

Semantics, languages and algorithms for multicore programming

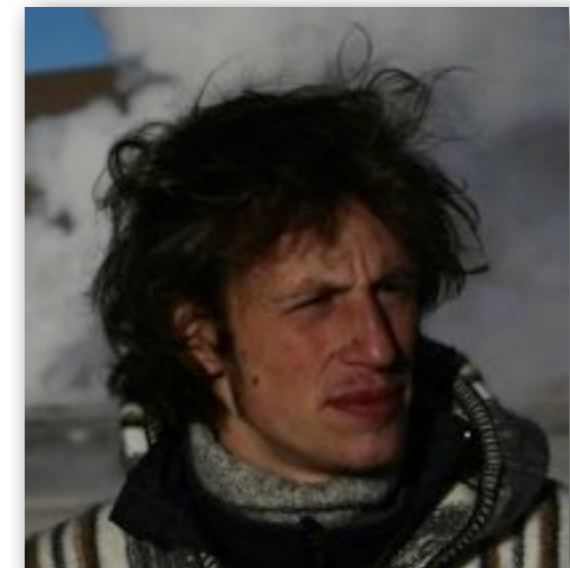
Albert Cohen



Luc Maranget



Francesco Zappa Nardelli



Vote: topics for my next lecture

1. The lwarx and stwcx Power instructions 1
- 2. Hunting compiler concurrency bugs 8**
3. Operational and axiomatic formalisation of x86-TSO 2
4. Fence optimisations for x86-TSO 1
5. The Java memory model 1
- 6. The C11/C++11 memory model 7**
- 7. Static and dynamic techniques for data-race detection 6**
8. What about the Linux kernel 3



1. *A word on techniques for data-race detection*

Recall: Data-race freedom

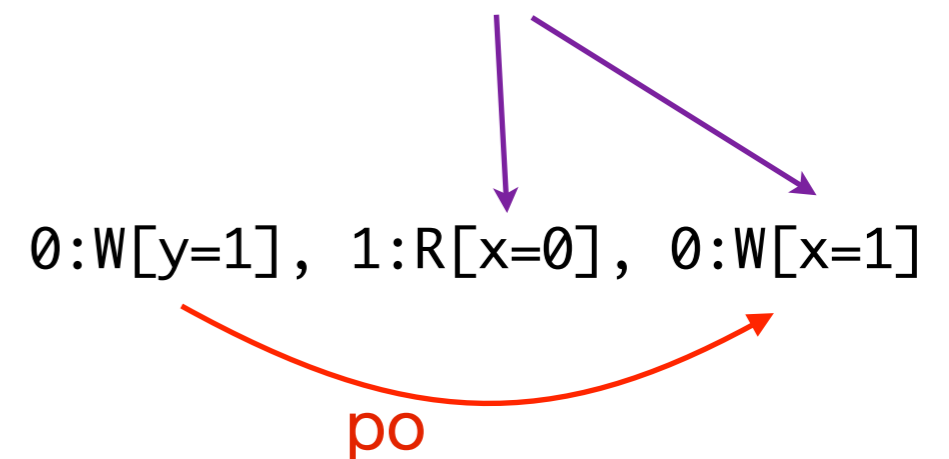
Definition [data-race-freedom]: A program (traceset) is **data-race free** if none of its executions has two adjacent conflicting actions from different threads.

Equivalently, a program is data-race free if in all its executions all pairs of conflicting actions are ordered by happens-before.

A racy program

Thread 0	Thread 1
<code>*y = 1</code>	<code>if *x == 1</code>
<code>*x = 1</code>	<code>then print *y</code>

Two conflicting accesses
not related by happens before.



Recall: Happens-before

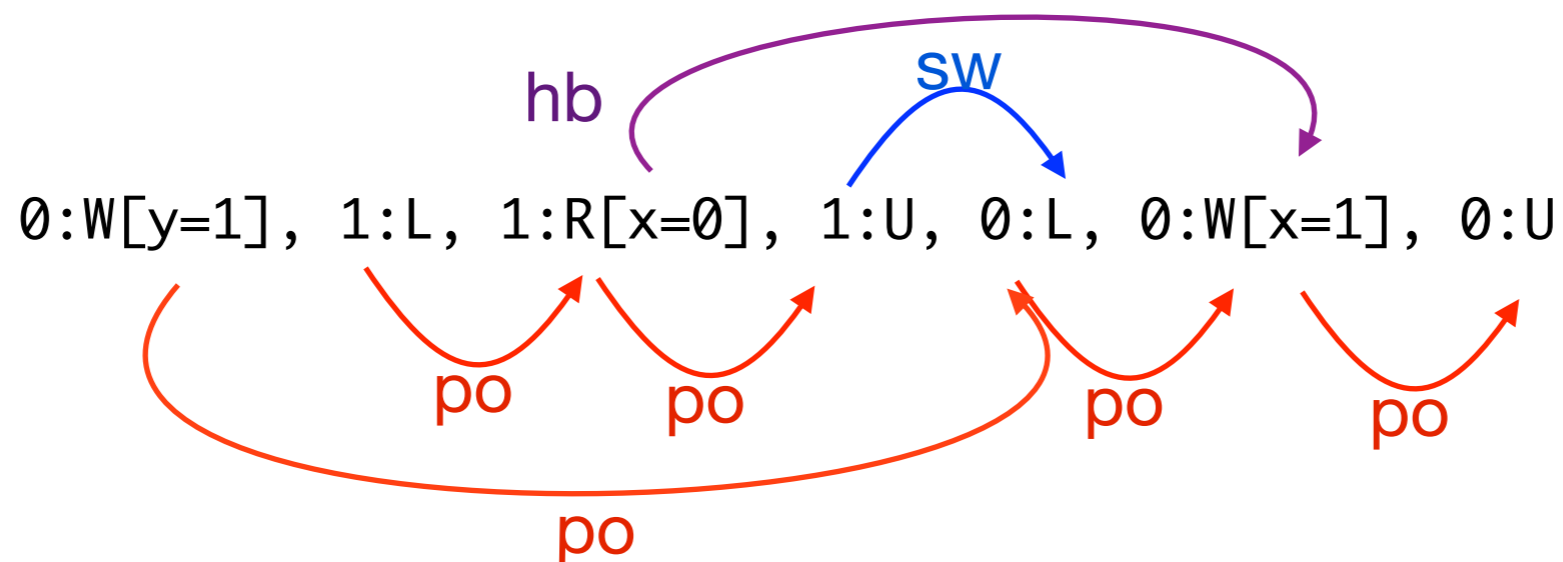
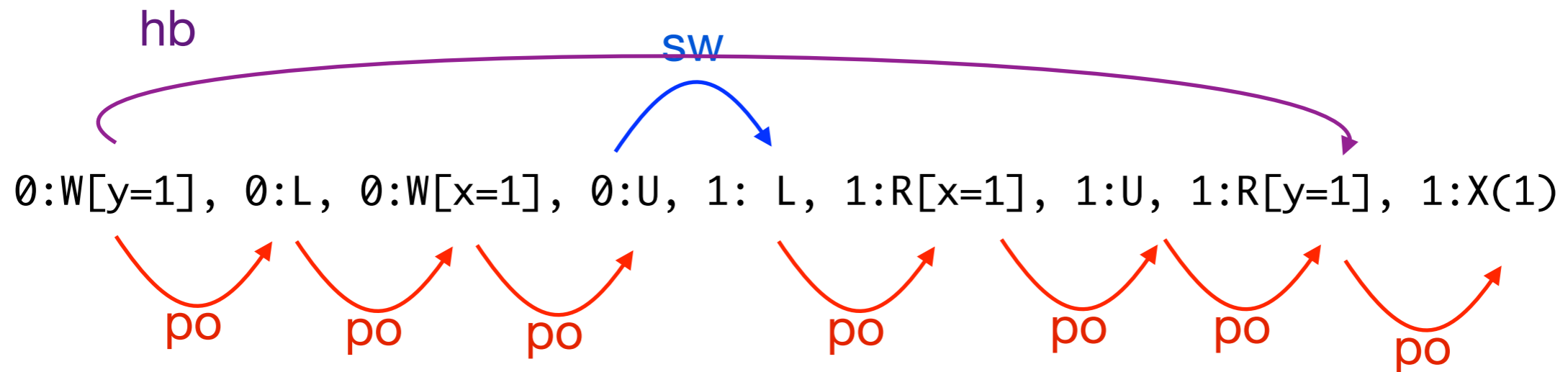
Definition [program order]: **program order**, $<_{po}$, is a total order over the actions of *the same thread in an interleaving*.

Definition [synchronises with]: in an interleaving I , index i **synchronises-with** index j , $i <_{sw} j$, if $i < j$ and $A(I_i) = U$ (unlock), $A(I_j) = L$ (lock).

Definition [happens-before]: **Happens-before** is the transitive closure of program order and synchronises with.

Examples of happens before

Thread 0	Thread 1
<code>*y = 1</code> <code>lock();</code> <code>*x = 1</code> <code>unlock();</code>	<code>lock();</code> <code>tmp = *x;</code> <code>unlock();</code> <code>if tmp = 1</code> <code>then print *y</code>



S(tid) actions omitted.

Data race detection: dynamic approaches

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 two conflicting
accesses are not related by hb

no false positives

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)		

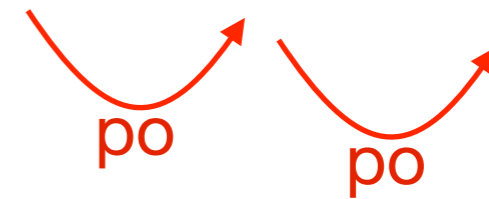
1:L(b);1:L(a);1:Wx1;1:U(a);2:L(c);2:Wx2;2:U(c)

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

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

lockset vs happens-before

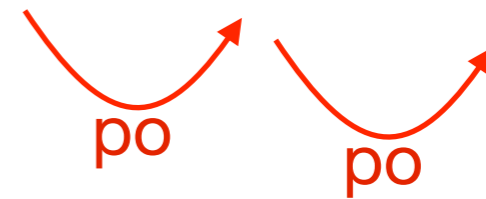
<code>y=1</code>		<code>lock(a)</code>
<code>lock(a)</code>		<code>x=2</code>
<code>x=1</code>		<code>unlock(a)</code>
<code>unlock(a)</code>		<code>y=2</code>



lockset vs happens-before

<code>y=1</code>		<code>lock(a)</code>
<code>lock(a)</code>		<code>x=2</code>
<code>x=1</code>		<code>unlock(a)</code>
<code>unlock(a)</code>		<code>y=2</code>

This program has a race on y

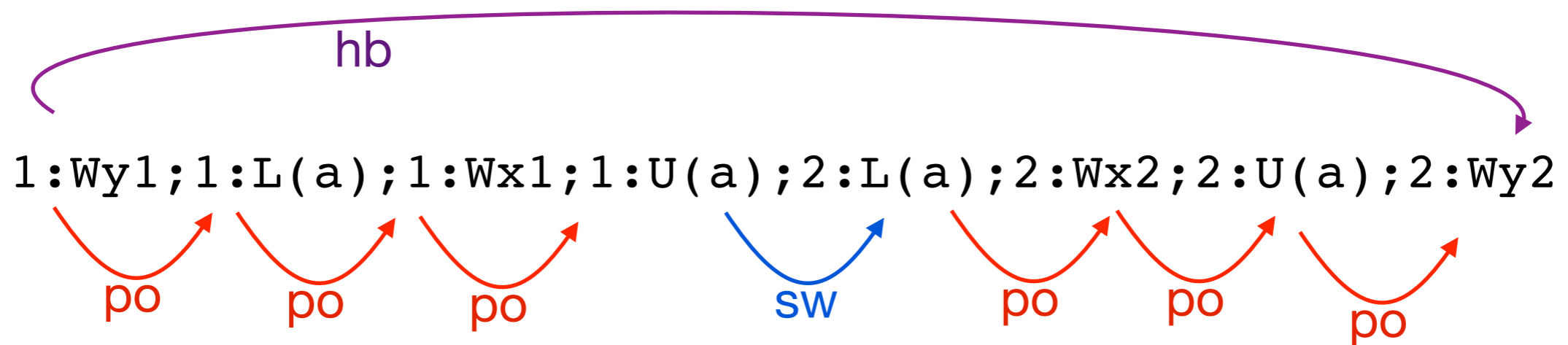


lockset vs happens-before

<code>y=1</code>		<code>lock(a)</code>
<code>lock(a)</code>		<code>x=2</code>
<code>x=1</code>		<code>unlock(a)</code>
<code>unlock(a)</code>		<code>y=2</code>

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

lockset vs happens-before (2)

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

This program instead is DRF.

lockset vs happens-before (2)

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

lock(a)
tmp=x
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 of 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

Current state of the art:

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

Helgrind, RaceFuzzer, ThreadSanitizer...

Impressive:

*tolerable slowdown on large applications
found thousands races*

Still not as reliable as the tool we dream of. Active area of research!

97

Data race detection: static approaches

Run a bunch of static analysis for inferring locksets.

Hard:

- aliasing on memory locations
- lock pointers
- must account all language features

Also done via fancy effect type-systems.

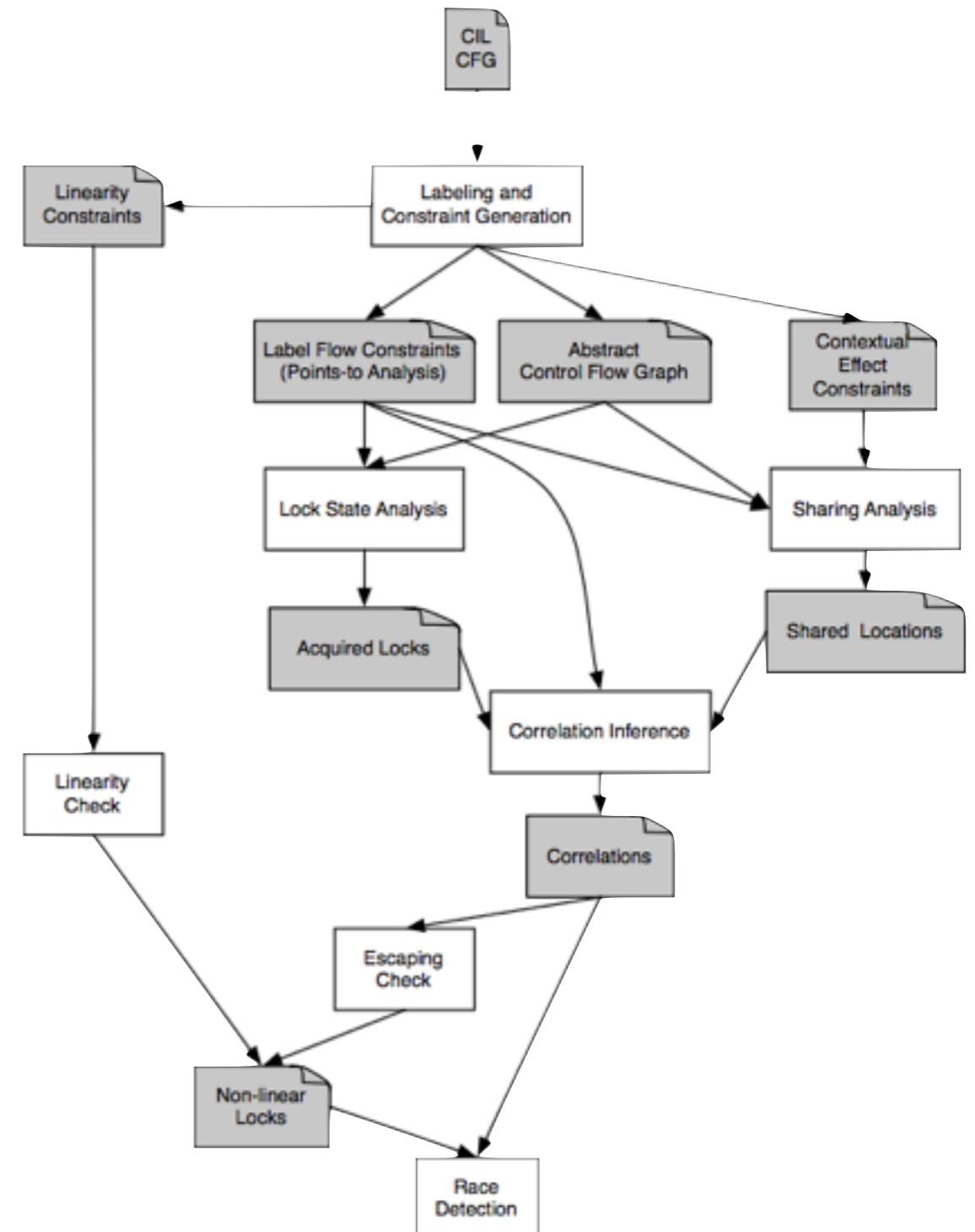


Fig. 1. LOCKSMITH architecture



2. The C++11 memory model

a good example of an axiomatic memory model



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.

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);
```



...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)  
        ;  
}
```

gcc 4.7 -O2



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

```
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
```

gcc 4.7 -O2



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

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
```


gcc 4.7 -O2



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

```
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
```

gcc 4.7 -O2



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

```
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
```

gcc 4.7 -O2



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

```
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
```

gcc 4.7 -O2



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

```
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
```

gcc 4.7 -O2



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

```
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
```

The compiled code saves and restores b

Correct in a sequential setting.

What about concurrency?

```
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 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 $X_{witness}$

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

3) searches the consistent executions for races and unconstrained 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.

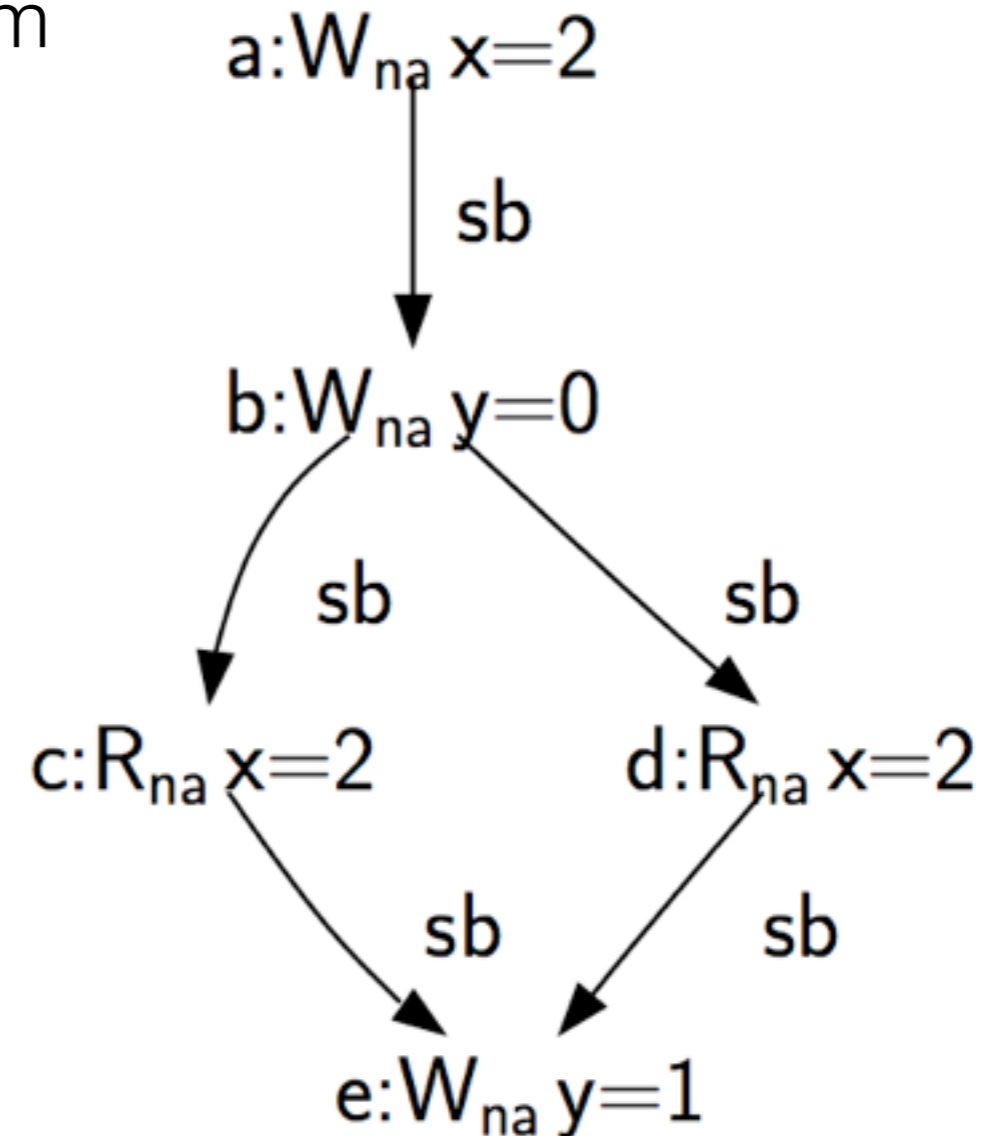
Formally

```
cpp_memory_model_opsem (p : program) =  
  let pre_executions =  
    { (Eopsem, Xwitness) . opsem p Eopsem ∧  
      consistent_execution (Eopsem, Xwitness) }  
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

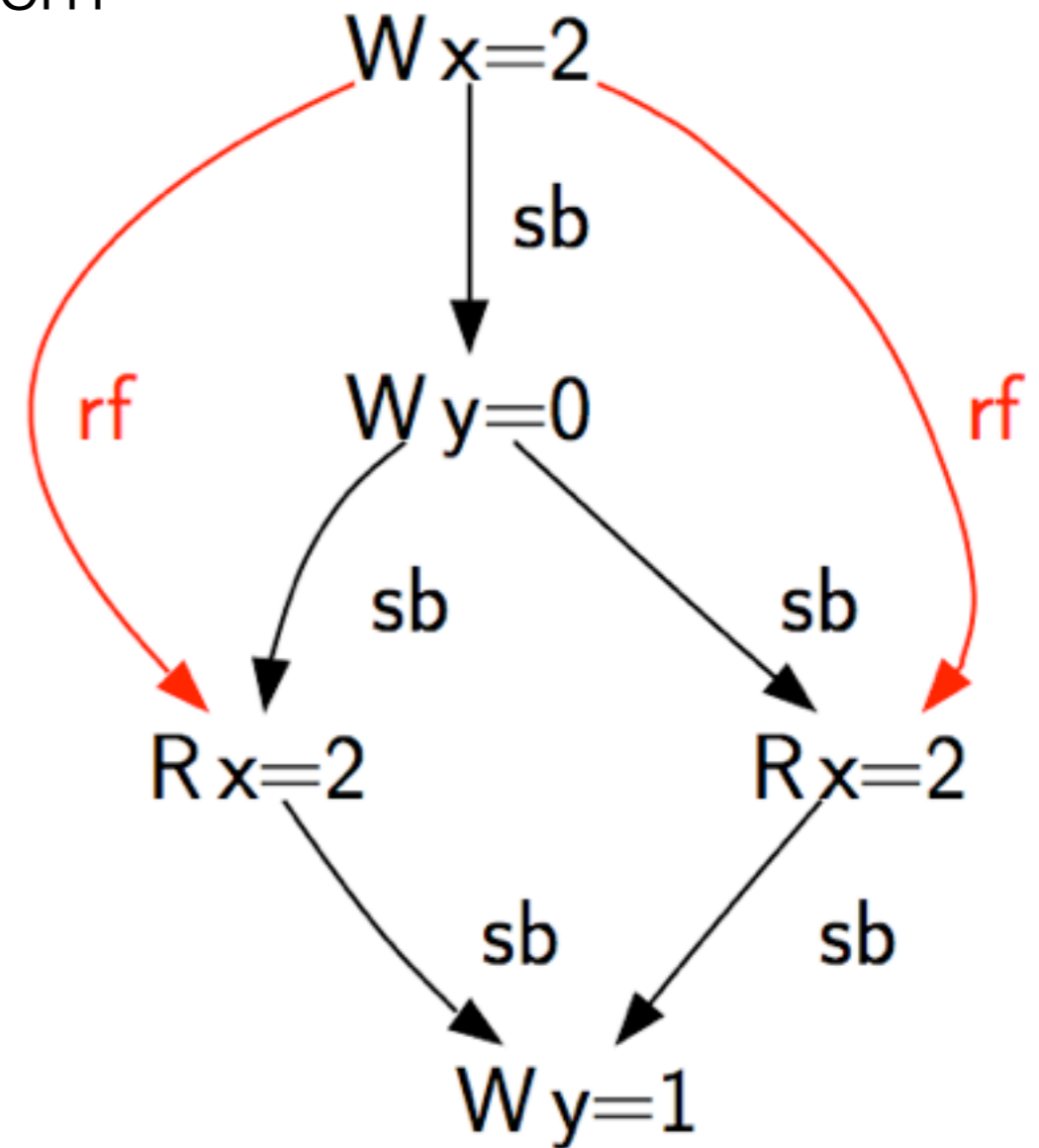
```
int main() {  
    int x = 2;  
    int y = 0;  
    y = (x==x);  
    return 0;  
}
```



A single-threaded example

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

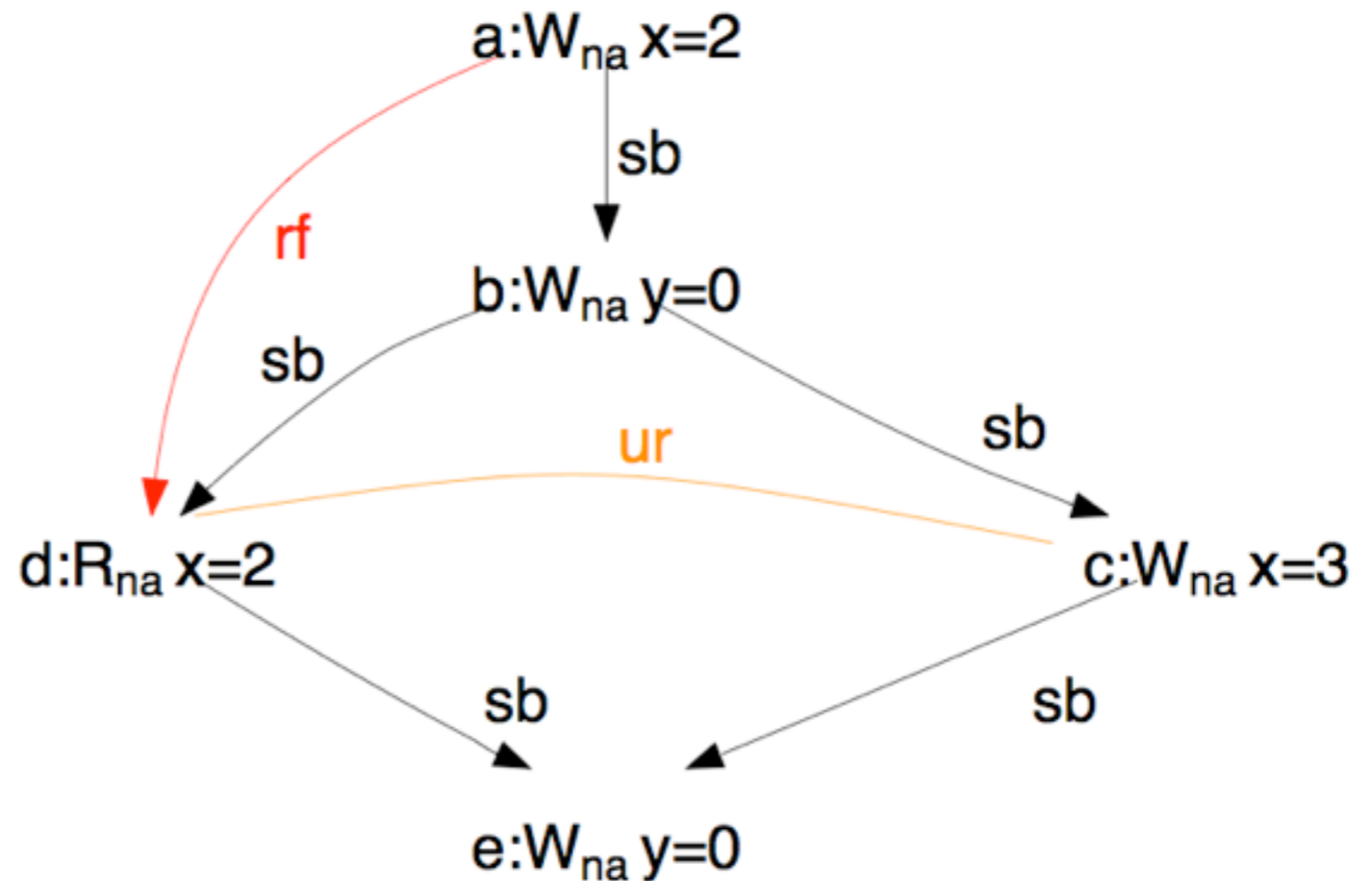
```
int main() {  
    int x = 2;  
    int y = 0;  
    y = (x==x);  
    return 0;  
}
```



A single-threaded ex. with undefined behaviour

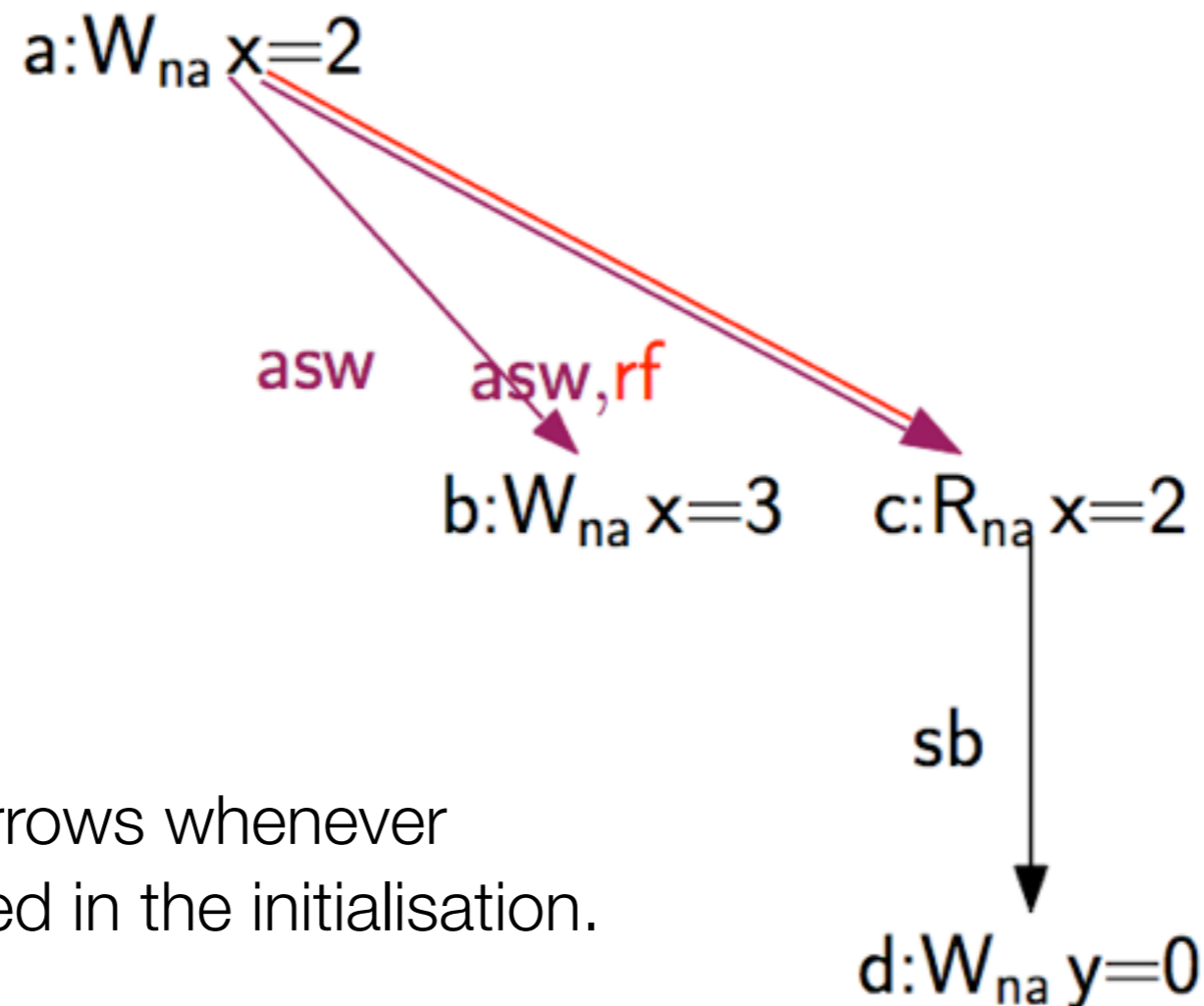
An unsequenced race.

```
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);
```



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

c:L mutex
sb ↓
d:W_{na} x=1
sb ↓
f:U mutex

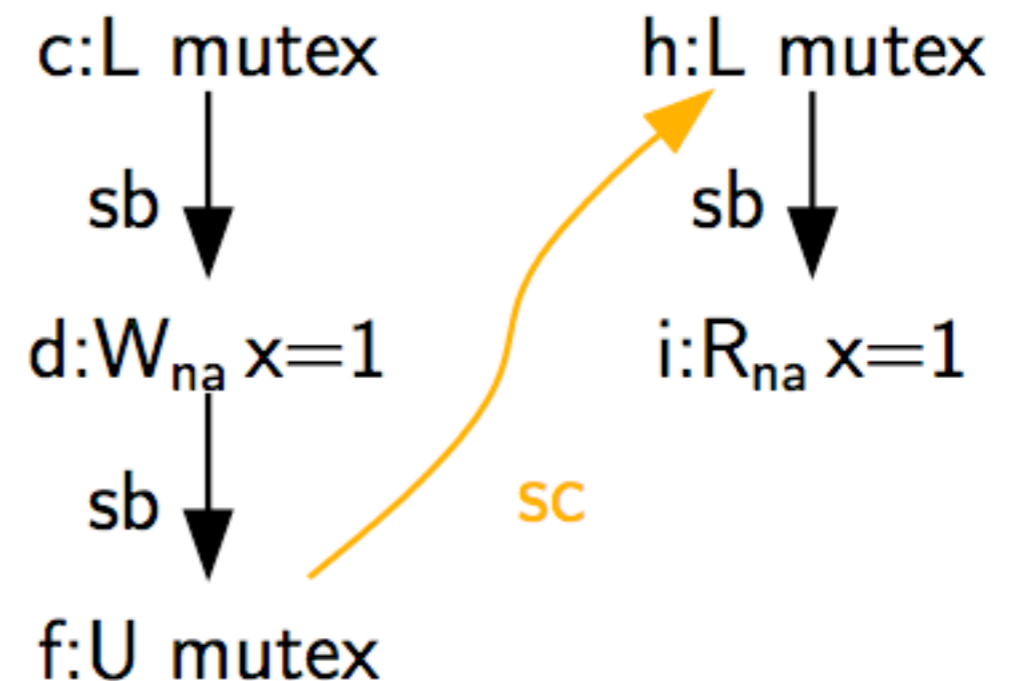
h:L mutex
sb ↓
i:R_{na} x=1

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)

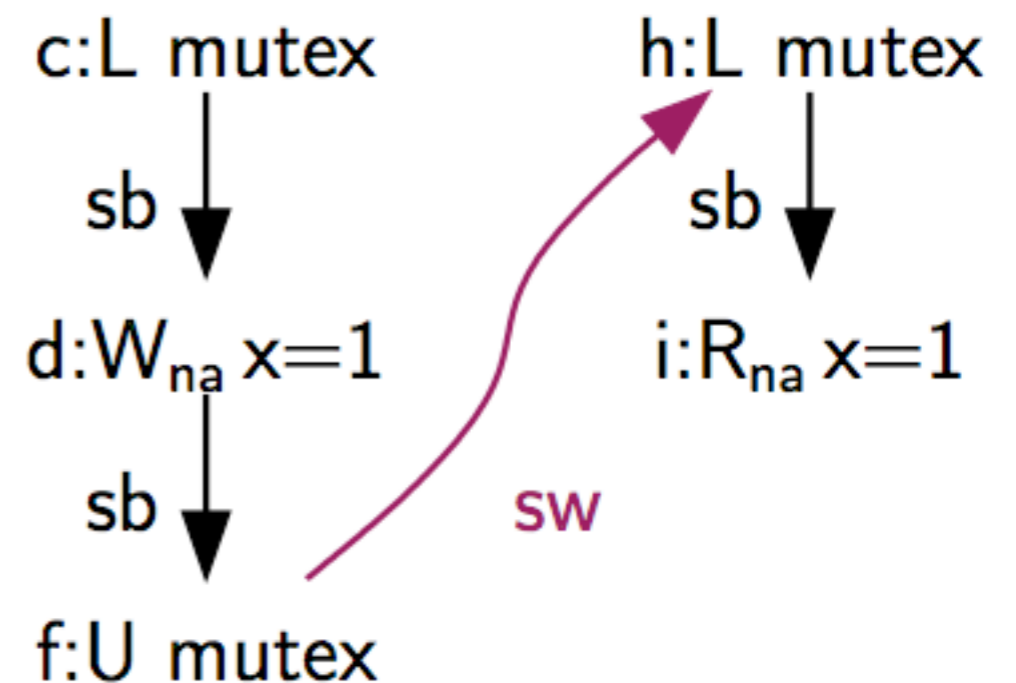


Locks and unlocks

```
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 synchronised-with relation



Locks and unlocks

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

```
int x, r;
```

```
mutex m;
```

```
m.lock();
```

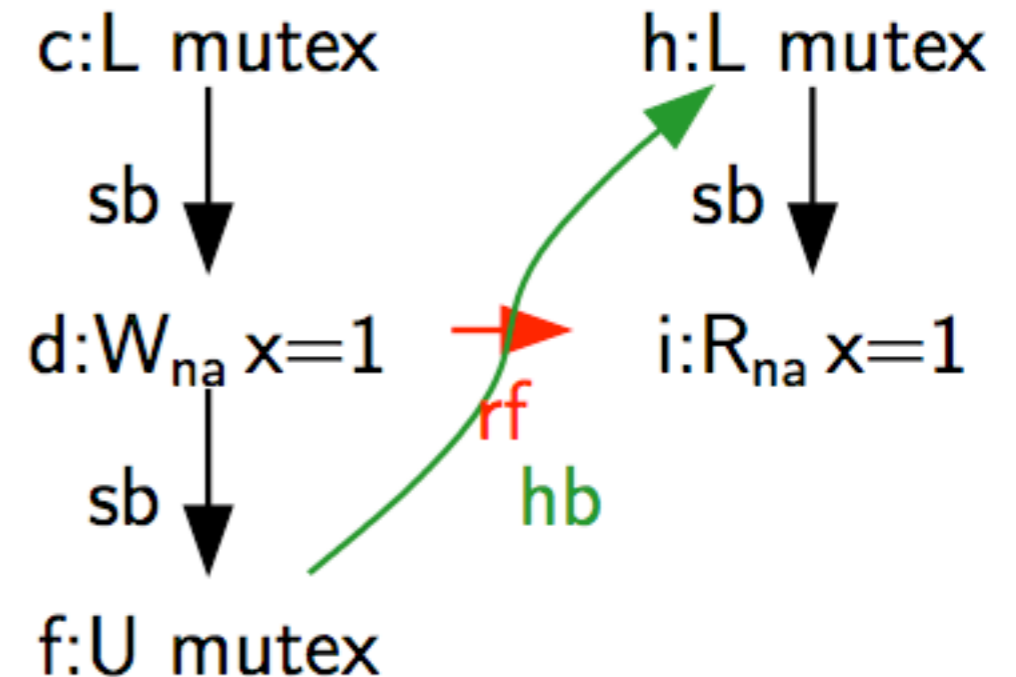
```
x = ...
```

```
m.unlock();
```

```
m.lock();
```

```
r = x;
```

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 synchronised-with relation
4. which in turn defines the happens-before relation...

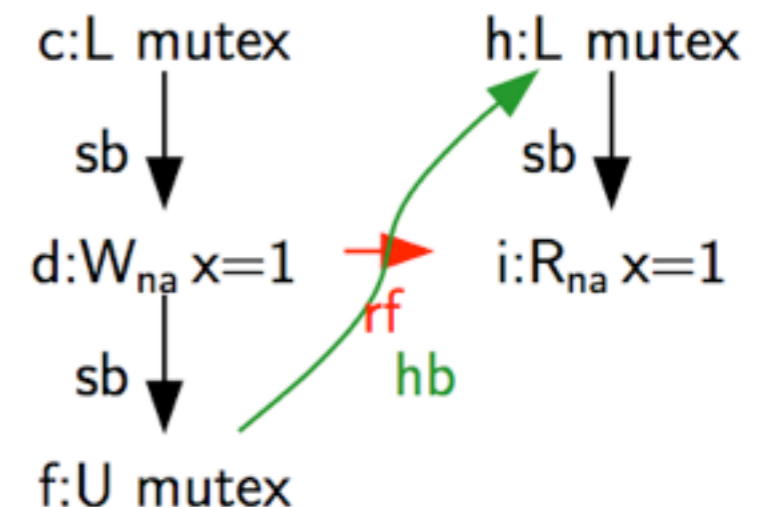


Happens before

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

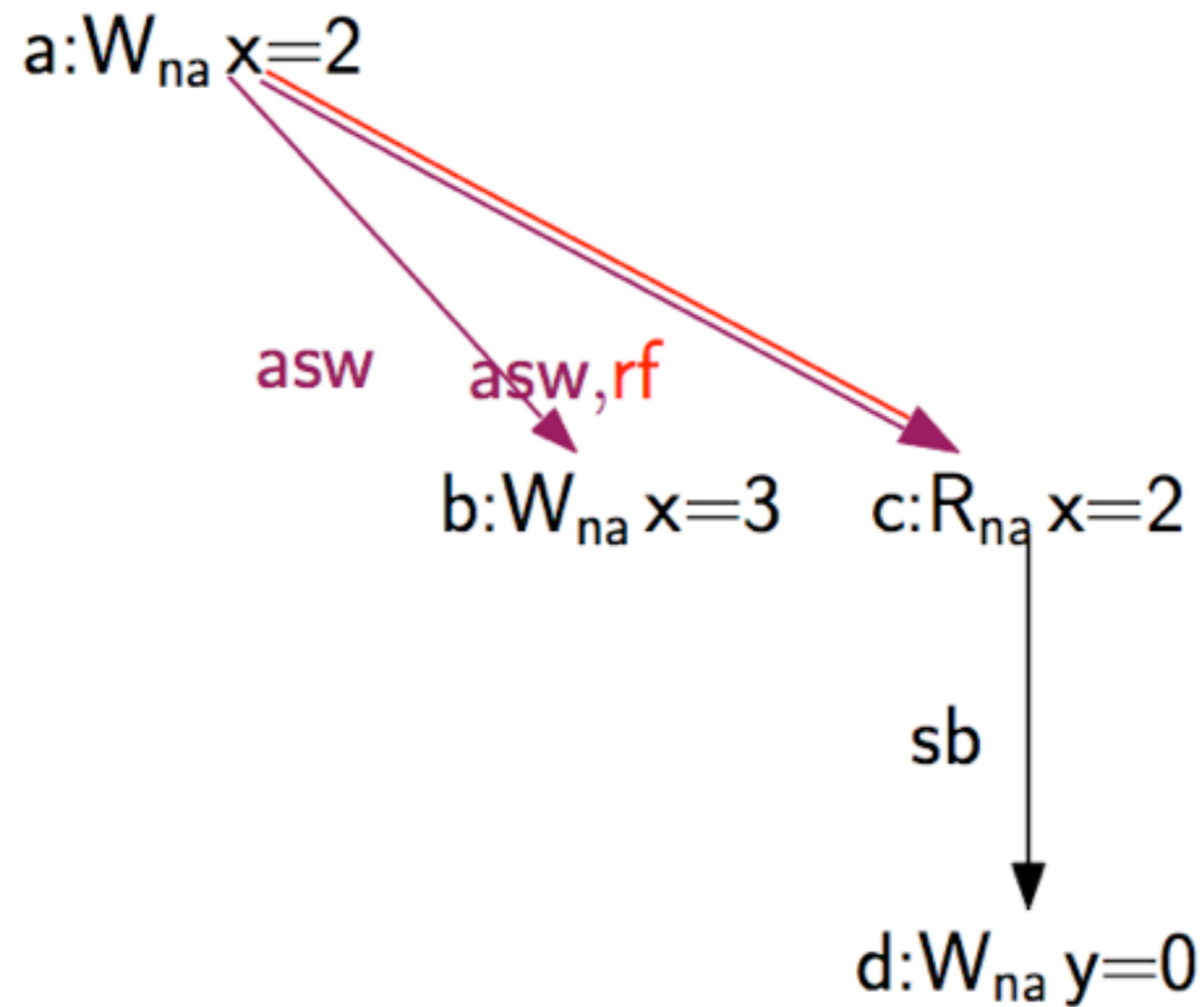
1. non-atomic loads read the most recent write in happens before.
(This is unique in DRF programs)
2. the story is more complex for atomics, as we shall see.
3. data races are defined as an absence of happens before between conflicting actions.

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



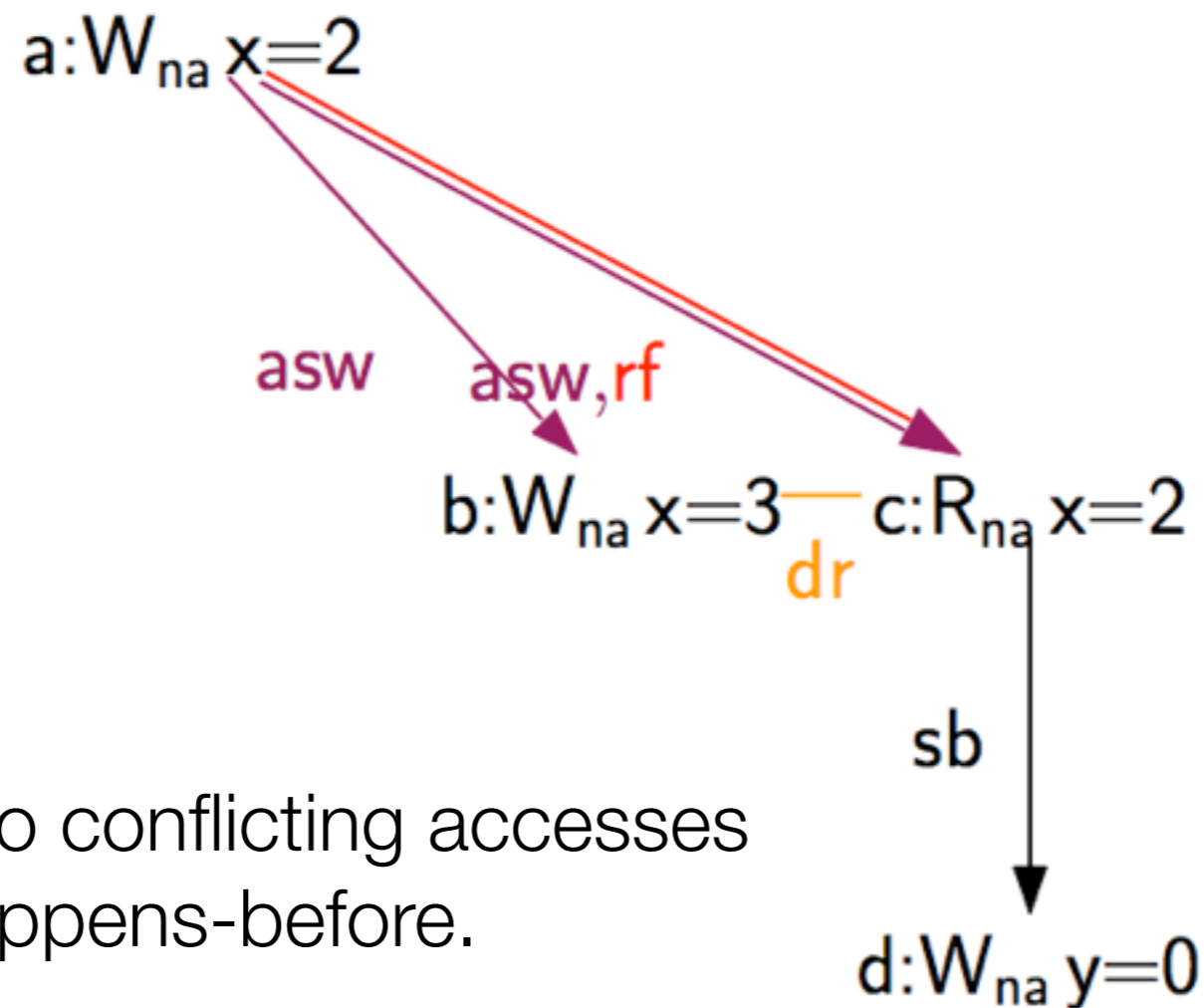
A data race

```
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

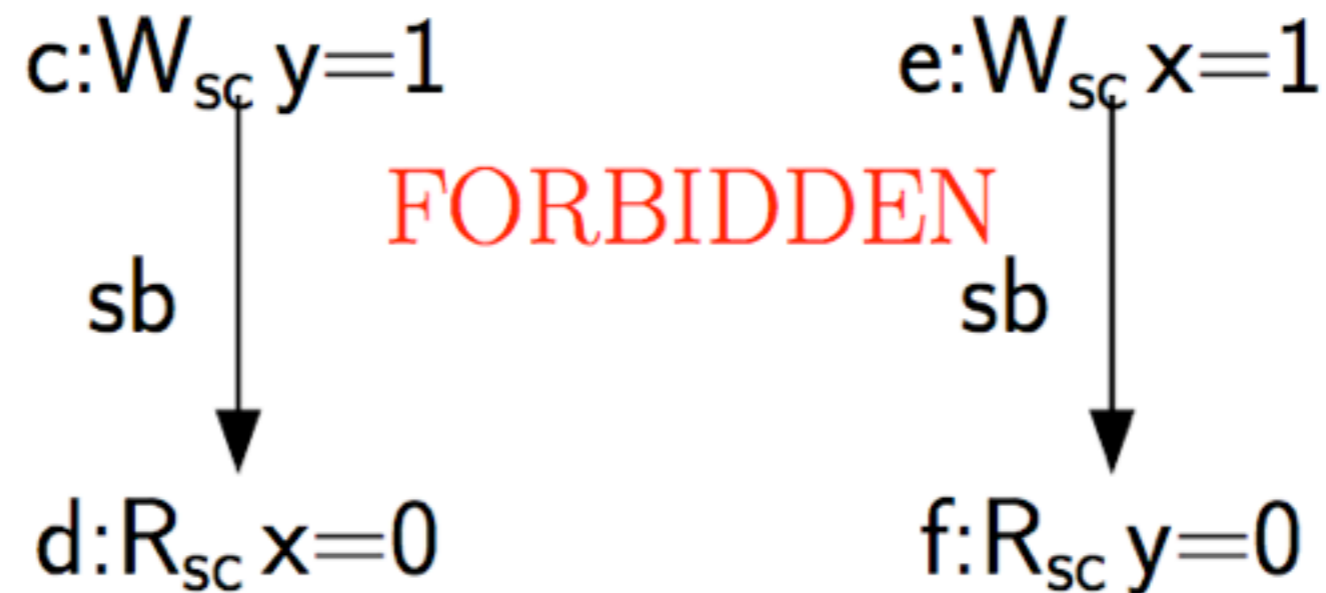
let $data_races\ actions\ hb =$
 $\{ (a, b) \mid \forall a \in actions\ b \in actions \mid$
 $\neg (a = b) \wedge$
 $same_location\ a\ b \wedge$
 $(is_write\ a \vee is_write\ b) \wedge$
 $\neg (same_thread\ a\ b) \wedge$
 $\neg (is_atomic_action\ a \wedge is_atomic_action\ b) \wedge$
 $\neg ((a, b) \in hb \vee (b, a) \in hb) \}$

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);
```

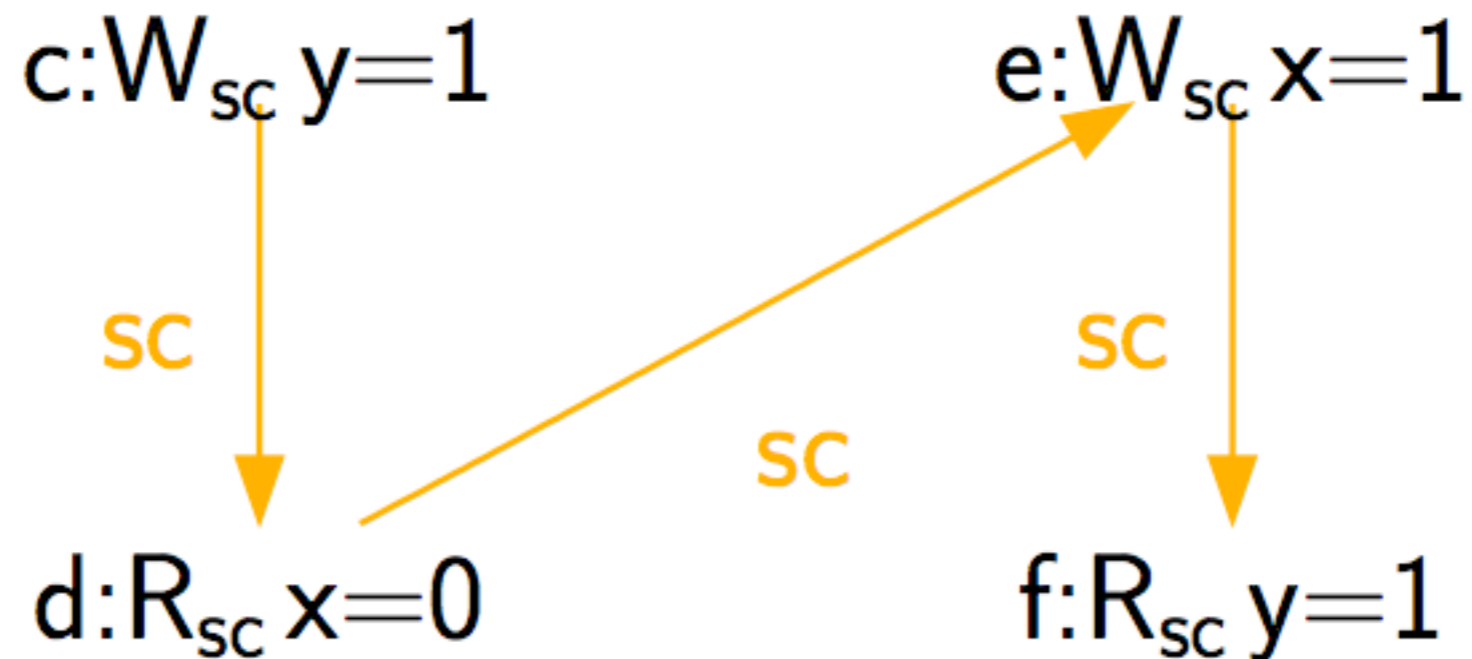


Why is this behaviour forbidden?

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);
```



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

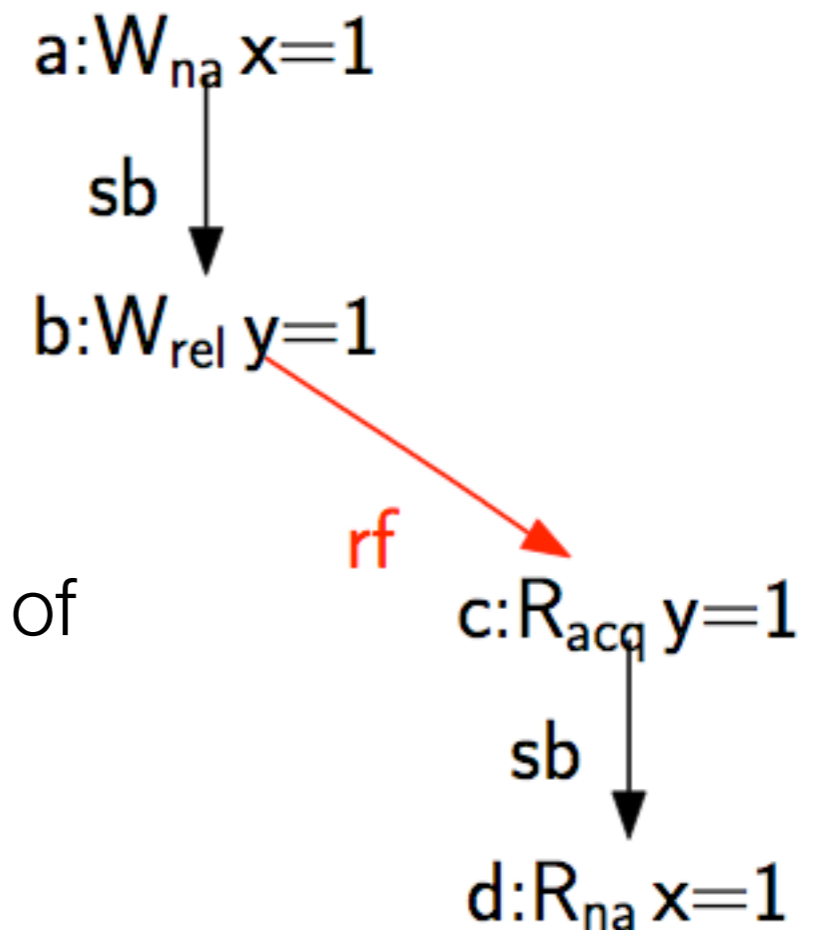
```
// sender
```

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

```
// receiver
```

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

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



Expert concurrency: the release-acquire idiom

```
// sender
```

```
x = ...
```

```
y.store(1, release);
```

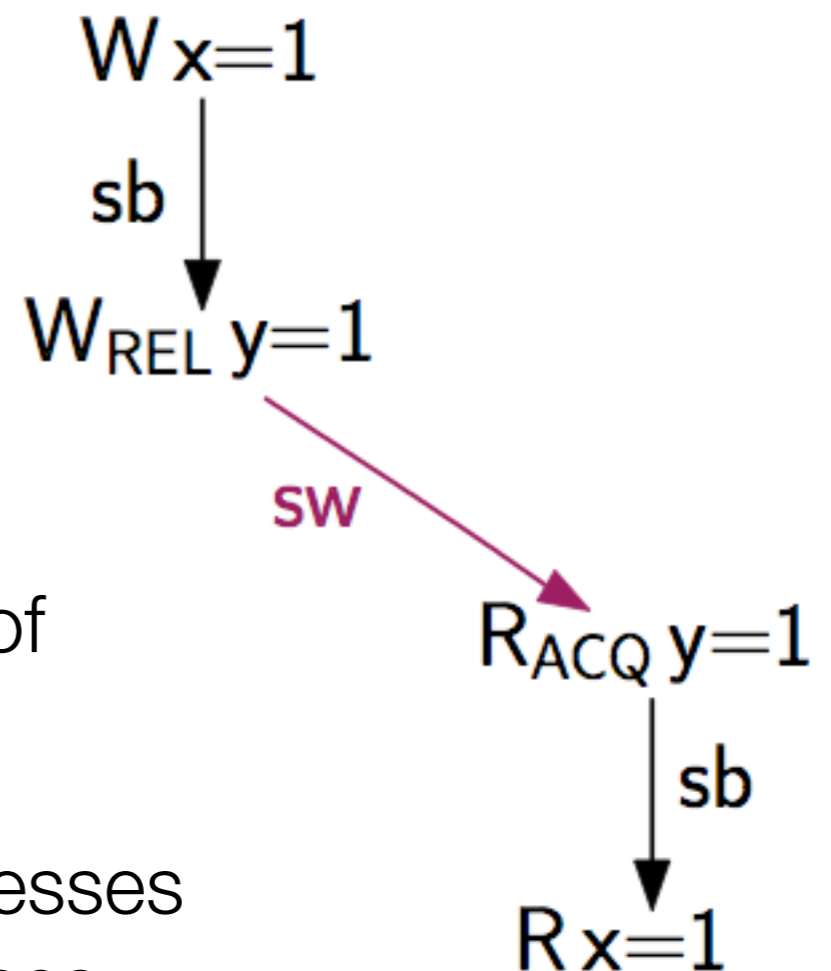
```
// receiver
```

```
while (0 == y.load(acquire));
```

```
r = x;
```

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

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



Expert concurrency: the release-acquire idiom

```
// sender
```

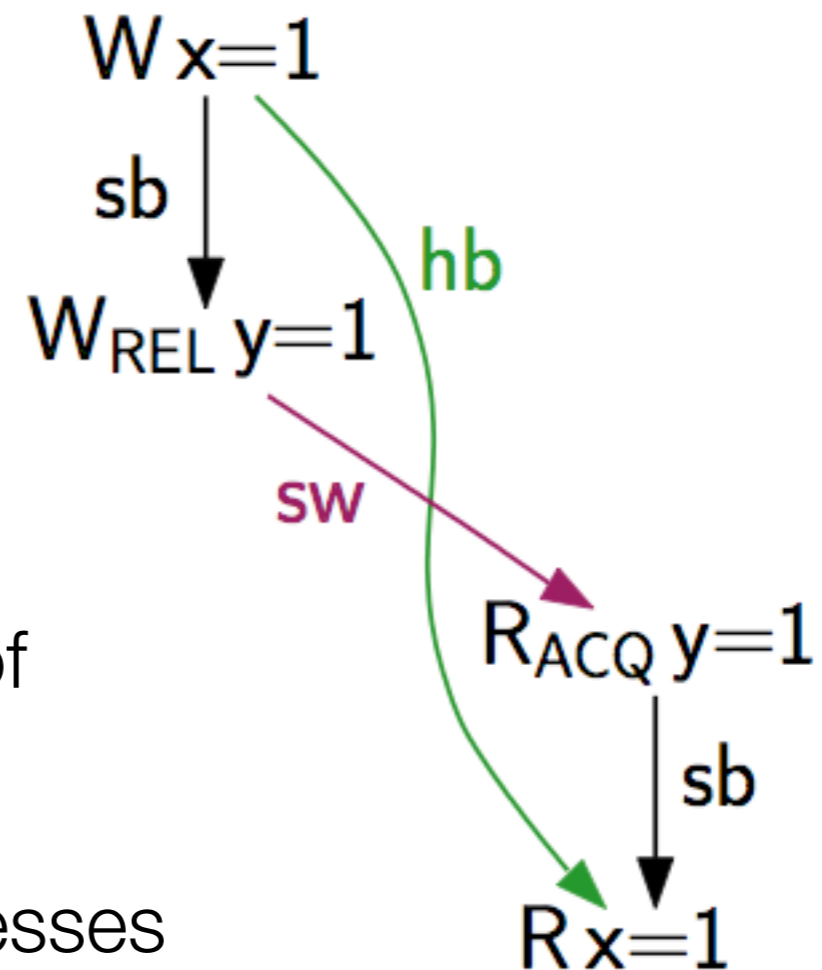
```
x = ...
y.store(1, release);
```

```
// receiver
```

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

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

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

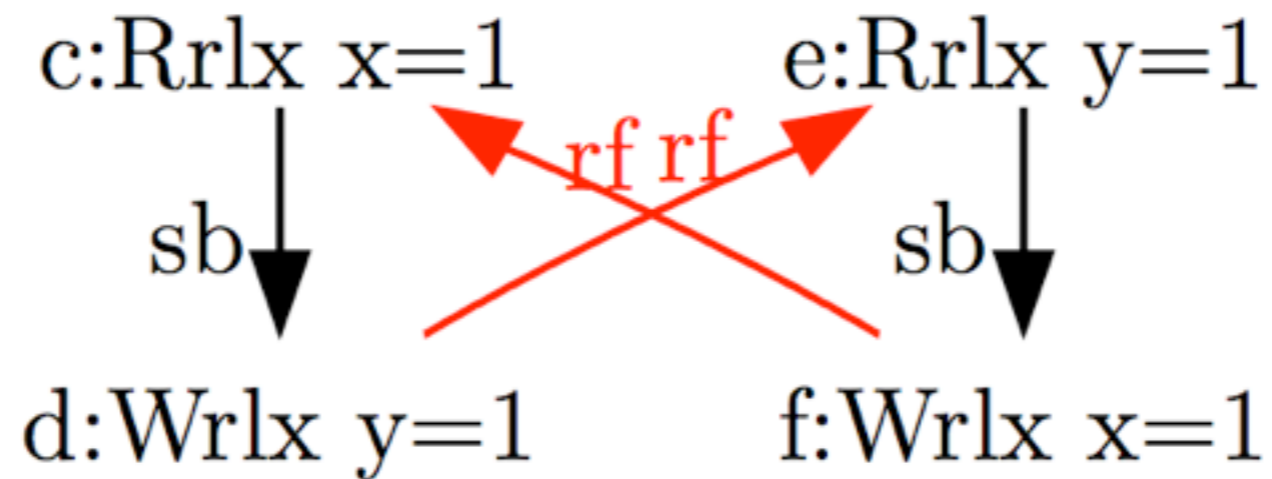


And in turn defines an **hb** constraint.

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

Relaxed writes

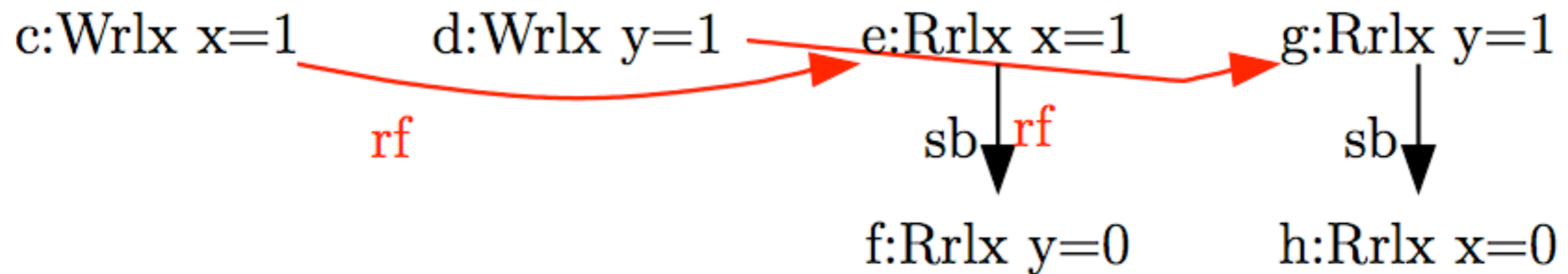
```
x.load(relaxed);      | y.load(relaxed);  
y.store(1, relaxed); | x.store(1, relaxed);
```



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); | y.load(relaxed); | x.load(relaxed);
```



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

Expert concurrency: fences avoid excess sync.

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

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

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

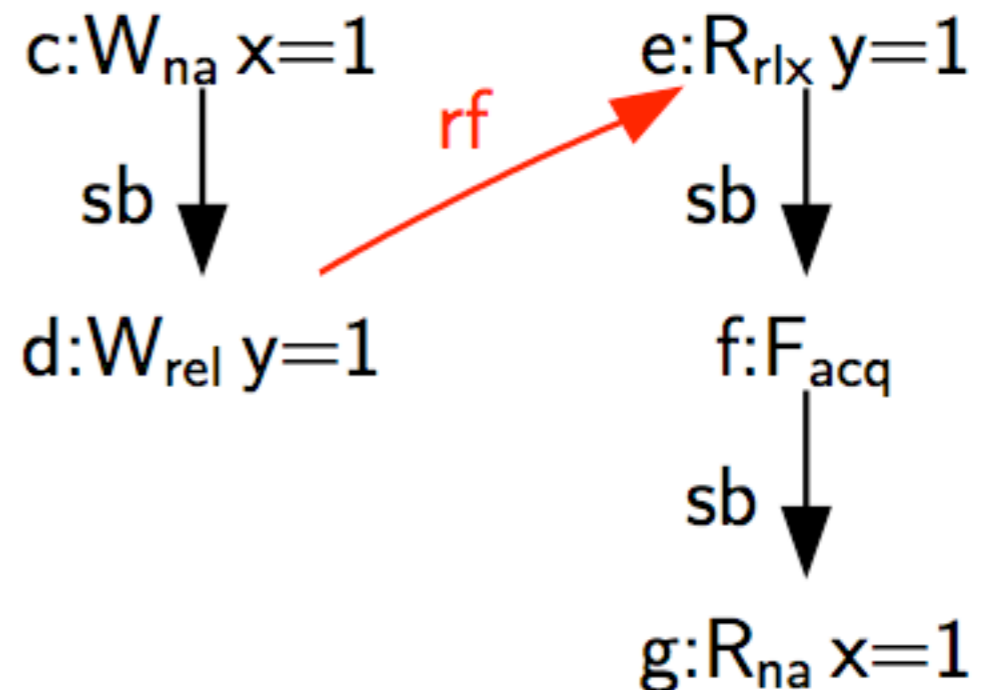
```
// receiver  
while (0 == y.load(relaxed));  
fence(acquire);  
r = x;
```

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 between a release write and a relaxed write.

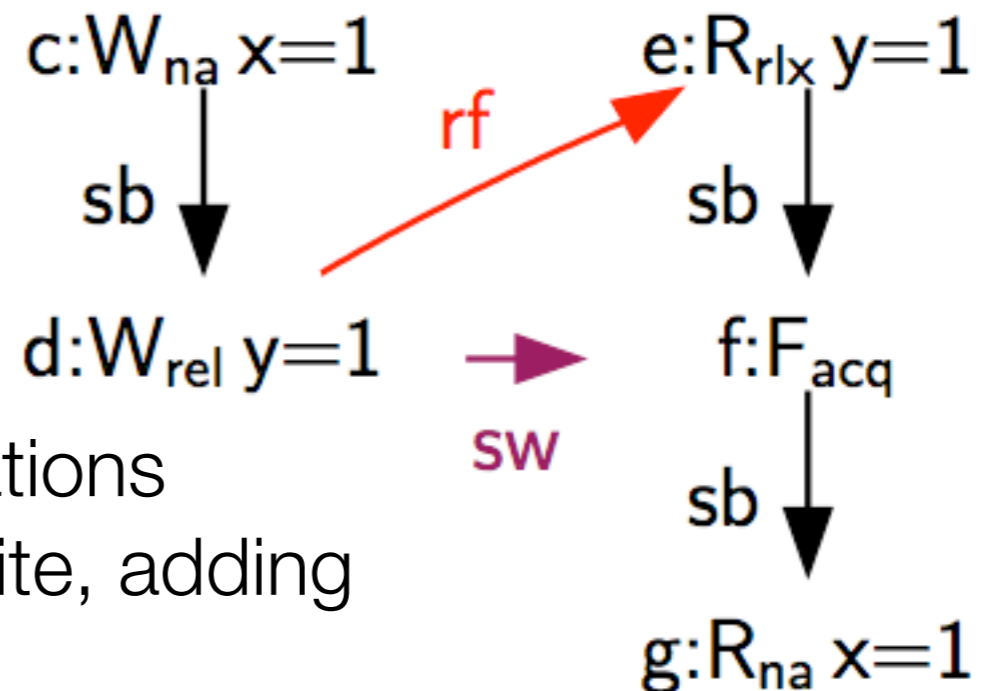


Expert concurrency: fences avoid excess sync.

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

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

Here we have an **rf** arrow between 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.

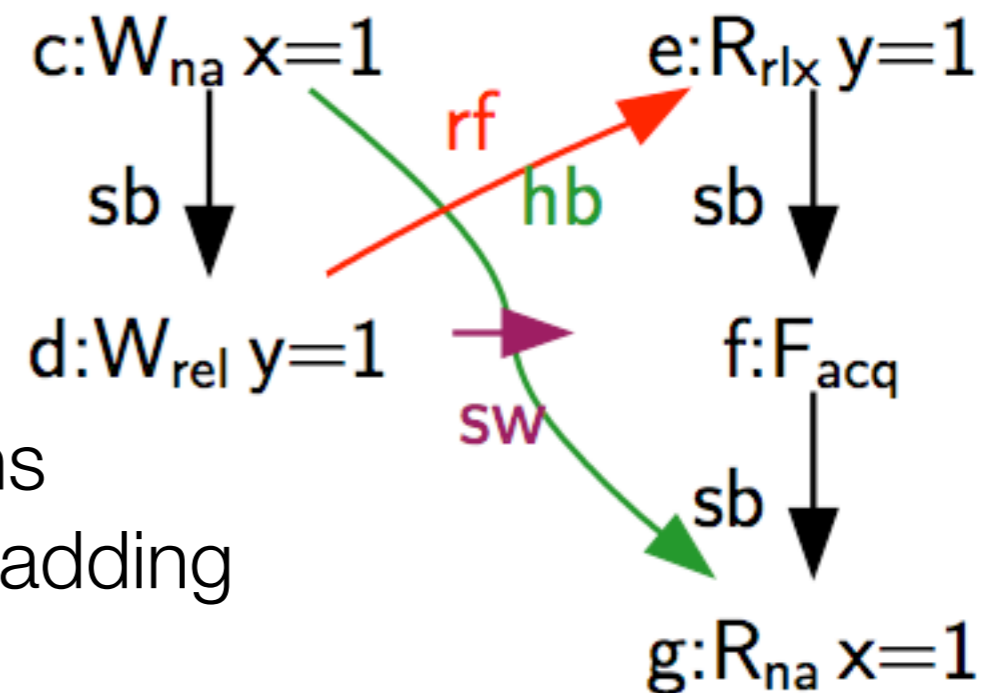
Expert concurrency: fences avoid excess sync.

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

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

Here we have an **rf** arrow between 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.



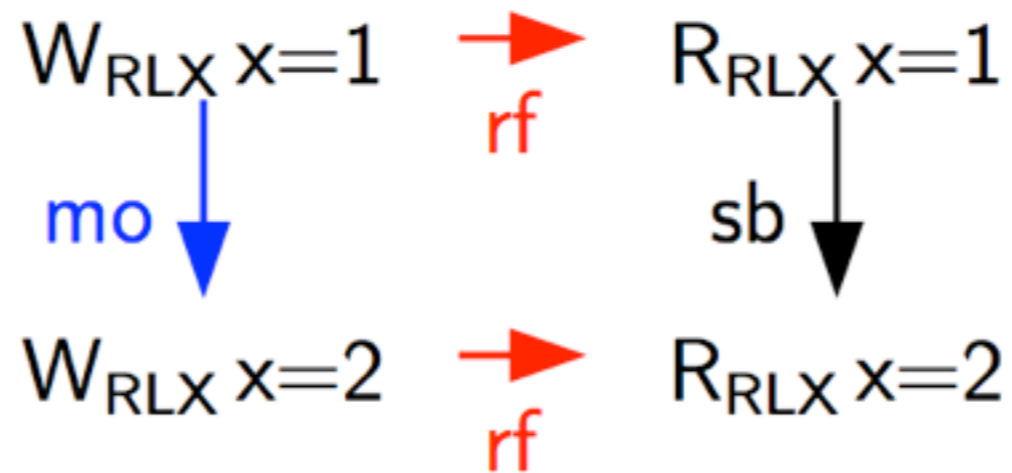
Happens-before follows as usual...

Modification order (aka coherence)

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

 ||

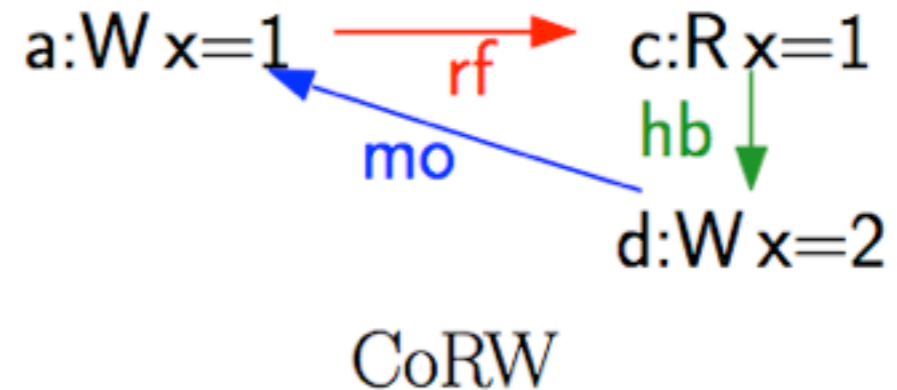
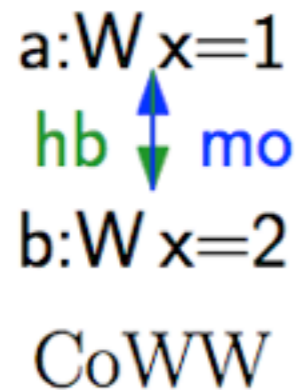
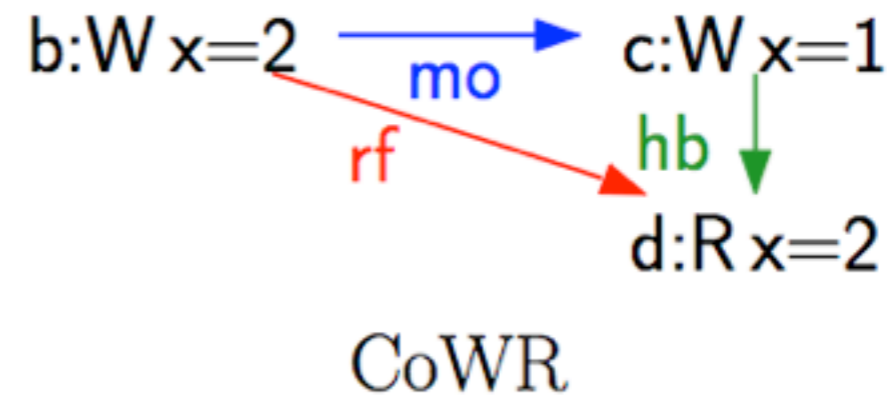
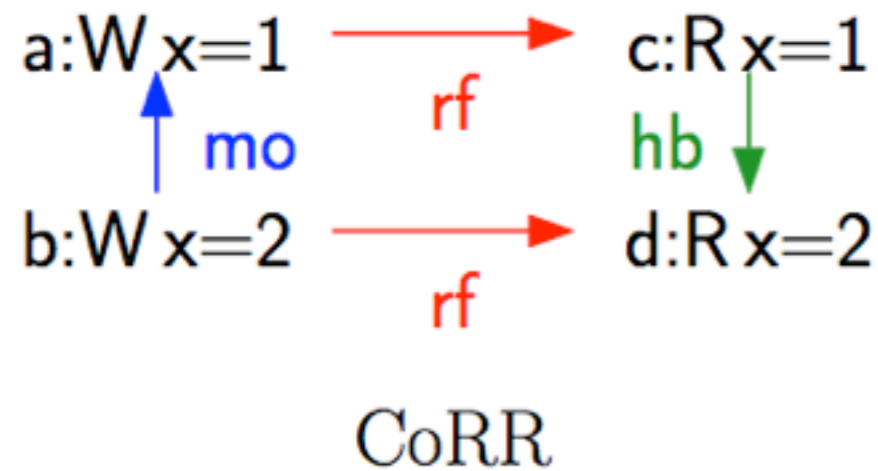
```
x.load(relaxed);  
x.load(relaxed);
```



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

Coherence and atomic reads

All forbidden:



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

Coherence and atomic reads

All forb

a:)

b:)

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.

b:W x=2

CoWW

d:W x=2

CoRW

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

Is C++11 hopelessly complicated?

Programmers cannot be given this model.

However, with a formal definition, **we can do proofs!** For instance:

- 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

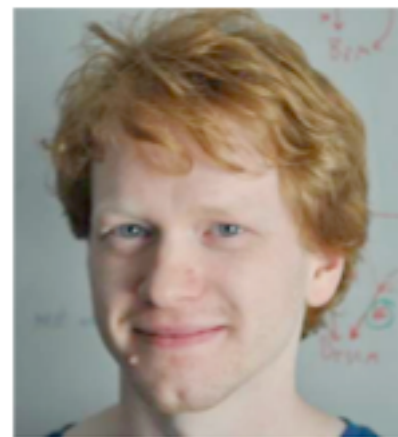
Is C++11 hopelessly complicated?

Simplifications:

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

Derivative models:

- without consume, happens-before is transitive



The current state of the standard

Fixed:

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

Not fixed:

- out of thin air reads (and self satisfying conditionals)
- `seq_cst` atomics do not guarantee SC



3. 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);
```

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 **MUST** only 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

```
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*

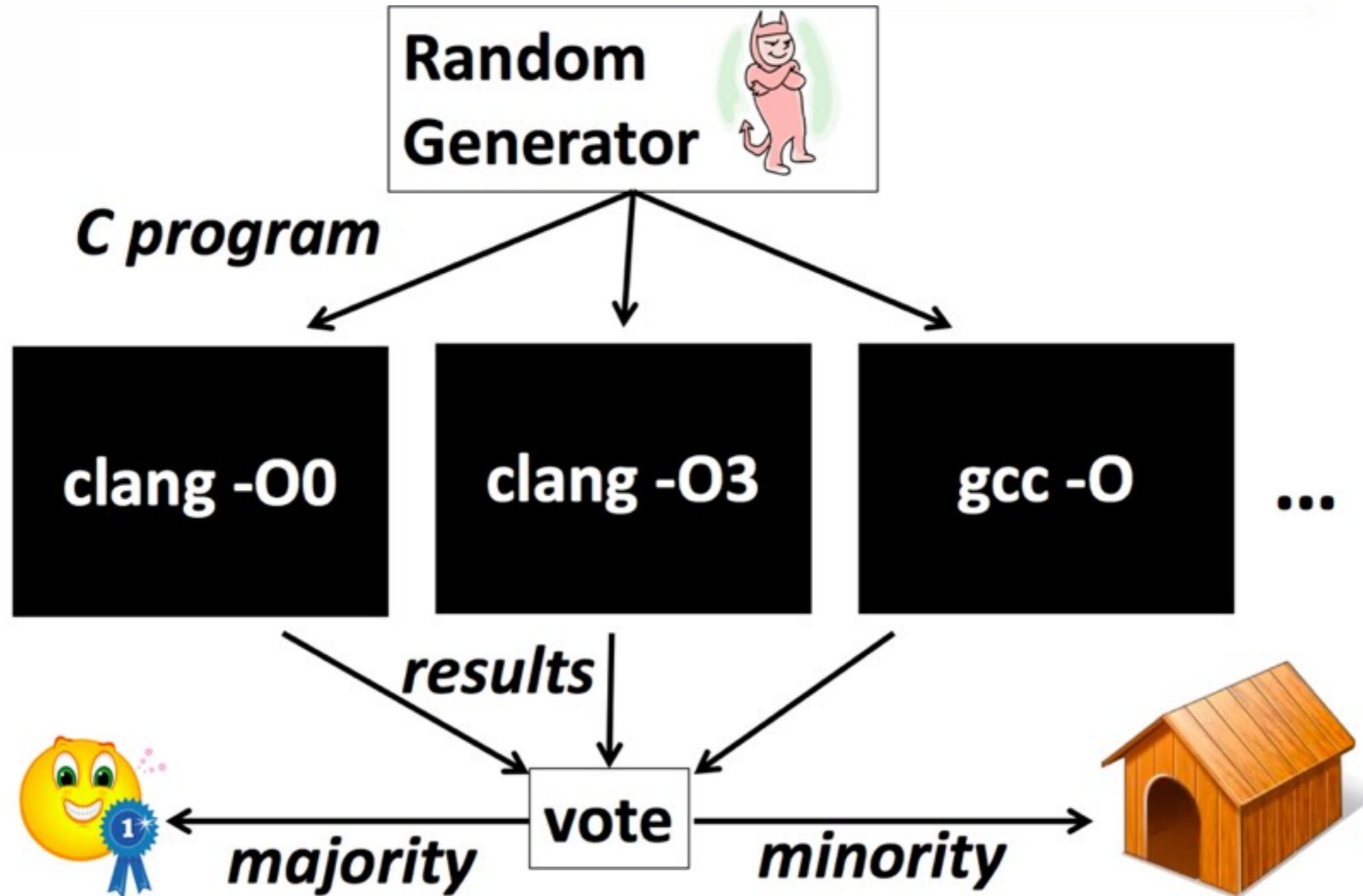
This is a *concurrency compiler bug*

```
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

Random



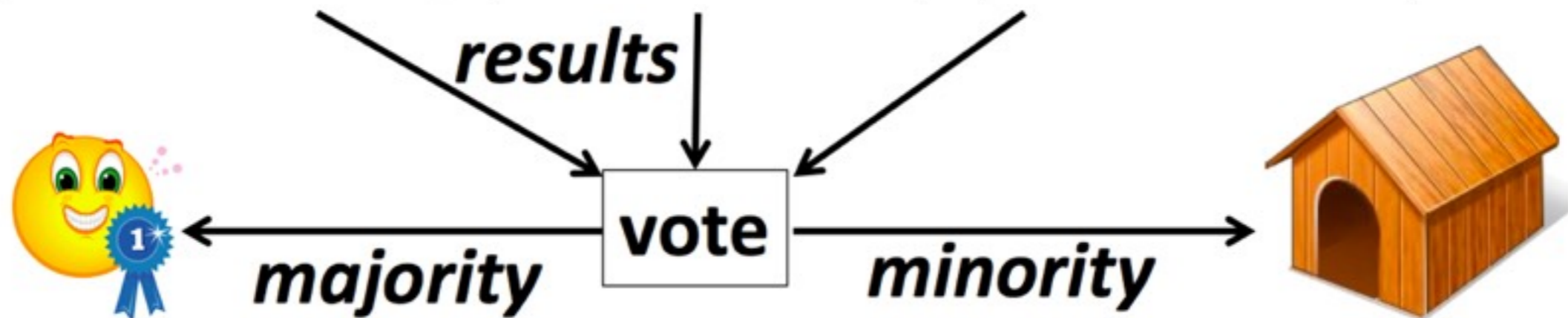
Reported hundreds of bugs
on various versions of gcc, clang and other compilers

clang -O0

clang -O3

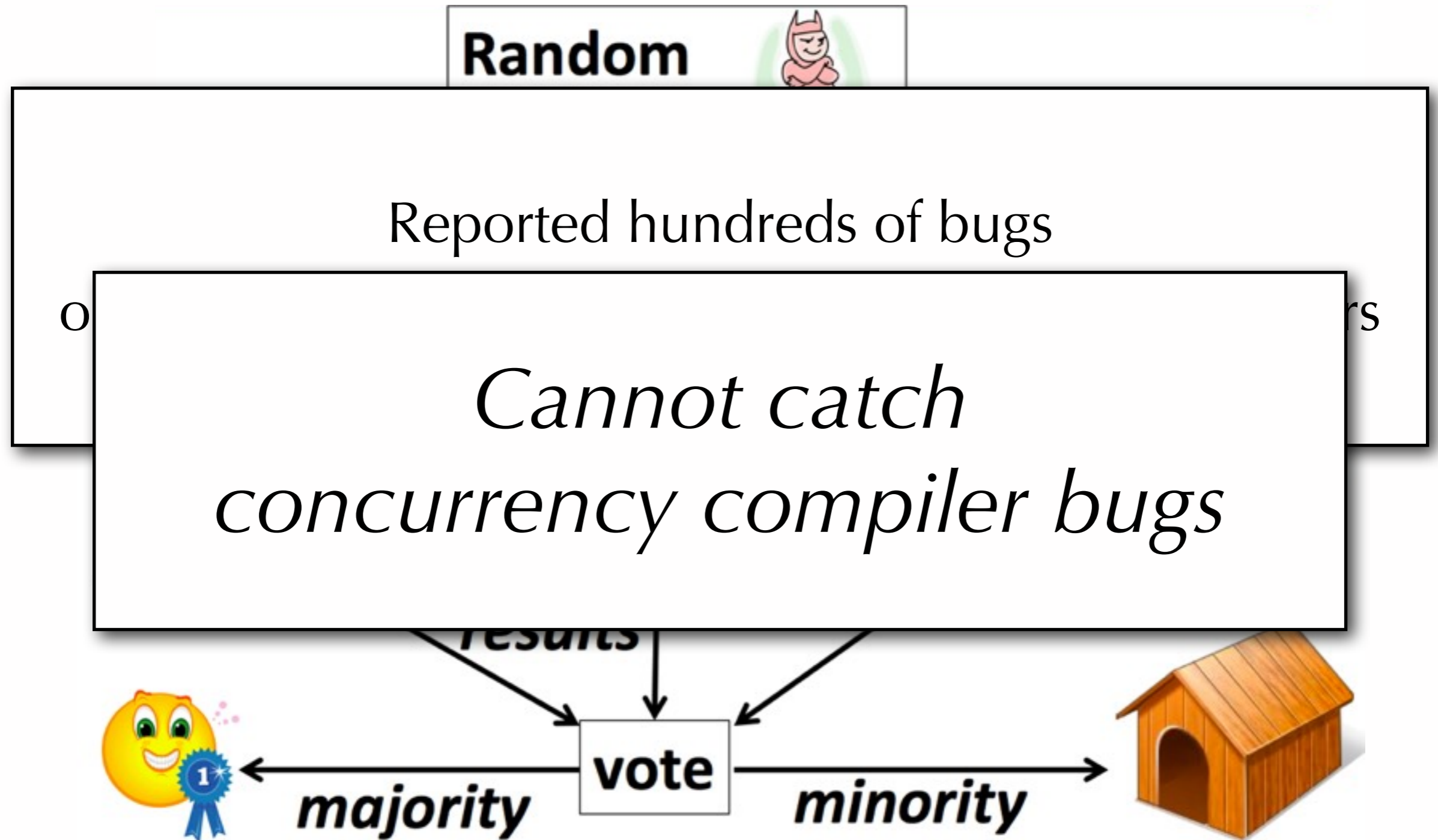
gcc -O

...



Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



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

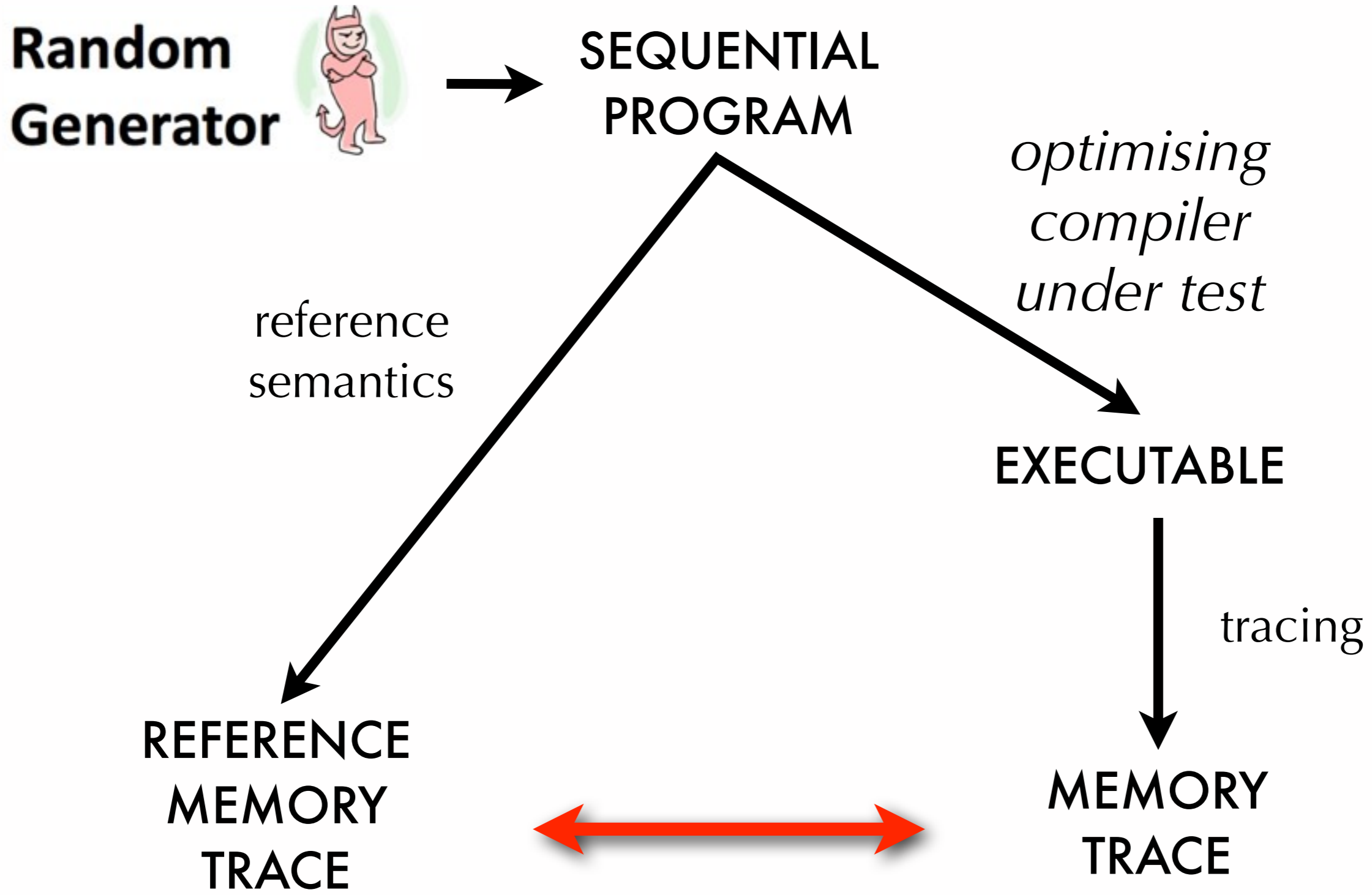


C/C++ compilers can only apply transformations sound
with respect to an arbitrary non-racy concurrent context

Hunt concurrency compiler bugs

=

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



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)  
        ;  
}
```

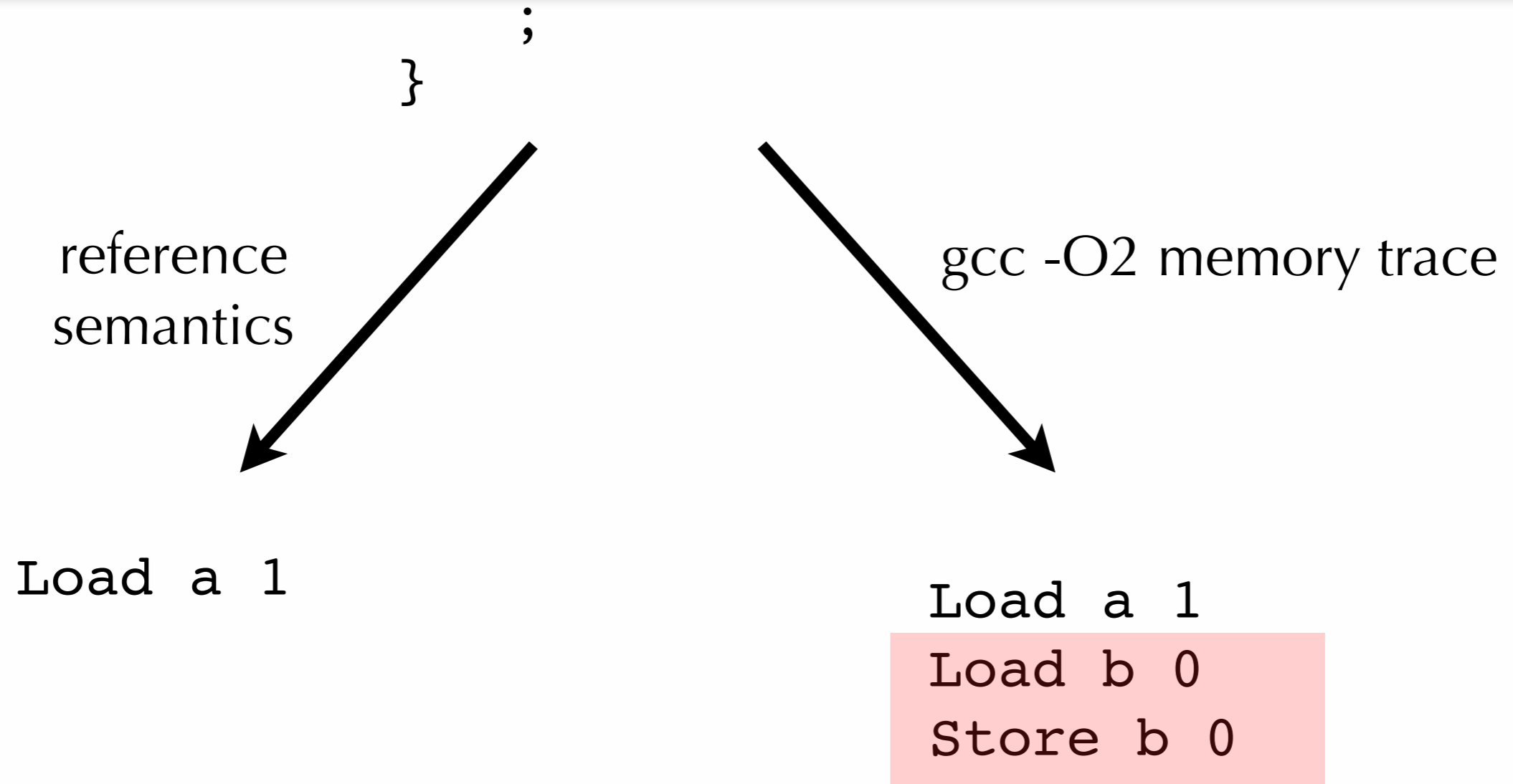
reference
semantics

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 ;  
}
```

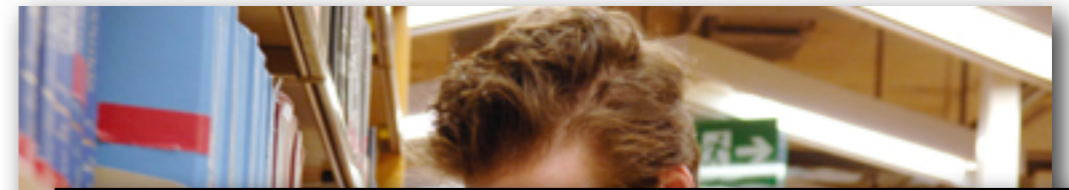
Example: loop invariant code motion

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

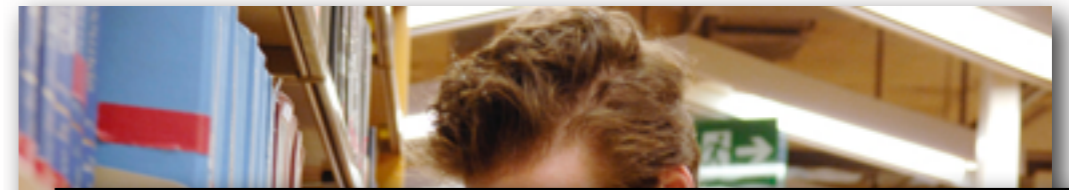
Compiler Writer



Sophisticated program analyses
Fancy algorithms
Source code or IR
Operations on AST

```
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
```

Example: loop invariant code motion

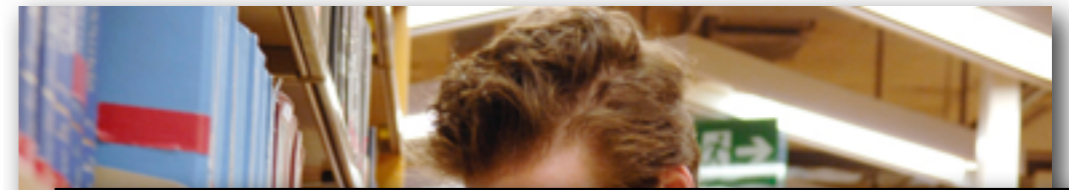
Compiler Writer



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 ;  
}
```

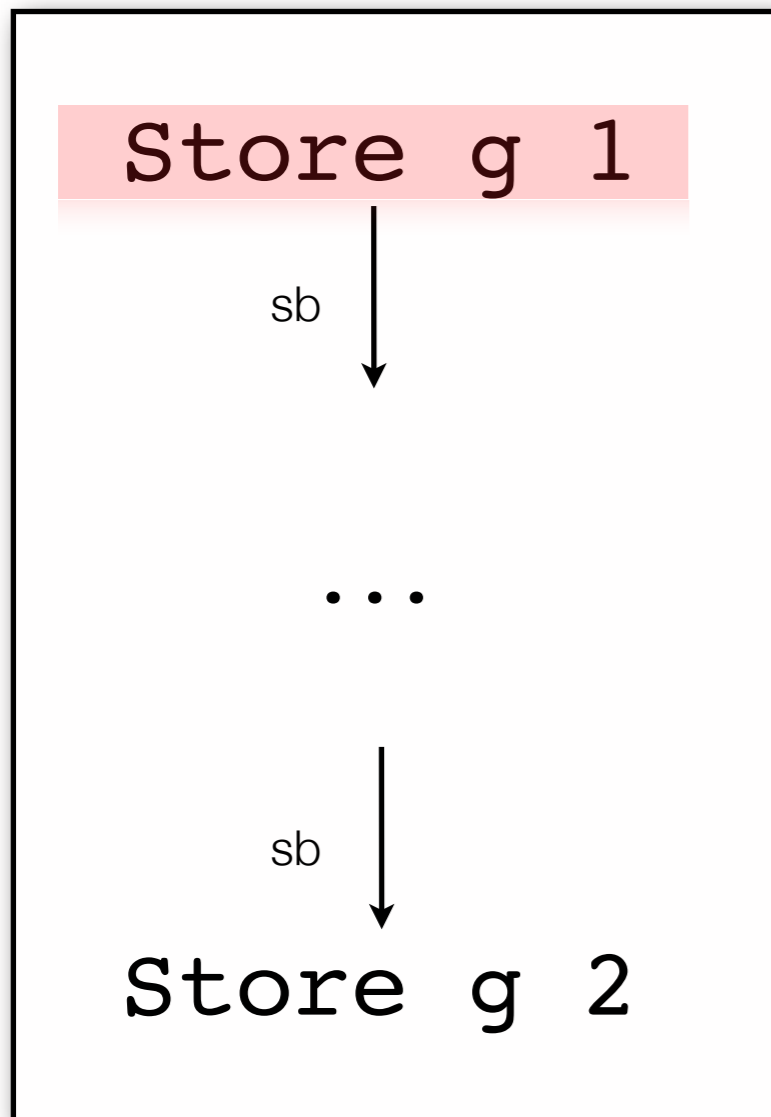
Semanticist



Elimination of run-time events
Reordering of run-time events
Introduction of run-time events
Operations on sets of events

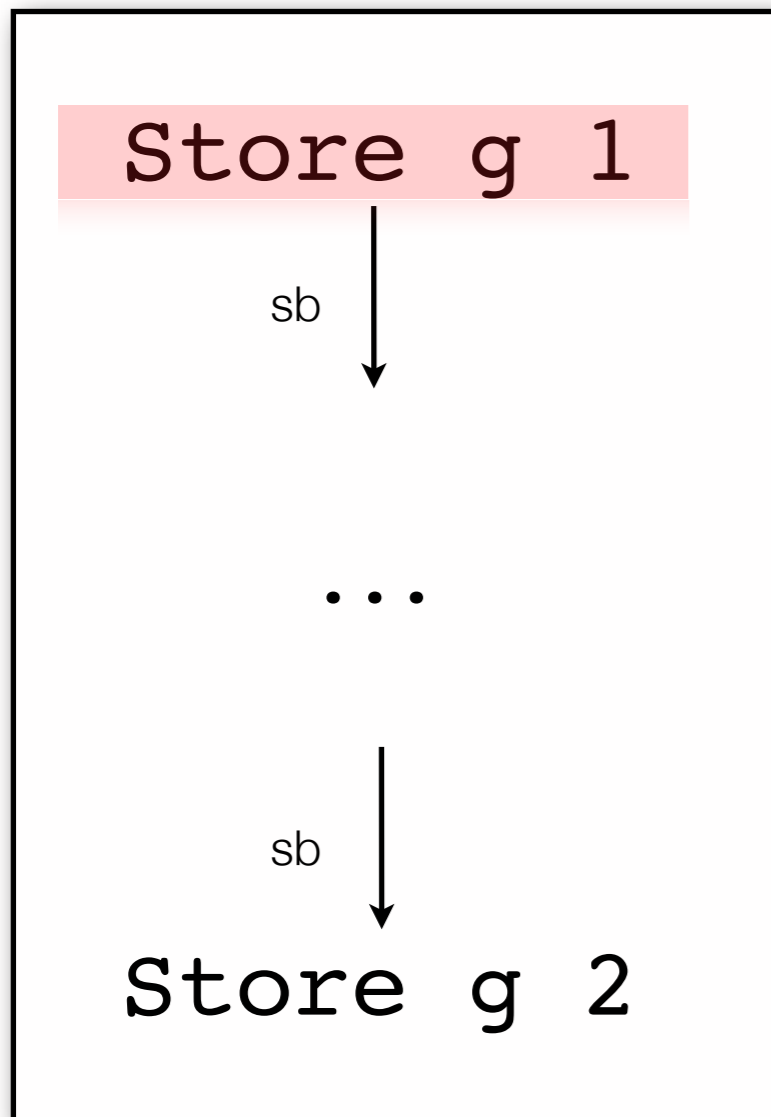
```
Load y 42  
Store z 0  
  
Store x[0] 43  
Store z 1  
  
Store x[1] 43
```


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);
```

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)
```

sync

The release/acquire synchronisation guarantees that:

- the program is DRF
- 42 is printed at the end of the execution

Remark: unlock \approx release, lock \approx acquire.

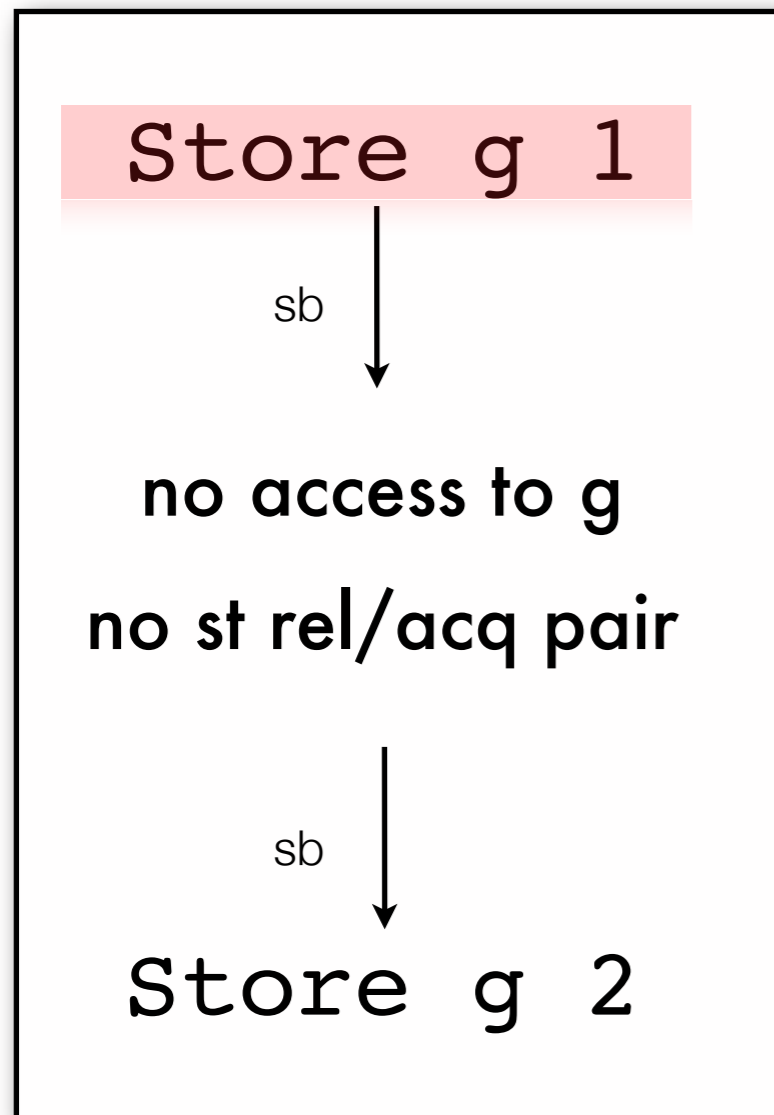
Same-thread release/acquire pairs

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

```
g = 1;
```

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

```
while(f2.load(ACQUIRE) == 0);
```

```
g = 2;
```

candidate overwritten write



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

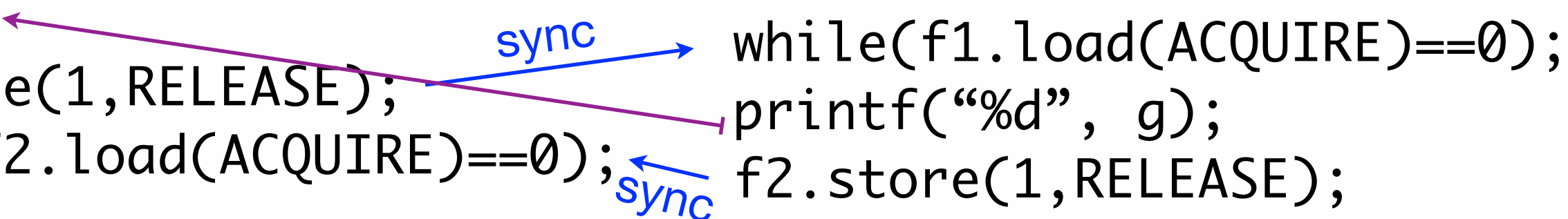
```
g = 0; atomic f1 = f2 = 0;
```

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

`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**

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;
```

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);
```

sync



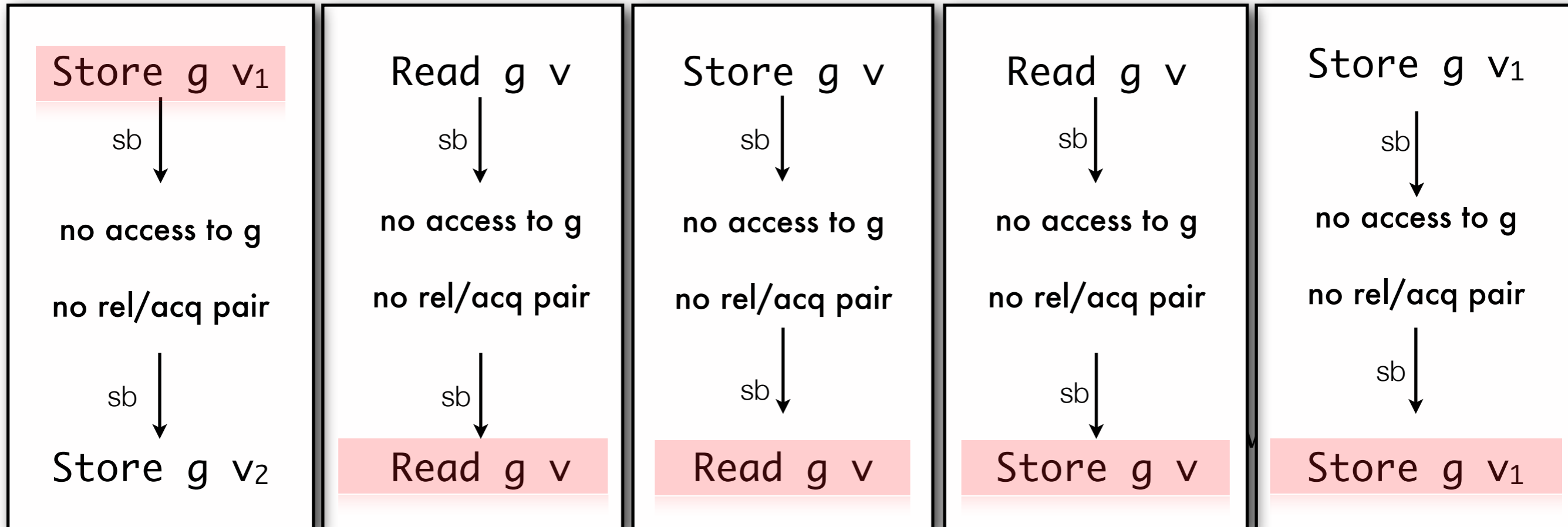
data race



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

It is sound to optimise the overwritten write.

Eliminations: bestiary



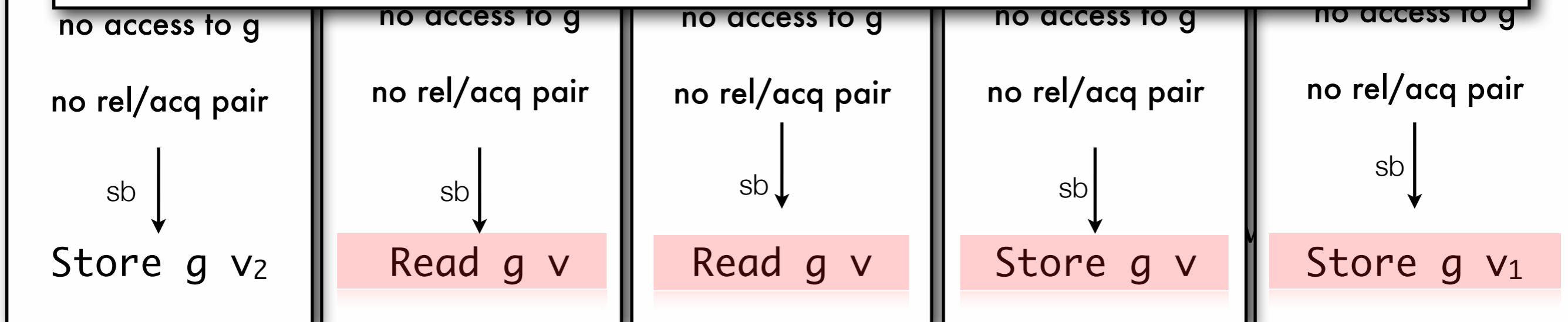
Overwritten-Write Read-after-Read Read-after-Write Write-after-Read Write-after-Write

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

Elimination: ~~beating~~

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

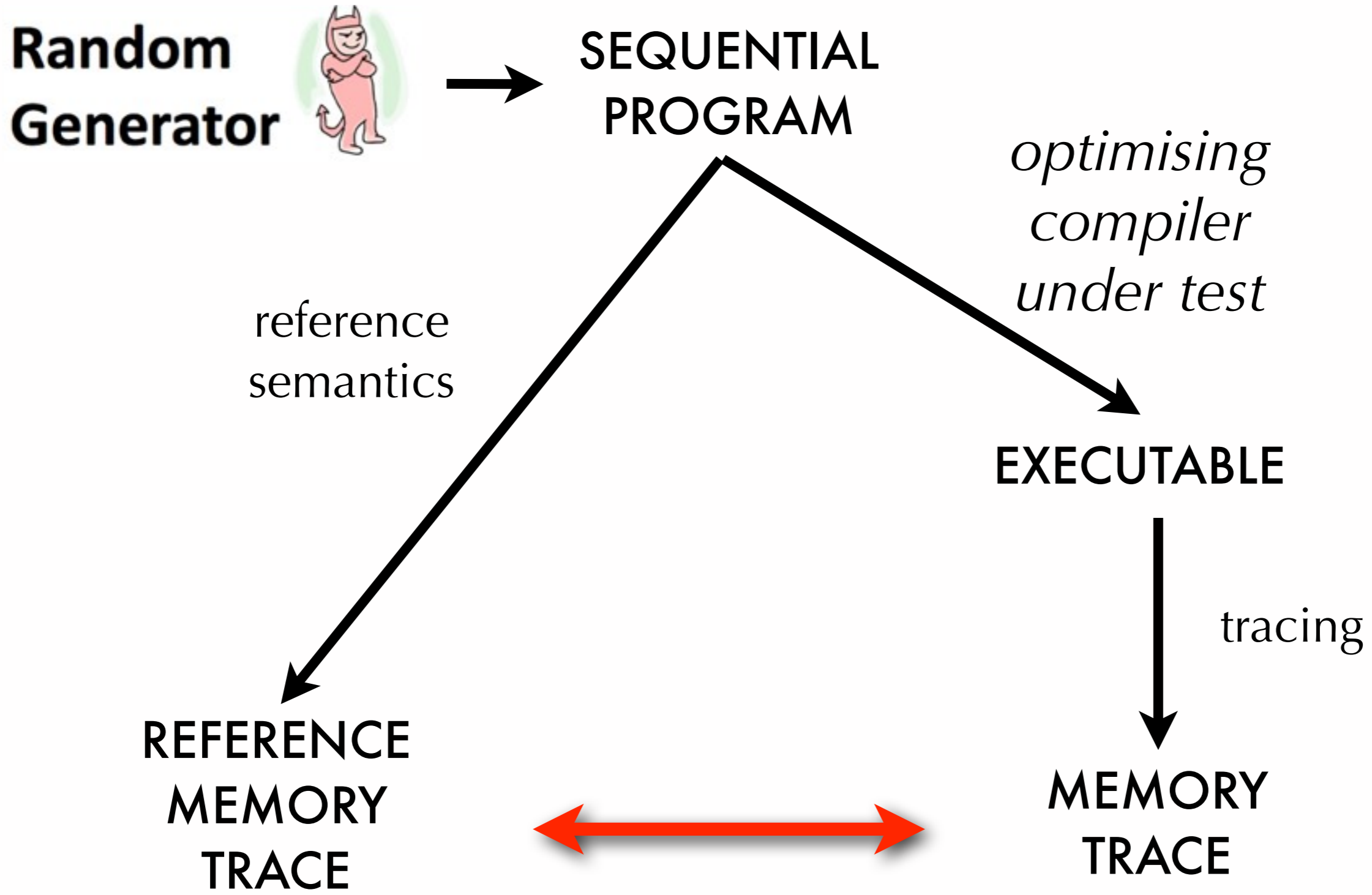
Roach-motel reordering (reordering across locks) not observed in practice

Read introductions observed in practice (gcc, clang).

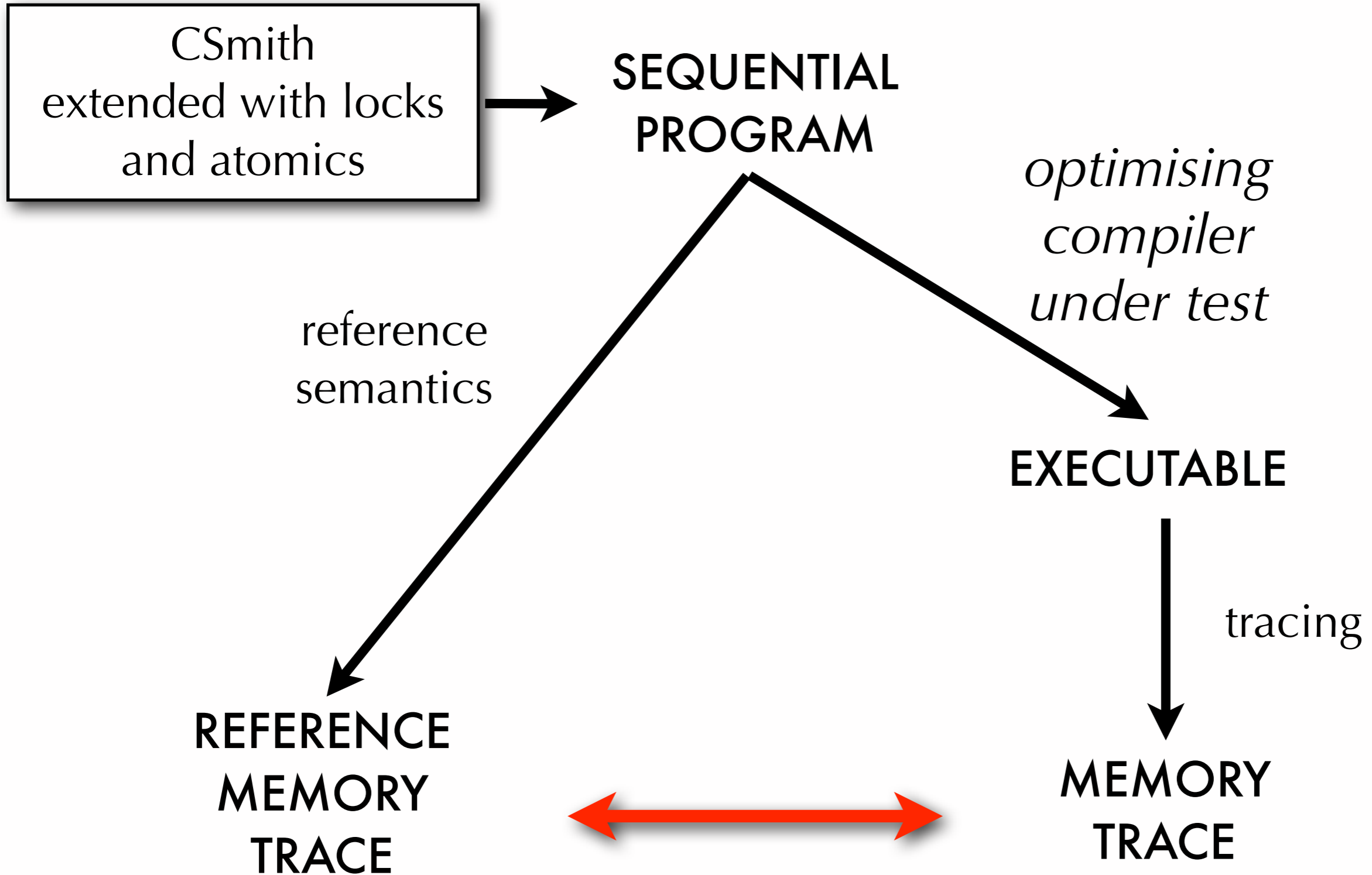
Introduction of eliminable reads proved correct.

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

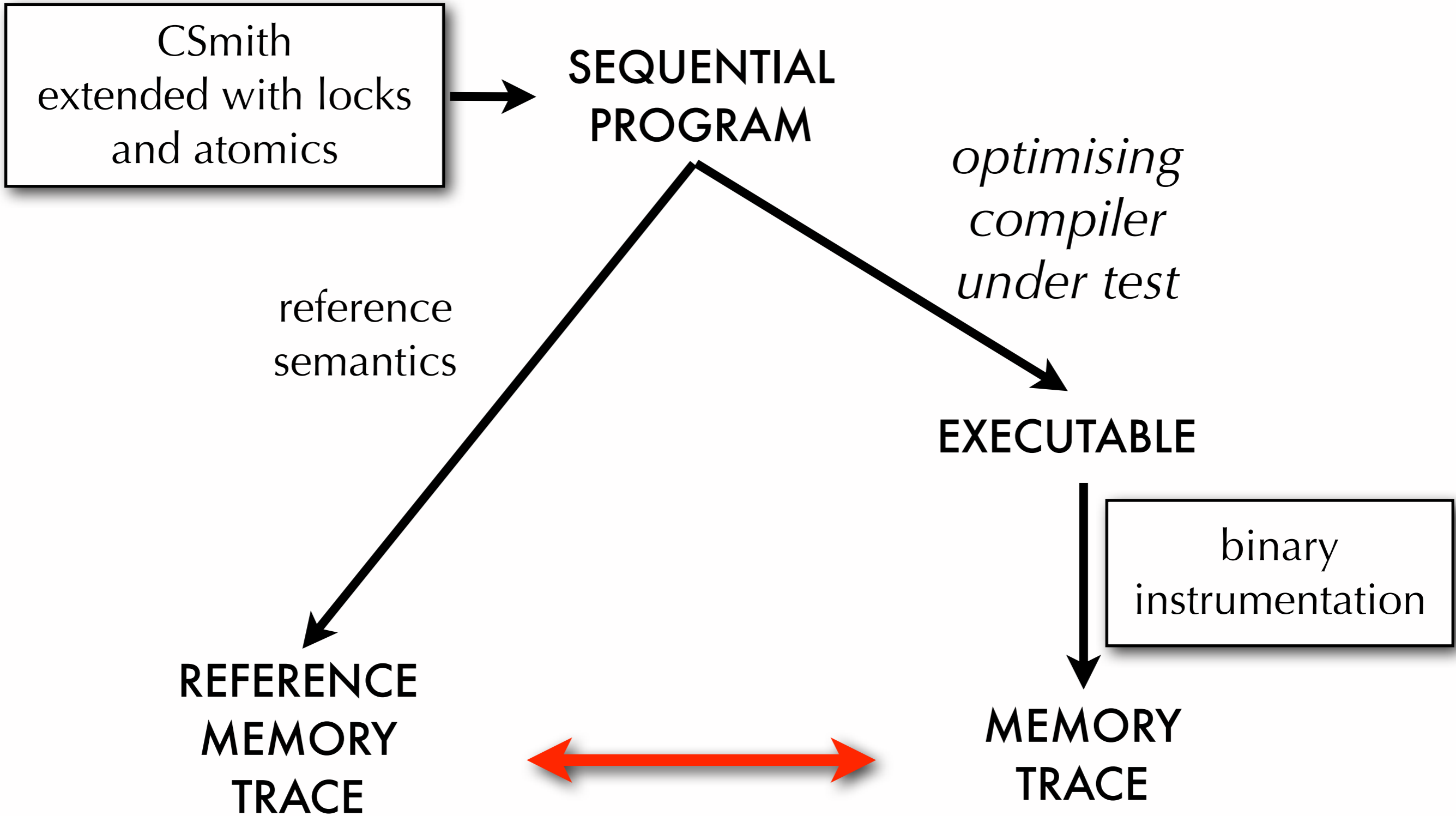
The CMMTEST Tool



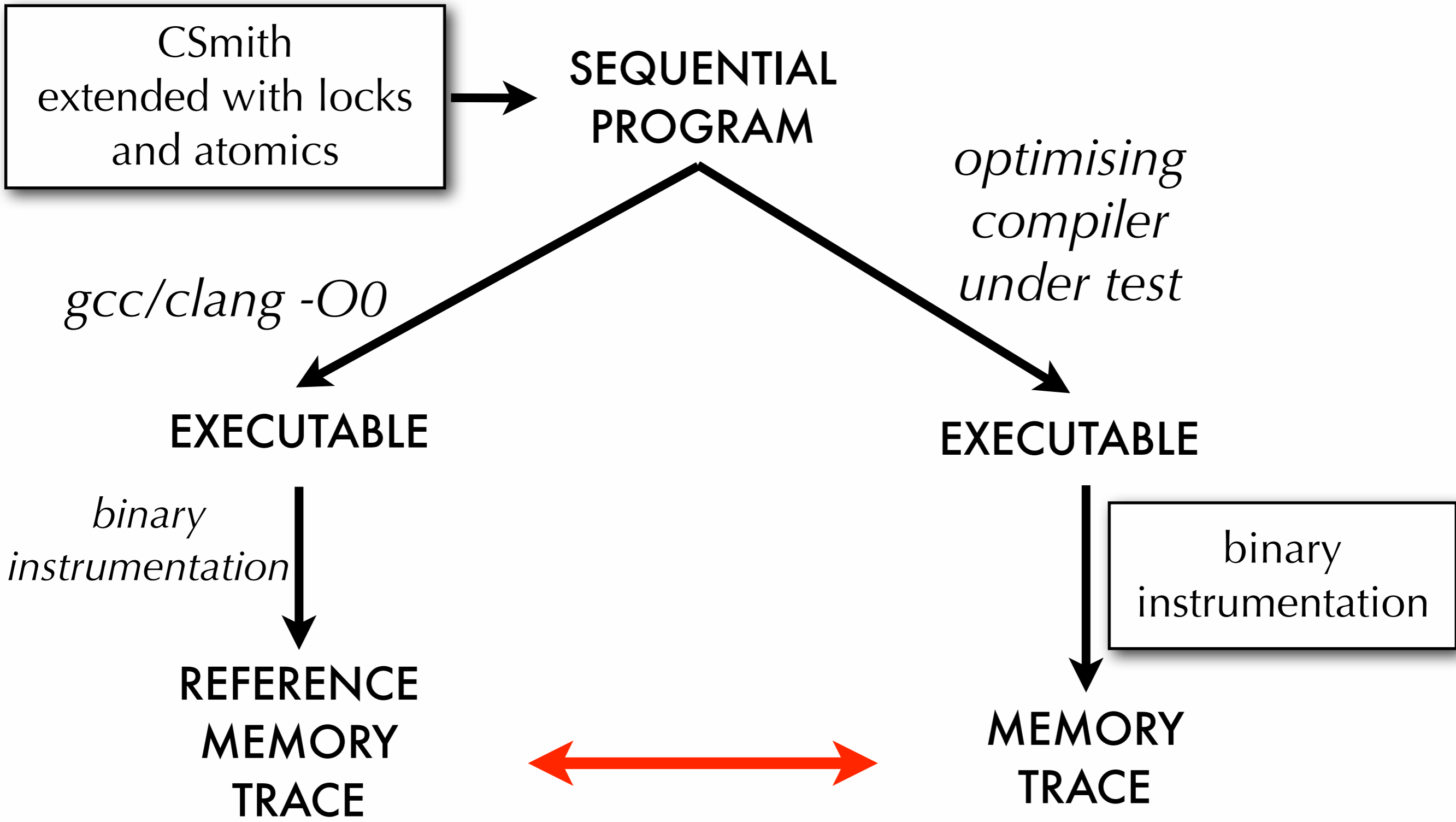
only transformations sound in any concurrent (non-racy) context?



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

*optimising
compiler
under test*

gcc/clang -O0

EXECUTABLE

EXECUTABLE

*binary
instrumentation*

REFERENCE
MEMORY
TRACE

binary
instrumentation

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*

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

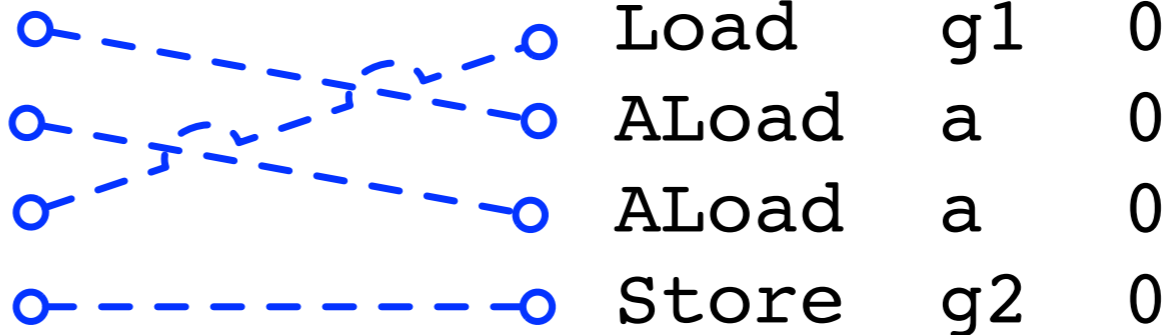
Baked this invariant into the tool and found a counterexample...

...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
```



3. Detecting unexpected behaviours

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.



4. Out of thin-air reads

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

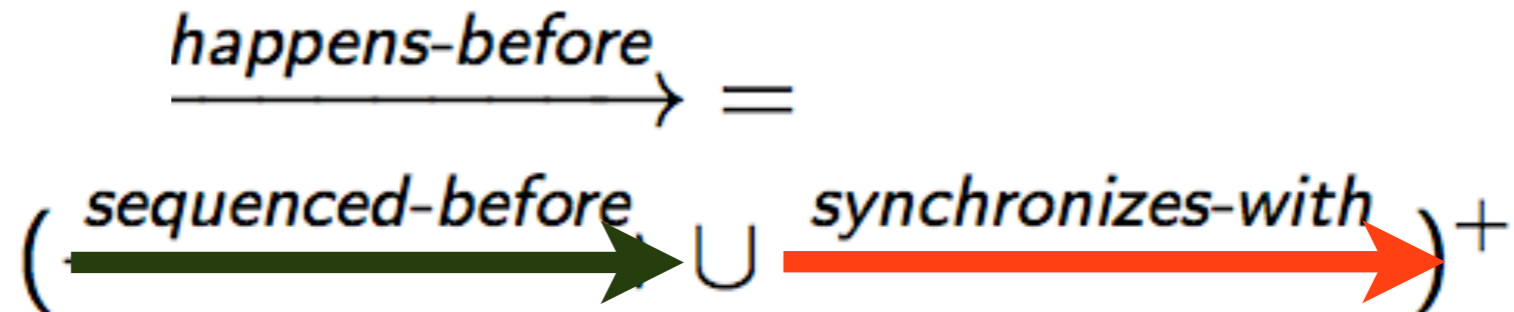
`x = y = 0`

Thread 1

Thread 2

`y = 1`
↓
`x.store(1, MO_RELEASE)`

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



Non-atomic loads must return the most recent write
in the happens-before order

Understanding MO_RELAXED

`x = y = 0`

Thread 1

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

Thread 2

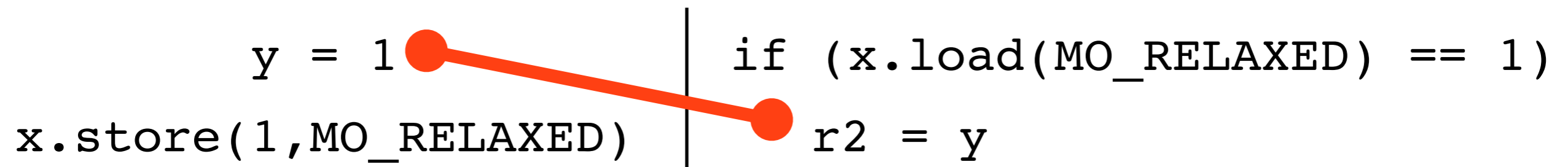
```
if (x.load(MO_RELAXED) == 1)  
    r2 = y
```

Understanding MO_RELAXED

$x = y = 0$

Thread 1

Thread 2



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

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)
```

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

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)
```

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

Shorthand

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

Out-of-thin-air

Thread 1

`r1 = x`

`y = r1`

`x = y = 0`

|

Thread 2

`r2 = y`

`x = 42`

Out-of-thin-air

Thread 1

$r1 = x$
 $y = r1$

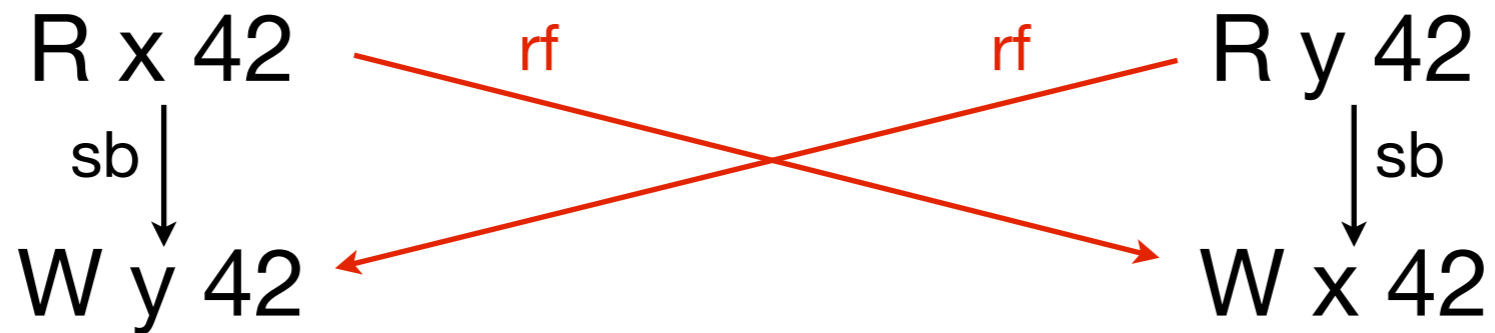
$x = y = 0$



Thread 2

$r2 = y$
 $x = 42$

$r1 = r2 = 42$
is a valid execution.



Intuition

the compiler (or hardware) can reorder independent accesses

Thread 1

$r1 = x$
 $y = r1$

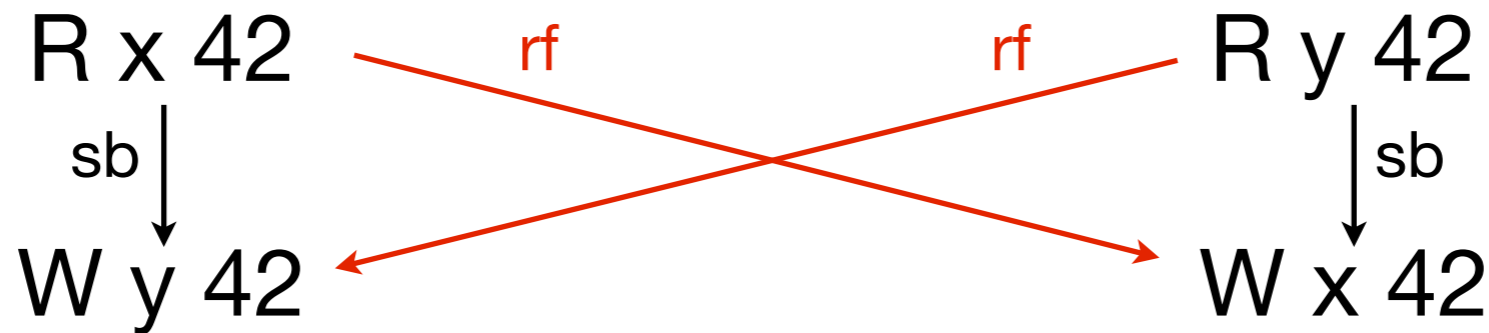
$x = y = 0$



Thread 2

$r2 = y$
 $x = 42$

$r1 = r2 = 42$
is a valid execution.



Out-of-thin-air reads

Thread 1

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

$x = y = 0$

|

Thread 2

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


Out-of-thin-air reads

Thread 1

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

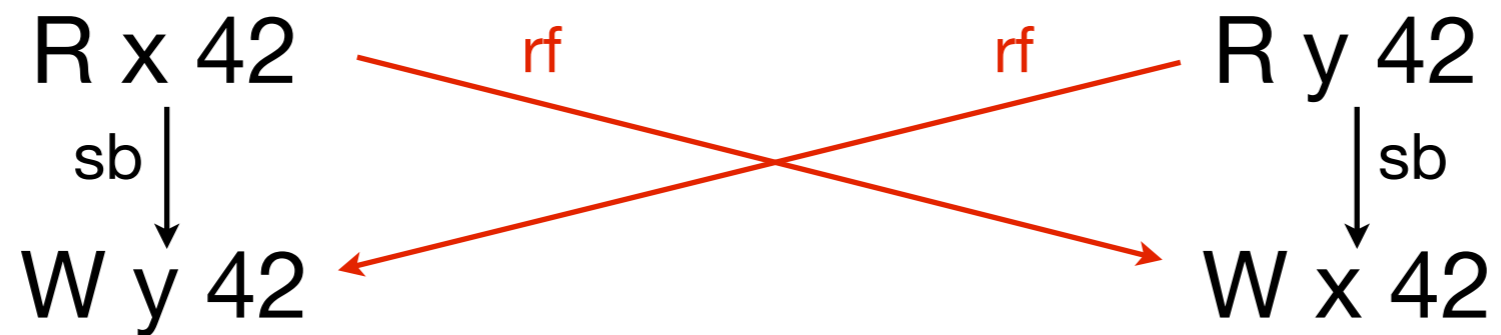
$x = y = 0$

Thread 2

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

$r1 = r2 = 42$

is also an allowed execution



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

Thread 1

r1 = x
y = r1

x = y = 0

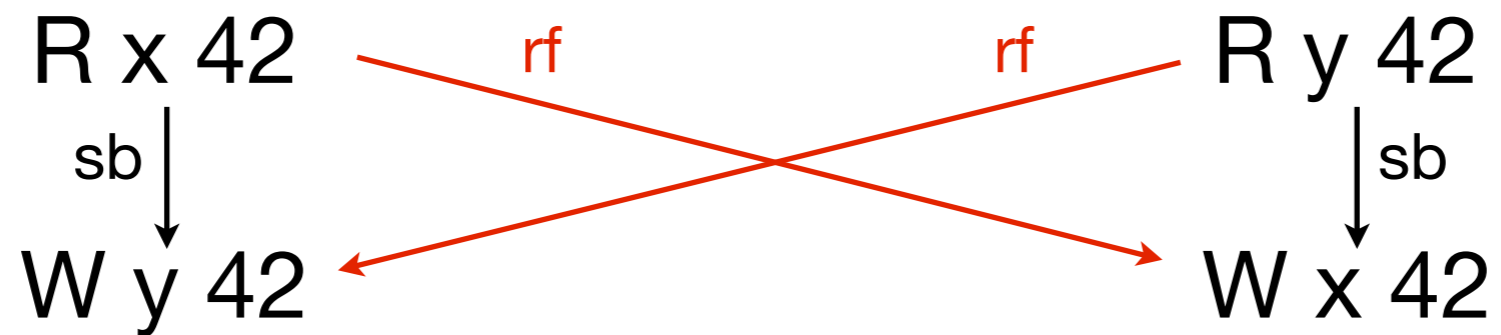


Thread 2

r2 = y
x = r2

r1 = r2 = 42

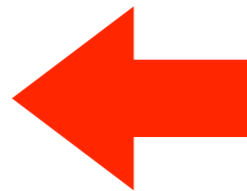
is also an allowed execution



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
```

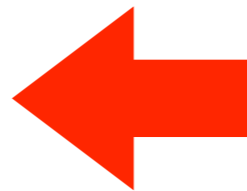


initially $x = y = 0$	
r1 := x	r2 := y
y := r1	x := r2
print r1	print r2

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
```



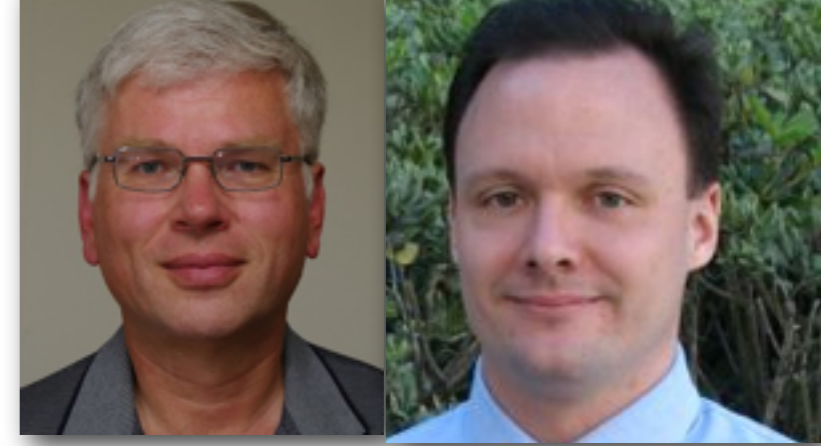
initially $x = y = 0$	
r1 := x	r2 := y
y := r1	x := r2
print r1	print r2

It does not happen in practice... even if it might!



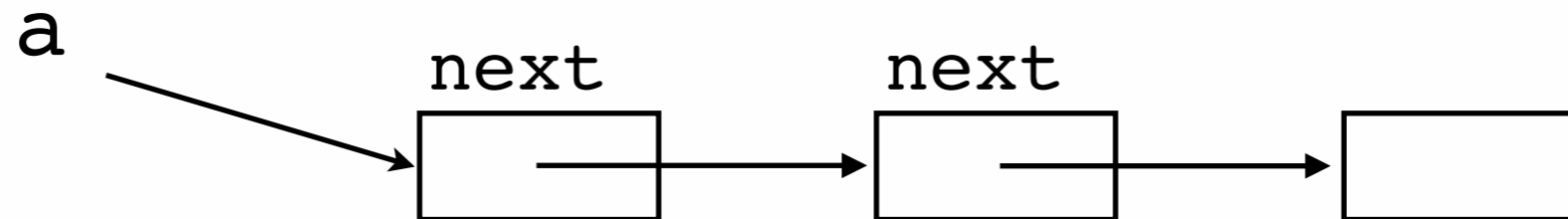
Consequences of out-of-thin-air reads

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

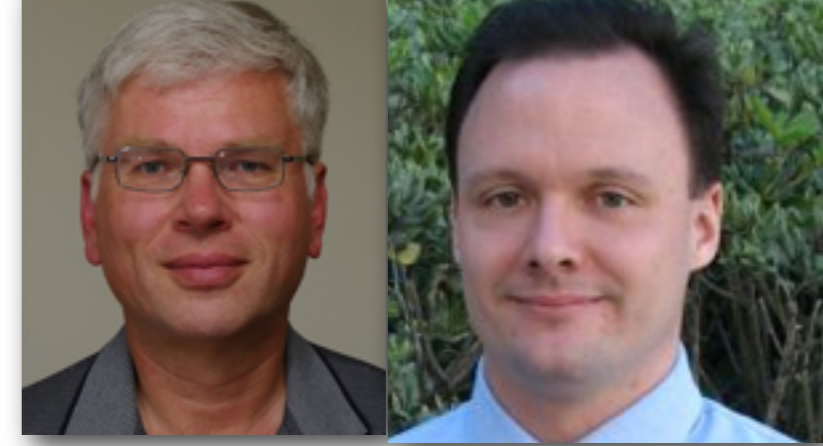


Thread 1

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

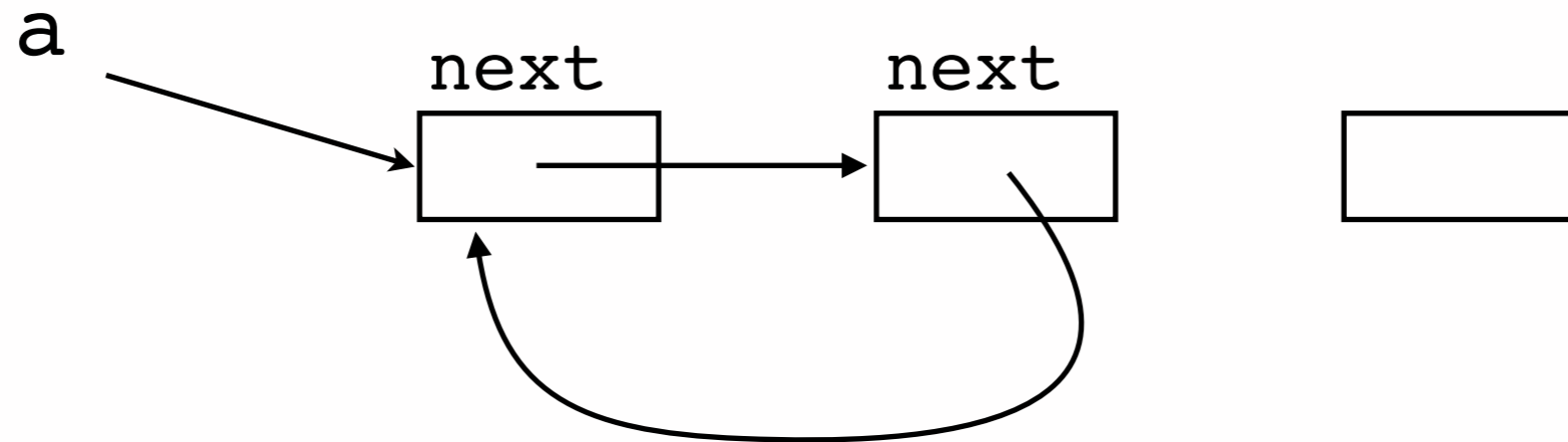


```
struct foo {  
    atomic<struct foo *> next;  
}  
struct foo *a;
```

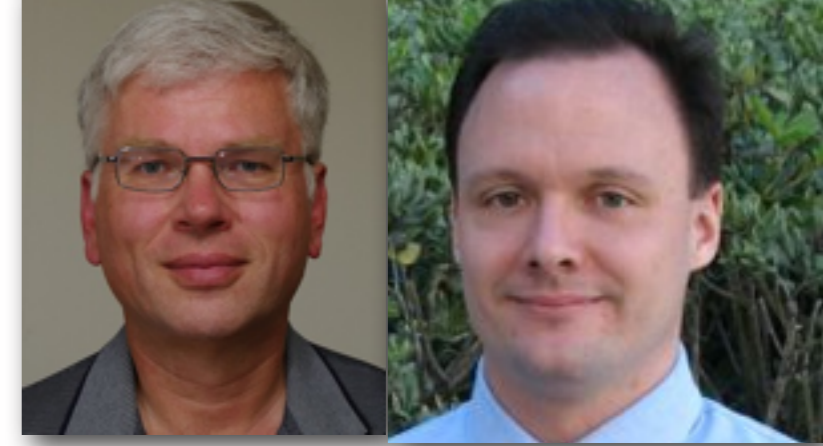


Thread 1

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



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



Thread 1

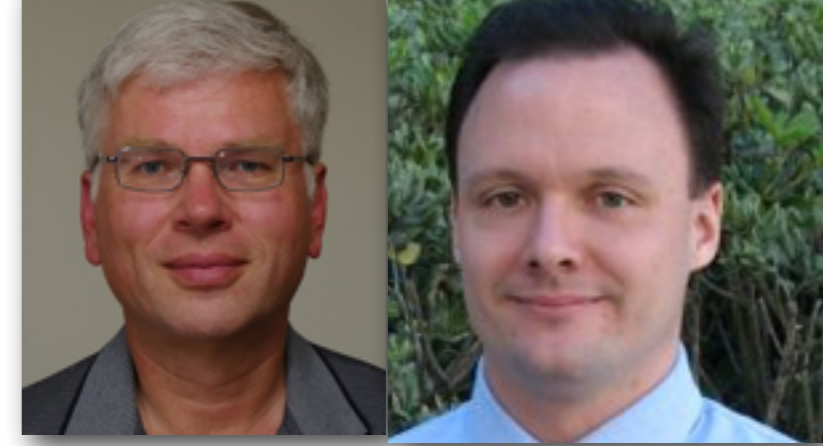
```
r1 = a->next  
r1->next = a
```

Thread 2

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



```
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 **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;
```

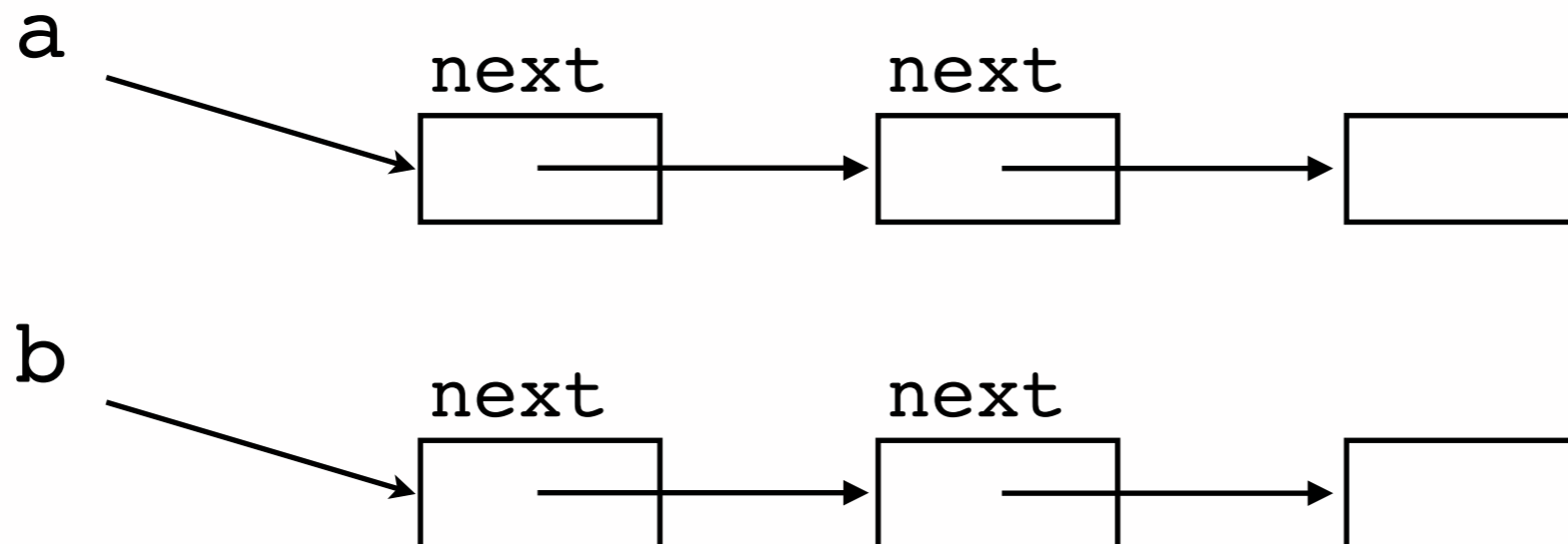


Thread 1

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

Thread 2

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



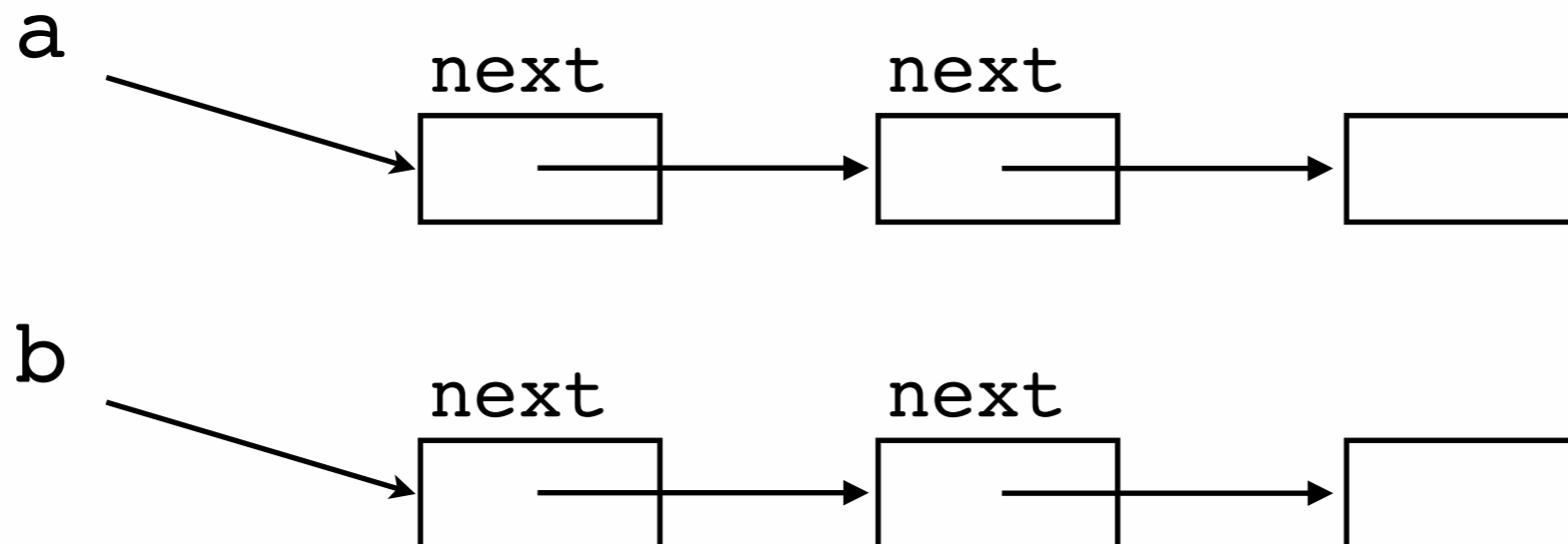
If the compiler speculates $r1=b$ and $r2=a$, then
the store $r1 \rightarrow next = a$ justifies $r2 = b \rightarrow next$ assigning $r2 = a$
(and symmetrically to justify $r1 = b$)

Thread 1

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

Thread 2

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



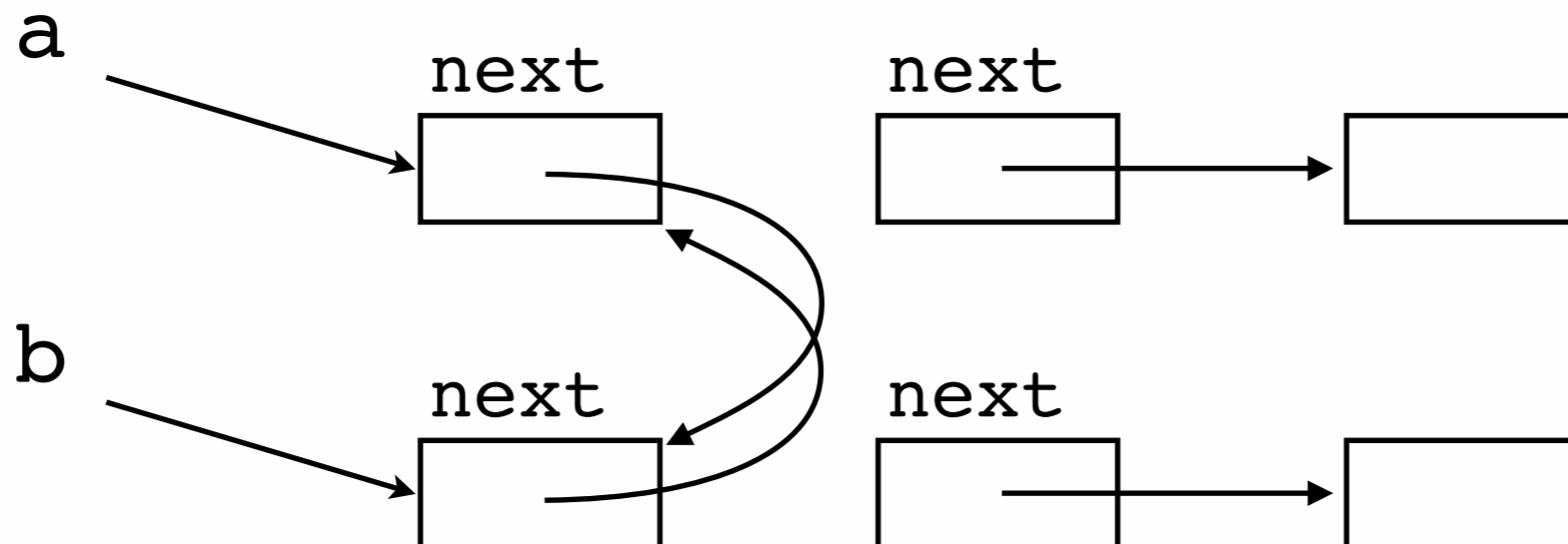
If the compiler speculates $r1=b$ and $r2=a$, then
the store $r1 \rightarrow next = a$ justifies $r2 = b \rightarrow next$ assigning $r2 = a$
(and symmetrically to justify $r1 = b$)

Thread 1

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

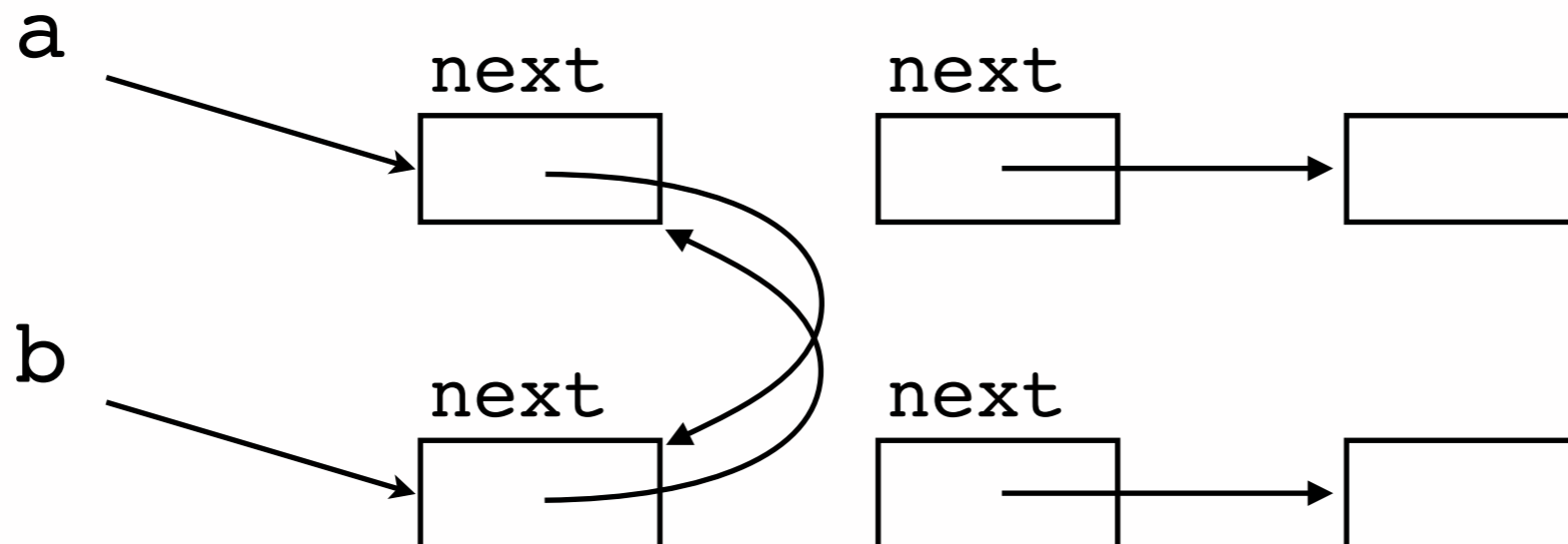
Thread 2

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



If the compiler speculates $r1=b$ and $r2=a$, then
the store $r1 \rightarrow \text{next}=a$ justifies $r2=b \rightarrow \text{next}$ assigning $r2=a$
(and symmetrically to justify $r1=b$)

Break our basic intuitions about memory and sharing!



Breaking news



20^H
WEEK-END

**Common compiler optimisations
are unsound in C11**

`x = y = a = 0`

<code>if (x.load(rlx)==42)</code>		<code>if (y.load(rlx)==42)</code>		<code>a = 1</code>
<code> y.write(42,rlx)</code>		<code> if (a==1)</code>		
		<code> x.write(42,rlx)</code>		

`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.

$x = y = a = 0$

<pre>if (x.load(rlx)==42) y.write(42,rlx)</pre>		<pre>if (y.load(rlx)==42) if (a==1) x.write(42,rlx)</pre>		<pre>a = 1</pre>
---	--	---	--	------------------

Remark 2

$a = 1 \wedge x = y = 0$

is the only possible final state

`x = y = a = 0`

<code>if (x.load(rlx)==42)</code>		<code>if (y.load(rlx)==42)</code>		<code>a = 1</code>
<code> y.write(42,rlx)</code>		<code> if (a==1)</code>		
		<code> x.write(42,rlx)</code>		

Consider sequentialisation:

`C || D` \implies `C ; D`

(ought to be correct)

x = y = a = 0

<pre>if (x.load(rlx)==42) y.write(42,rlx)</pre>	<pre>if (y.load(rlx)==42) if (a==1) x.write(42,rlx)</pre>	<pre>a = 1</pre>
---	---	------------------



<pre>if (x.load(rlx)==42) y.write(42,rlx)</pre>	<pre>a = 1 if (y.load(rlx)==42) if (a==1) x.write(42,rlx)</pre>
---	---

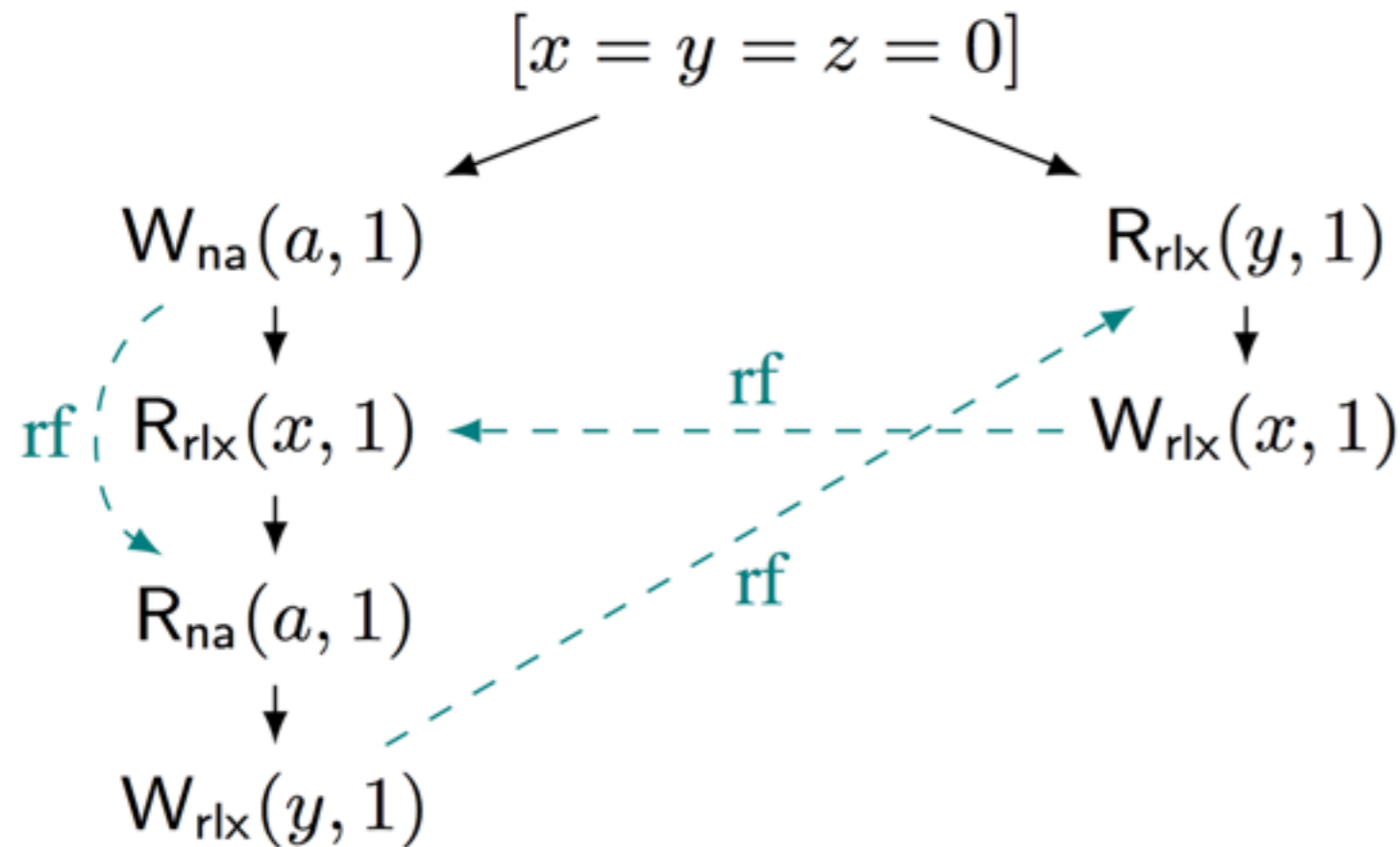
```
x = y = a = 0
```

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

`x = y = a = 0`

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

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



`a = 1`

`x = y = 42`
is also possible

`x = y = a = 0`

```
if (x.load(r1x) == 42) | a = 1
    y.write(42, r1x)    | if (y.load(r1x) == 42)
                        |     if (a == 1)
                        |         x.write(42, r1x)
```

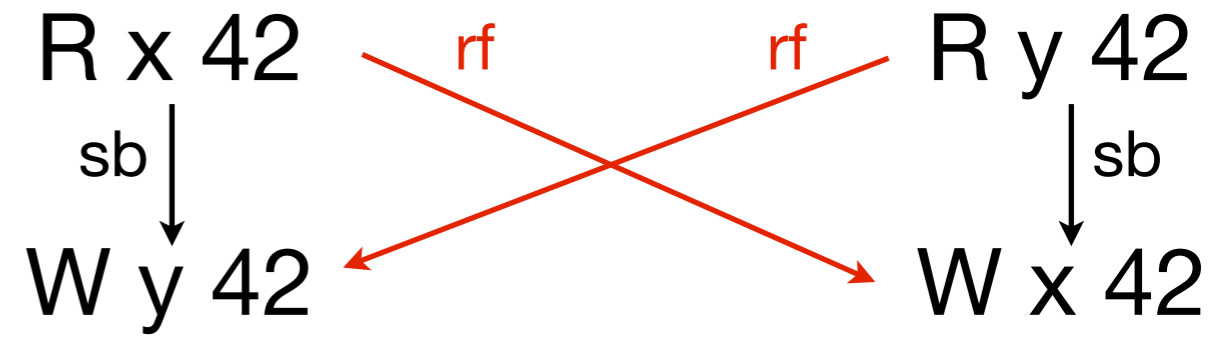
Break common source-to-source (or LLVM IR - to - LLVM IR) compiler optimisations

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

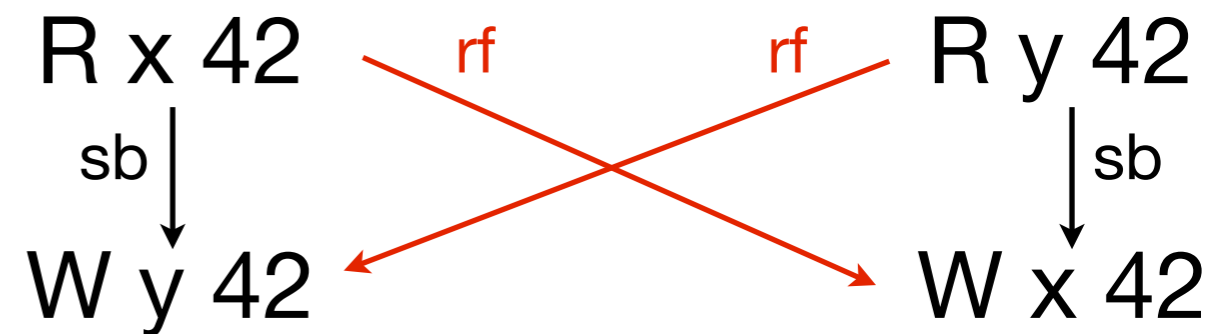


Are there any solutions?

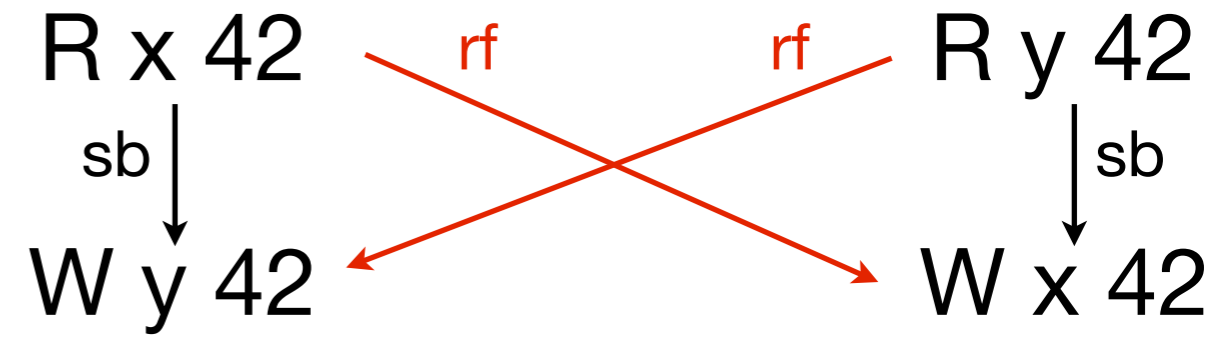
Thread 0	Thread 1
$r1 = x$ $y = r1$	$r2 = y$ $x = 42$



Thread 0	Thread 1
$r1 = x$ $y = r1$	$r2 = y$ $x = r2$

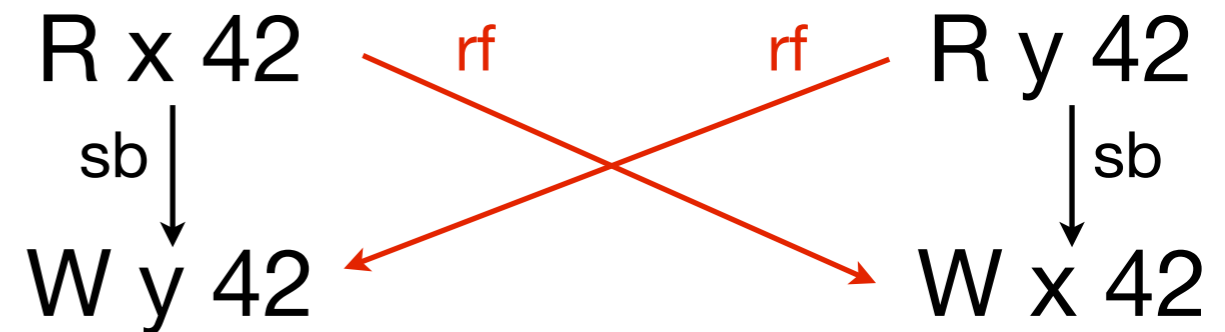


Thread 0	Thread 1
<code>r1 = x</code> <code>y = r1</code>	<code>r2 = y</code> <code>x = 42</code>

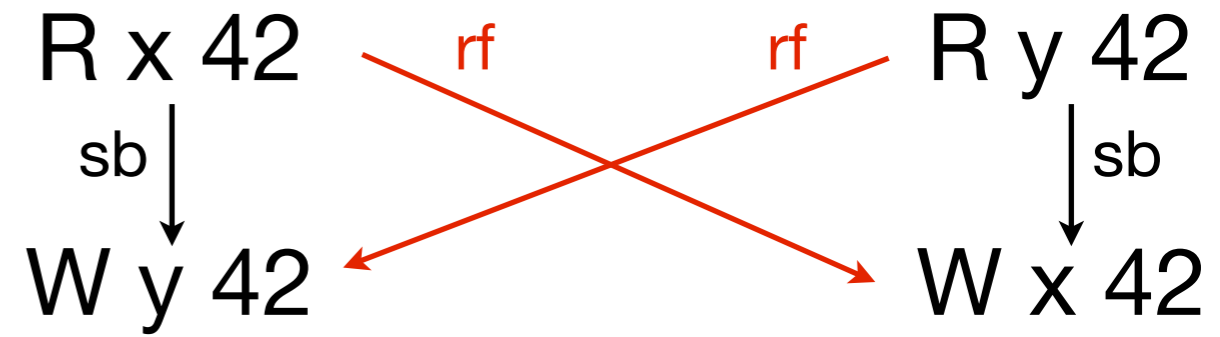


`r1 = r2 = 42.` Can you spot the difference?

Thread 0	Thread 1
<code>r1 = x</code> <code>y = r1</code>	<code>r2 = y</code> <code>x = r2</code>

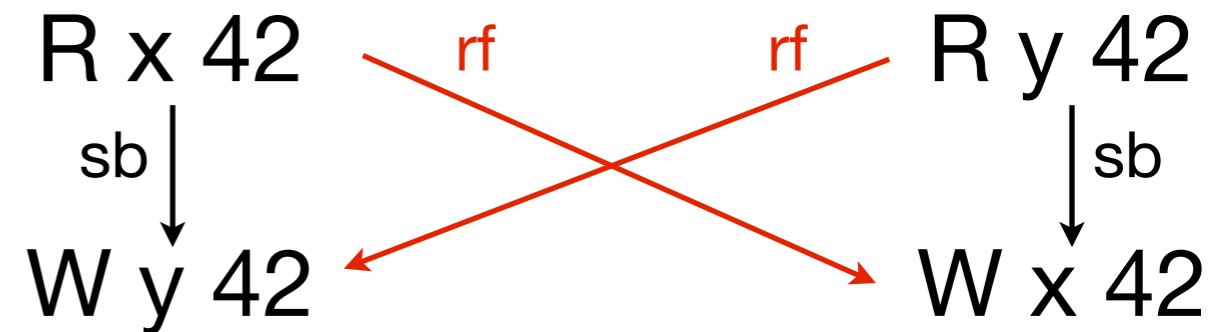


Thread 0	Thread 1
$r1 = x$ $y = r1$	$r2 = y$ $x = 42$



The “bad” example has a *cycle of dependencies*.

Thread 0	Thread 1
$r1 = x$ $y = r1$	$r2 = y$ $x = r2$

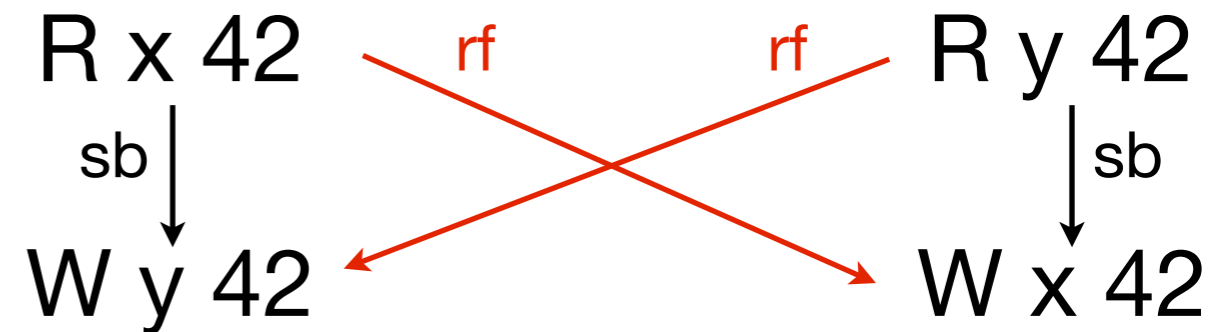


Solution 1.

Prohibit executions with dependency cycles

The “bad” example has a *cycle of dependencies*.

Thread 0	Thread 1
<code>r1 = x</code> <code>y = r1</code>	<code>r2 = y</code> <code>x = r2</code>



**Compiler writers
do not want to track all dependencies**

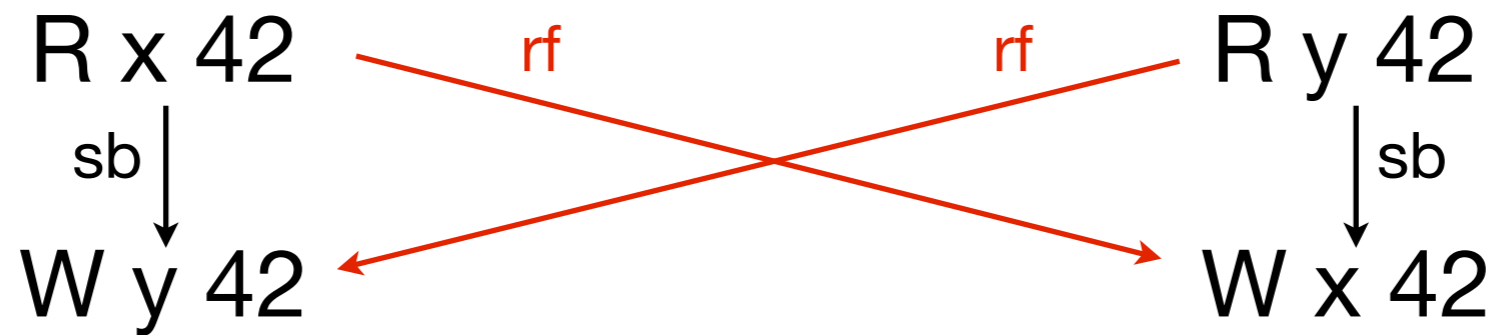
Compiler writers do not want to track all dependencies

```
if (x)
    a[i++] = 1;
else
    a[i++] = 2;
```

Does the store to *i* depend on the load of *x*?

Solution 2. Brute force

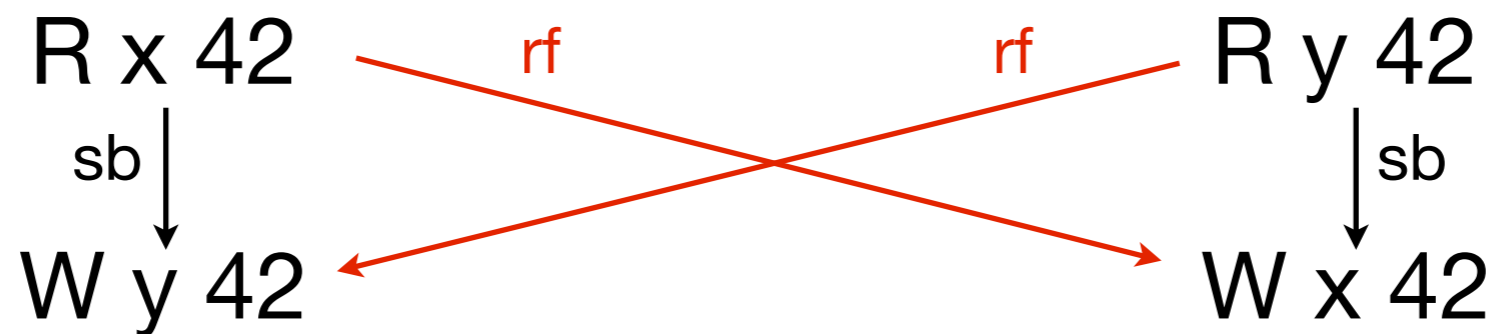
Disallow cycles altogether



$\text{acyclic}(\text{hb} \cup \{(a, b) \mid rf(b) = a\})$

Allows all source-to-source optimisations
(except for r/w reordering on atomics)
but expensive on ARM and GPUs

Disallow cycles altogether



$\text{acyclic}(\text{hb} \cup \{(a, b) \mid rf(b) = a\})$

Solution 3. less brute force

Allow cycles but make this racy
by allowing a to read 1

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


Efficient implementation of atomics on ARM/GPUs but all R/W reorderings are unsound

Allow cycles but make this racy
by allowing a to read 1

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

State of the art

“Implementations should ensure that no “out-of-thin-air” values are computed that circularly depend on their own computation.”

Current proposal for C14



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.

Still, many open problems...

attempt

els.

mmers

se

problems.



The memory models of modern hardware are better understood.

Still, many research opportunities!

attempt

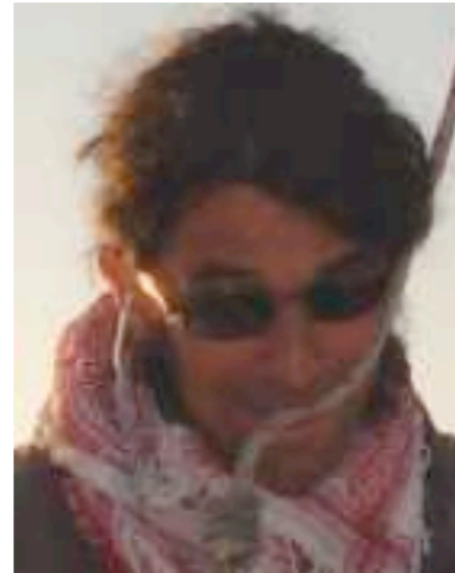
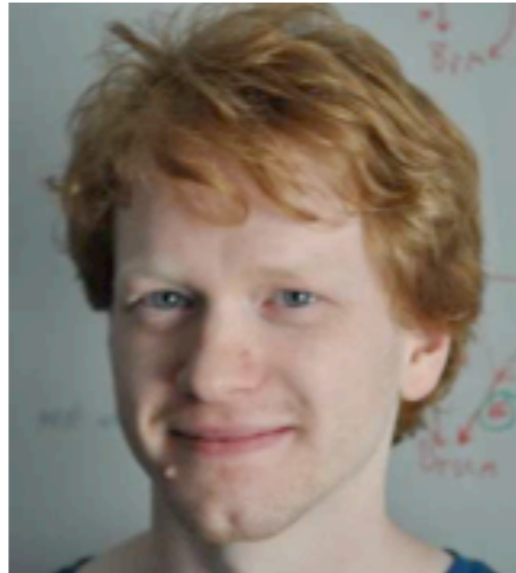
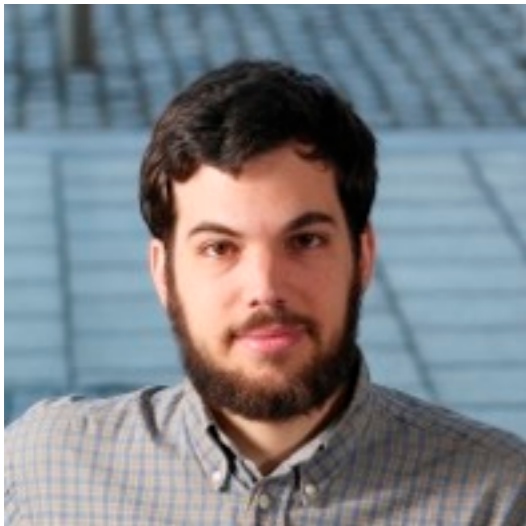
els.

mmers

se

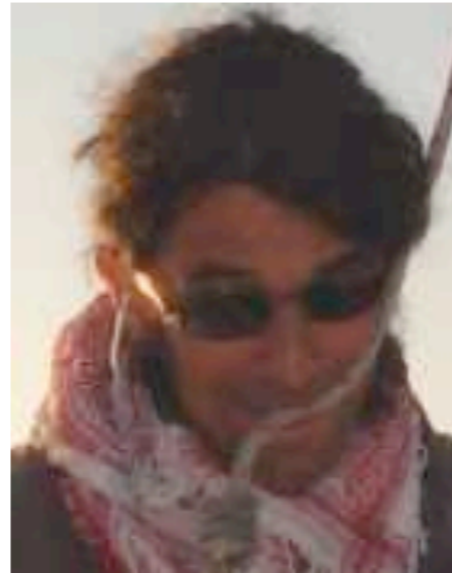
problems.

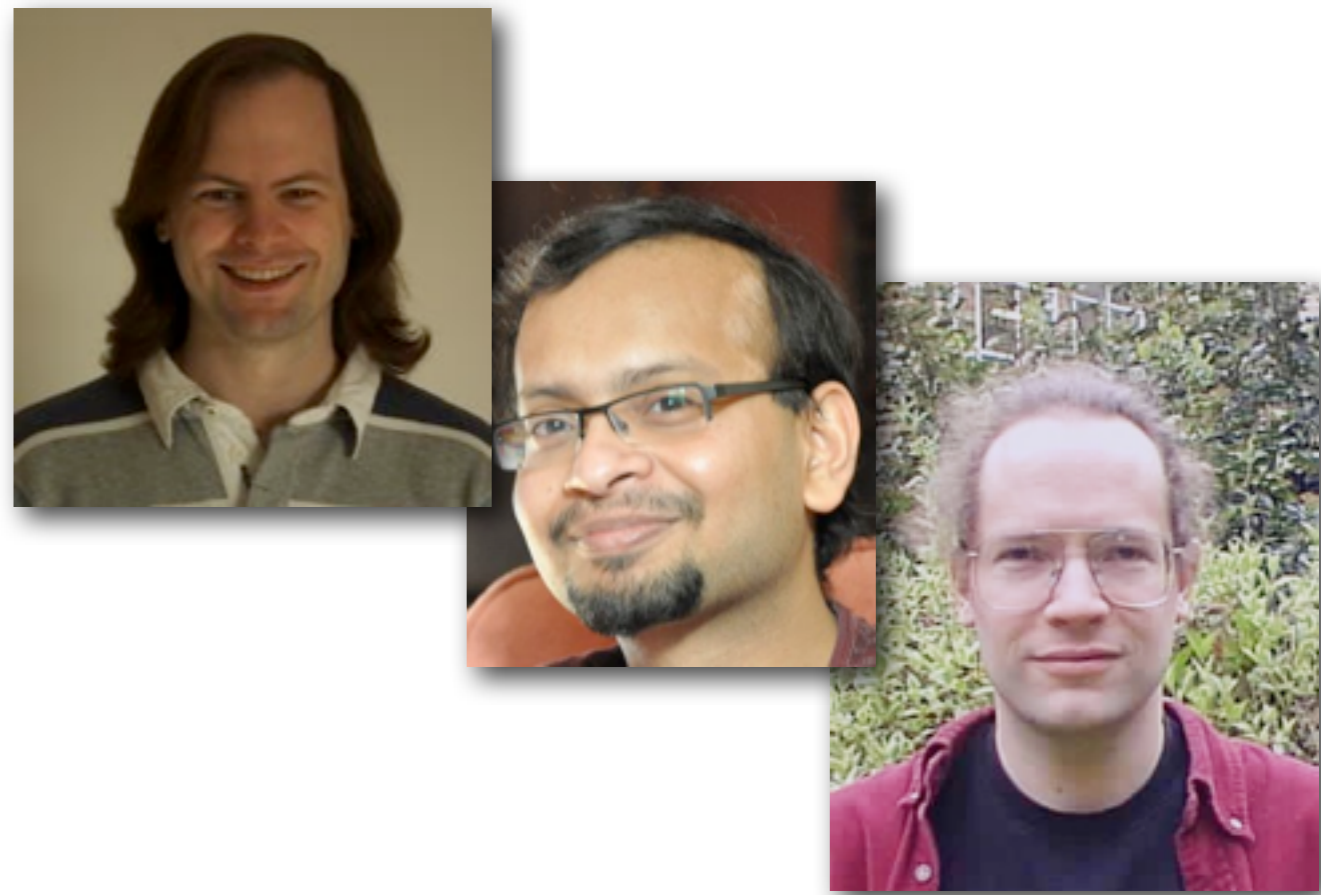
All these lectures are based on work done with/by my colleagues. Thank you!



And thank you all for attending these lectures!

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





4. Sketch of an operational formalisation of x86-TSO

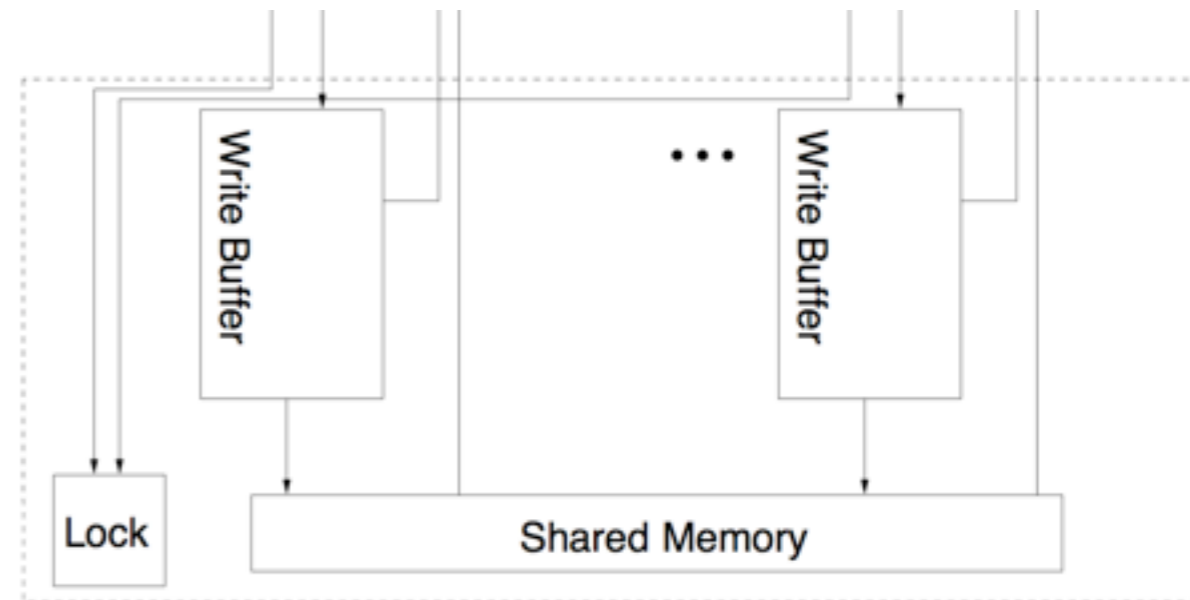
...starting with a formalisation of SC

Separate language and memory semantics

```
1  class ArrayWrapper
2  {
3      public:
4          ArrayWrapper (int n)
5              : _p_vals( new int[ n ] )
6              , _size( n )
7          {}
8          // copy constructor
9          ArrayWrapper (const ArrayWrapper& other)
10             : _p_vals( new int[ other._size ] )
11             , _size( other._size )
12         {
13             for ( int i = 0; i < _size; ++i )
14             {
15                 _p_vals[ i ] = other._p_vals[ i ];
16             }
17         }
18         ~ArrayWrapper ()
19         {
20             delete [] _p_vals;
21         }
22     private:
23         int *_p_vals;
24         int _size;
25 };
```

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

Separate language and state semantics
proved to be a very good choice
in many (unrelated) projects I worked on!

```
1  class Arr
2  {
3      public
4      A
5
6
7      {
8      /
9      A
10
11
12      {
13
14
15
16
17      }
18      ~
19      {
20
21      }
22      priva
23      int *
24      int
25  };
```

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

<i>expression, e</i>	::=	expression	
		<i>n</i>	integer literal
		<i>*x</i>	read from pointer
		<i>*x = e</i>	write to pointer
		<i>e; e'</i>	sequential composition
		<i>e + e'</i>	plus

<i>process, p</i>	::=	process	
		<i>t:e</i>	thread
		<i>p p'</i>	parallel composition

What can a thread do in isolation?

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

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

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

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

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

$$\frac{e_1 \xrightarrow{l} e'_1}{e_1; e_2 \xrightarrow{l} e'_1; e_2} \quad \text{SEQ_CONTEXT}$$

$$\frac{e_1 \xrightarrow{l} e'_1}{e_1 + e_2 \xrightarrow{l} e'_1 + e_2} \quad \text{PLUS_CONTEXT_1}$$

$$\frac{e_2 \xrightarrow{l} e'_2}{n_1 + e_2 \xrightarrow{l} n_1 + e'_2} \quad \text{PLUS_CONTEXT_2}$$

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

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

Example

Show that the expression:

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

can perform the following trace:

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

Lifting to processes

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

$$\frac{e \xrightarrow{l} e'}{t:e \xrightarrow{l_t} t:e'} \quad \text{THREAD}$$
$$\frac{p_1 \xrightarrow{l_t} p'_1}{p_1|p_2 \xrightarrow{l_t} p'_1|p_2} \quad \text{PAR_CONTEXT_LEFT}$$
$$\frac{p_2 \xrightarrow{l_t} p'_2}{p_1|p_2 \xrightarrow{l_t} p_1|p'_2} \quad \text{PAR_CONTEXT_RIGHT}$$

Actions are labelled by the thread that performed the action.

Free interleaving.

A sequentially consistent memory

Take M to be a function from addresses to integers.

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

$$\frac{M(x) = n}{M \xrightarrow{R\ x=n} M} \quad \text{MREAD}$$

$$\frac{}{M \xrightarrow{W\ x=n} M \oplus (x \mapsto n)} \quad \text{MWRITE}$$

SC semantics: whole system transitions

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

$$\frac{\begin{array}{c} p \xrightarrow{R_t x=n} p' \\ M \xrightarrow{R x=n} M' \end{array}}{\langle p, M \rangle \xrightarrow{R_t x=n} \langle p', M' \rangle} \quad \text{SREAD}$$

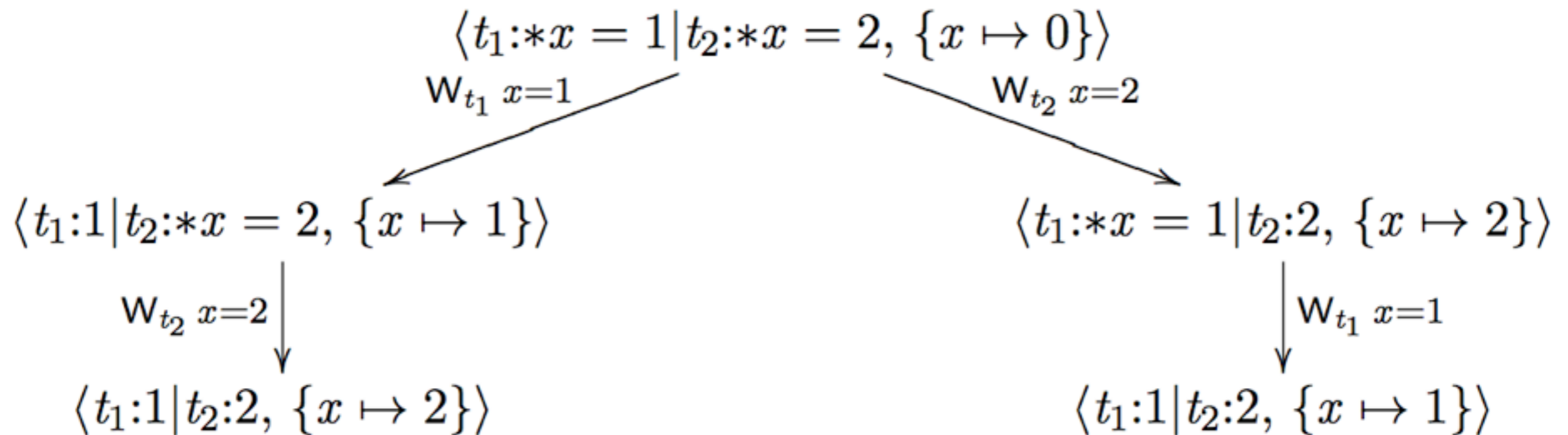
Synchronising between the processes and the memory.

$$\frac{\begin{array}{c} p \xrightarrow{W_t x=n} p' \\ M \xrightarrow{W x=n} M' \end{array}}{\langle p, M \rangle \xrightarrow{W_t x=n} \langle p', M' \rangle} \quad \text{SWRITE}$$

$$\frac{p \xrightarrow{\tau_t} p'}{\langle p, M \rangle \xrightarrow{\tau_t} \langle p', M \rangle} \quad \text{STAU}$$

SC semantics, example

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



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

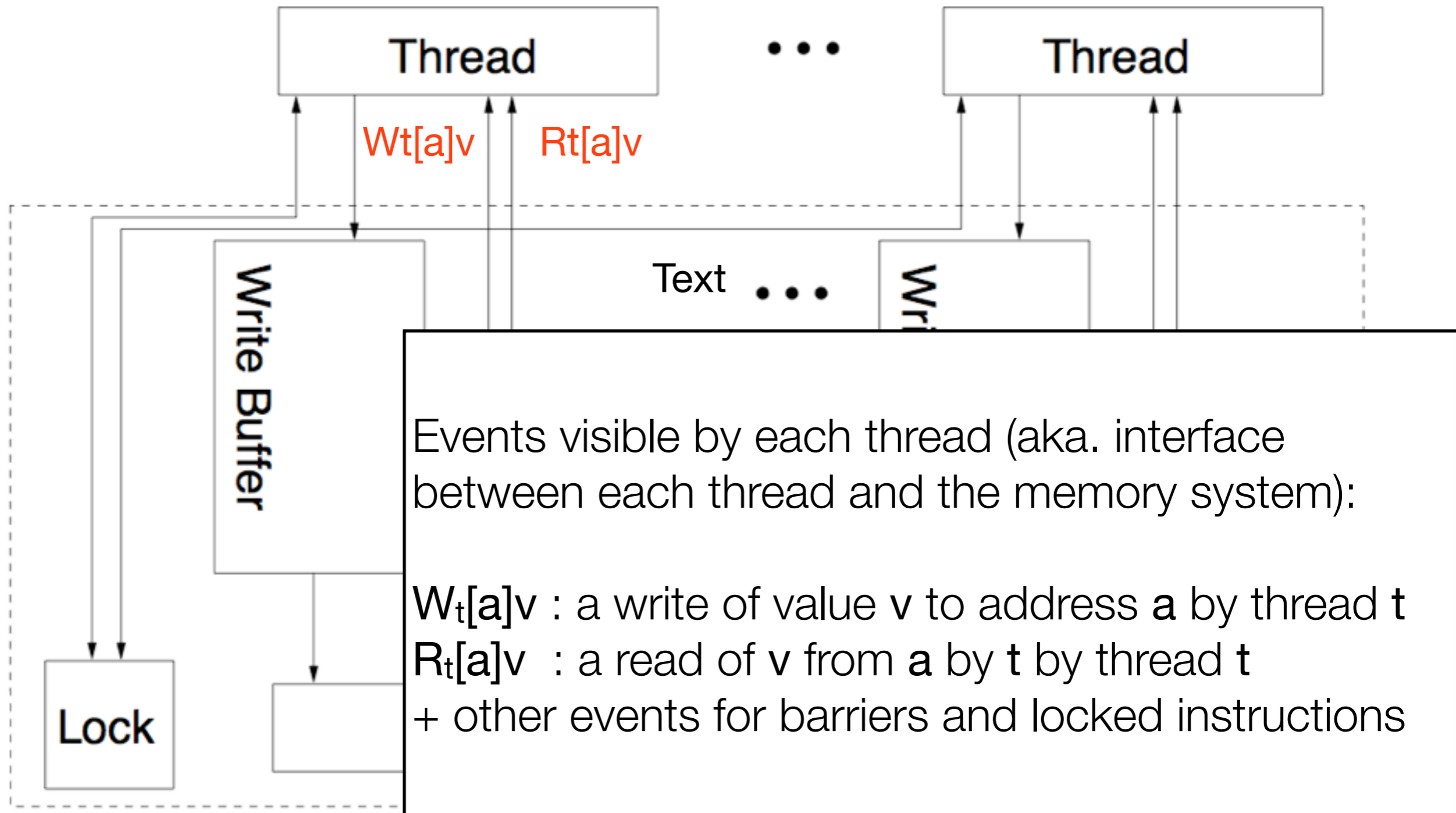
Take M to be a function from addresses to integers.

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

$$\frac{M(x) = n}{M \xrightarrow{R\ x=n} M} \quad \text{MREAD}$$

$$\frac{}{M \xrightarrow{W\ x=n} M \oplus (x \mapsto n)} \quad \text{MWRITE}$$

x86-TSO abstract machine



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.

ues

x86-tso: a formalisation using an LTS

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

M : address \rightarrow value option

B : tid \rightarrow (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

$\text{not_blocked}(s, t)$

$s.M(x) = v$

$\text{no_pending}(s.B(t), x)$

$s \xrightarrow{R_t x=v} s$

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

$\text{not_blocked}(s, t)$

$\exists b_1 b_2. s.B(t) = b_1 \text{ ++ } [(x, v)] \text{ ++ } b_2$

$\text{no_pending}(b_1, x)$

$$s \xrightarrow{R_t x=v} s$$

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

$$s \xrightarrow{W_t x=v} s \oplus \langle B := s.B \oplus (t \mapsto ([x, v] ++ s.B(t))) \rangle$$

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

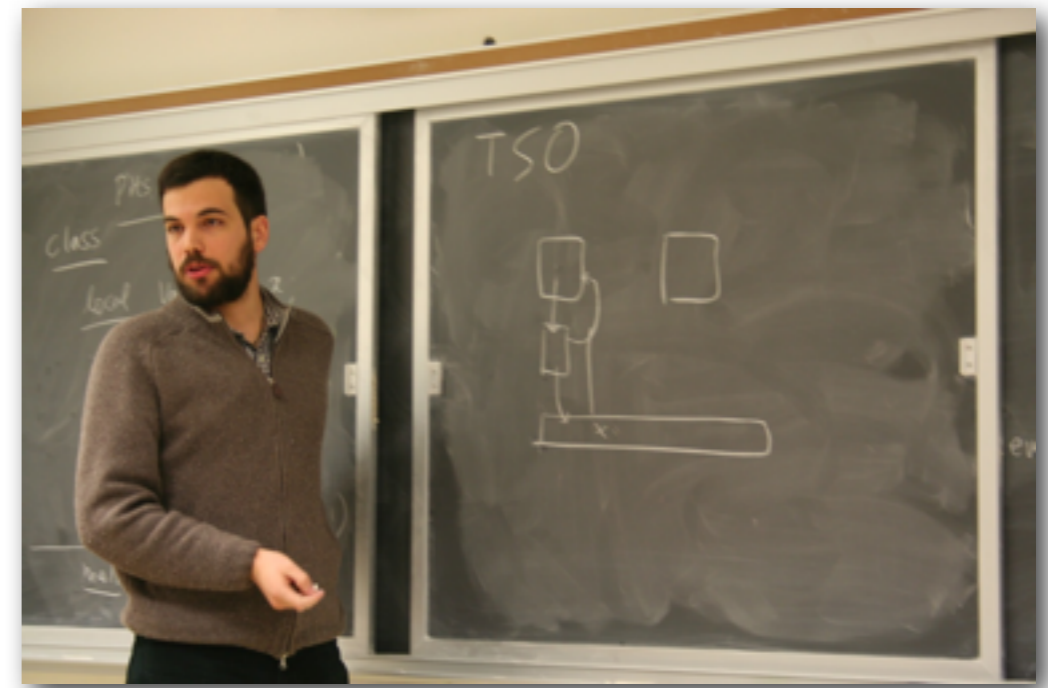
WM: Write from write buffer to memory

not_blocked(s, t)

$s.B(t) = b ++ [x, v]$

$$s \xrightarrow{\tau_t x=v} s \oplus \langle M := s.M \oplus (x \mapsto v) \rangle \oplus \langle B := s.B \oplus (t \mapsto b) \rangle$$

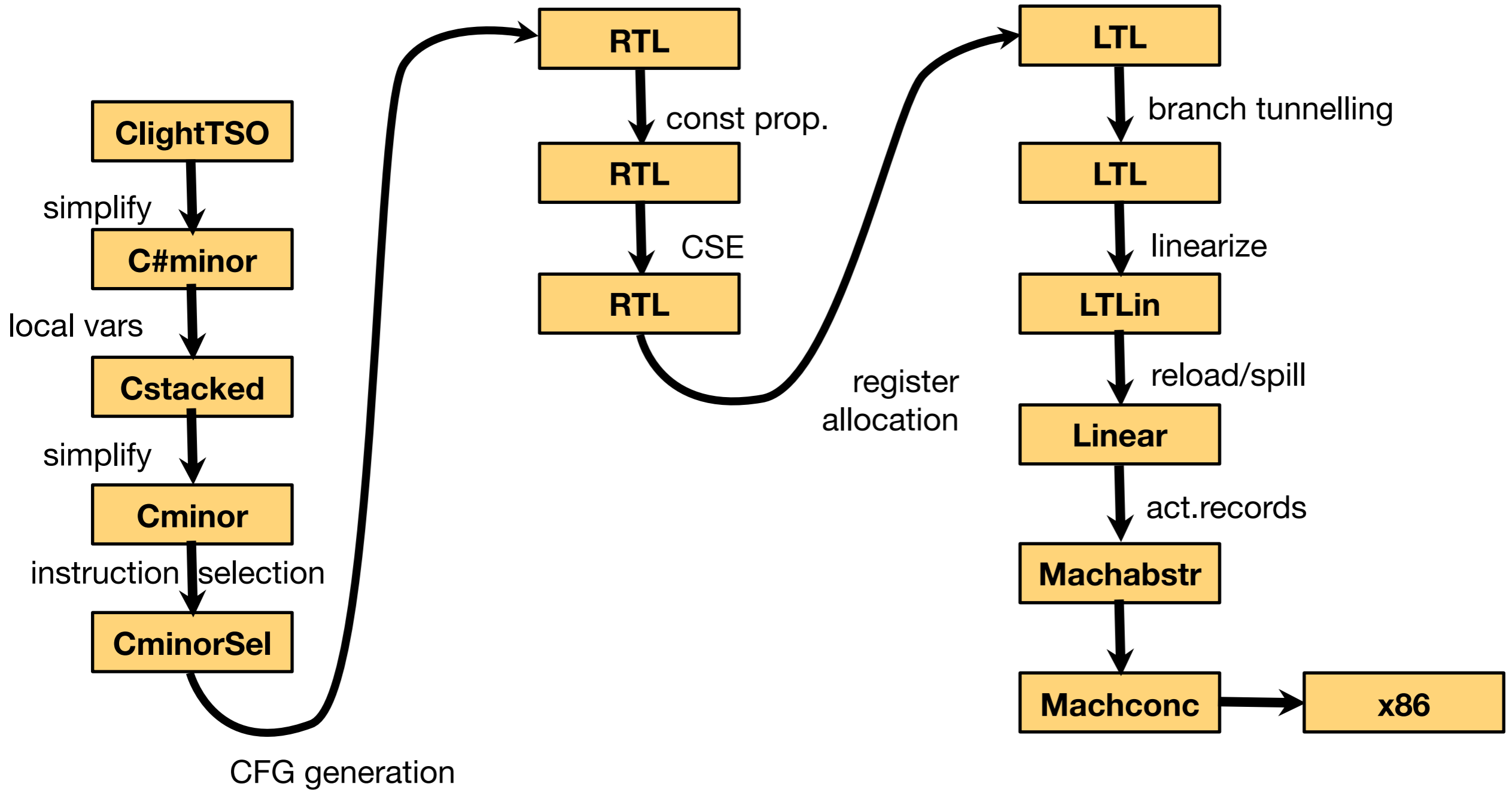
If t is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread



5. Verifying fence elimination optimisations

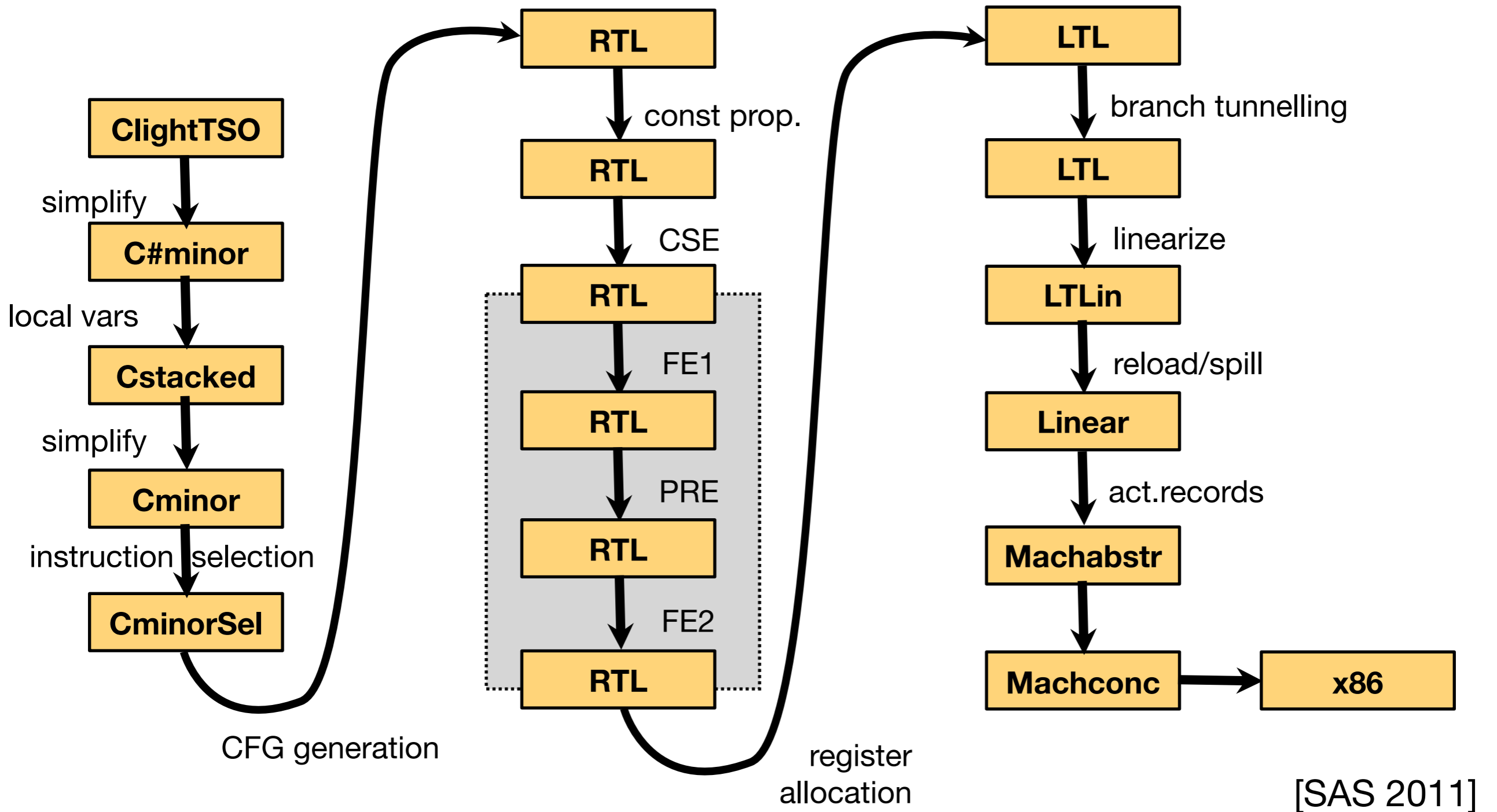
aka reasoning on the x86TSO operational memory model
and compiler correctness

CompCertTSO



[POPL 2011]

CompCertTSO + fence optimisations



Compilers are *ideal* for verification



Compilers are:

- Basic computing infrastructure
- Generally reliable, but nevertheless contain many bugs
e.g., Yang et al. [PLDI 2011] found 79 `gcc` & 202 `llvm` bugs
- “Specifiable”: compiler correctness = preservation of behaviours
- Interesting: naturally higher-order, involve clever algorithms
- Big, but modular

Language semantics

The semantics of all the CompCertTSO languages is defined by:

- a type of programs, *prg*
- a type of states, *states*
- a set of initial states for each program, $\text{init} \in \text{prg} \rightarrow \mathbb{P}(\text{states})$
- a transition relation, $\rightarrow \in \mathbb{P}(\text{states} \times \text{event} \times \text{states})$

`call, return, fail, oom, τ`

The visible behaviour of a program is defined by the external function calls (`call`) and returns (`return`), errors (`fail`), and running out of memory (`oom`).

Traces

- *Finite sequences* of call & return events ending with:
 - end**: successful termination,
 - inftau**: infinite execution that stops performing visible events
 - oom**: execution runs out of memory
- *Infinite sequences* of call & return events;

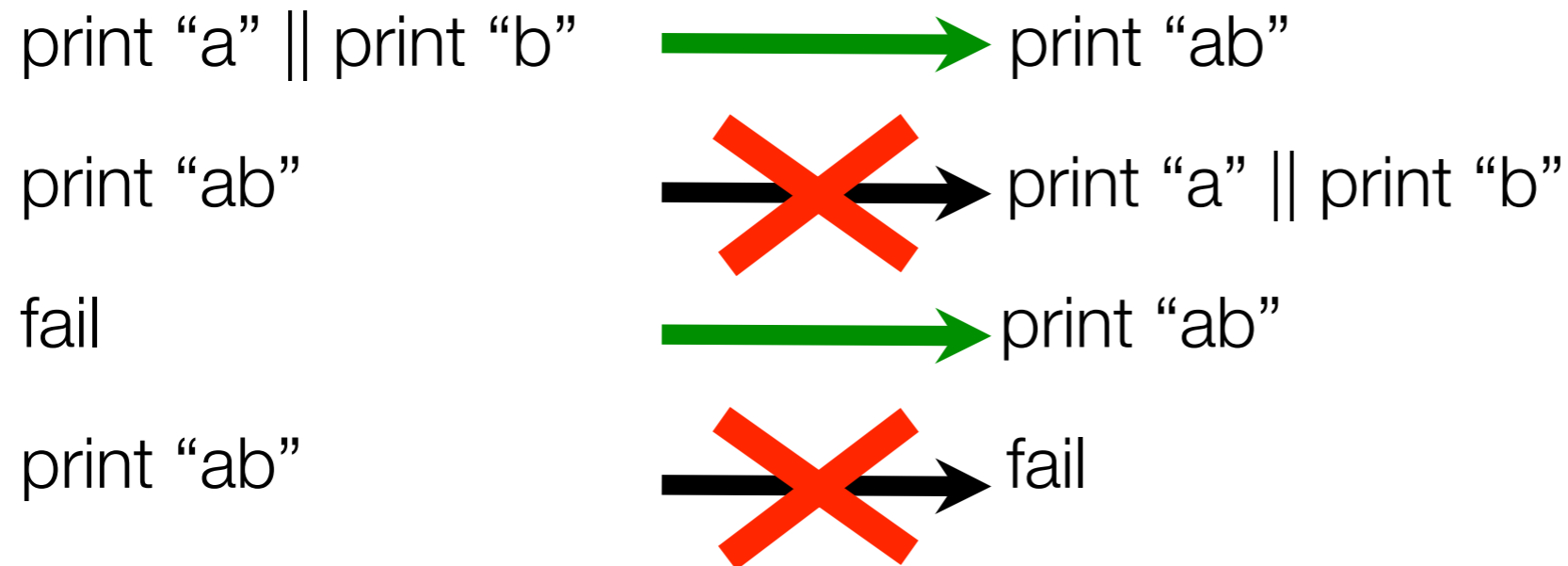
$$\begin{aligned} \text{traces}(p) \stackrel{\text{def}}{=} & \{ \ell \cdot \text{end} \mid \exists s \in \text{init}(p). \exists s'. s \xRightarrow{\ell} s' \wedge s' \not\rightarrow \} \\ & \cup \{ \ell \cdot tr \mid \exists s \in \text{init}(p). \exists s'. s \xrightarrow{\ell \cdot \text{fail}} s' \} \\ & \cup \{ \ell \cdot \text{inftau} \mid \exists s \in \text{init}(p). \exists s'. s \xRightarrow{\ell} s' \wedge \text{inftau}(s') \} \\ & \cup \{ \ell \cdot \text{oom} \mid \exists s \in \text{init}(p). \exists s'. s \xRightarrow{\ell} s' \} \\ & \cup \{ tr \mid \exists s \in \text{init}(p). s \text{ can do the infinite trace } tr \} \end{aligned}$$

NB: Erroneous computations become undefined after the first error.

Compiler correctness



$$\text{traces}(\text{source_program}) \supseteq \text{traces}(\text{target_program})$$



Fence instructions prevent hardware reorderings

E.g., on x86-TSO:

$[x]=[y]=0$

Thread 0	Thread 1
MOV $[x] \leftarrow 1$	MOV $[y] \leftarrow 1$
MOV $EAX \leftarrow [y]$	MOV $EBX \leftarrow [x]$

$EAX = EBX = 0$
allowed

$[x]=[y]=0$

Thread 0	Thread 1
MOV $[x] \leftarrow 1$	MOV $[y] \leftarrow 1$
MFENCE	MFENCE
MOV $EAX \leftarrow [y]$	MOV $EBX \leftarrow [x]$

$EAX = EBX = 0$
forbidden

Who inserts fences?

1. The *programmer*, explicitly. Example: Fraser's lockfree-lib:

```
/*
 * II. Memory barriers.
 * MB(): All preceding memory accesses must commit before any later accesses.
 *
 * If the compiler does not observe these barriers (but any sane compiler
 * will!), then VOLATILE should be defined as 'volatile'.
 */
#define MB() __asm__ __volatile__ ("lock; addl $0,0(%%esp)" : : : "memory")
```

2. The *compiler*, to implement a high-level memory model,
e.g. `SEQ_CST` C++0x low-level atomics on x86:

Load `SEQ_CST`: `MFENCE; MOV`

Store `SEQ_CST`: `MOV; MFENCE`

Fence instructions

1. *Fences are necessary*

to implement locks & not fully-commutative linearizable objects (e.g., stacks, queues, sets, maps).

[Attiya et al., POPL 2011]

2. *Fences can be expensive*


Redundant fences (1)

If we have two consecutive fence instructions, we can remove the *latter*:

MFENCE		MFENCE
MFENCE		NOP

The *buffer is already empty* when the second fence is executed.

Generalisation:

MFENCE		MFENCE
NON-WRITE INSTR		NON-WRITE INSTR
...		...
NON-WRITE INSTR		NON-WRITE INSTR
MFENCE		NOP

FE1

A fence is redundant if it always follows a previous fence or locked instruction in program order, and no memory store instructions are in between.

A *forward* data-flow problem over the boolean domain $\{\perp, \top\}$

Associate to each program point:

\perp : along all execution paths there is an atomic instruction *before* the current program point, with no intervening writes;

\top : otherwise.

$T_1(\text{nop}, \mathcal{E})$	$= \mathcal{E}$
$T_1(\text{op}(op, \vec{r}, r), \mathcal{E})$	$= \mathcal{E}$
$T_1(\text{load}(\kappa, addr, \vec{r}, r), \mathcal{E})$	$= \mathcal{E}$
$T_1(\text{store}(\kappa, addr, \vec{r}, src), \mathcal{E})$	$= \top$
$T_1(\text{call}(sig, ros, args, res), \mathcal{E})$	$= \top$
$T_1(\text{cond}(cond, args), \mathcal{E})$	$= \mathcal{E}$
$T_1(\text{return}(optarg), \mathcal{E})$	$= \top$
$T_1(\text{threadcreate}(optarg), \mathcal{E})$	$= \top$
$T_1(\text{atomic}(aop, \vec{r}, r), \mathcal{E})$	$= \perp$
$T_1(\text{fence}, \mathcal{E})$	$= \perp$

$$\mathcal{FE}_1(n) = \begin{cases} \top & \text{if predecessors}(n) = \emptyset \\ \bigsqcup_{p \in \text{predecessors}(n)} T_1(\text{instr}(p), \mathcal{FE}_1(p)) & \text{otherwise} \end{cases}$$

FE1

A fence is redundant if it always follows a previous fence or locked instruction in program order, and no memory store instructions are in between.

A forward data-flow problem over

$T_1(\text{nop}, \mathcal{E})$

the bo

Assoc

\perp : alo

is a

cur

no

\top : oth

Implementation:

1. Use CompCert implementation of Kildall algorithm to solve the data-flow equations.
2. Replace **MFENCES** for which the analysis returns \perp with **NOP** instructions.

= \mathcal{E}
 = \mathcal{E}
 = \mathcal{E}
 = \top
 = \top
 = \top
 = \mathcal{E}
 = \top
 = \top
 = \perp
 = \perp

$$\mathcal{FE}_1(n) = \begin{cases} \bigsqcup_{p \in \text{predecessors}(n)} T_1(\text{instr}(p), \mathcal{FE}_1(p)) & \text{otherwise} \end{cases}$$

\emptyset

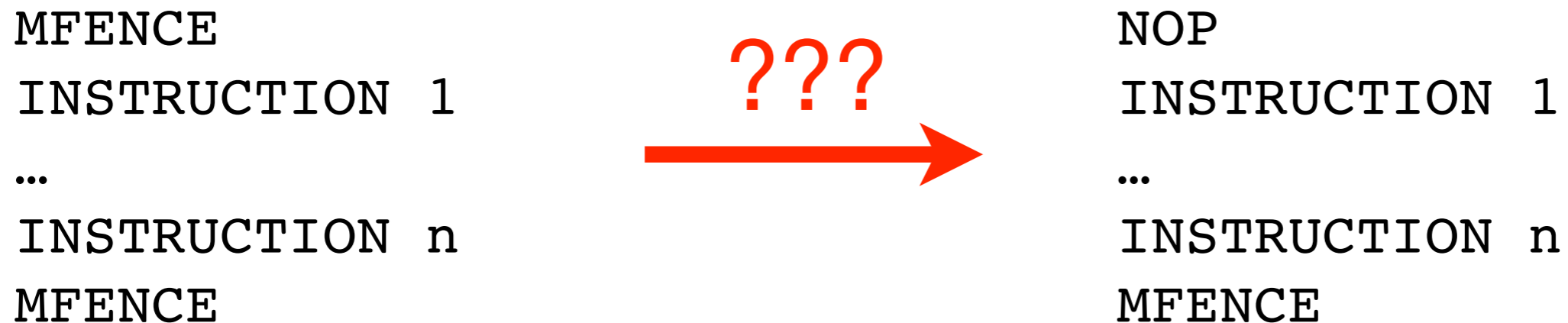
Redundant fences (2)

If we have two consecutive fence instructions, we can remove the *former*:



Intuition: the visible effects initially published by the former fence, are now published by the latter, and nobody can tell the difference.

Generalisation:



Redundant fences (2)

If there are reads in between the fences...

$[x]=[y]=0$

Thread 0	Thread 1
MOV [x] ← 1 MFENCE MOV EAX ← [y] MFENCE	MOV [y] ← 1 MFENCE MOV EBX ← [x]

EAX = EBX = 0
forbidden

but

$[x]=[y]=0$

Thread 0	Thread 1
MOV [x] ← 1 NOP MOV EAX ← [y] MFENCE	MOV [y] ← 1 MFENCE MOV EBX ← [x]

EAX = EBX = 0
allowed

Redundant fences (2)

If there are reads in between the fences...

[x]=[y]=0

Thread 0	Thread 1
MOV [x] ← 1	MOV [y] ← 1
MFENCE	MFENCE
MOV EAX ← [y]	
MFENCE	

EAX = EBX = 0
forbidden

but

If there are reads in between, the optimisation is unsound.

[x]=[y]=0

Thread 0	Thread 1
MOV [x] ← 1	MOV [y] ← 1
NOP	MFENCE
MOV EAX ← [y]	MOV EBX ← [x]
MFENCE	

EAX = EBX = 0
allowed

Redundant fences (2)

Swapping a `STORE` and a `MFENCE` is sound:

`MFENCE; STORE`  `STORE; MFENCE`

1. transformed program's behaviours \subseteq source program's behaviours
(source program might leave pending write in its buffer)
2. There is the new intermediate state if the buffer was initially non-empty, but this intermediate state *is not observable*.
(a local read is needed to access the local buffer)

Intuition: Iterate this swapping...

FE2

A fence is redundant if it always precedes a later fence or locked instruction in program order, and no memory read instructions are in between.

A *backward* data-flow problem over the boolean domain $\{\perp, \top\}$

Associate to each program point:

\perp : along all execution paths there is an atomic instruction *after* the current program point, with no intervening reads;

\top : otherwise.

$T_2(\mathbf{nop}, \mathcal{E})$	$= \mathcal{E}$
$T_2(\mathbf{op}(op, \vec{r}, r), \mathcal{E})$	$= \mathcal{E}$
$T_2(\mathbf{load}(\kappa, addr, \vec{r}, r), \mathcal{E})$	$= \top$
$T_2(\mathbf{store}(\kappa, addr, \vec{r}, src), \mathcal{E})$	$= \mathcal{E}$
$T_2(\mathbf{call}(sig, ros, args, res), \mathcal{E})$	$= \top$
$T_2(\mathbf{cond}(cond, args), \mathcal{E})$	$= \mathcal{E}$
$T_2(\mathbf{return}(optarg), \mathcal{E})$	$= \top$
$T_2(\mathbf{threadcreate}(optarg), \mathcal{E})$	$= \top$
$T_2(\mathbf{atomic}(aop, \vec{r}, r), \mathcal{E})$	$= \perp$
$T_2(\mathbf{fence}, \mathcal{E})$	$= \perp$

$$\mathcal{FE}_2(n) = \begin{cases} \top & \text{if successors}(n) = \emptyset \\ \bigsqcup_{s \in \text{successors}(n)} T_2(instr(s), \mathcal{FE}_2(s)) & \text{otherwise} \end{cases}$$

FE1 and FE2 are both useful

Removed by FE1 but not FE2:

```
MFENCE
MOV EAX <- [y]
MFENCE
MOV EBX <- [y]
```

Removed by FE2 but not FE1:

```
MOV [x] <- 1
MFENCE
MOV [x] <- 2
MFENCE
```

Informal correctness argument

Intuition: FE2 can be thought as iterating

MFENCE; STORE



STORE; MFENCE

MFENCE; non-mem



non-mem; MFENCE

and then applying

MFENCE; MFENCE



NOP; MFENCE

This argument works for *finite traces*, but not for *infinite traces* as the later fence might never be executed:

MFENCE;
STORE;
WHILE(1);
MFENCE



NOP;
STORE;
WHILE(1);
MFENCE

Basic simulations

A pair of relations

$$\sim \in \mathbb{P}(\text{src.states} \times \text{tgt.states}) \qquad > \in \mathbb{P}(\text{tgt.states} \times \text{tgt.states})$$

is a *basic simulation* for $\text{compile} : \text{src.prg} \rightarrow \text{tgt.prg}$ if:

$$\text{sim_init} : \forall p p'. \text{compile}(p) = p' \implies \forall t \in \text{init}(p'). \exists s \in \text{init}(p). s \sim t$$

$$\text{sim_end} : \forall s t. s \sim t \wedge t \not\rightarrow - \implies s \not\rightarrow -$$

$$\text{sim_step} : \forall s t t' ev. s \sim t \wedge t \xrightarrow{ev} t' \wedge ev \neq \text{oom} \implies$$

$$\begin{aligned} & (s \xrightarrow{\tau}^* \xrightarrow{\text{fail}} -) && \text{— } s \text{ reaches a failure} \\ \vee & (\exists s'. s \xrightarrow{\tau}^* \xrightarrow{ev} s' \wedge s' \sim t') && \text{— } s \text{ does matching step sequence} \\ \vee & (ev = \tau \wedge t > t' \wedge s \sim t'). && \text{— } s \text{ stutters (only allowed if } t > t') \end{aligned}$$

Exhibiting a basic simulation implies:

$$\text{traces}(\text{compile}(p)) \setminus \{t \cdot \text{inf}\tau \mid t \text{ trace}\} \subseteq \text{traces}(p)$$

“simulation can stutter forever”

Usual approach: measured simulations

Definition 2 (Measured sim.). *A measured simulation is any basic simulation $(\sim, >)$ such that $>$ is well-founded.*

Theorem 1. *If there exists a measured simulation for the compilation function `compile`, then for all programs p , $\text{traces}(\text{compile}(p)) \subseteq \text{traces}(p)$.*

Simulation for FE2

$s \equiv_i t$ iff thread i of s and t have identical pc, local states and buffers

$s \sim_i s'$ iff thread i of s can execute zero or more `NOP`, `OP`, `STORE` and `MFENCE` instructions and end in the state s'

$s \sim t$ iff

- t 's CFG is the optimised version of s 's CFG; and

- s and t have identical memories; and

- \forall thread i , either $s \equiv_i t$ or

the analysis for i 's pc returned \perp and $\exists s', s \sim_i s'$ and $s' \equiv_i t$

“ s is some instructions behind and can catch up”

Stutter condition:

$t > t'$ iff $t \rightarrow t'$ by a thread executing a `NOP`, `OP`, `STORE` or `MFENCE`
(and t 's buffer being non-empty)

Simulation for FE2

$s \equiv_i t$ iff thread i of s and t have identical pc, local states and buffers

$s \sim_i s'$ iff th

MFENCE

$s \sim t$ iff

– t 's CFG

– s and t

– \forall thread

But if (1) all threads have non-empty buffers, and
 (2) are stuck executing infinite loops, and
 (3) no writes are ever propagated to memory,
 then we can stutter forever.

(i.e., $>$ is not well-founded.)

the analysis for i 's pc returned \perp and $\exists s', s \sim_i s'$ and $s' \equiv_i t$

“ s is some instructions behind and can catch up”

Stutter condition:

$t > t'$ iff $t \rightarrow t'$ by a thread executing a NOP, OP, STORE OR MFENCE
 (and t 's buffer being non-empty)

Simulation for FE2

$s \equiv_i t$ iff thread i of s and t have identical pc, local states and buffers

$s \sim_i s'$ iff th

MFE

$s \sim t$ iff

– t 's CFG

– s and t

– \forall thread

But if (1) all threads have non-empty buffers, and
 (2) are stuck executing infinite loops, and
 (3) no writes are ever propagated to memory,
 then we can stutter forever.

Solution 1: Assume this case never arises (*fairness*)

Solution 2: Do a case split.

- If this case does not arise, we are done.
- If it does, use a different (weaker) simulation to construct an infinite trace for the source

Stutter condition

$t > t'$ iff $t \rightarrow$

(and

$i t$

Weaktau simulation

Definition 3 (Weaktau sim.). A weaktau simulation consists of a basic simulation $(\sim, >)$ with and an additional relation between source and target states, $\simeq \in \mathbb{P}(\text{src.states} \times \text{tgt.states})$ satisfying the following properties:

$$\text{sim_weaken} : \forall s, t. s \sim t \implies s \simeq t$$

$$\text{sim_wstep} : \forall s t t'. s \simeq t \wedge t \xrightarrow{\tau} t' \wedge t > t' \implies$$

$$\begin{aligned} & (s \xrightarrow{\tau^*} \text{fail}) && \text{— } s \text{ reaches a failure} \\ \vee & (\exists s'. s \xrightarrow{\tau^*} s' \wedge s' \simeq t') && \text{— } s \text{ does a matching step sequence.} \end{aligned}$$

Theorem 2. If there exists a weaktau-simulation $(\sim, >, \simeq)$ for the compilation function `compile`, then for all programs p , $\text{traces}(\text{compile}(p)) \subseteq \text{traces}(p)$.

Remarks:

- Once the simulation game moves from \sim to \simeq , stuttering is forbidden;
- Can view difference between \sim and \simeq as a boolean prophecy variable.

Weaktau simulation for FE2

$s \sim t$, $t > t'$ as before.

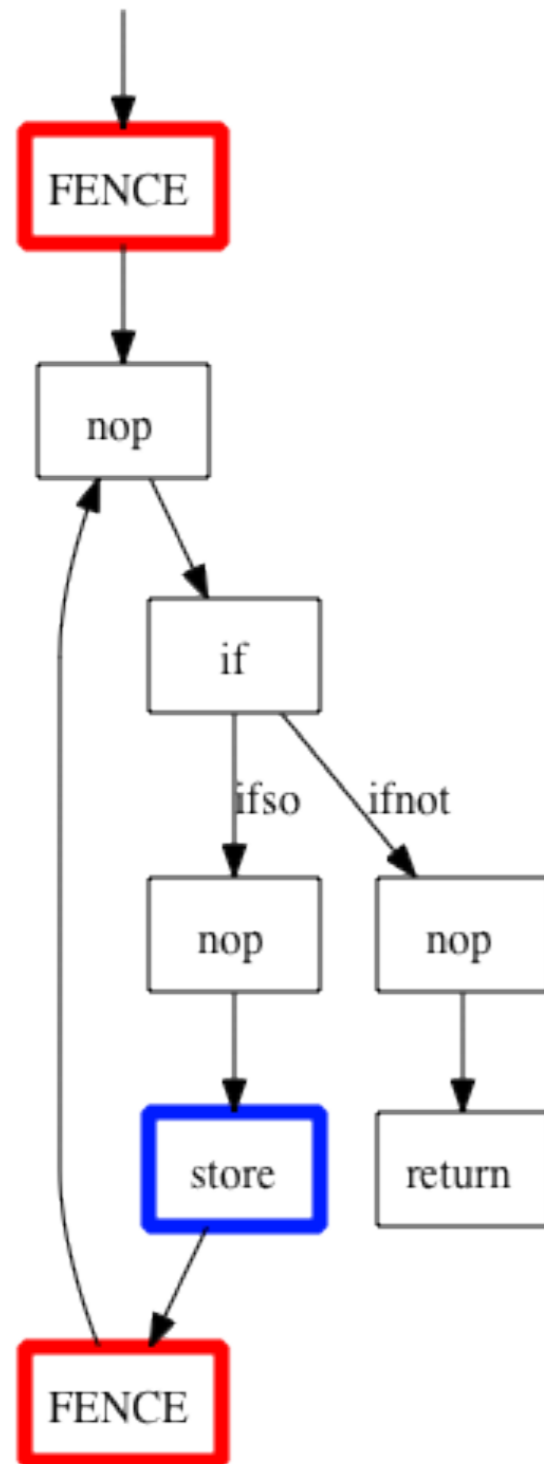
$s \simeq t$ iff

– t 's CFG is the optimised version of s 's CFG; and

– $\forall i, \exists s'$ s.t. $s \sim_i s' \equiv_i t$.

(i.e., same as $s \sim t$ except that the memories are unrelated.)

A closer look at the RTL

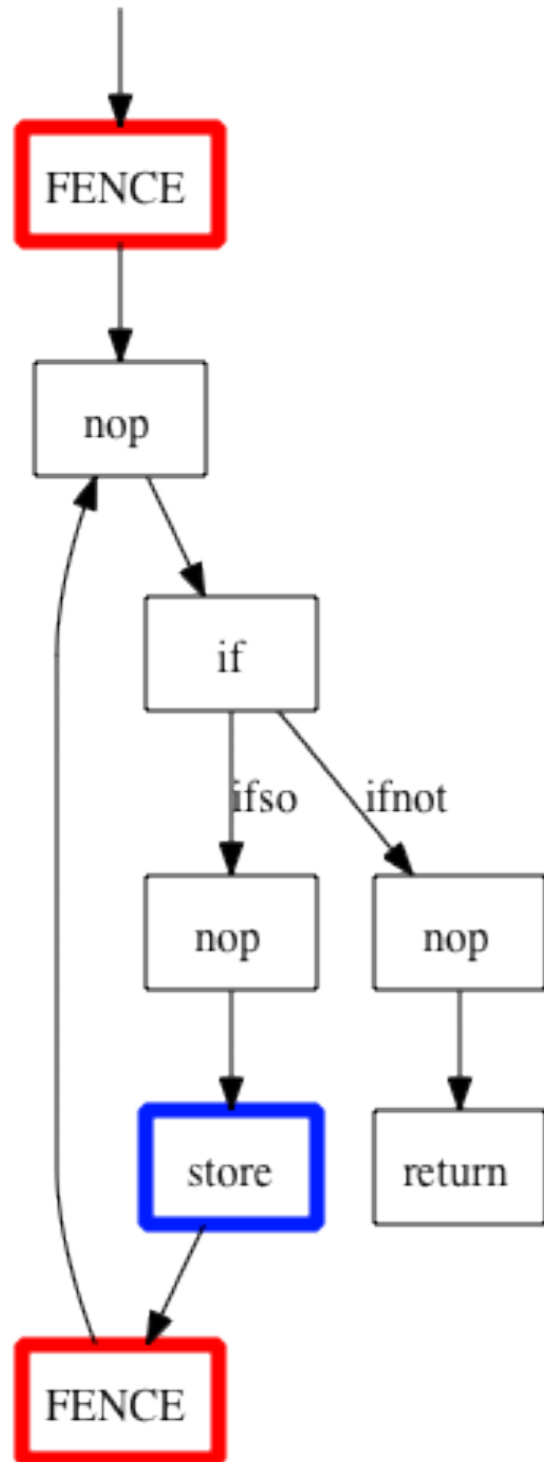


Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

It would be nice to hoist those fences out of the loop.

A closer look at the RTL



Patterns like that on the left are common.

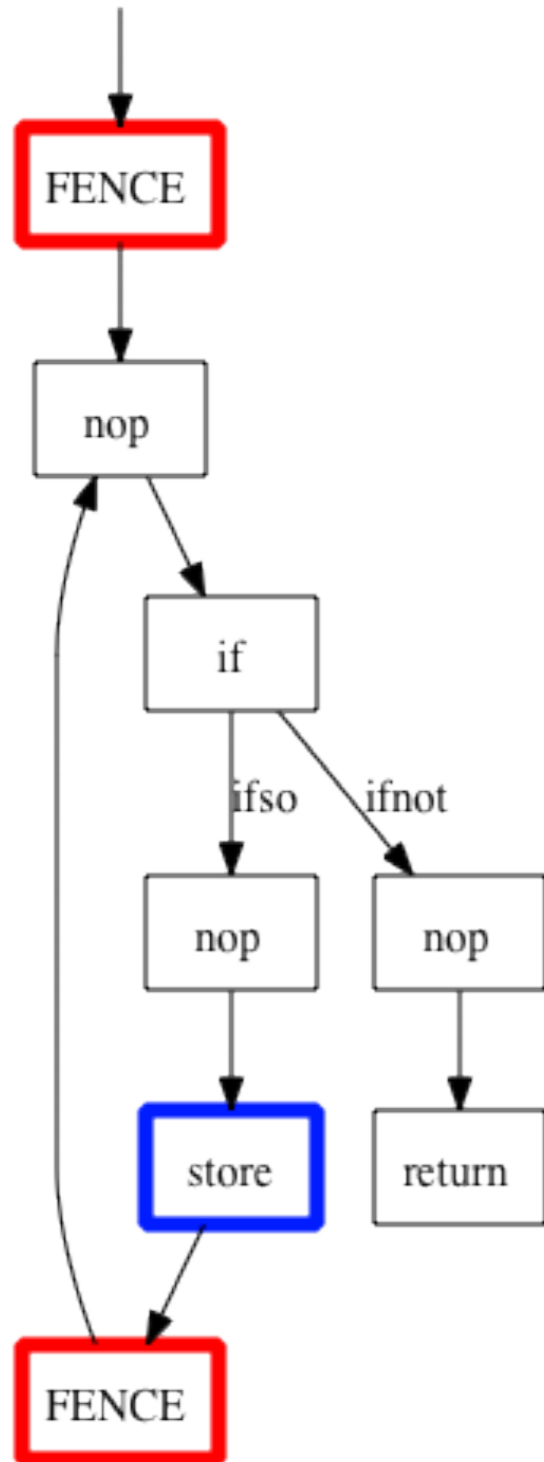
FE1 and FE2 do not optimise these patterns.

It would be nice to have these fences out of the loop.

Do you perform PRE?



A closer look at the RTL



Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

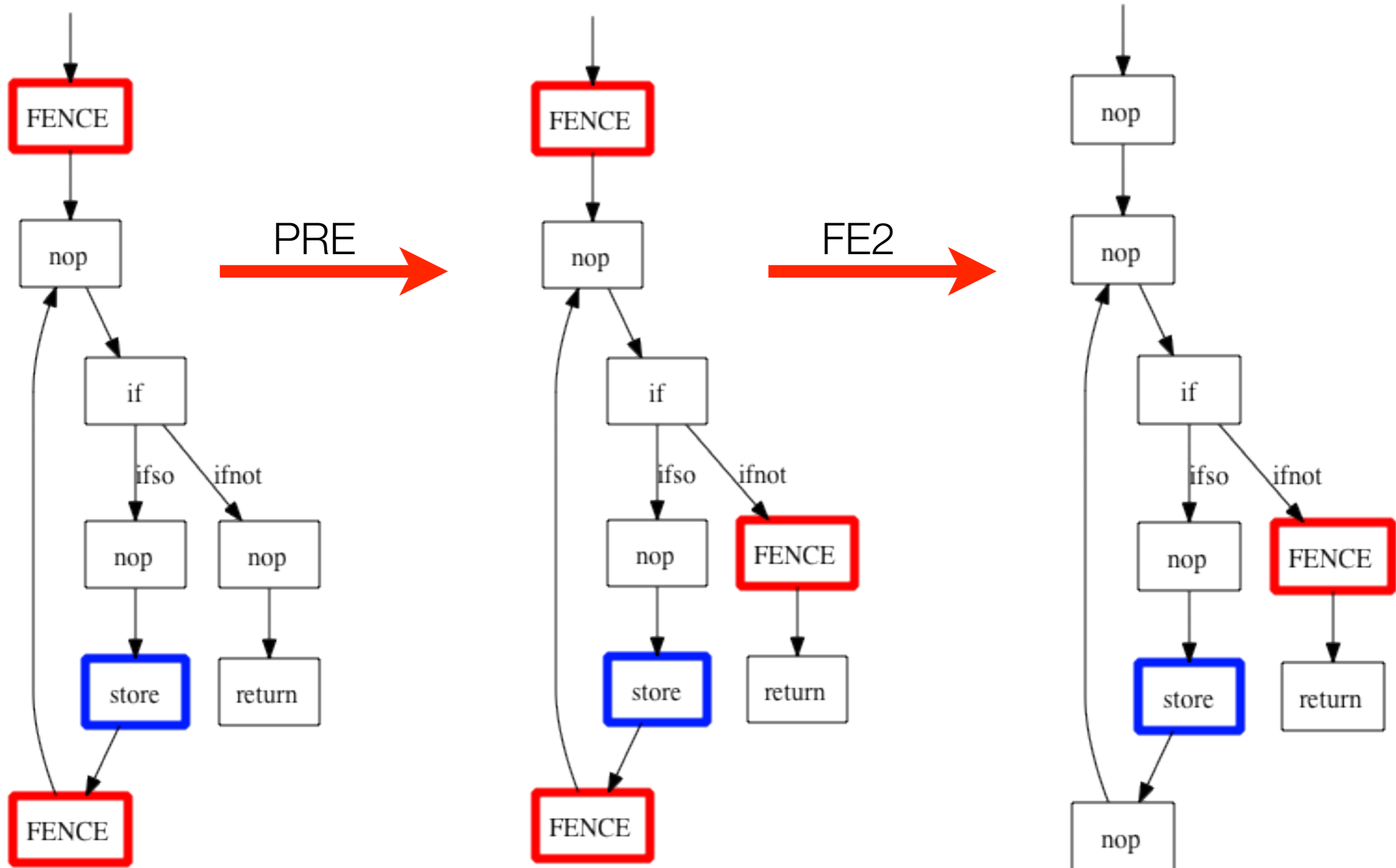
It would be nice to be able to move the fence out of the loop.

Do you perform PRE?



...adding a fence is always safe...

Partial redundancy elimination



Evaluation of the optimisations

- Insert `MFENCES` *before every read* (br), or *after every write* (aw).
- Count the `MFENCE` instructions in the generated code.

	br	br+FE1	aw	aw+FE2	aw+PRE+FE2
Dekker	3	2	5	4	4
Bakery	10	2	4	3	3
Treiber	5	2	3	1	1
Fraser	32	18	19	12	11
TL2	166	95	101	68	68
Genome	133	79	62	41	41
Labyrinth	231	98	63	42	42
SSCA	1264	490	420	367	367

Evaluation of the optimisations

– Insert `MFENCES` *before every read* (br), or *after every write* (aw).

– Cour

Important remark for your future work:

..this is not a proper evaluation: we know nothing about real code, and the number of fences is not a good measure. But unclear how to do better...

`http://evaluate.inf.usi.ch/`

Labyrinth	231	98	63	42	42
SSCA	1264	490	420	367	367