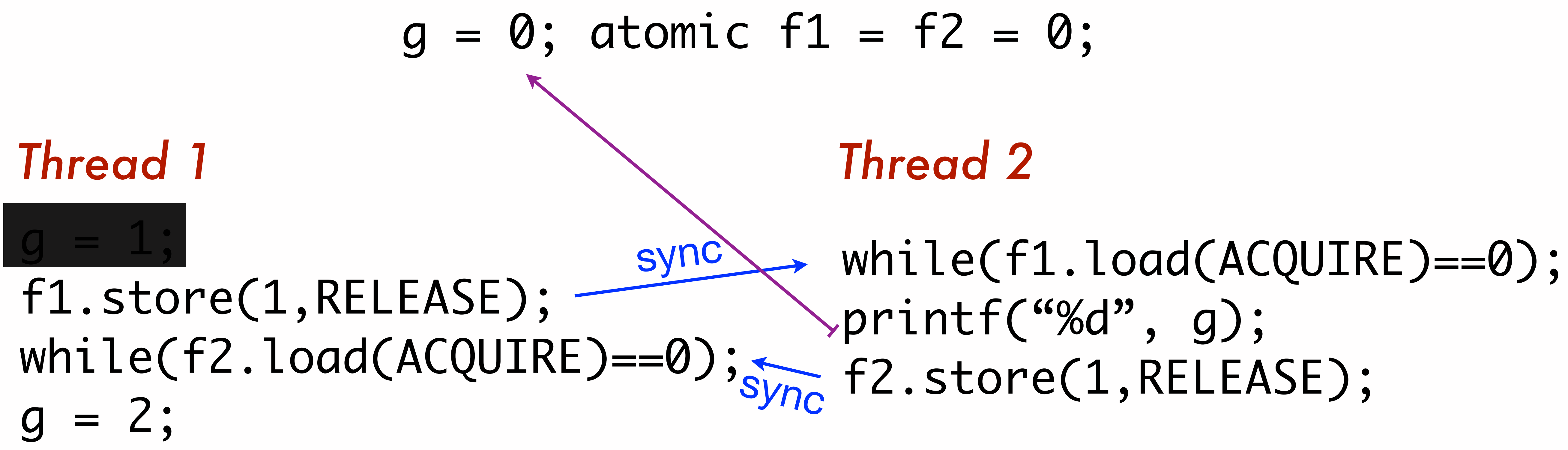
```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

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`

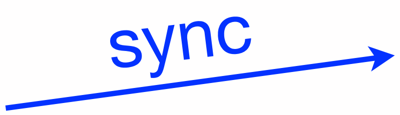
The soundness condition

Shared memory

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

Thread 1

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



Thread 2

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

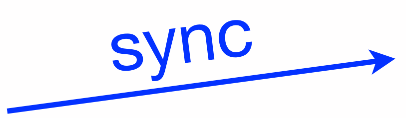
The soundness condition

Shared memory

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

Thread 1

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

g = 2;
```



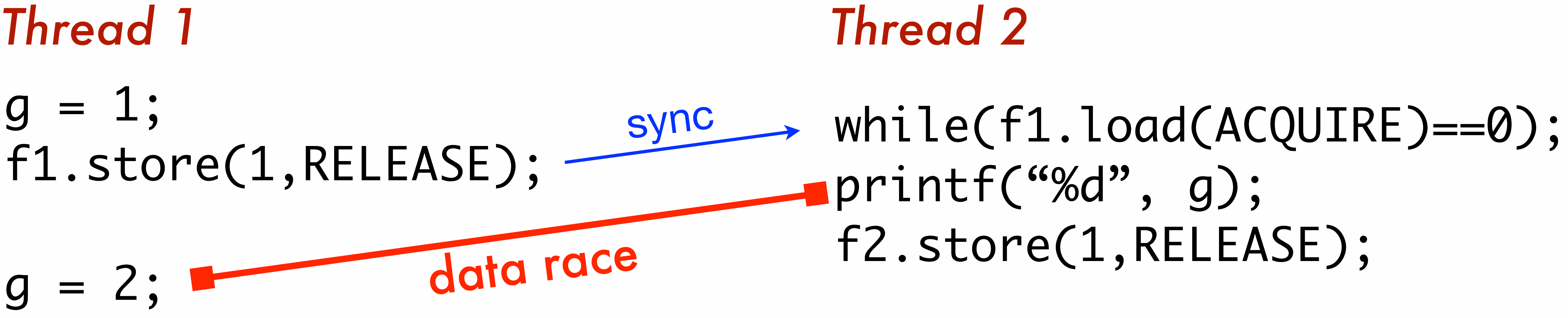
Thread 2

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

The soundness condition

Shared memory

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



Thread 1

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

g = 2;
```

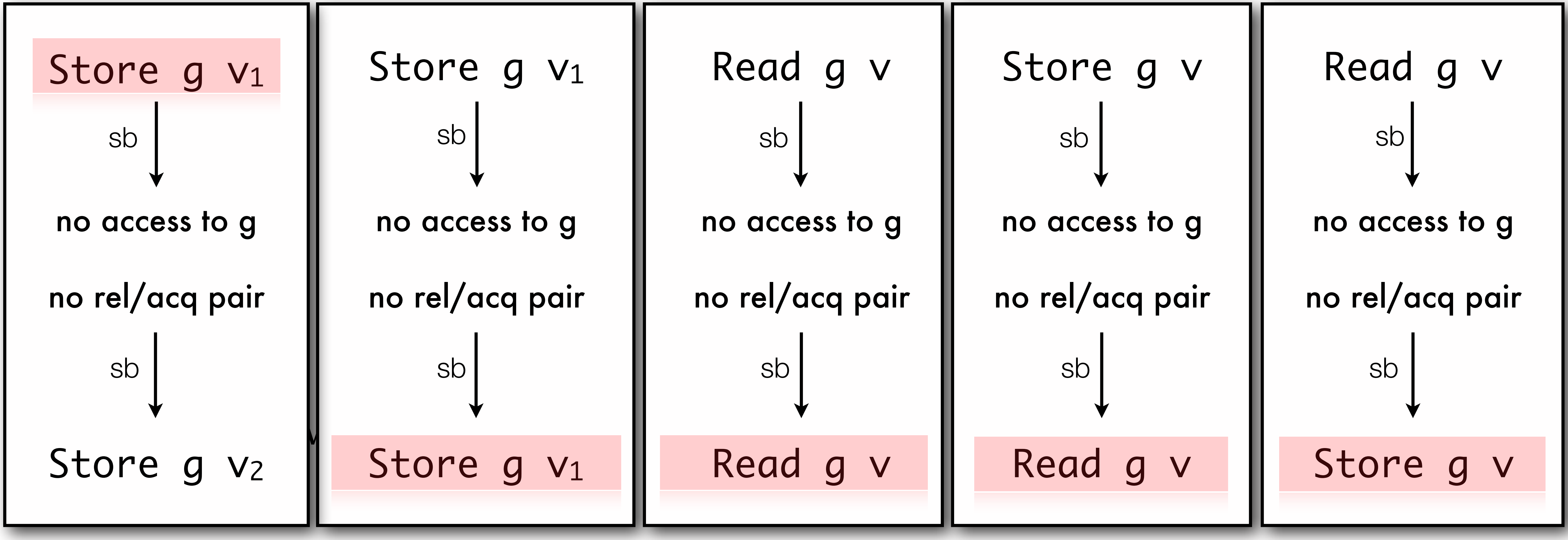
Thread 2

```
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



Store g v_1
sb
no access to g
no rel/acq pair
sb
Store g v_2

Overwritten-Write

Store g v_1
sb
no access to g
no rel/acq pair
sb
Store g v_1

Write-after-Write

Read g v
sb
no access to g
no rel/acq pair
sb
Read g v

Read-after-Read

Store g v
sb
no access to g
no rel/acq pair
sb
Read g v

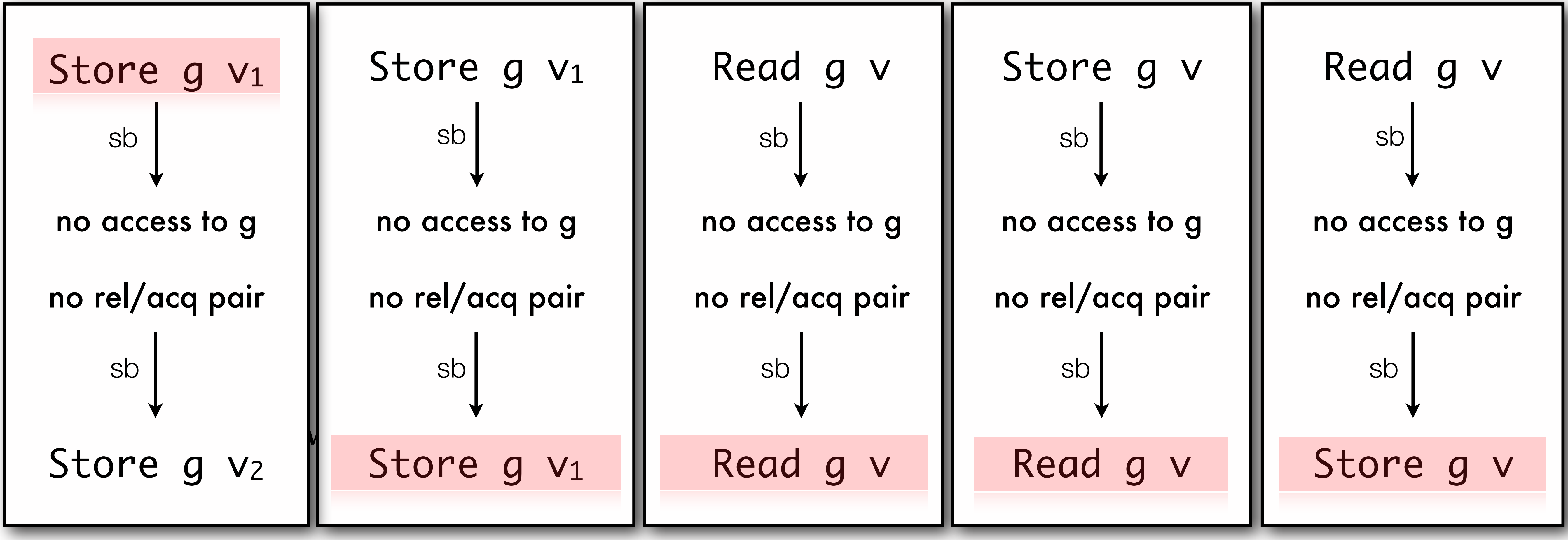
Read-after-Write

Read g v
sb
no access to g
no rel/acq pair
sb
Store g v

Write-after-Read

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

Also correctness statements for **reorderings**, **merging**, and **introductions** of events.



Store g v_1
sb
no access to g
no rel/acq pair
sb
Store g v_2

Overwritten-Write

Store g v_1
sb
no access to g
no rel/acq pair
sb
Store g v_1

Write-after-Write

Read g v
sb
no access to g
no rel/acq pair
sb
Read g v

Read-after-Read

Store g v
sb
no access to g
no rel/acq pair
sb
Read g v

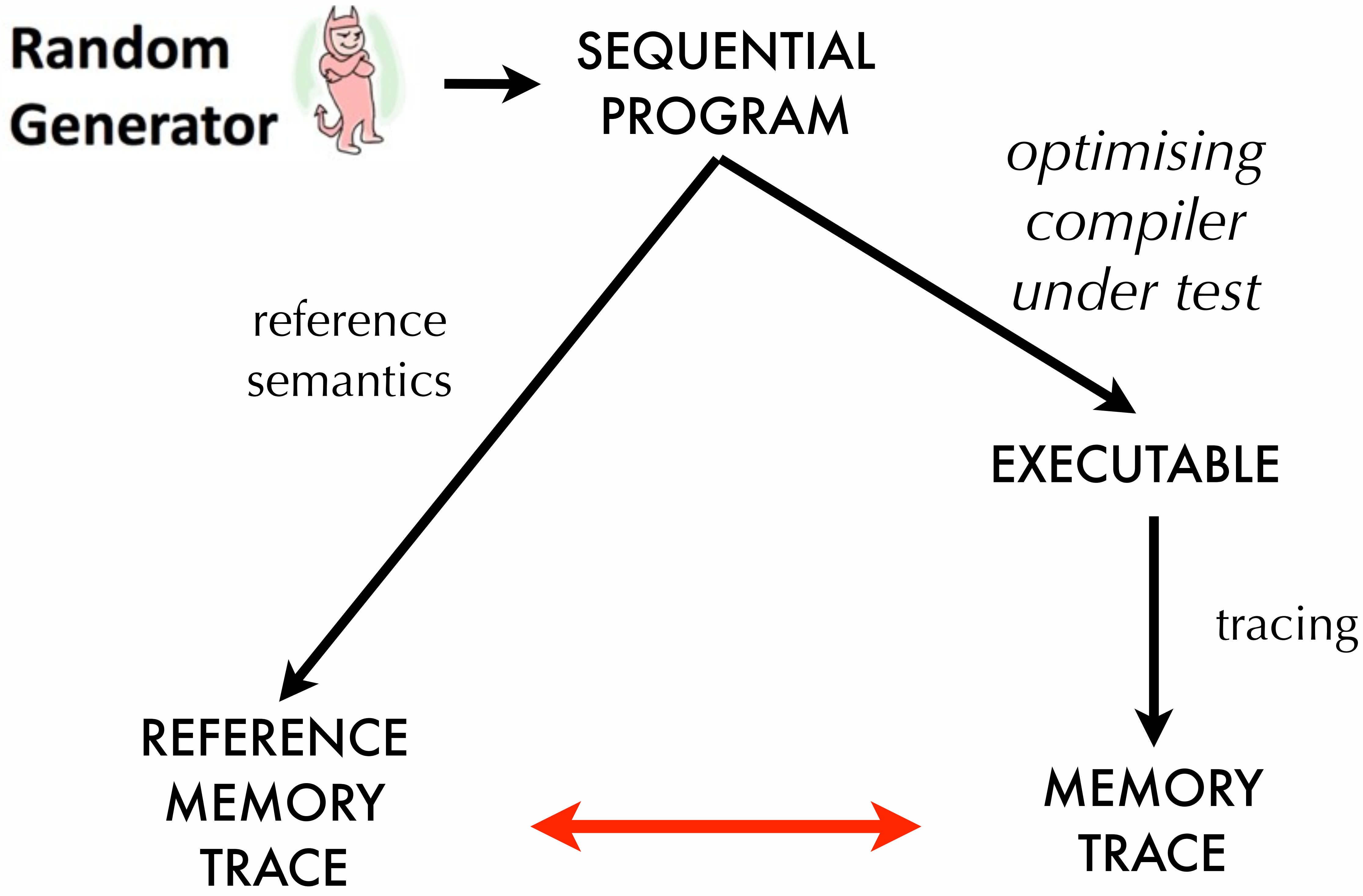
Read-after-Write

Read g v
sb
no access to g
no rel/acq pair
sb
Store g v

Write-after-Read

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

From theory to the Cmmtest tool



Random Generator

SEQUENTIAL PROGRAM

optimising compiler under test

reference semantics

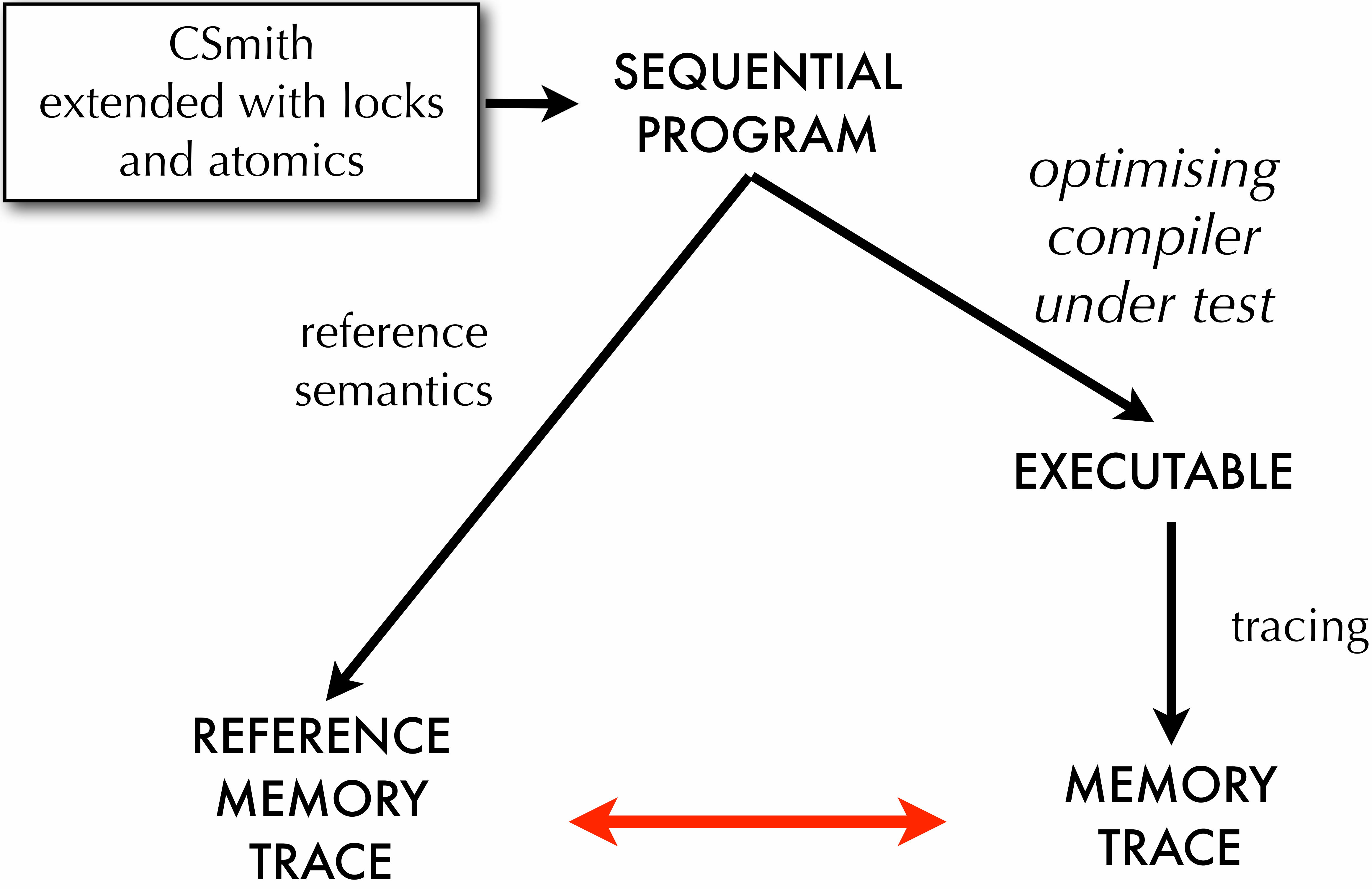
EXECUTABLE

tracing

REFERENCE MEMORY TRACE

MEMORY TRACE

Check: only transformations sound in any concurrent non-racy context



CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

*optimising
compiler
under test*

reference
semantics

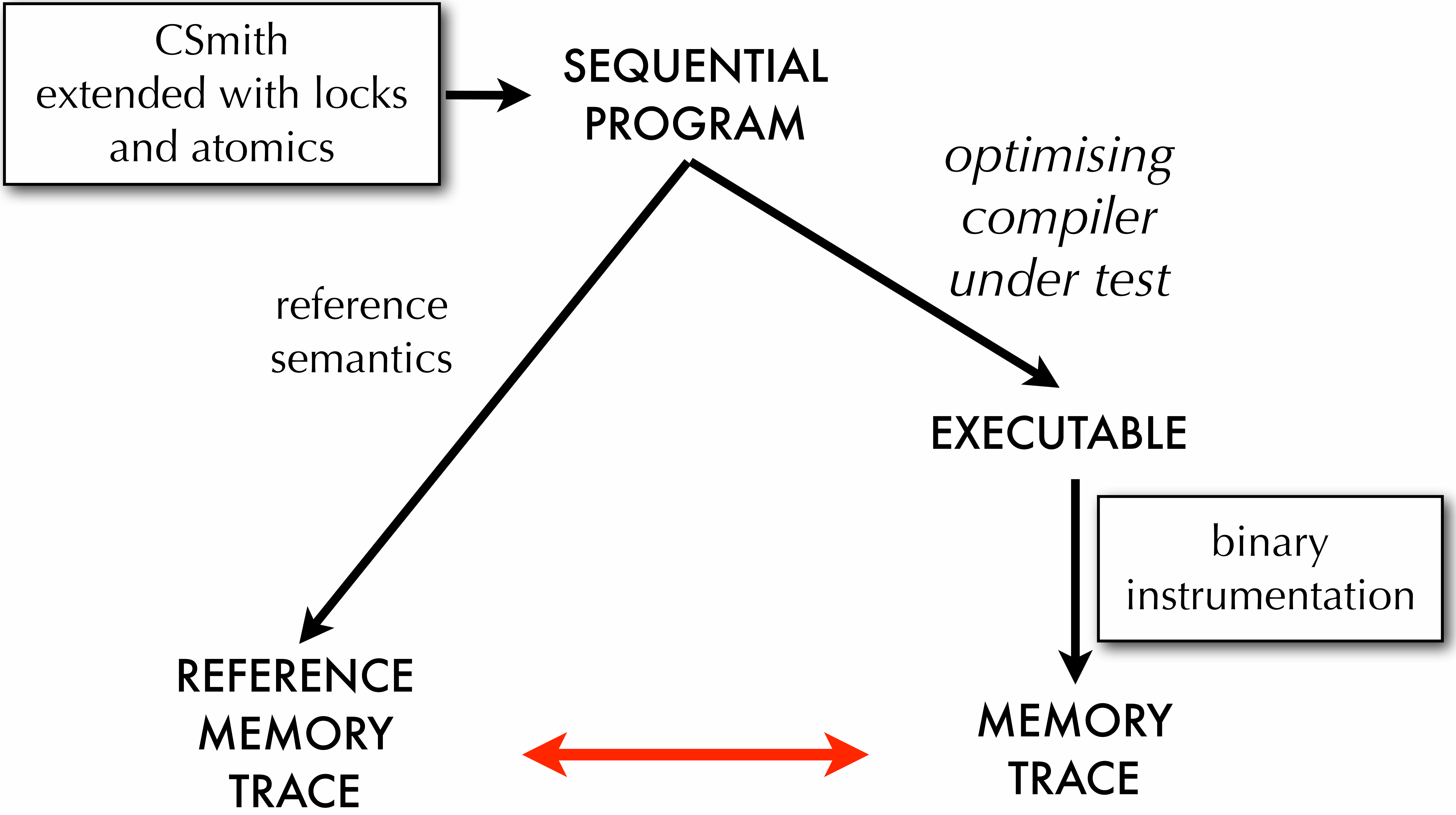
EXECUTABLE

tracing

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

Check: only transformations sound
in any concurrent non-racy context



CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

reference
semantics

optimising
compiler
under test

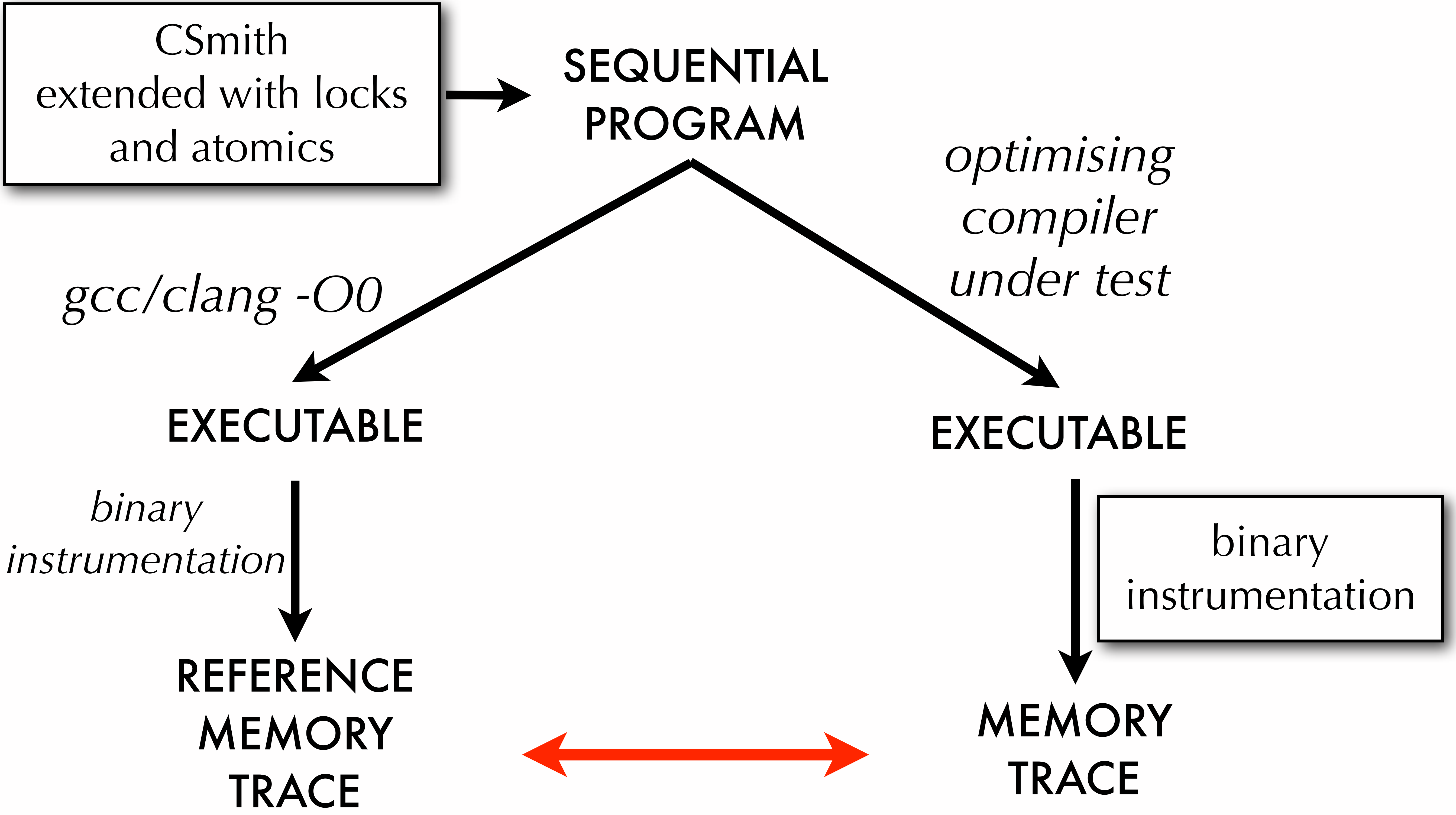
EXECUTABLE

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

Check: 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

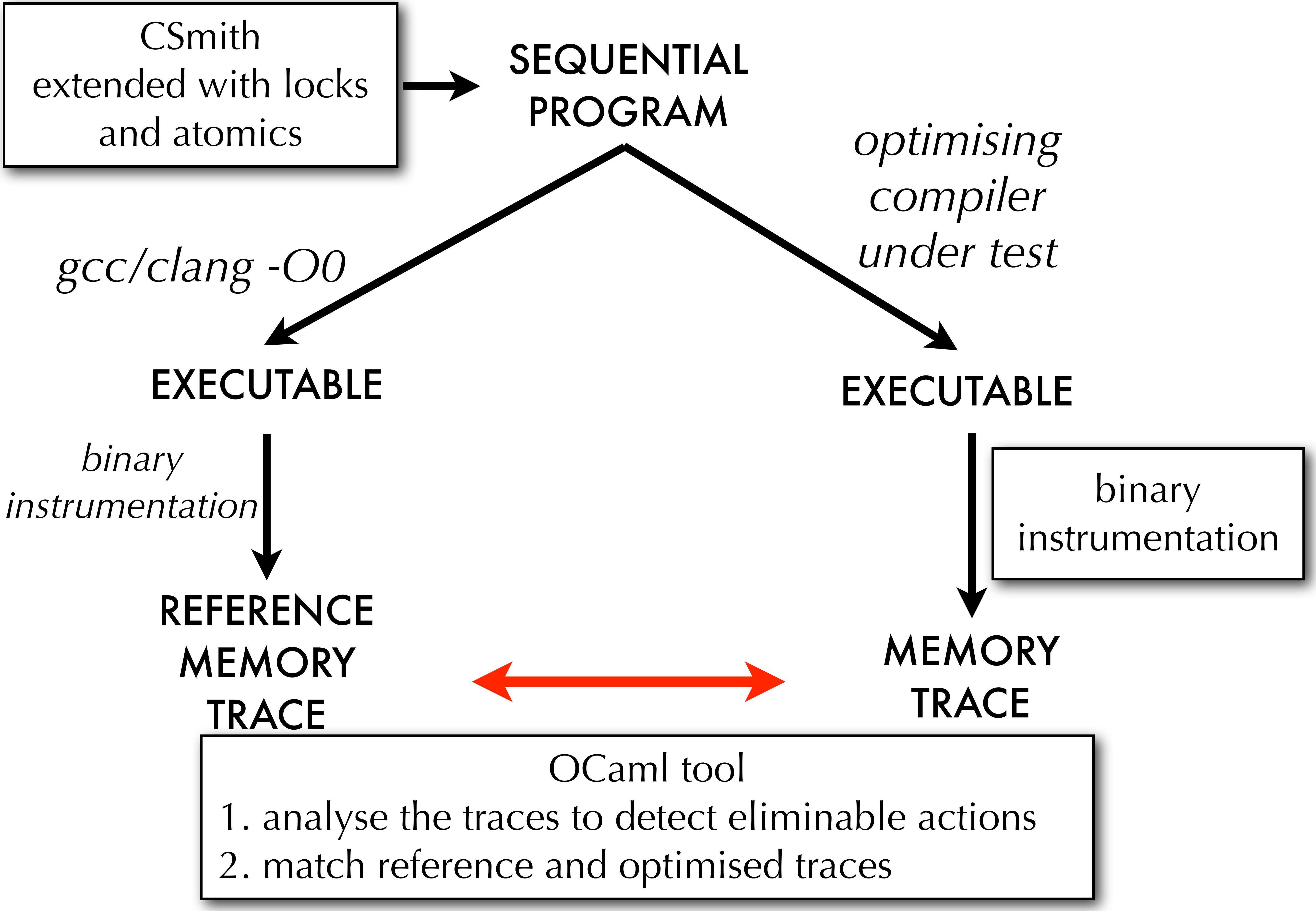
binary
instrumentation

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

Check: 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

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

 void func_1(void){
     int *l8 = &g6;
     int l36 = 0x5E9D070FL;
     unsigned int l107 = 0xAA37C3ACL;
     g4 &= g3;
     g5++;
     int *l102 = &l36;
     for (g6 = 4; g6 < (-3); g6 += 1);
     l102 = &g6;
     *l102 = ((*l8) && (l107 << 7)*(*l102));
 }
```

Start with a randomly generated well-defined program

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

```c
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

reference semantics

```
Load  g4 1
Store g4 0
Load  g5 1
Store g5 2
Store g6 4
Load  g6 4
Load  g6 4
Load  g6 4
Store g6 1
Load  g4 0
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

reference semantics

```
Load  g4 1
Store g4 0
Load  g5 1
Store g5 2
Store g6 4
Load  g6 4
Load  g6 4
Load  g6 4
Store g6 1
Load  g4 0
```

gcc -O2 memory trace

```
Load  g5 1
Store g4 0
Store g6 1
Store g5 2
Load  g4 0
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

reference semantics

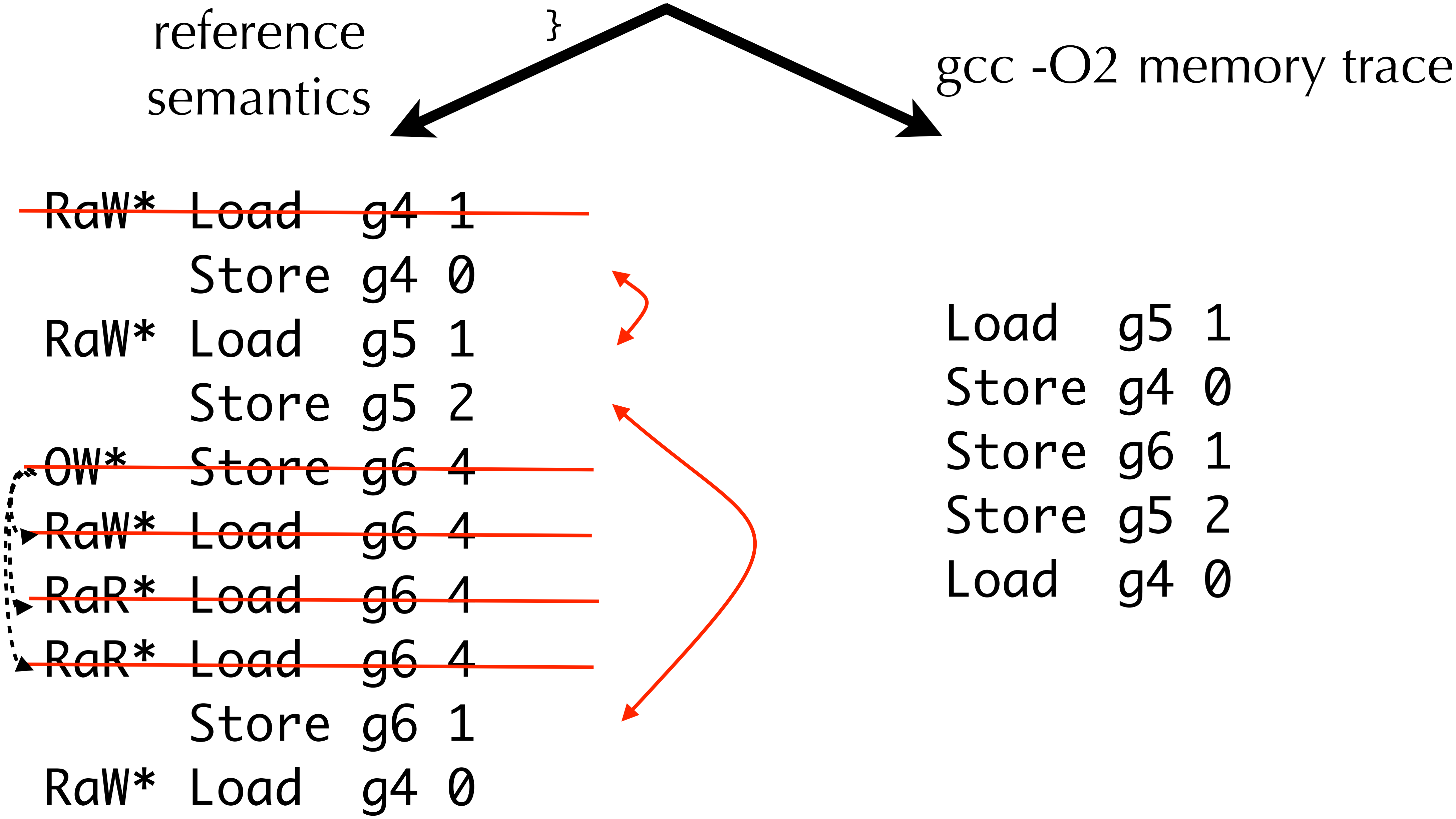
```
RaW* Load  g4 1
     Store g4 0
RaW* Load  g5 1
     Store g5 2
OW*  Store g6 4
RaW* Load  g6 4
RaR* Load  g6 4
RaR* Load  g6 4
     Store g6 1
RaW* Load  g4 0
```

gcc -O2 memory trace

```
Load  g5 1
Store g4 0
Store g6 1
Store g5 2
Load  g4 0
```

Init g3 0
Init g4 1
Init g5 1
Init g6 6

```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```



reference semantics

gcc -O2 memory trace

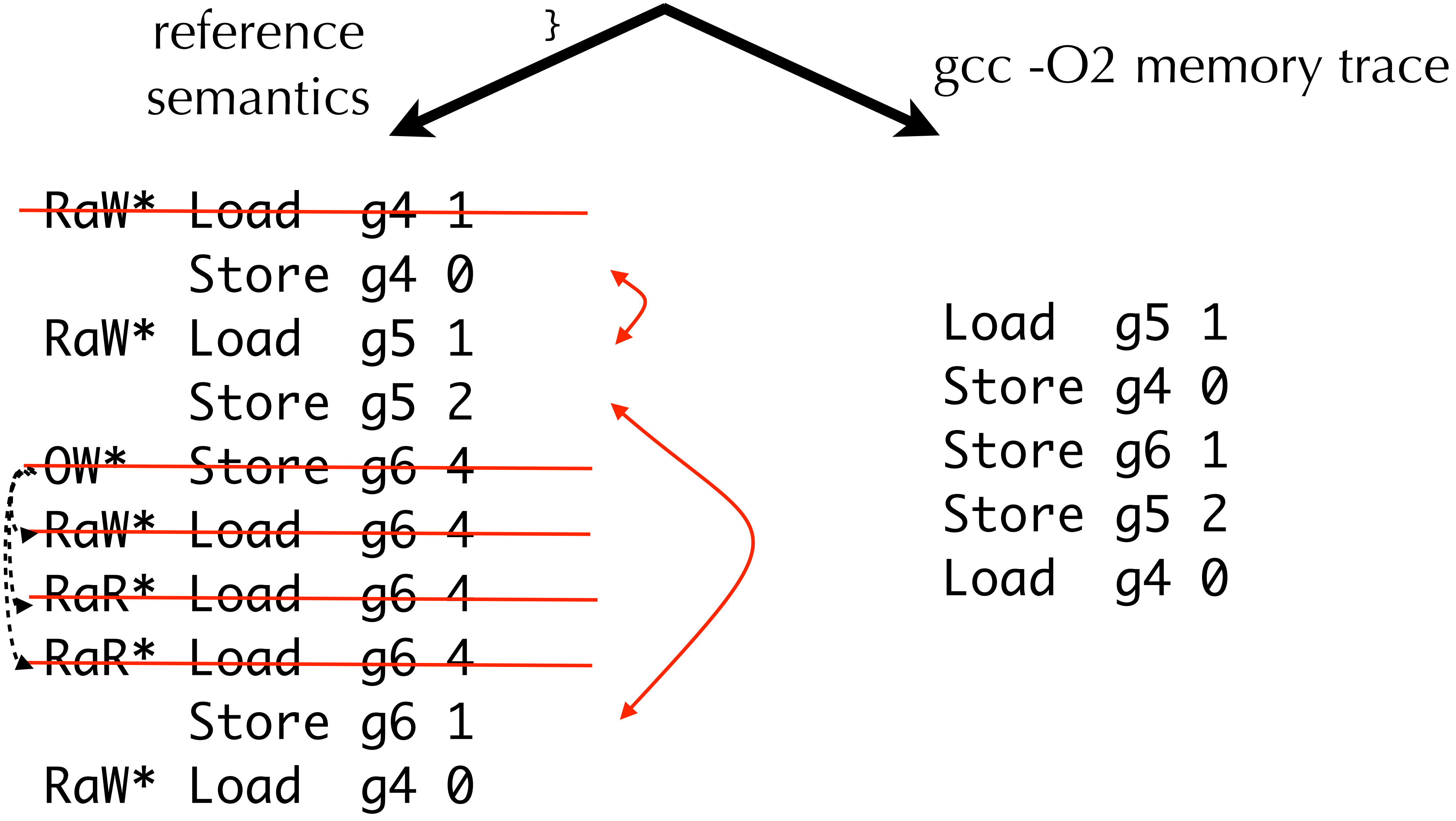
~~RaW* Load g4 1~~
Store g4 0
RaW* Load g5 1
Store g5 2
~~OW* Store g6 4~~
~~RaW* Load g6 4~~
~~RaR* Load g6 4~~
~~RaR* Load g6 4~~
Store g6 1
RaW* Load g4 0

Load g5 1
Store g4 0
Store g6 1
Store g5 2
Load g4 0

Init g3 0

```
void func_1(void){
    int *l8 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

Can match applying
only correct eliminations and reorderings



reference semantics

gcc -O2 memory trace

~~RaW* Load g4 1~~
Store g4 0
RaW* Load g5 1
Store g5 2
~~OW* Store g6 4~~
~~RaW* Load g6 4~~
~~RaR* Load g6 4~~
~~RaR* Load g6 4~~
Store g6 1
RaW* Load g4 0

Load g5 1
Store g4 0
Store g6 1
Store g5 2
Load g4 0

```
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)
    ;
```

If we focus on the miscompiled initial example...

```
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)
    ;
}
```

```
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
```

```
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 ⟶ detect compiler bug

```
          ;
      }
```

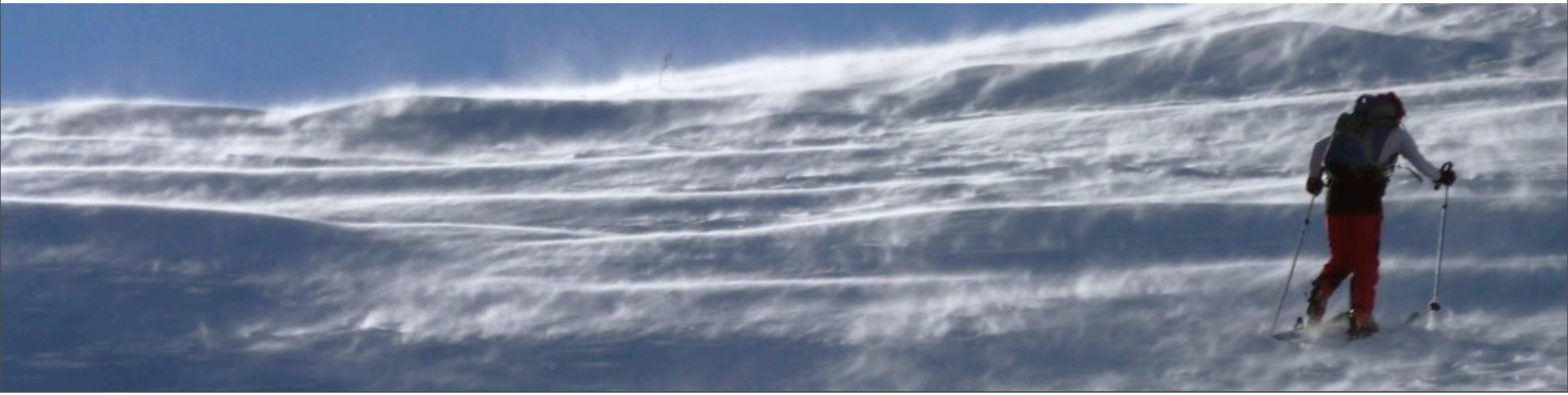
reference
semantics

```
Load a 1
```

gcc -O2 memory trace

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

Applications



1. Testing C compilers (GCC, Clang, ICC)

Some concurrency compiler bugs found
in the latest version of GCC.

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

Remark: these bugs break the Posix thread model too.

All promptly fixed.

2. Checking compiler invariants

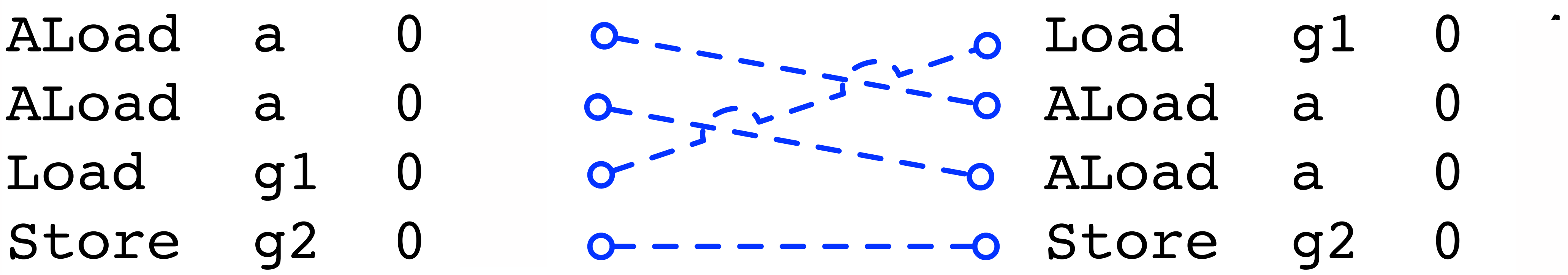
GCC internal invariant: never reorder with an atomic access

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                        uint16_t g

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

Correct or not?

3. Detecting unexpected behaviours

```
uint16_t g                          uint16_t g

for (; g==0; g--);      →           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.*

3. Detecting unexpected behaviours

```
uint16_t g                           uint16_t g

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

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

`Load    g   0`

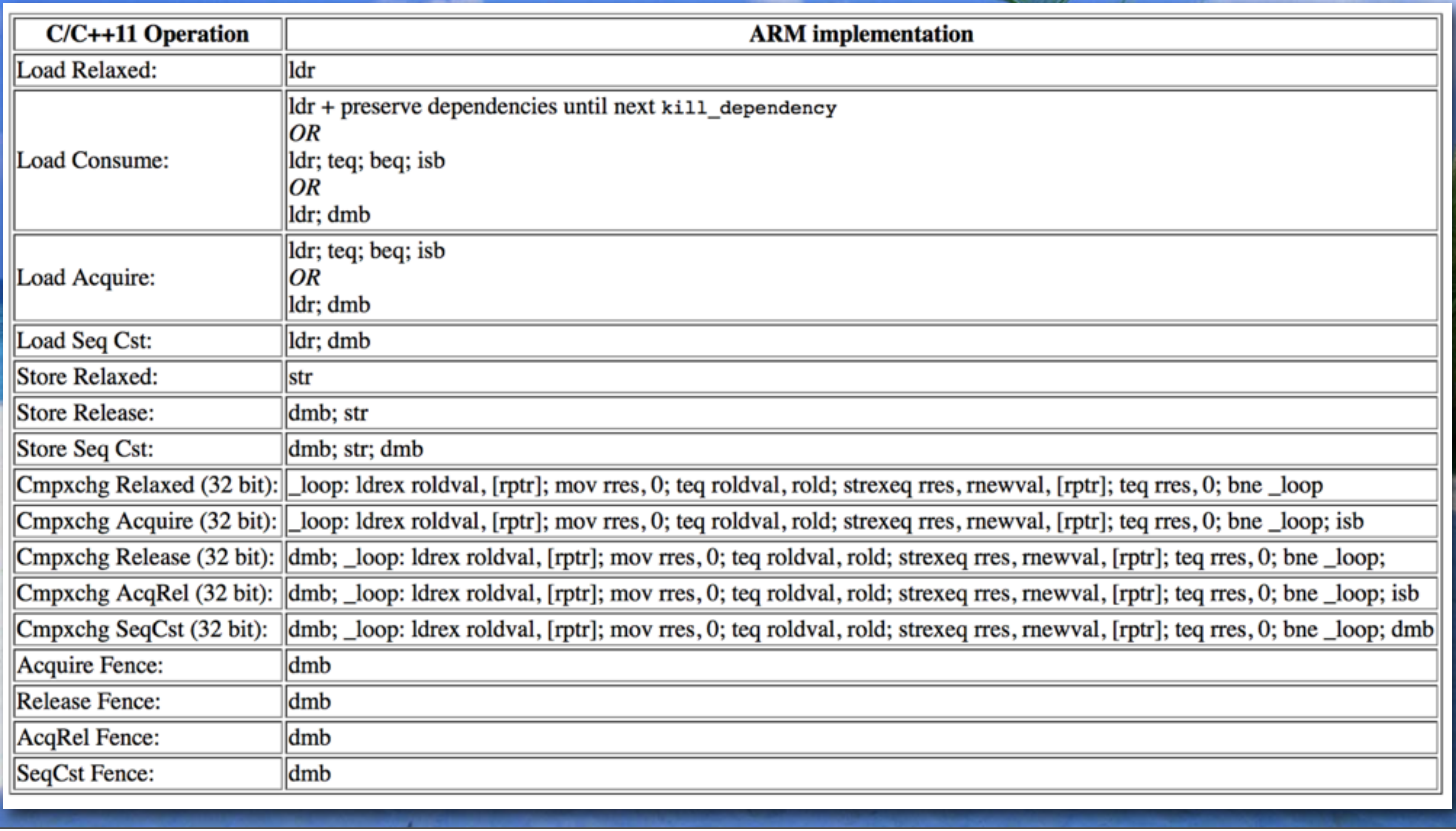
`Store   g   0`

False positives in Thread Sanitizer

The formalisation of the C11 memory model enables compiler testing... what else?

Proving the correctness of mappings for atomics

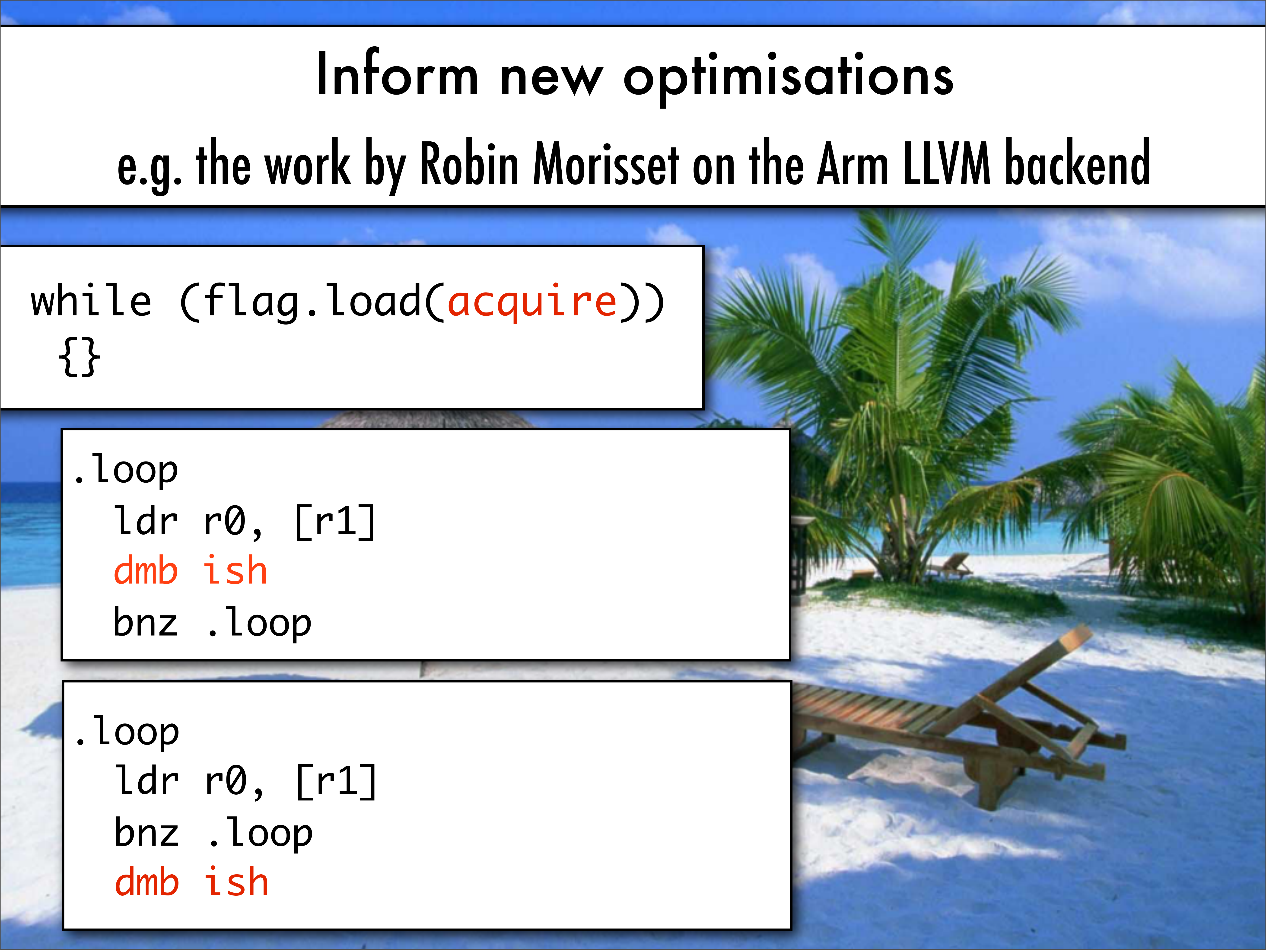
https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html



C/C++11 Operation	ARM implementation
Load Relaxed:	ldr
Load Consume:	ldr + preserve dependencies until next `kill_dependency` *OR* ldr; teq; beq; isb *OR* ldr; dmb
Load Acquire:	ldr; teq; beq; isb *OR* ldr; dmb
Load Seq Cst:	ldr; dmb
Store Relaxed:	str
Store Release:	dmb; str
Store Seq Cst:	dmb; str; dmb
Cmpxchg Relaxed (32 bit):	_loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop
Cmpxchg Acquire (32 bit):	_loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb
Cmpxchg Release (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop;
Cmpxchg AcqRel (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb
Cmpxchg SeqCst (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; dmb
Acquire Fence:	dmb
Release Fence:	dmb
AcqRel Fence:	dmb
SeqCst Fence:	dmb

Inform new optimisations

e.g. the work by Robin Morisset on the Arm LLVM backend



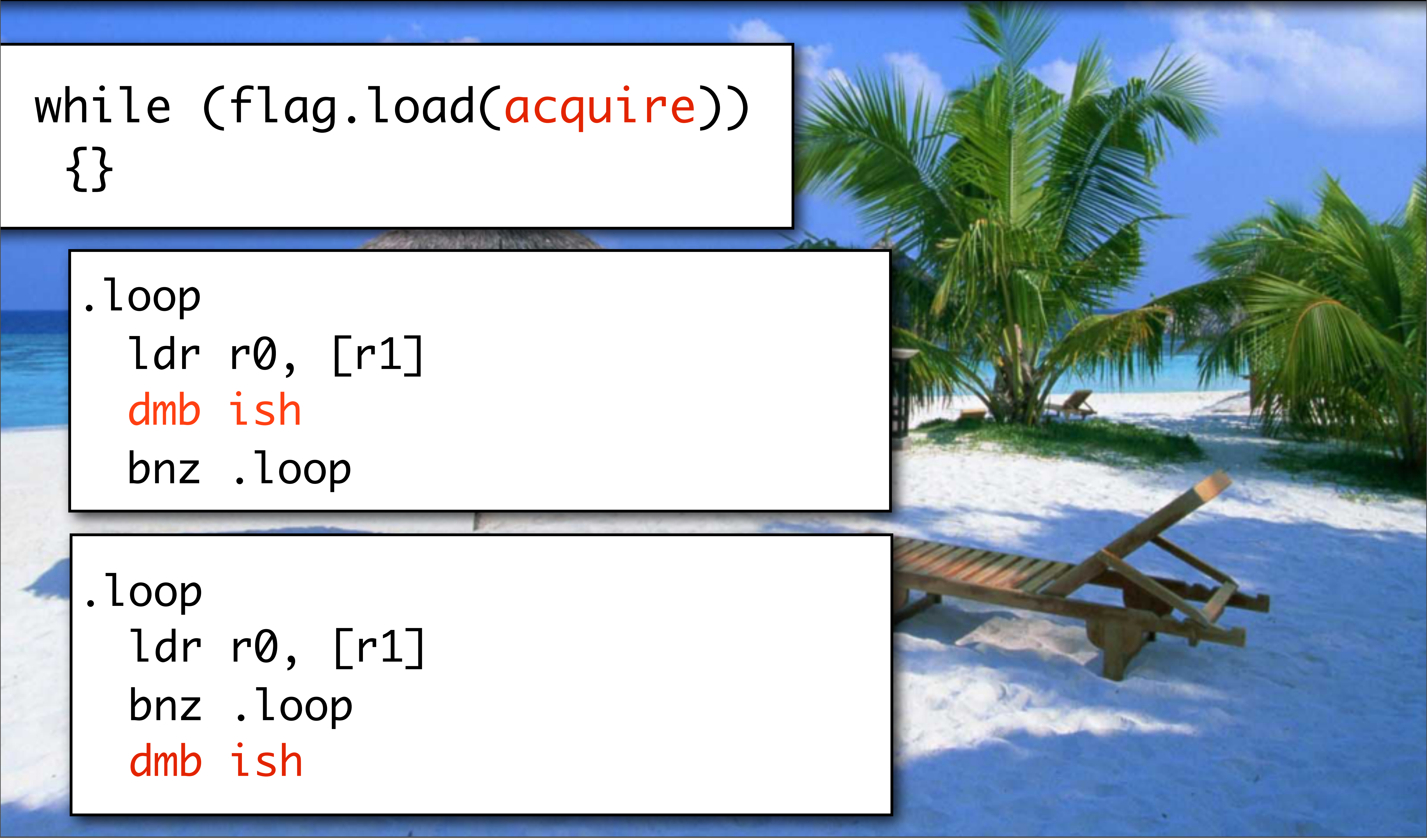
```
while (flag.load(acquire))
 {}
```

```
.loop
  ldr r0, [r1]
  dmb ish
  bnz .loop
```

```
.loop
  ldr r0, [r1]
  bnz .loop
  dmb ish
```

Inform new optimisations

e.g. the work by Robin Morisset on the Arm LLVM backend



```
while (flag.load(acquire))
  {}
```

```
.loop
  ldr r0, [r1]
  dmb ish
  bnz .loop
```

```
.loop
  ldr r0, [r1]
  bnz .loop
  dmb ish
```

Not all of C/C++11 is good

A second look at qualifiers

LESS RELAXED

`MO_SEQ_CST`

`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

`MO_RELAXED`

MORE RELAXED

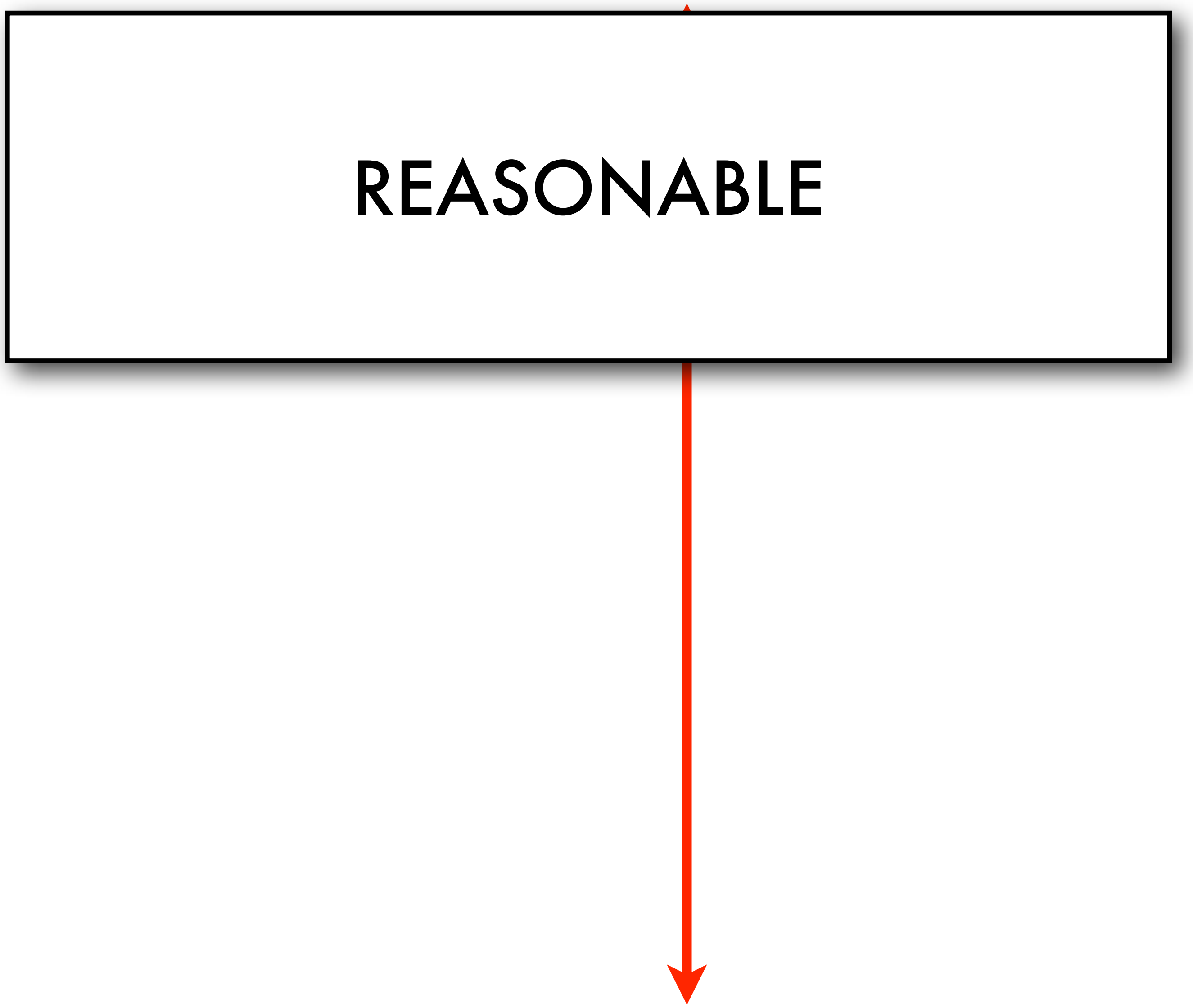
A second look at qualifiers

`MO_SEQ_CST`

`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

`MO_RELAXED`



LESS RELAXED

REASONABLE

MORE RELAXED

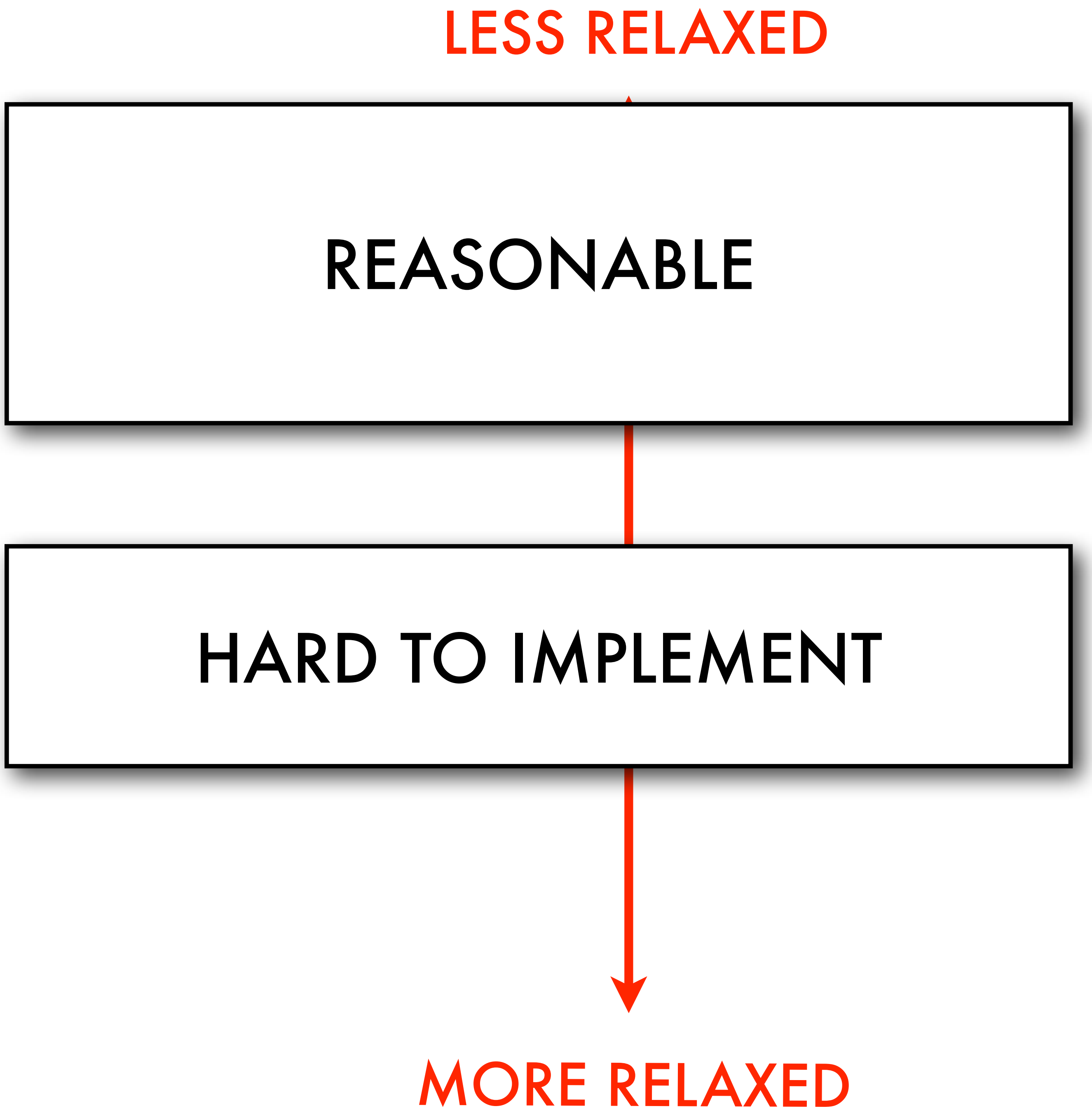
A second look at qualifiers

`MO_SEQ_CST`

`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

`MO_RELAXED`



LESS RELAXED

REASONABLE

HARD TO IMPLEMENT

MORE RELAXED

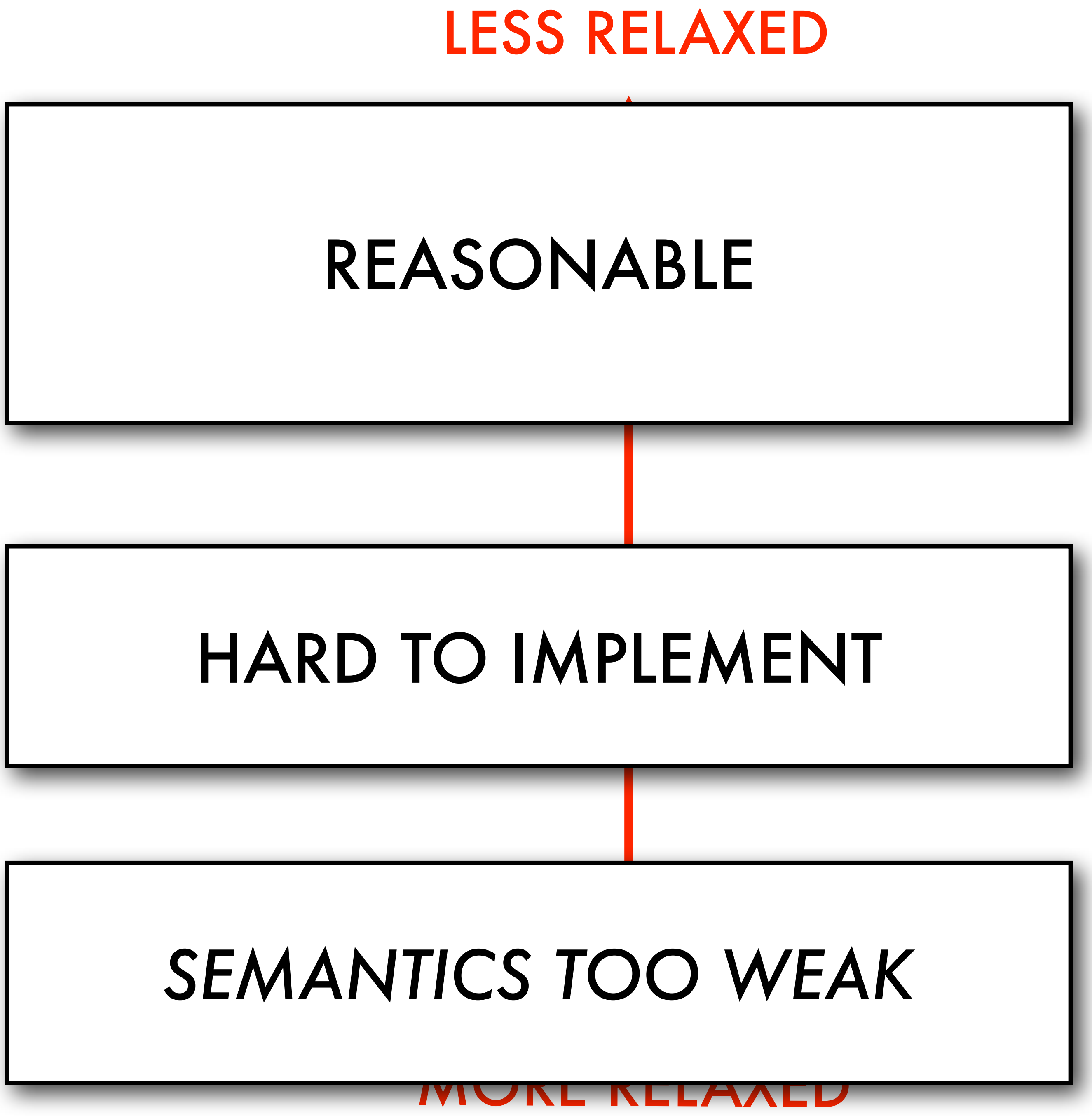
A second look at qualifiers

`MO_SEQ_CST`

`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

`MO_RELAXED`



LESS RELAXED

REASONABLE

HARD TO IMPLEMENT

SEMANTICS TOO WEAK

MORE RELAXED

Out of thin air reads

Shorthand

from now on, all the memory accesses are atomic with MO_RELAXED semantics

Relaxed atomics

```
x = y = 0
```

Thread 1

Thread 2

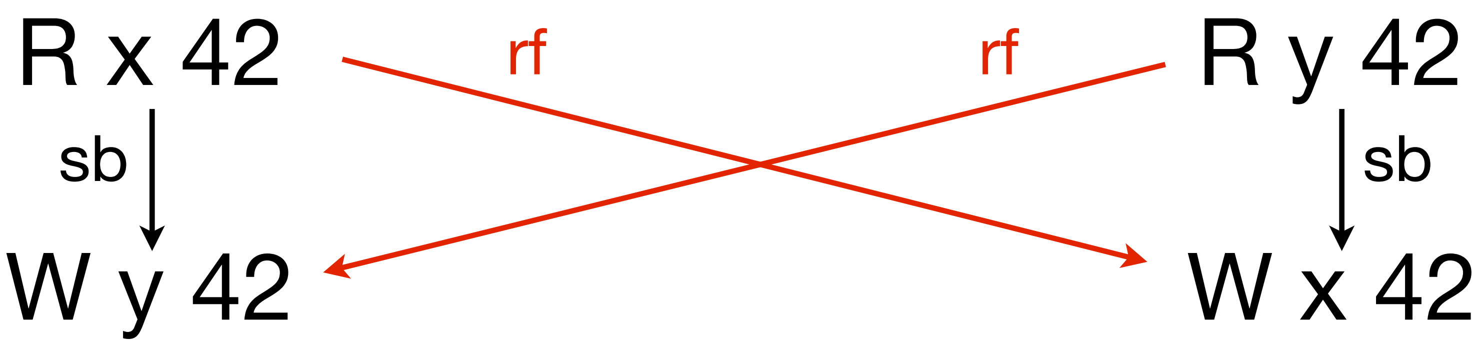
```
r1 = x        |    r2 = y
y = r1        |    x = 42
```

Relaxed atomics

```
                 x = y = 0
Thread 1                              Thread 2

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

r1 = r2 = 42

is a valid execution.



R x 42

sb

W y 42

rf

rf

R y 42

sb

W x 42

Out-of-thin-air reads

`x = y = 0`

Thread 1

```
r1 = x
y = r1
```

Thread 2

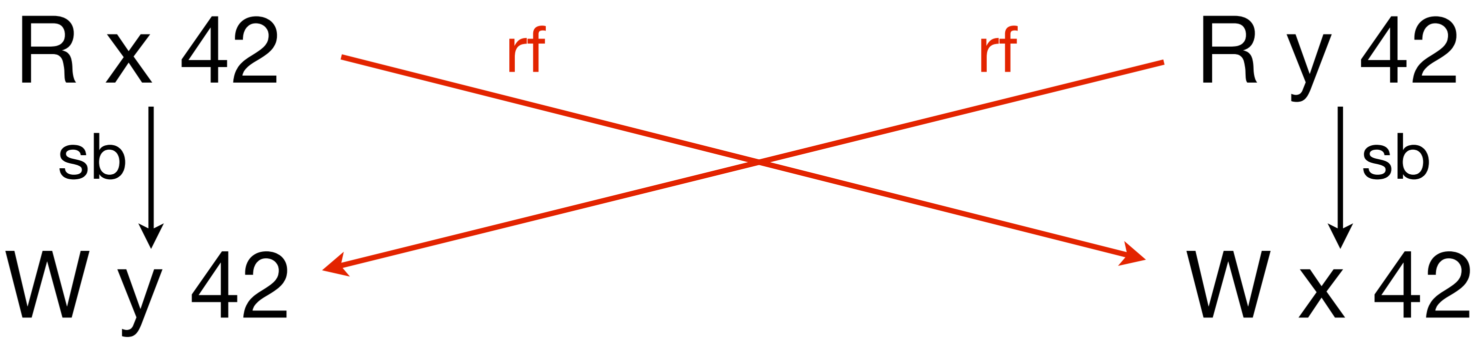
```
r2 = y
x = r2
```

Out-of-thin-air reads

```
                x = y = 0
Thread 1                          Thread 2

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

r1 = r2 = 42

is also an allowed execution

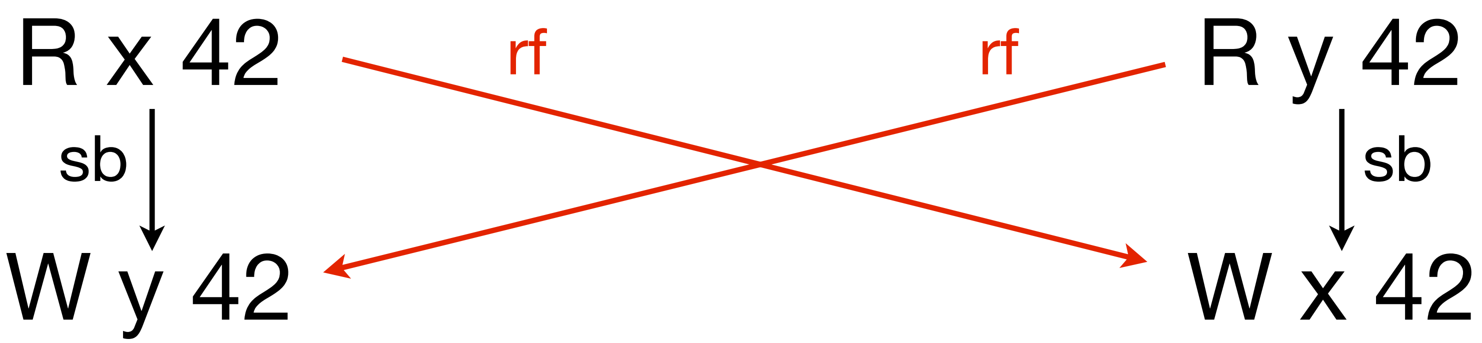


the value 42 appears *out-of-thin-air*

```
Thread 1                x = y = 0                Thread 2

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

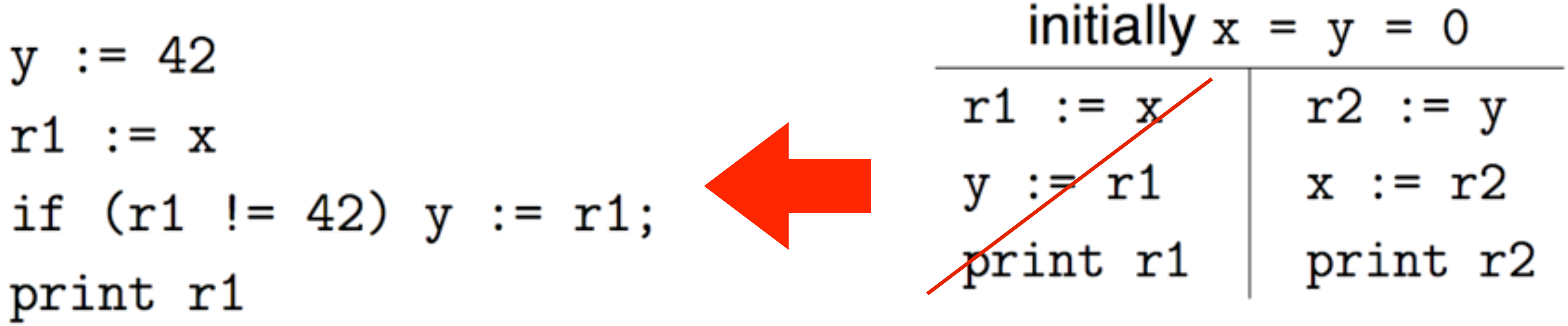
$$r1 = r2 = 42$$

is also an allowed execution



Speculation can justify out-of-thin-air reads

If the compiler states that x is likely to hold 42...



```
y := 42
r1 := x
if (r1 != 42) y := r1;
print r1
```

initially x = y = 0	
~~r1 := x~~	r2 := y
~~y := r1~~	x := r2
~~print r1~~	print r2

Speculation can justify out-of-thin-air reads

If the compiler states that x is likely to hold 42...

```
y := 42
r1 := x
if (r1 != 42) y := r1;
print r1
```

←

initially `x = y = 0`

~~`r1 := x`~~	`r2 := y`
~~`y := r1`~~	`x := r2`
~~`print r1`~~	`print r2`

It does not happen in practice...

(a big thank you to compiler and hardware developers)

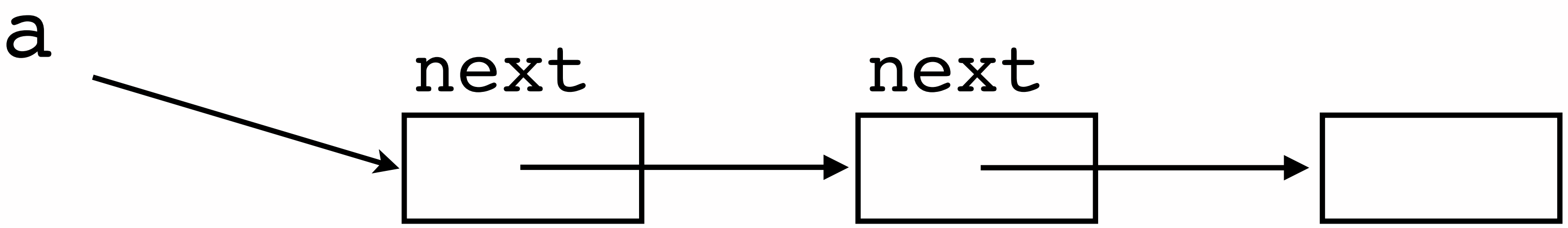
...but allowed by the standard

Consequences of out-of-thin-air reads

```
struct foo {
  atomic<struct foo *> next;
}
struct foo *a;
```

Thread 1

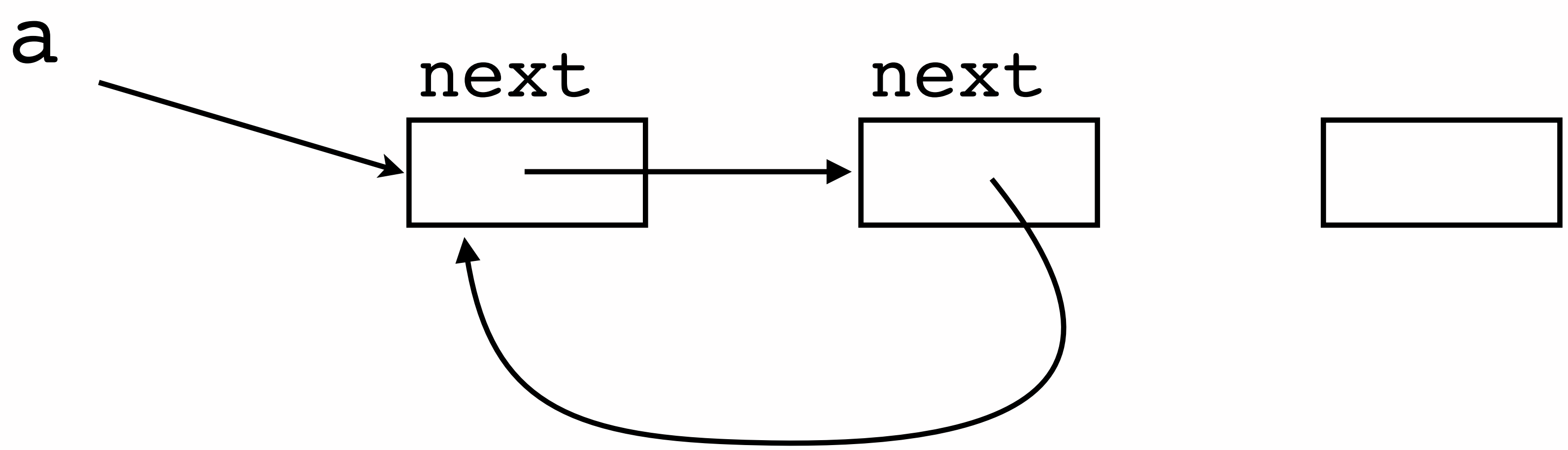
```
r1 = a->next
r1->next = a
```



```
struct foo {
  atomic<struct foo *> next;
}
struct foo *a;
```

Thread 1

```
r1 = a->next
r1->next = a
```



```
struct foo {
  atomic<struct foo *> next;
}
struct foo *a, *b;
```

Thread 1	Thread 2
`r1 = a->next`	`r2 = b->next`
`r1->next = a`	`r2->next = b`

```
struct foo {
  atomic<struct foo *> next;
}
struct foo *a, *b;
```

Thread 1

Thread 2

Thread 1	Thread 2
`r1 = a->next`	`r2 = b->next`
`r1->next = a`	`r2->next = b`

If `a` and `b` initially reference disjoint data-structures

we expect `a` and `b` to remain disjoint

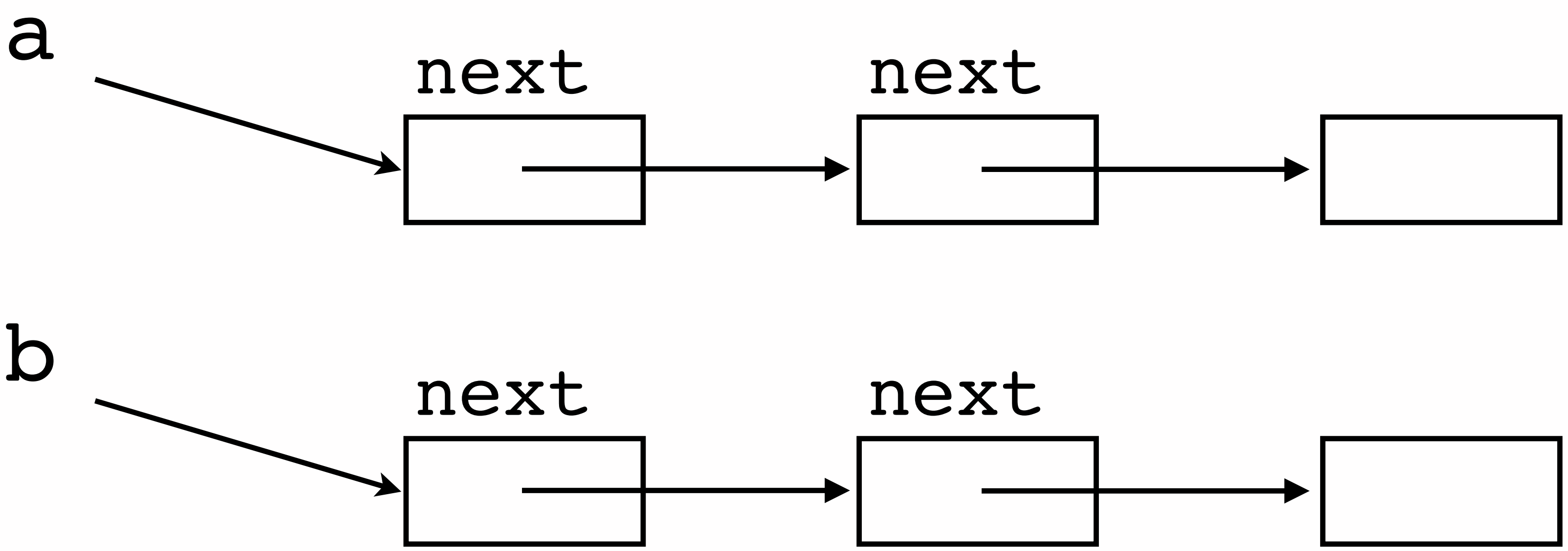
```
struct foo {
  atomic<struct foo *> next;
}
struct foo *a, *b;
```

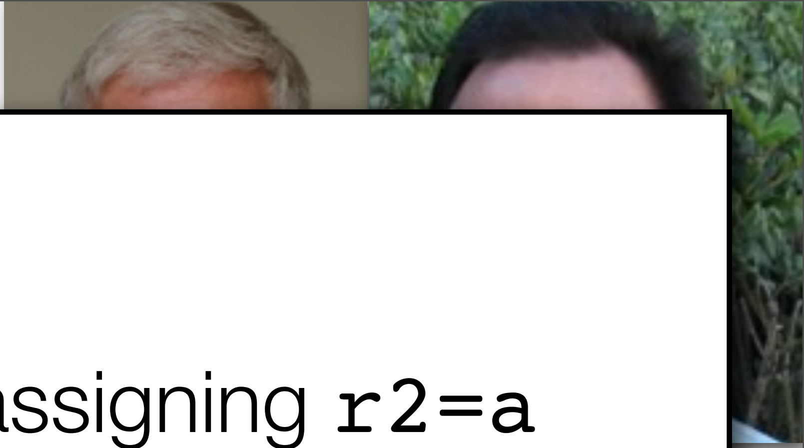
Thread 1

```
r1 = a->next
r1->next = a
```

Thread 2

```
r2 = b->next
r2->next = b
```



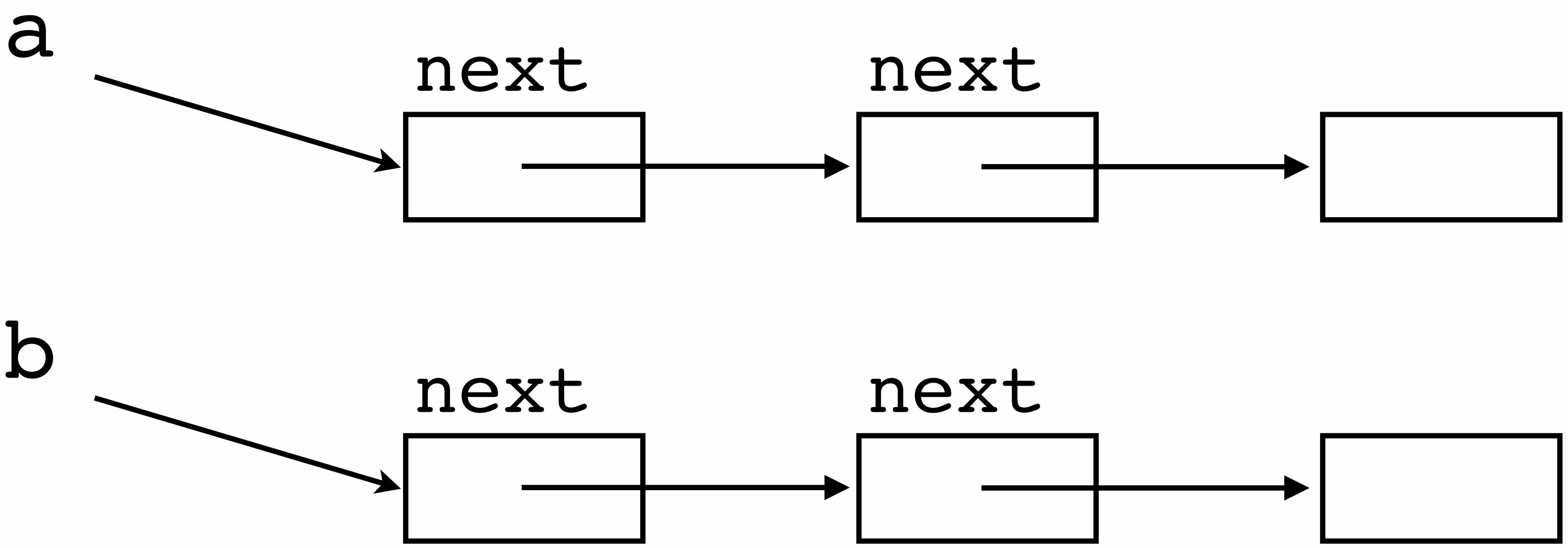


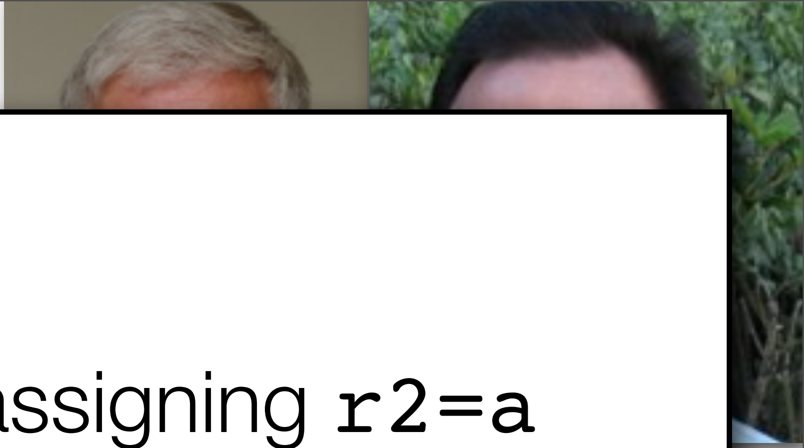
If the compiler speculates `r1=b` and `r2=a`, then

the store `r1->next=a` justifies `r2=b->next` assigning `r2=a`

(and symmetrically to justify `r1=b`)

Thread 1	Thread 2
`r1 = a->next`	`r2 = b->next`
`r1->next = a`	`r2->next = b`

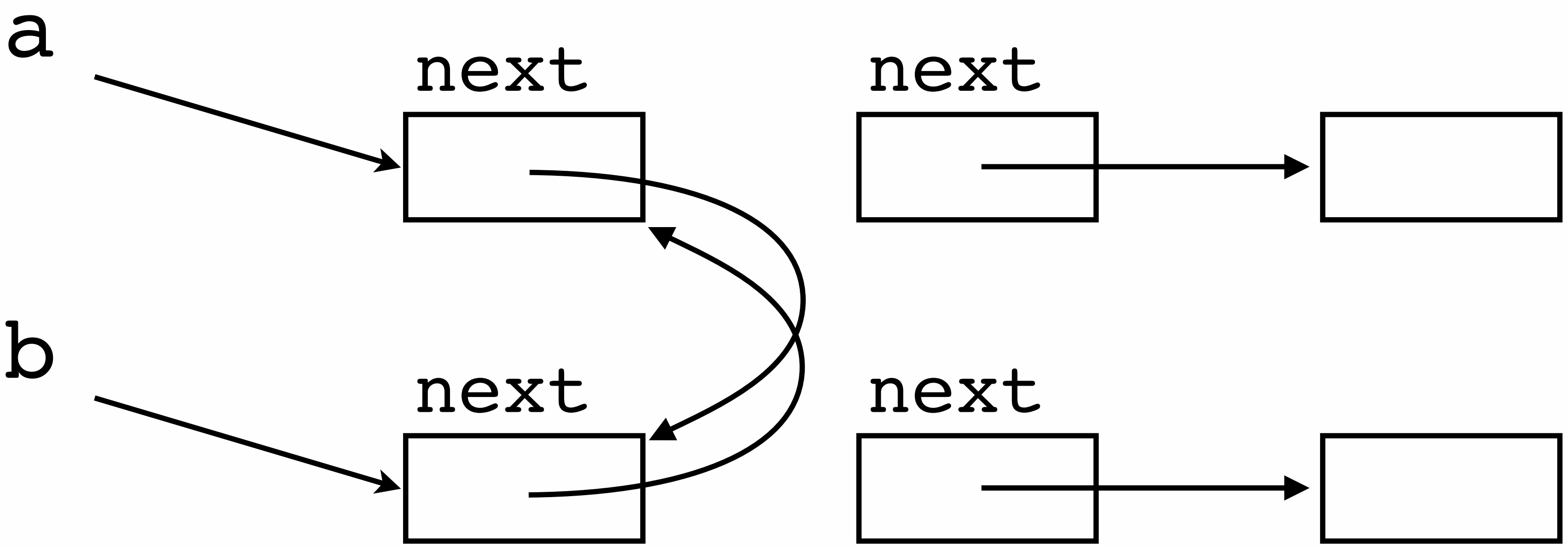


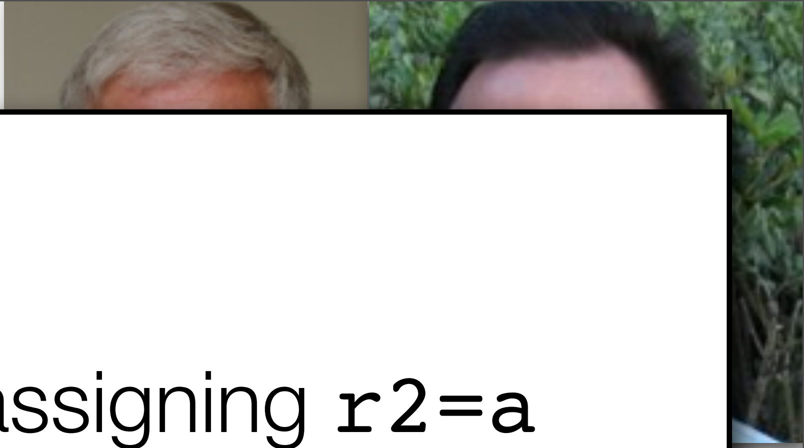


If the compiler speculates `r1=b` and `r2=a`, then
the store `r1->next=a` justifies `r2=b->next` assigning `r2=a`
(and symmetrically to justify `r1=b`)

Thread 1 | Thread 2

```
r1 = a->next          r2 = b->next
r1->next = a          r2->next = b
```



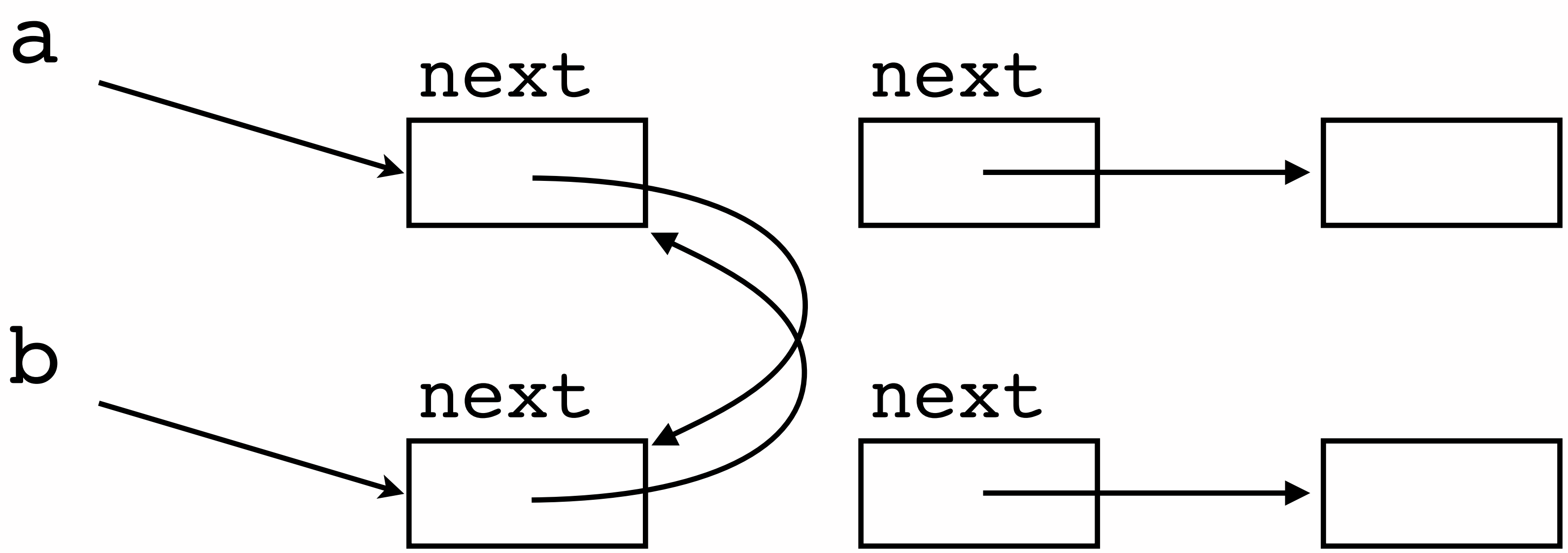


If the compiler speculates `r1=b` and `r2=a`, then

the store `r1->next=a` justifies `r2=b->next` assigning `r2=a`

(and symmetrically to justify `r1=b`)

Break our basic intuitions about memory and sharing!



```
                     x = y = a = 0

if (x.load(rlx)==42)| if (y.load(rlx)==42) | a = 1
   y.write(42,rlx)  |   if (a==1)          |
                    |     x.write(42,rlx)  |
```

```
x = y = a = 0

if (x.load(rlx)==42)| if (y.load(rlx)==42) | a = 1
   y.write(42,rlx)  |    if (a==1)         |
                    |      x.write(42,rlx) |
```

Remark 1

This code is not racy!

There is no consistent execution in which the read of `a` occurs.

```
                        x = y = a = 0

if (x.load(rlx)==42) | if (y.load(rlx)==42)  | a = 1
   y.write(42,rlx)   |    if (a==1)          |
                     |       x.write(42,rlx) |
```

Remark 2

$$\texttt{a} = 1 \wedge \texttt{x} = \texttt{y} = 0$$

is the only possible final state

```
                   x = y = a = 0

if (x.load(rlx)==42) | if (y.load(rlx)==42)   | a = 1
   y.write(42,rlx)   |    if (a==1)           |
                     |      x.write(42,rlx)   |
```

Consider sequentialisation:

$$C \,||\, D \implies C\,;\,D$$

(ought to be correct)

```
                    x = y = a = 0

if (x.load(rlx)==42) | if (y.load(rlx)==42) | a = 1
   y.write(42,rlx)   |    if (a==1)          |
                     |      x.write(42,rlx)  |
```

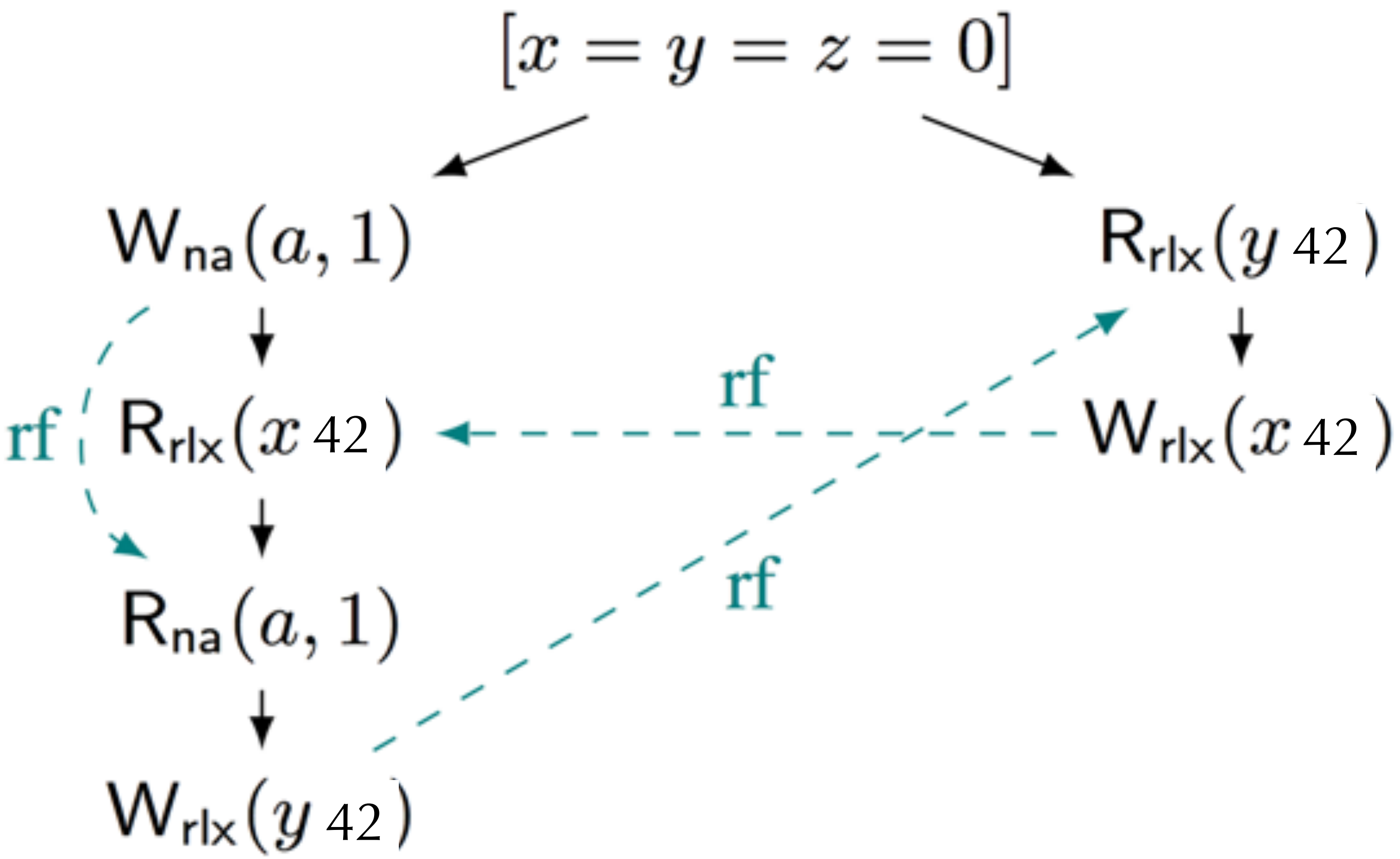
↓

```
                     | a = 1
                     |
if (x.load(rlx)==42) | if (y.load(rlx)==42)
   y.write(42,rlx)   |    if (a==1)
                     |      x.write(42,rlx)
```

```
x = y = a = 0
```

```
                         | a = 1
if (x.load(rlx)==42)     | if (y.load(rlx)==42)
   y.write(42,rlx)       |    if (a==1)
                         |      x.write(42,rlx)
```

```
                   x = y = a = 0

                              | a = 1
if (x.load(rlx)==42)          | if (y.load(rlx)==42)
   y.write(42,rlx)            |    if (a==1)
                              |      x.write(42,rlx)
```



$a = 1$

$x = y = 42$

is also possible

```
                 x = y = a = 0
                     | a = 1
if (x.load(rlx)==42) | if (y.load(rlx)==42)
   y.write(42,rlx)   |    if (a==1)
                     |      x.write(42,rlx)
```

Break common source-to-source (or LLVM IR - to - LLVM IR) compiler optimisations

including *expression linearisation* and *roach-motel reorderings*

We still lack a really satisfactory proposal for the semantics of a general-purpose shared-memory concurrent programming language.

The way forward

Understand the effects of what compilers implement and programmers rely on

Build on that...

Beyond concurrency

Can one do < comparison or pointer arithmetic between pointers to separately allocated objects?

Routinely done in Linux kernel

Forbidden by ISO standard

tinyurl.com/csurvey2

A web survey of 15 questions to investigate what C is in current practice: what behaviour is implemented by mainstream compilers and relied on by systems programmers

tinyurl.com/csurvey2

Eventual outcome: clear descriptions of what people can rely on and what compilers *in practice* should implement, what alias analysis and optimisation passes should (and should not) be allowed to do, etc.

tinyurl.com/csurvey2

Thank you.
Questions?