

Semantics, languages and algorithms for multicore programming

Albert Cohen



Luc Maranget



Francesco Zappa Nardelli



Vote: topics for my next lecture

1. The Iwarx and stwcx Power instructions



- 3. Operational and axiomatic formalisation of x86-TSO 2
- 4. Fence optimisations for x86-TSO
- 5. The Java memory model
- 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





Recall: Data-race freedom

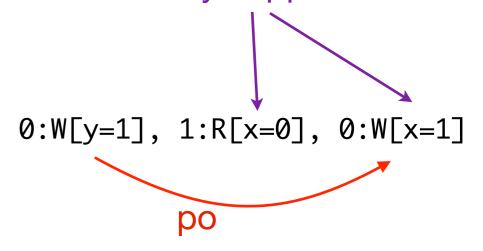
Definition [data-race-freedom]: A program (traceset) is data-race free if none of its executions has two adjacent conflicting actions from different threads.

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
*y = 1	if *x == 1
*x = 1	then print *y

Two conflicting accesses not related by happens before.



Recall: Happens-before

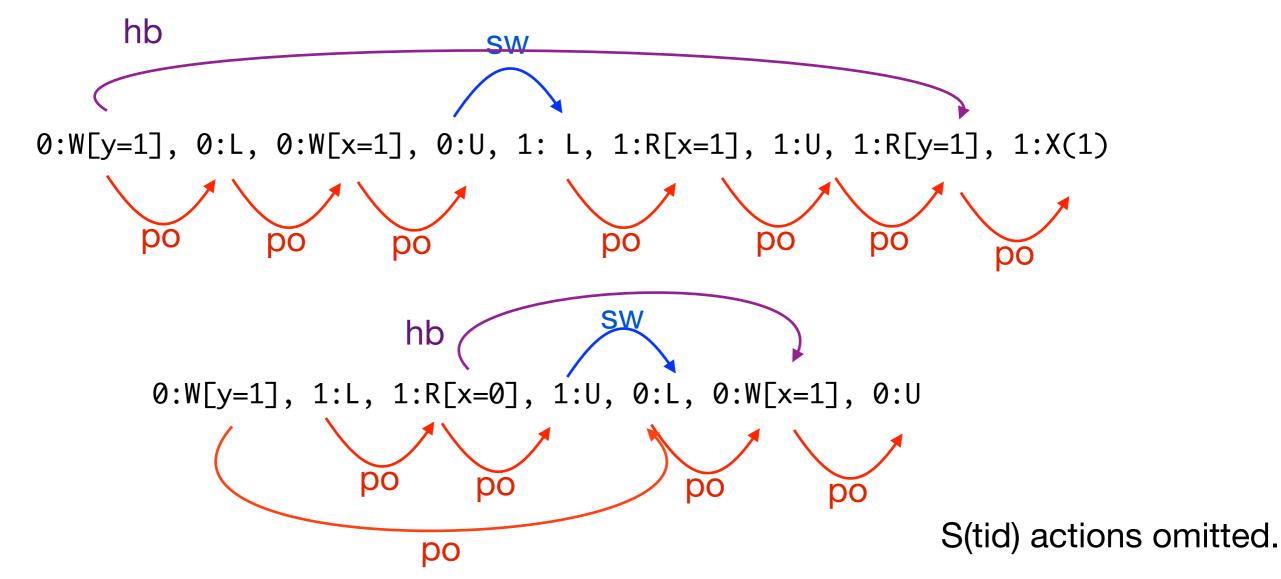
Definition [program order]: **program order**, $<_{po}$, is a total order over the actions of the same thread in an interleaving.

Definition [synchronises with]: in an interleaving I, index i synchroniseswith 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
*y = 1	lock();
lock();	lock(); tmp = *x;
*x = 1	unlock();
unlock();	if tmp = 1
	then print *y



Data race detection: dynamic approaches

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

happens-before ordering

lockset computation

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

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

no false positives

popularised by Eraser (Savage et al.) '97

drawback: misses races occurring on rare executions

can detect races not observed in the execution being monitored

drawback: unsound (false positives)

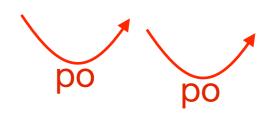
Examples of lockset computation

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

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

lockset vs happens-before

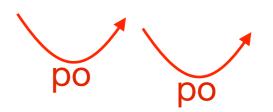
```
y=1
lock(a)
x=2
x=1
unlock(a)
unlock(a)
```



lockset vs happens-before

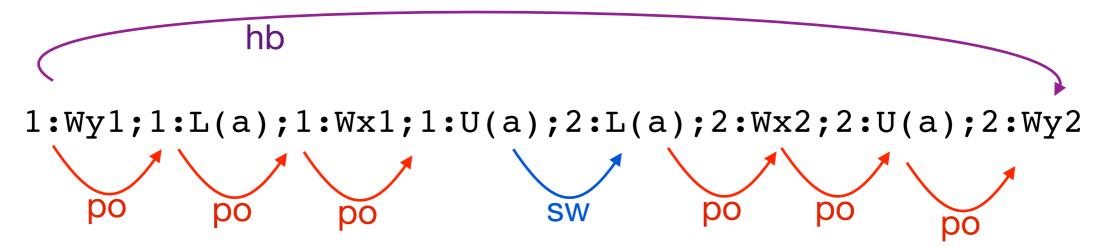
$$\begin{array}{c|c} y=1 & | & lock(a) \\ lock(a) & | & x=2 \\ x=1 & | & unlock(a) \\ unlock(a) & | & y=2 \end{array}$$

This program has a race on y



lockset vs happens-before

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)

lockset vs happens-before (2)

```
y=1
lock(a)
tmp=x
x=1
unlock(a)
tmp=x
then print y
This program instead is DRF.
```

lockset vs happens-before (2)

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

lockset computation

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

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

sound

popularised by Eraser (Savage et al.) '97

drawback: misses races occurring on rare executions

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 rec happens-before ordering + other dynamic annotations rei 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!

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.

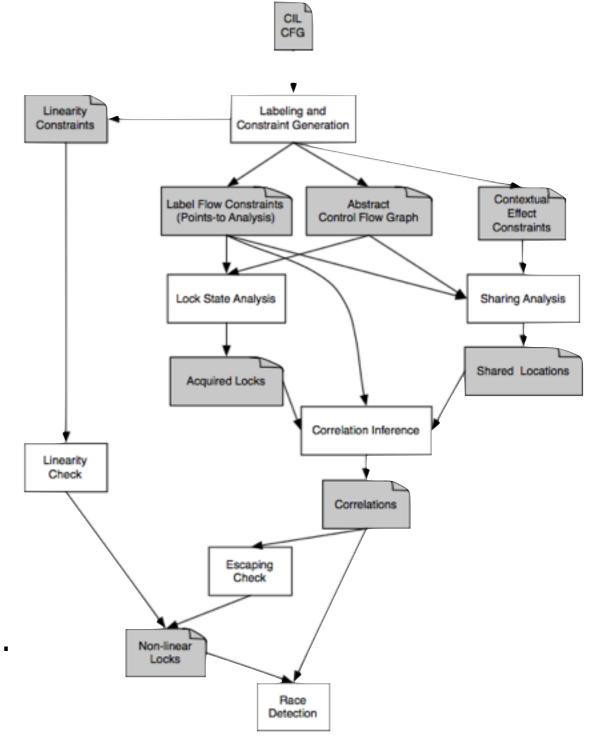


Fig 1 LOCKSMITB are nited ure



2. The C++11 memory model

a good example of an axiomatic memory model



```
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)
   ;
}
```

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

```
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)
   ;
}
```

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

```
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)
   ;
}
```

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

```
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)
   ;
}
```

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

```
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)
   ;
}
```

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

```
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)
   ;
}
```

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

```
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.

```
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.

```
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)
   ;
}
```

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

```
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);
```

```
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
```

```
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  # 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
```

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



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);

$$P \longrightarrow E_1, \ldots, E_n$$

2) an axiomatic memory model judges *E*opsem paired with a memory ordering *X*witness

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

3) searches the consistent executions for races and uncostrained reads is there an X_{ij} with a race?

Relations

An E_{opsem} part containing:

sb sequenced before, program order

asw additional synchronizes with, inter-thread ordering

An X_{witness} part containing:

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.

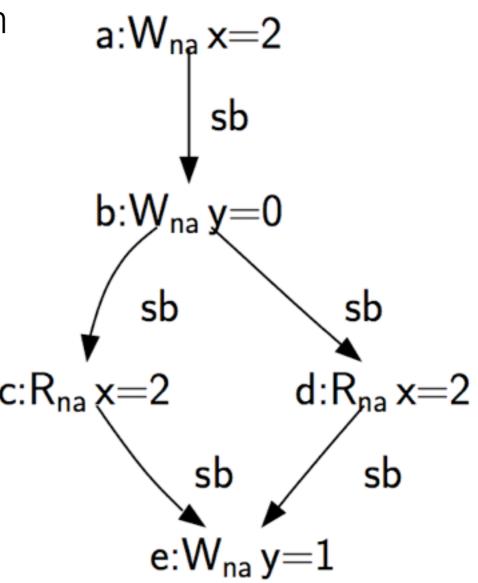
Formally

```
cpp memory model opsem (p : program) =
let pre executions =
  { (Eopsem, Xwitness). opsem p Eopsem \( \Lambda \)
     consistent execution (Eopsem, Xwitness) }
in
if \exists X \in \text{pre executions.}
    (indeterminate reads X = {}) \times
    (unsequenced races X = {}) V
    (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;
}
```

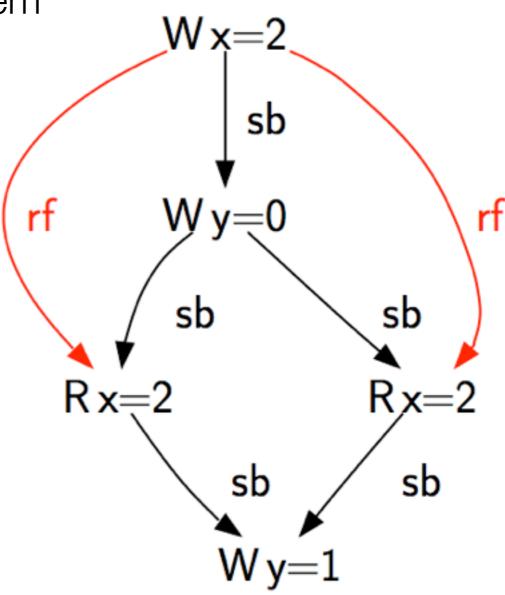


A single-threaded example

1. sequenced before (sb) - given by opsem

2. read-from (rf) - part of the witness

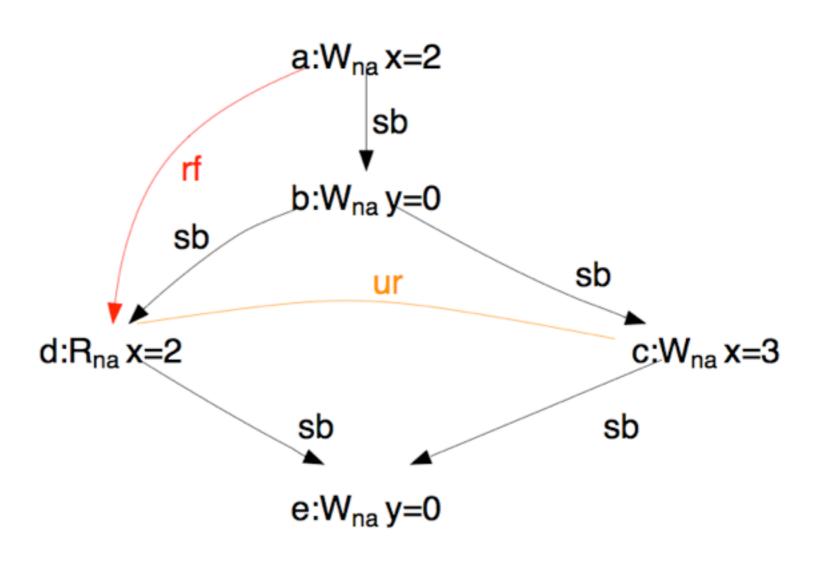
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



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;
$$x = 3; \qquad |y = (x==3)|$$

$$a:W_{na}x=2$$

$$b:W_{na}x=3 \quad c:R_{na}x=2$$

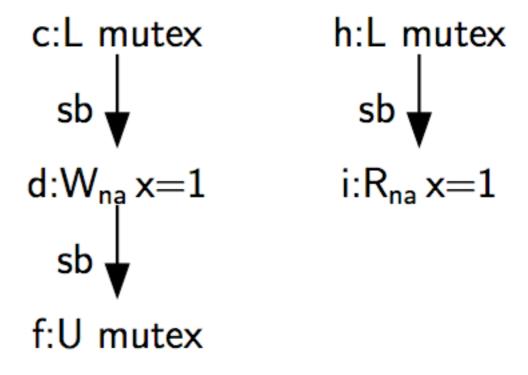
$$b:W_{na}x=3 \quad c:R_{na}x=2$$

$$b:W_{na}x=3 \quad c:R_{na}x=2$$

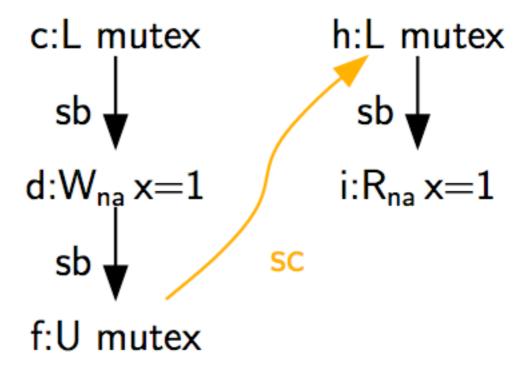
$$c:R_{na}x=2$$

$$c:R_{na}x=2$$

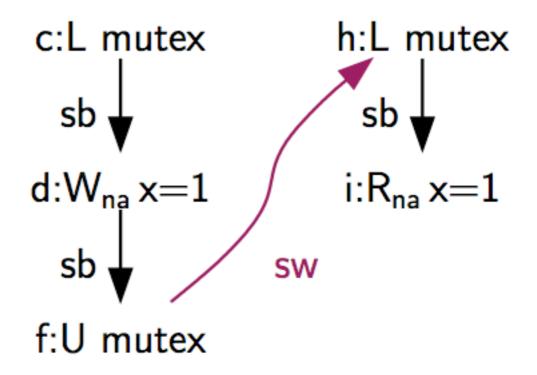
1. the operational semantics defines the sb arrows



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



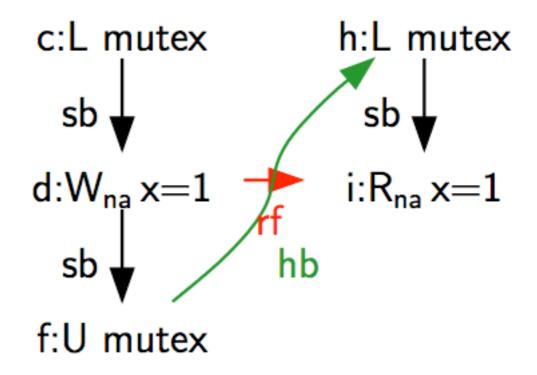
- 1. the operational semantics defines the sb arrows
- 2. guess an sc order on Unlock/Lock actions (part of the witness)
- 3. the sc order is included in the syncronised-with relation



```
\dfrac{\textit{simple-happens-before}}{\left( \dfrac{\textit{sequenced-before}}{} \cup \dfrac{\textit{synchronizes-with}}{} \right)^+}
```

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

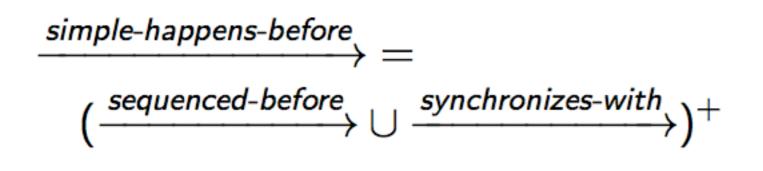
- 1. the operational semantics defines the sb arrows
- 2. guess an sc order on Unlock/Lock actions (part of the witness)
- 3. the sc order is included in the syncronised-with relation
- 4. which in turn defines the happens-before relation...

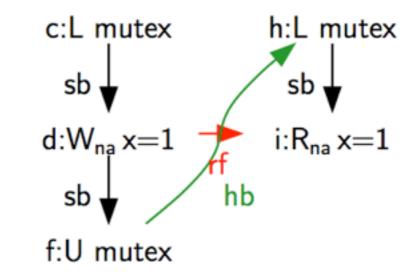


Happens before

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

- 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.





A data race

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

a:W_{na} x=2
b:W_{na} x=3 c:R_{na} x=2
sb
d:W_{na} y=0

A data race

int y,
$$x = 2$$
; $x = 3$; $y = (x==3)$; $y =$

Data race definition

```
let data\_races actions hb = \{ (a, b) \mid \forall \ a \in actions \ b \in actions \mid \neg (a = b) \land \\ same\_location \ a \ b \land \\ (is\_write \ a \lor is\_write \ b) \land \\ \neg (same\_thread \ a \ b) \land \\ \neg (is\_atomic\_action \ a \land is\_atomic\_action \ b) \land \\ \neg ((a, b) \in hb \lor (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);
```

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);
                x.load(seq_cst);
y.load(seq_cst);
     c:W_{sc}y=1
                              e:W_{sc}x=1
                                SC
       SC
                        SC
                               f:R_{sc}y=1
     d:R_{sc}x=0
```

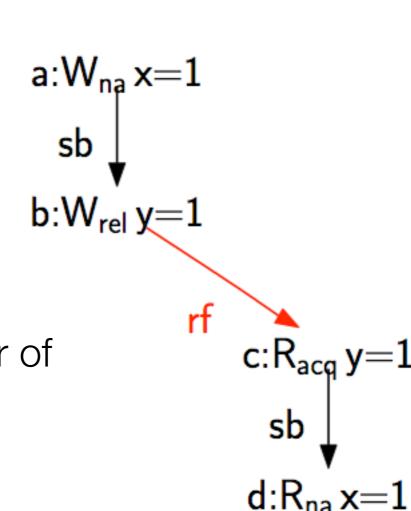
The sc relation must define a total order over unlocks/locks and seq_cst accesses... sc is included in hb, an rf must respect hb.

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.



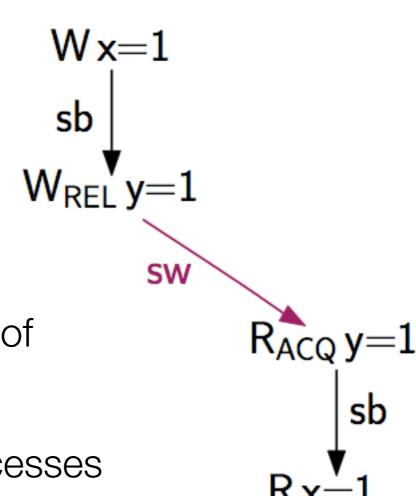
Expert concurrency: the release-acquire idiom

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

release/acquire accesses.

Here we have an rf arrow between a pair of

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



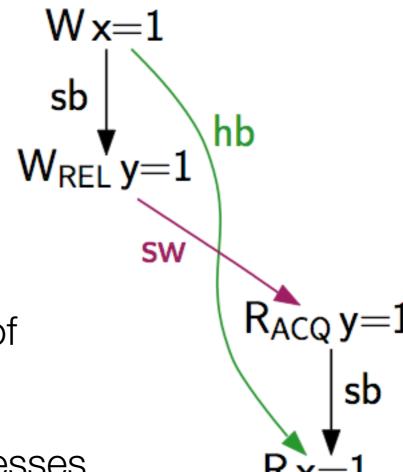
Expert concurrency: the release-acquire idiom

```
// sender
X = \dots
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. simple-happens-before



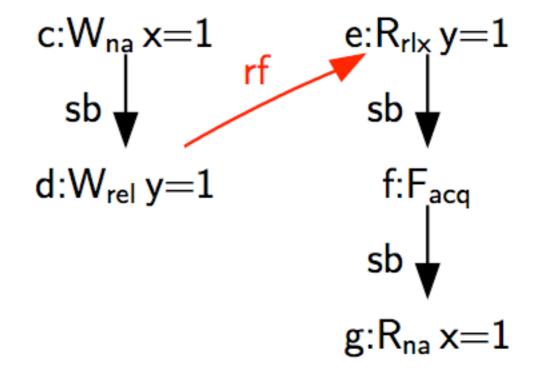
Relaxed writes

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

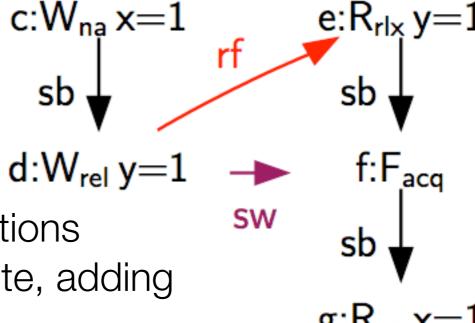
Relaxed writes, ctd.

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

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



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.

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

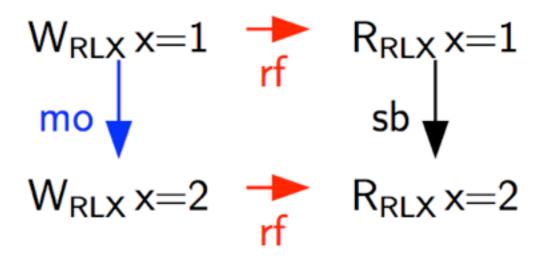
c:W_{na} x=1 e:R_{rlx} y=1 sb $\sqrt{ f:F_{acq}}$ school of the sb $\sqrt{ f:F_{acq}}$ adding $\sqrt{ g:R_{na}}$ x=1

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)

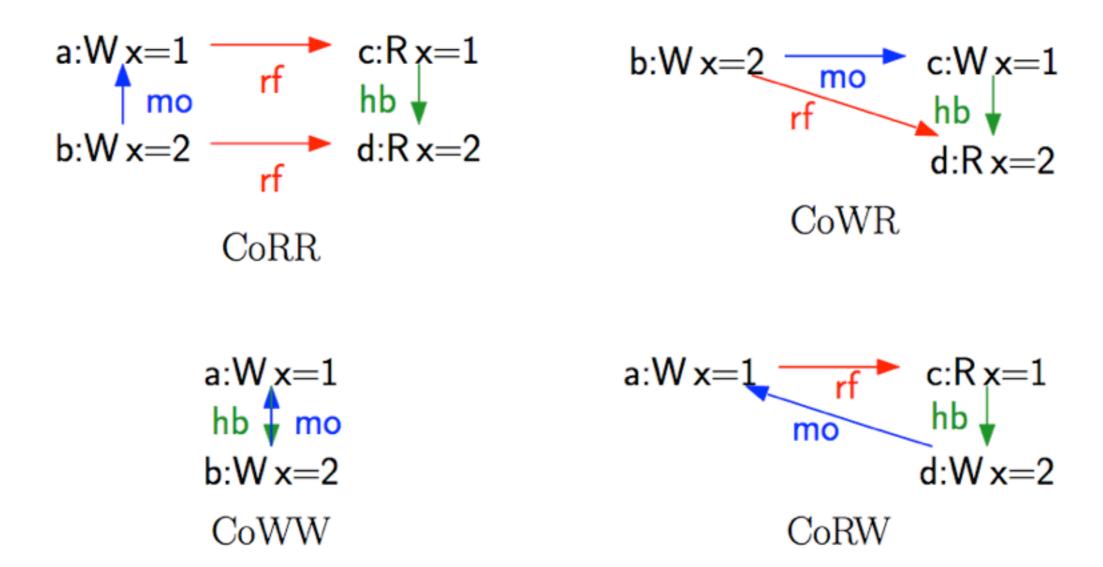
```
atomic_int x = 0;
x.store(1, relaxed); x.load(relaxed);
x.store(2, relaxed); x.load(relaxed);
```



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

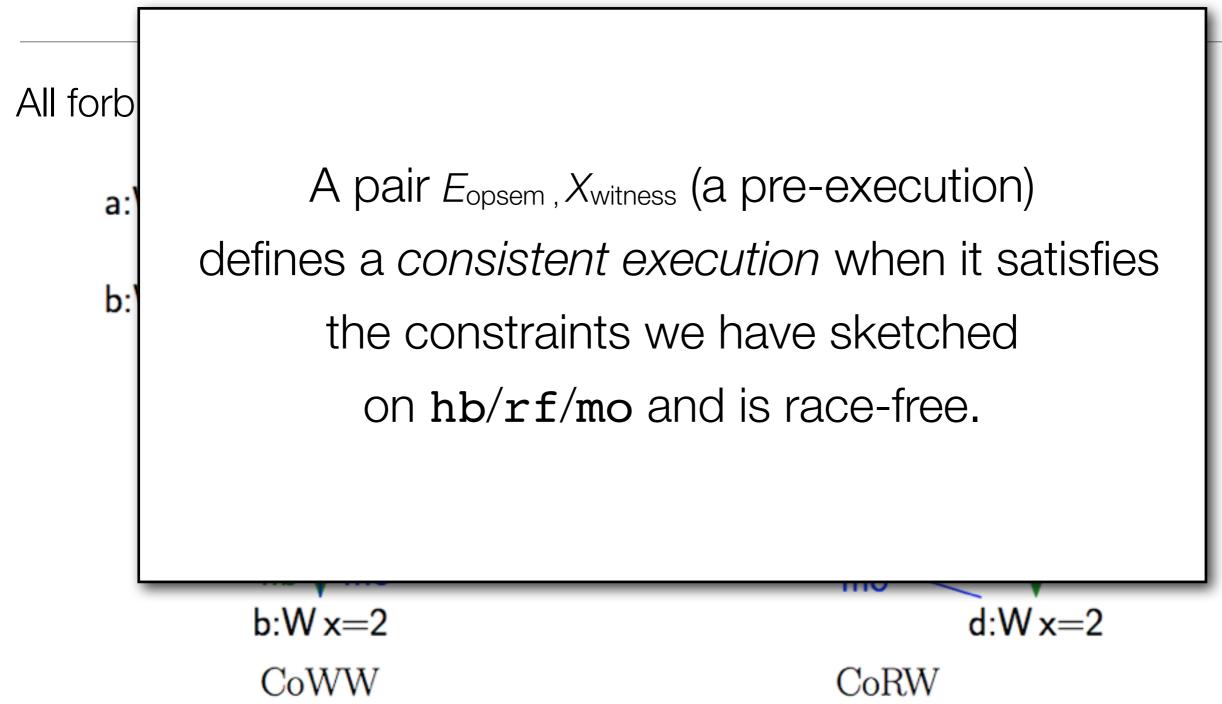
Coherence and atomic reads

All forbidden:



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

Coherence and atomic reads



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

The full model

		7	
$a \xrightarrow{r} b = (a, b) \in r$	is_store a = case a of Store → T _ → F		visible_side_effect_set actions threads location-kind sequenced before additional-gnchronized-with data-dependency central-dependency happens-before =
$a r b = (a, b) \in r$	is_fence $a = case a$ of Fence $a \to T \parallel a \to F$	rs_element rs_head a = same_thread a rs_head V is_utomic_rmw a	(ab c. Apperes defere active (1.50) = ab in the control of the con
	is_lock_or_unlock a = is_lock a v is_unlock a	$release_vegenece = \lambda_{sl} \xrightarrow{nlaw sequence} b =$	visiba-represent-side-effects and = visiba-represent-side-effects and variety was head b =
$a \stackrel{a}{\Rightarrow} b = (a, b) \notin r$	is_atomic_action a = is_atomic_load a \times is_atomic_store a \times is_atomic_rmw a	is_at_atomic_location $b \wedge is_a = b_a = b_a + b_a = b$	{c. vse_phad multication units c ∧ nature before - nature before - nature be
$\stackrel{r}{\rightarrow} = r$		$\begin{array}{cccccccccccccccccccccccccccccccccccc$	(% xxx, had xxx had xxx had xxx xxx xx xx xx xx xx xx xx xx xx
$a \xrightarrow{s} b \xrightarrow{s} c = a \xrightarrow{s} b \wedge b \xrightarrow{s} c$	is_load_or_store a = is_load a ∨ is_store a	$(rc. a_{nl} \longrightarrow c \longrightarrow b \Longrightarrow rr. riement a_{nl} c)))$	
relation_over s $rel = domain \ rel \subseteq s \land range \ rel \subseteq s$	is_read z = is_atomic_load z∨is_atomic_rmw z∨is_load z	release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order =	
$\frac{nl}{nl} _{s} = rel \cap (s \times s)$	is_write a =	release_sequence: actions threads location-kind sequenced-before additional-genchronized-with data-dependency control-dependency modification-order a b)	visible_sequence_sd_side_effects = visible_sequences_sd_side_effects = \(\lambda \text{test}, \text{ best} \text{ best} \) \(\lambda \text{ its state} \text{ best} \text{ best} \) (it if it is, it state) \(\text{ best} \text{ its state} \)
	is_atomic_store a V is_atomic_rmw a V is_store a	hypothetical_release_sequence = a hypothetical-nineas-sequence, b = is.ast.atomic.location b / i.	(vss_had) \\ \text{visibs_veoquence_of_side_rifects_tail_vsse_head b}
$rel _s = rel \cap (s \times s)$	is_nequire a = (case memory_order a of	$(b=3) \lor (rs_s c) = rs_s \land s \xrightarrow{modification order} b \land \land$	ctuc ())
\xrightarrow{nd} _s = rel \cap (s \times s)	SOME $mem_ord \rightarrow$ $(mem_ord \in$	$(\forall c, a \text{ and fiction order} c \text{ confiction order} b \Longrightarrow rs_e \text{ inserts } a c)))$	visible_sequences_of_aide_effects_set actions threads location-kind sequenced before additional-synchronized-with data-dependency control-dependency modification-order happens before visible-side-effect = myimage (visible_sequences_of_aide_effects actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect myimage (visible_effects) actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect myimage (visible_effects) actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect myimage (visible_effects) actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect myimage (visible_effects) actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect myimage (visible_effects) action kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect myimage (visible_effects) myimage (vis
$ret _s = ret \cap (s \times s)$	{Mo_ACQUIRE, Mo_ACQ_REL, Mo_SEQ_CST} ∧ (is_rend a ∨ is_fence a)) ∨ (* 29.85 states that consume fences are acquire fences. *)	hypothetical_release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order =	
strict_preorder ord = irreflexive ord ∧ trans ord	((mem_ord = MO_CONSUME) ∧ is_fence a) NONE → is_lock a)	hypothetical_release_sequence actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order a b)	consistent_armds_from_manpping = consistent_armds_from_manpping = $ (\forall h \; (\text{ix.red} \; h) \; i_{\text{ix.red}} \; ho_{\text{ix.red}} \; ho_{\text{ix.red}}$
total_over s ord =	is_consume a = is_read a A (memory_order a = SOME MO_CONSUME)	synchronins_with = 2 endousine with b = (* - additional synchronization, from thread create etc. − *)	then $(3_{m-2}, \frac{1}{m} \sin \sin \sin \sin \frac{1}{m}) \land s_m = s)$ $\sin (3_{m-2}, \frac{1}{m} \sin \sin \sin \sin \frac{1}{m})$
relation_over s ord \land $(\forall x \in s. \forall y \in s. x. \xrightarrow{ord} y \lor y \xrightarrow{ord} x \lor (x = y))$		2 additional symmetric from coreacy create exc. —)	(th. (in_read $b \land in_s n_s n_s n_s n_s n_s n_s n_s n_s n_s $
strict_total_order_over s ord =	is_release a = (case memory_order a of Souse memory_order)	(same_location $ab \land a \in actions \land b \in actions \land ($ (" - mutex specimenization - ") (S_unified: $ab \land b \land$	then $(\widetilde{a}(y, wu))$ is visible explanation of $\widetilde{a}(x)$ in (x, y, y) in (x, y) in $($
strict_preorder ord \(\tau \tau \tau \tau \tau \tau \tau \tau	SOME mem_ord -> mem_ord -> Mo_release, Mo_acq_rel, Mo_seq_cst} \(\) (is, write a V is, fence a)	(* - release/acquire synchronization - *)	che ¬(∃x → ¬(±y))) ∧
$x \xrightarrow{cod}_{pred} y = pred x \wedge x \xrightarrow{sod}_{y} y \wedge -(\exists x. pred x \wedge x \xrightarrow{sod}_{y} x \xrightarrow{sod}_{y} y)$	NONE → is_unlock a)	(is_release $a \land$ is_nequire $b \land \neg$ same_thread $a \not b \land$ (is_c $a \xrightarrow{\text{other sequence}} c \xrightarrow{e^a} b)) \lor$	$(\forall (x, a) \in \vec{A},$ $\forall (y, b) \in \vec{A}.$
	is_seq_cst a = (memory_order a = Some Mo_seq_cst)	(* – fence synchronization – *) (is_fence a ∧ is_release a ∧ is_fence b ∧ is_acquire b ∧	
$x \xrightarrow{ord} y = x \xrightarrow{ord} y \land \neg(\exists z. x \xrightarrow{ord} z \xrightarrow{ord} y)$	location_kind = MUTEX	(2x. 2y. same-bontion x y Λ is atomic action x Λ is Latonic action y Λ is, write x Λ y-generalizable y _λ γ y separate below β _λ Λ	$ = (x = y) \times x = y \times y$
well_founded $r = wf r$	NON_ATOMIC ATOMIC	$(\exists z. \ x \xrightarrow{hgothetical-odiano-assance} z \xrightarrow{d} y))) \lor$	Vc. c * b h k write a hume location a b h is net_atomic_location b
type_abbrev action.id: string	actions_respect_location_kinds = actions_respect_location_kinds =	(is_fence a \(\) is_grelense \(a \) \(\) is_atomic_action \(b \) is accopain \(b \) \(\) (2x. sum_becomes \(a \) \(b \) \(\)	$\Rightarrow (c = a) \lor a$ $\xrightarrow{\text{modification entry}} c \land \land$
	∀a. case location a of SOME I → (case location-kind I of	is atomic action $x \wedge f_{\infty}$ write $x \wedge g_{\infty}$ and g_{∞}	$(\forall A, b) \in \frac{I_{topous below,}}{VC}$, VC .
type_abbrev thread_id:string	MUTEX >> is_lock_or_unlock a NON_ATOMIC -> is_load_or_store a ATOMIC -> is_load_or_store a \(\) is_atomic_action a)	(3x x hymthetis/nines separate x c d b))) ∨ (is_attomic_action 2 ∧ is_relesse a ∧	C = 3 A & write b \(\) same_location \(2 \) b \(\) i_A d_atomic_location \(2 \) was \(\) matter most same_location \(2 \) b \(\) \(\)
type_abbrev location:string	\parallel None \rightarrow T	is_fence $b \land is$ _nequire $b \land$ $(\exists x. same_l-beation x \land x \land f_{is_1} tomis_{-net} ton x \land$	(ria.b) = ^{di} , is atomic rune b
type_abbrev val:string	is_at_location_kind = is_at_location_kind = case location x of	$\begin{array}{c} x \xrightarrow{\text{missance below }} b \land \\ (2z. x \xrightarrow{\text{missance missance }} z \xrightarrow{d'} x))))) \end{array}$	\Rightarrow a mildicate-order \Rightarrow
memory_order_enum = Mo_seq_cst	SOME $l \rightarrow (location-kind \ l = lk0)$ $\parallel \text{None} \rightarrow \mathbf{F}$	synchronizes_with_set actions through location-kind sequenced-before additional-synchronized-with data-dependency control-dependency of modification-order sc release-sequence hypothetical-release-sequence =	$ (r(x,b)) \in \stackrel{f}{\sim} h_{x}(s_{x}r_{y}) \leq h_{x$
Mo_release Mo_release	is_at_mutex_location z =	synchronizes, with actions through location-kind sequenced-before additional-synchronized-with data-dependency control-dependency of modification-order sc release-sequence hypothetical-release-sequence a b)	(* feme articles-*)
Mo_course Mo_consume Mo_co_rel	is_at_location_kind a MUTEX	carries A_{a} dependency, $D = 2$ carries adaptatory to $b = 1$	
	is_nt_non_atomic_location_s = is_nt_location_kind_s Non_atomic	$2((\frac{d}{d})\cap \frac{\text{expected-define}}{2}) \cup \frac{\text{deta-dependency}}{2})^+ b$	(is, frace x ∧ is, usequext x ∧ is, ustonic, action b ∧ is, write 2 ∧ same, location a b ∧
LOCK of action_id thread_id location UNLOCK of action_id thread_id location	is_at_atomic_location z =	carries_a_dependency_to_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf =	$z \stackrel{\leftarrow}{\sim} x \wedge y \stackrel{\leftarrow}{\sim} b$ $\longrightarrow (y = z) \vee z \stackrel{\text{millionia order}}{\longrightarrow} y) \wedge$
ATOMIC_LOAD of action_id thread_id memory_order_enum location val ATOMIC_STORE of action_id thread_id memory_order_enum location val	is_at_location_kind 2 ATOMIC	carries_a_dependency_to actions threads location kind sequenced before additional-synchronized with data-dependency control-dependency of a b)	(* 29.34 *)
ATOMIC_RMW of action_id thread_id memory_order_enum location val val			$(v a,x) \in \frac{suspensed before}{s}, v(y,b) \in \vec{z}_i$.
LOAD of action_id thread_id location val STORE of action_id thread_id location val	same_thread $ab = (thread_id_of \ a = thread_id_of \ b)$	dependency archived, before $=2$ dependency arterial before g d $=$ g \in actions f , d \in actions f	(is_atomic_action a) is_fence x \(\lambda\) is_expect x \(\lambda\) is_vviite a\(\lambda\) same_closulum a \(\lambda\)\(\lambda\) \(\sigma\)_v=\(\lambda\)\(\lambda\) is a somic action \(\lambda\)\(\lambda\)
LOAD of action.id thread.id location val STORE of action.id thread.id location val FENCE of action.id thread.id memory_order_enum	same,thread $a b = (\text{thread-id-of } a) = \text{thread-id-of } b)$ threadwise,relation, over $a r d = \text{ristion.over } a r d \land (r b, b) \in r d$, same,thread $a b$	$a \in actions \land d \in actions \land$ b, b , b , c	(bactonic-action a Λ is favore $x \land b$ is any $x \land b$ is a favore $x \land b$ is a favore $x \land b \land x \land b$ is a favore $x \land b \land x \land x$
Load of action, if thread of location val Stront of action, if thread of location val Fibror of action, if thread of location val Fibror of action, if the location of	threadwise, relation, over s $rd = relation$, over s $rd = rd$, same, thread s b)	a c actions h of C actions h (lib. kurchouse h h is consume h h (lie. kurchouse h h is consume h h (lie. h	(bactonic-action 2 h lactures x h bacoquet x ∧ bacoquet x h x 2 h h bacoquet x b h x 2 h h bacoquet x b h h x 2 h bacoquet x y h x b x b x b x b x b x b x b x b x b x
Load of action-of thread-of hostion val Stront of action-of thread hostion val Stront of action-of thread-of money-order-one Fixed of action-of hosting-order-or	threadwise_pulsion_over s rd — relation_over s rd \sim relation_over s rd \wedge $(\forall a,b) \in rd$. same_thread s b) same_location s b — (becation s = bostion b)	a c actions A d c actions A (Bb. kurdness A is parameter b A (Bc	(backonia_action a) is favore x ∧ favore x
[Loan of action, id thread-di location val STORGE of action, di thread-di location val FINCHE of action, di thread-di memory_order_enum (action_id_off_(LOCK) air) = airf_) \ (action_id_off_(NOCK) air) = airf_) \ (action_id_off_(NOCK) action_id_of) = airf_) \ (action_id_off_(NOCK) action_id_of	threadwise, relation, over s $rd = relation$, over s $rd = rd$, same, thread s b)	is a carious A 6 actions A (Bis. Lardones A his commune b A (Bis. Lardon	(inclumination a h is famous x ∧ is compact x ∧ is control to h is control to h ∧ x = 5 ∧ h action at a h ∧ x = 5 ∧ h action at a n h ∧ x = 5 ∧ h action at a n h ∧ y = 0 ∧ h action at a n h ∧ y = 0 ∧ h action at a n h ∧ y = 0 ∧ h action at a n h action at x ∈ 0 ∧ h
Load of action of thread-of location val	threadwise_relation_over x $xd = relation_over x xd = relation_over x xd \in N(A_0, b) \in rel. same_thread x b) same_location x b = (bocation x) = bocation b) location_out x x x x x x x x x x$	a c actions A d c actions A (Bb. kurdness A is parameter b A (Bc	(inclusion action a h is favore $x \land h$ increases $h \land h$ incre
Loan of action, if thread-of location val	threadring_relation_core s set = relation_core s set = relation_core s set / (v(s, b) ∈ set. some_abrend s b) some_location s b = (location s = boution b) location_cof serious = (f. Sa (location s = Soute f)) well_formed_action s = case s of Arronn_colon_core_core_core_core_core_core_core_core	is a carbon A of a action A (Bis kurdeness A his common b A (Bis a submanus, at b) (by a submanus, at b) dependency, andreal, before actions through location kind sequenced before additional synchronized with data-dependency control-dependency of modification order release sequence carries a dependency-to a dependency, andreal, before actions through location-kind sequenced before additional synchronized with data-dependency control-dependency of modification order release sequence carries a dependency-to a b) simple Auguston. Leftor a common sequenced before additional synchronized with data-dependency control-dependency or modification order release sequence carries a dependency-to a b) simple Auguston. Leftor a common sequenced before additional synchronized with data-dependency control-dependency or modification order release sequence carries a dependency-to a b) consistent unique.	(inclusion action 2 is fileway to 5 incomposed x ∧ incomposed x of x ≥ 5 ∧ incomposed x ≥ 5 ∧ incomp
Loan of action, it thread the location val Stront of actional thread of location value Stront of actional value Stront of location value Stront of action value Stront of location value Stront of	threshvine, platine, over a pl = relation, over a pl = relation, over a ml \(\tilde{\pi}(\pi_A \beta) \in m\), some, thread a b) sum_bostion a b = (breation a = bostion b) location, of actions = (f. 2a (bostion a = SOME f)) will, formulation a = ATOMIC_LOAD	is a carbon A of a action A (3b. increases b A increases b A (3b. increases (3b. in	(inclusion action 2 of inference $x \land incompant x \land incom$
Loan of action, its thread-of location val Strong of action, all thread location val Firete of actional thread of location val Firete of actional thread of memory order anum (action_id_id_i) (Lock $sid_{-j} = sid_j) \wedge (action_id_id_i) (Lock sid_{-j} = sid_j) \wedge (action_id_id_i) (Arounclassus sid_{-j} = sid_j) \wedge (action_id_id_i) (Arounclassus sid_{-j} = sid_j) \wedge (action_id_id_i) (Rounclassus sid_{-j} = sid_j) \wedge (action_id_id_i) (Rounclassus sid_{-j} = sid_j) \wedge (action_id_id_id_i) (Rounclassus sid_{-j} = sid_j) \wedge (action_id_id_id_i) (Rounclassus sid_{-j} = sid_j) \wedge (action_id_id_id_id_i) (Rounclassus sid_{-j} = sid_j) \wedge (action_id_id_id_id_id_id_id_id_id_id_id_id_id_$	throshvin-printing, over a pri- enthion, over a rad \(\text{V(f_A, b)} \in \text{d} \), some, showed a b) sum_location a b = (breation a = breation b) location, of actions = (l. lin (breation a = State f)) will formulaction a = ATOMICJOAD = mem_ord = mem_ord \(\text{ATOMICJOAD} \), mem_ord = mem_ord \(\text{ATOMICJOAD} \).	ica serion A 6 a action A (6 action A) (Bis increase A his community b A) (Bis increase A his community b A) (Bis increase A his community of 4 (b) and (b)) (dependency action of the control of the	(inclusion action a β is former α is increased α inclusion as β in the second α is a β in the second α included β in α is β in
Load of action is threaded location val Strong of action is threaded location val Strong of actional threaded location val Fixture of actional thread bloomensy color_numm (action_id_id_i (Locat_id_i)_ = 3 elf) \ (color_num_id_id_i (Locat_id_id_i)_ = 3 elf) \ (color_num_id_id_i (Notion_id_id_id_id_id_id_id_id_id_id_id_id_id_	throubviou_velation_over x rd = enhiston_over x rd \(\cdot \text{V}(\text{V}_0 \text{\tex	ice actions A & actions A (a action A) (Bis increases A increasement b A) (Bis increased A increasement A) (Bis increased A increasement A) (Bis increased A increasement B A) (Bis increased A increasement B A) (Bis increase and a increas	(material and factors of lawrence of lawre
LOAD of action of thread-of location val STORE of actional thread of location value (actional of LOCATION - - - - - - - - -	threadwiss_relation_over s nd = relation_over s nd \(\cdots \) (\(\lambda \) (\	carbon of c actions of c acti	(inclusion action a 7 in Section 4 × 5 incomposed 4 × 5 i
Load of action of thread of location val	throudsins_relation_over s rd = enhiston_over s rd \(\cdot \times \) (\(\times \) (\(\times \) (\(\times \) (\(\times \) \) (\(\times \)	ice actions A & actions A (a action A) (Bis increases A increasement b A) (Bis increased A increasement A) (Bis increased A increasement A) (Bis increased A increasement B A) (Bis increased A increasement B A) (Bis increase and a increas	(inclusionic action a finite or in lower part in lower par
Loan of action, if thread-th location val Strong of action, all thread blocation val Firete of action, all thread blocation val Firete of action, all thread blocation val Firete of action, all thread blocation val (action, all, all (Loans all _ ,) = adf) \ (action, all, all (Loans all _ ,) = adf) \ (action, all, all (Arounc, aroune all _ , .) = adf) \ (action, all, all (Arounc, aroune all _ , .) = adf) \ (action, all, all (Loans all _ , .) = adf) \ (action, all, all (Loans all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all (Firete all _ , .) = adf) \ (action, all, all _ ,	throubries_relation_over x nd - estation_over x nd \(\cdot \times \) \(\times \) throubries_relation_over x nd \(\cdot \times \) \(cardinal A & actions A cardinal A card	(inclusions a final former of Newmorth of
Load of action of thread of location val	threshvin_polation_ever s st = relation_ever s st = relation_ever s st / (v(s, b) ∈ st. sume_thread s b) sume_location s b = (location s = bootton b) location_et scrions = (f, 3a (location s = Soute f)) well_formed_action s = case s of Arounc_colon_ever_Mon_	cardinar A & cardinar A cardinar ca	Consideration and the Section Association of the Consideration of the
Loan of action, it threads the tourism val	throubviou_velation_over s rd = enhistion_over s rd \(\cdot (\pi_A) \in \cdot	carbon A of carbon A (cames b)	(inclusions actions a finishment of locations of the company of a finishment of the company of t
LOAD of action.id thread.de location val	threshrin, polation, over a rd = relation_over a rd \(\text{reshring_polation_polation_set a rd \(\text{v(f_a b)} \) \(\text{continum } a = \text{boottom } b \) \(\text{some_shrouts } a b = (\text{boottom } a = \text{boottom } b) \) \(\text{location} a b = (\text{boottom} a) = \text{boottom} b) \) \(location_polat	cardinar of c actions of c actions of c actions of c continued	(inclusions) a file former of isological of inclusions and inclusions of
LOAD of action.id thread.de location val	throubsing_relation_aver x nl = enthiologous x nl \(\(\forall (\forall \) \) \(\int \) \	is excisent of a states of (a serious of (b layer and black of the serious of (b layer and black of the serious of (b layer and black of the serious of the	(**Landermicantina of histories
LOAD of action-if thread-de location val	throubvious_relation_vere x nd ~ (v(f_A, b) ∈ nd. sume_throud x b) sums_bootton x b = (breation x = breation b) location_vef x critics = (f_A : 2a_(breation x = breation b) will_formul_action x = Arround_LOAD	consistent_simple_happens_before = consistent_sim	Consistent and a final form of a final power of a final
LOAD of action.id thread.de location val STORE of action.id thread.de location val STORE of action.id thread.de location val STORE of action.id thread de location val STORE of action.id thread.de location.geography. STORE of action.id.de (LOCE ad.,) = 361 / (section.id.de (ACOMIC.ADOM ad.,) = 361 / (section.id.de (threshvin_polation_row s st or relation_rows st of or relation_rows st of \(\text{V(S_A} \) \) \(\text{S} \) \(\text{contion} \) s b = (beatinn s = beatinn b) blocation_sd strions = (f. 3a (beatinn s = beatinn b) will_formed_action n = case s of ATOMIC_TOME, \(\text{S} \) \(\text{AD_CONNEME} \) \(AD_CON	is excisent of a states of (a serious of (b layer and black of the serious of (b layer and black of the serious of (b layer and black of the serious of the	(Cast) Consideration of the force of interpret of its sequence of the additional synchronized with data dependency of medication order is reduced by the consideration of the con
LOAD of action-id thread-of boxelion val STORIG of action-id thread-of boxelion val STORIG of action-id thread-of boxelion val STORIG of action-id bread-of boxelion val STORIG of action-id-of bread-of boxelion value Content-of-of (DOCK of action-id-of) Content-of-of (DOCK of action-id-of) Content-of-of (ACONICLOUS of action-id-of) Content-of-of (ACONICLOUS of action-id-of) Content-of-of-of-of-of-of-of-of-of-of-of-of-of-	throubrios_relation_rows x nd ~ enthios_rows x nd ~ (r(f_A, b) ∈ nd. sums_throad x b) sums_bostion x b = (broation x = broation b) location_rows x nd \(r(f_A, b) ∈ nd. sums_throad x b) location_rows x nd \(r(f_A, b) ∈ nd. sums_throad x b) well_formed_action x =	care actions of c actions of (2s. actions of c actions of c actions of c (2s. acti	(custimate and manufactions of a bit of the second of the
LOAD of action-it thread-it beaution val STORE of action-it thread the location val STORE of action-it thread the location val STORE of action-it thread thread-it memory-actions STORE of action-it thread thread-it memory-action STORE of action-it thread thread-it memory-action STORE of action-it thread thread-it thread-it memory-action STORE of action-it thread (NOTICE of action-it thread STORE of act	throubvious_relation_rows x xt = relation_rows x xt = relation_rows x xt / \(\text{(f_A, b)} \in \text{(f_A, b)} \in (f_	Section A of a strong	Continued and the analysis of the August 1 A playing
LOAD of action-it thread-of beaution val STORE of action-it thread Action STORE of action-it thread Action Action Action STORE of action Action Action Action STORE of action Action Action Action STORE of action Action Action STORE of action Action Action STORE of action Action Action Action Action Action STORE of action Action Action Action Action Action STORE of action Action Action Action Action Action Action STORE of action	throubrios_relation_rows x nd ~ enthios_rows x nd ~ (r(f_A, b) ∈ nd. sums_throad x b) sums_bostion x b = (broation x = broation b) location_rows x nd \(r(f_A, b) ∈ nd. sums_throad x b) location_rows x nd \(r(f_A, b) ∈ nd. sums_throad x b) well_formed_action x =	Section A of a striam	Continued and the problems of the problems o
LOAD of action.id thread.id location val	throubrios_relation_rows x xt = ministron_rows x xt = ministron_rows x xt \(\text{N}(\text{T}_k) \) \(\text{N} \) \(N	See actions A of a sations	(Continue actions a 7 h. Section of 7 h. Secti
Load of action.id thread.id location val	threshvin_polation_row s st or relation_rows st of a relation_rows st of \(\text{V(S,B)} \) \(\in \text{d. same_shread } a \) \) smm_location a \(b = \text{(location } a = \text{bootion } b \) \] location_rows a \(c = \text{(location } a = \text{bootion } b \) \] location_rows a \(c = \text{(location } a = \text{bootion } b \) \] well_formed_action a = \(c = \text{case } a = \text{bootion } b \) \(\text{bootion_rows} \) \[\lambda \(\text{bootion_rows} b = \text{bootion_rows} a = \text{bootion_rows} a \) \[\lambda \(\text{bootion_rows} b = \text{bootion_rows} a	Secretaria of a section	(Continue actions of a plus failure or a plus against or against aga
LOAD of action.id thread.de location val	threadmin_point in_move s st or relation_power s st of verification_power s st of v(st_s) \(\)	Section A Canton	(Captionis actions of a Nacione of Alexander
Load of action.id thread.id location val	throubrios_polation_rows x xt = relation_rows x xt = relation_rows x xt / \(\(\forall (\forall \) \) \(\int \) \	Section 1 of Extracts	Continued actions of \$1 inching in \$1 \text{ inching in \$1 \tex
Load of action.id thread delicents val	throughting_polation_rows x xt = relation_rows x xt = relation_rows x xt / (*(f_A)) \(\times \) c. st. summ_through x b \(\times \) (station x = botation b) location_xt xt xt / (*(f_A)) \(\times \) c. st. summ_through x b \(\times \) (incation x = b (botation x = botation b) vell_formulation_xt = c.c. x xt x	Section 1 of Extracts	Continued that is a big former or big form
LOAD of action.id thread decision val	thresdering_relation_rows s st or relation_rows s st or relation_rows s st of \(\text{(fig. b)} \) \(\text{ continue} \) a = boots b \(\text{)} \) bootston_rows s = \(\text{(fig. b)} \) \(\text{ continue} \) \(\text{)} \) bootston_rows s = \(\text{(fig. b)} \) \(\text{ continue} \) \(\text{)} \) well_formed_action s = \(\text{ case s of } \) A FORT (ASSETS S) (ASSETS S) (ASSETS S) (ASSETS S) (ASSETS S) (ASSETS S) \(\text{(fig. continue} \) \((fig. cont	action of C extions ((i) but human to bit ((ii) but human to bit ((iii) but hu	Constitution The State of t
Load of action.id thread delicents val	throubrios_polation_rows x xt = relation_rows x xt = relation_rows x xt / (*(f_A b) ∈ xt. sums_throud x b) sums_bostion x b = (broation x = bostion b) location_rows xt = (f_A is (broation x = bostion b) voil_formul_stribus x = (f_A is (broation x = bostion f)) voil_formul_stribus x = (f_A is (broation x = bostion f)) voil_formul_stribus x = (f_A is (broation x = bostion f)) voil_formul_stribus x = (f_A is (broation x = bostion f)) voil_formul_stribus x = (f_A is (broation x = bostion f)) (MO_DELEXITIN_DO_ROW_ENTRY_DO_ROW_ENT	actions of Extract N (Sh. Surfames As June June 19) dependency and red. June June 19) dependency and red. Jule 19) dependency and red. Jule 19) dependency and red. Jule 19) action to the second dependency of modification order release sequence carriers - dependency to a dependency and red. Jule 19) action to the second dependency carried dependency control dependency of modification order release sequence carriers - dependency to a bit property and red. Jule 19) action to the red. Jule 19) action to the second dependency control dependency of modification order release sequence carriers - dependency to a bit property dependency of modification order release sequence carriers - dependency to a bit property dependency of modification order release sequence carriers - dependency to a bit property dependency of modification order release sequence carriers - dependency to a bit property dependency of modification order release sequence carriers - dependency to a bit property dependency of modification order release sequence carriers - dependency to a bit property dependency of modification order release sequence carriers - dependency to a bit property dependency of modification order release sequence carriers - dependency to a bit property dependency of modification order release sequence carriers - dependency to a bit property dependency or modification order release sequence carriers - dependency to a bit property dependency or modification order release sequence carriers - dependency to a bit property dependency or modification order release sequence carriers - dependency to a bit property dependency or modification order release sequence carriers - dependency to a bit property dependency or modification order release sequence carriers - dependency to a bit property dependency or modification order release sequence carriers - dependency to a bit property dependency or modification order release sequence carriers - dependency to a bit property dependency or modification order release sequence carriers - de	Constitution of the form of
LOAD of action.id thread decision val	threshvin_polation_rows s st = relation_rows s st = relation_rows st = f(\sigma	a continued and produced and a continued and a	The continue contin

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?

can we comple to zee.

- Ca	า we	comp	oile to	Pow	er?
------	------	------	---------	-----	-----

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

C++0x Operation	POWER Implementation
Non-atomic Load	ld
Load Relaxed	ld
Load Consume	1d (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?

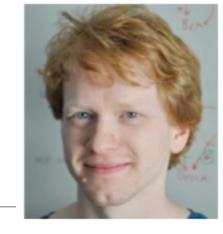
Simplifications:

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

Derivative models:

- without consume, happens-before is transitive





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



```
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);
```

```
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)

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

Thread 1

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

```
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.

```
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);
```

```
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);
```

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

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

```
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

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
```

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

ret

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

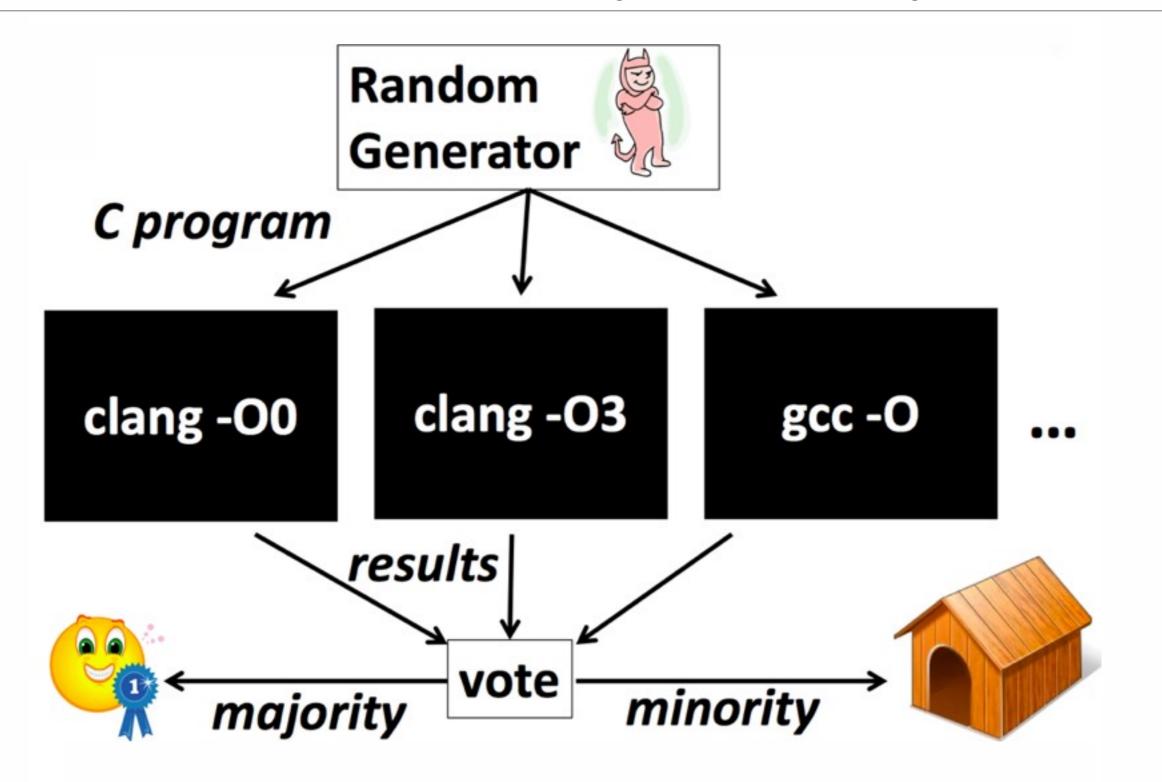
This is a concurrency compiler bug

```
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



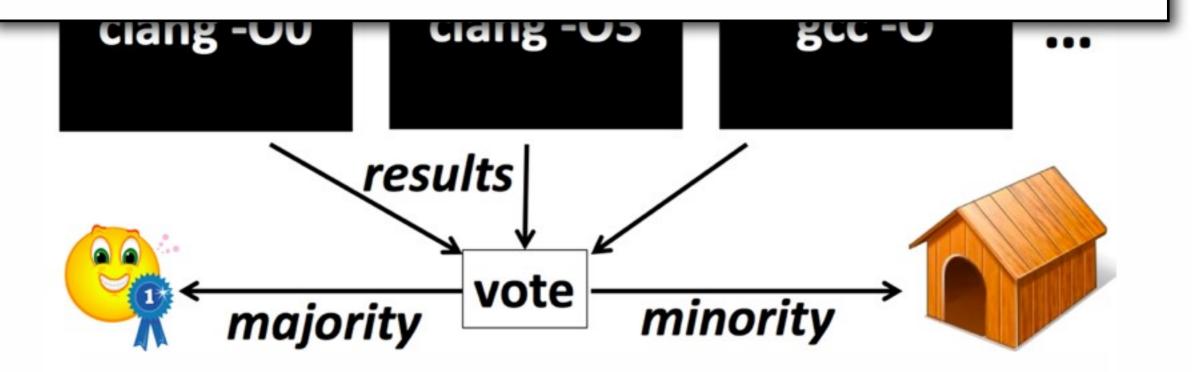
Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

Random

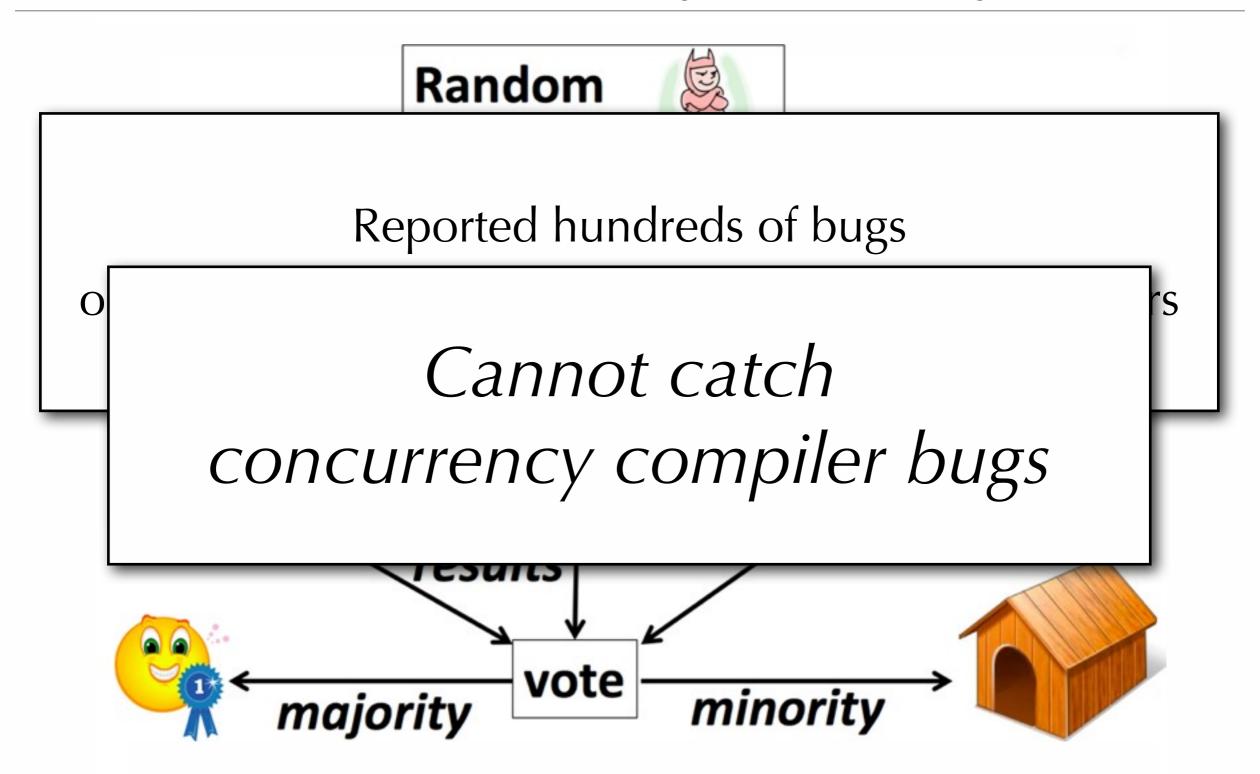


Reported hundreds of bugs on various versions of gcc, clang and other compilers



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

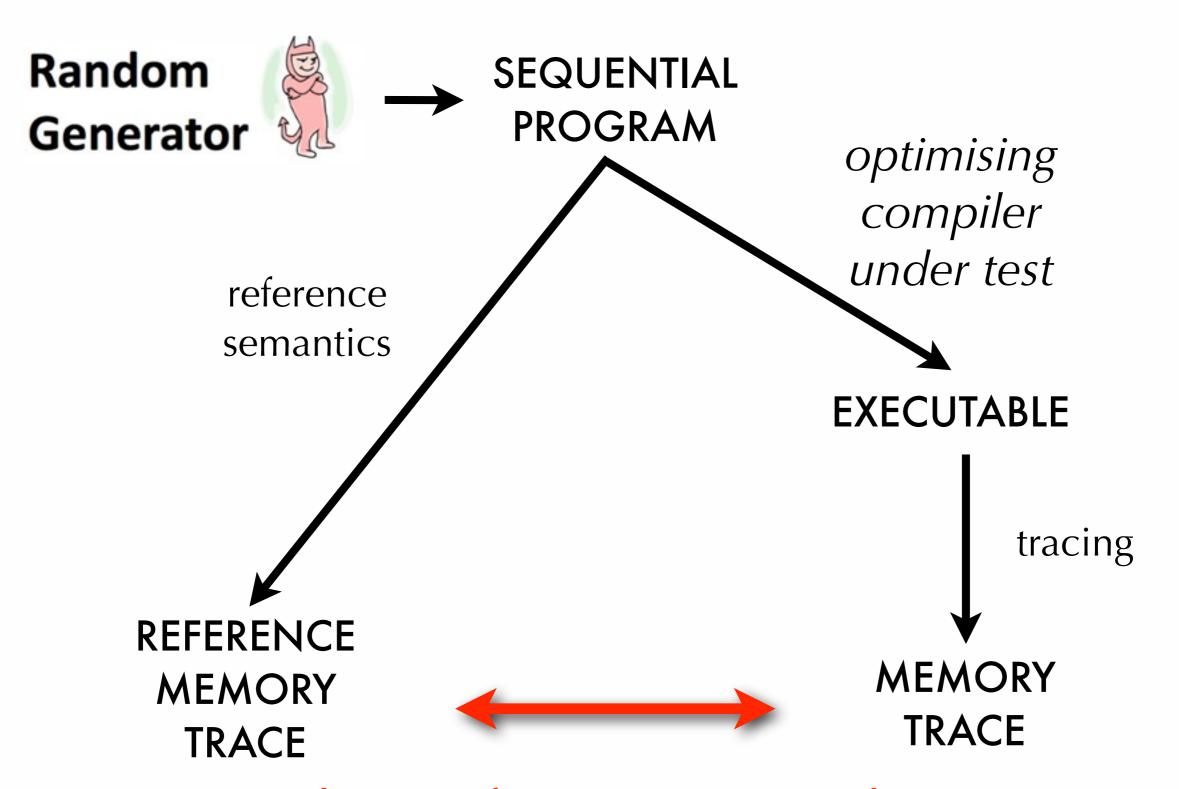


C/C++ compilers can only apply transformations sound with respect to an arbitrary non-racy concurrent context

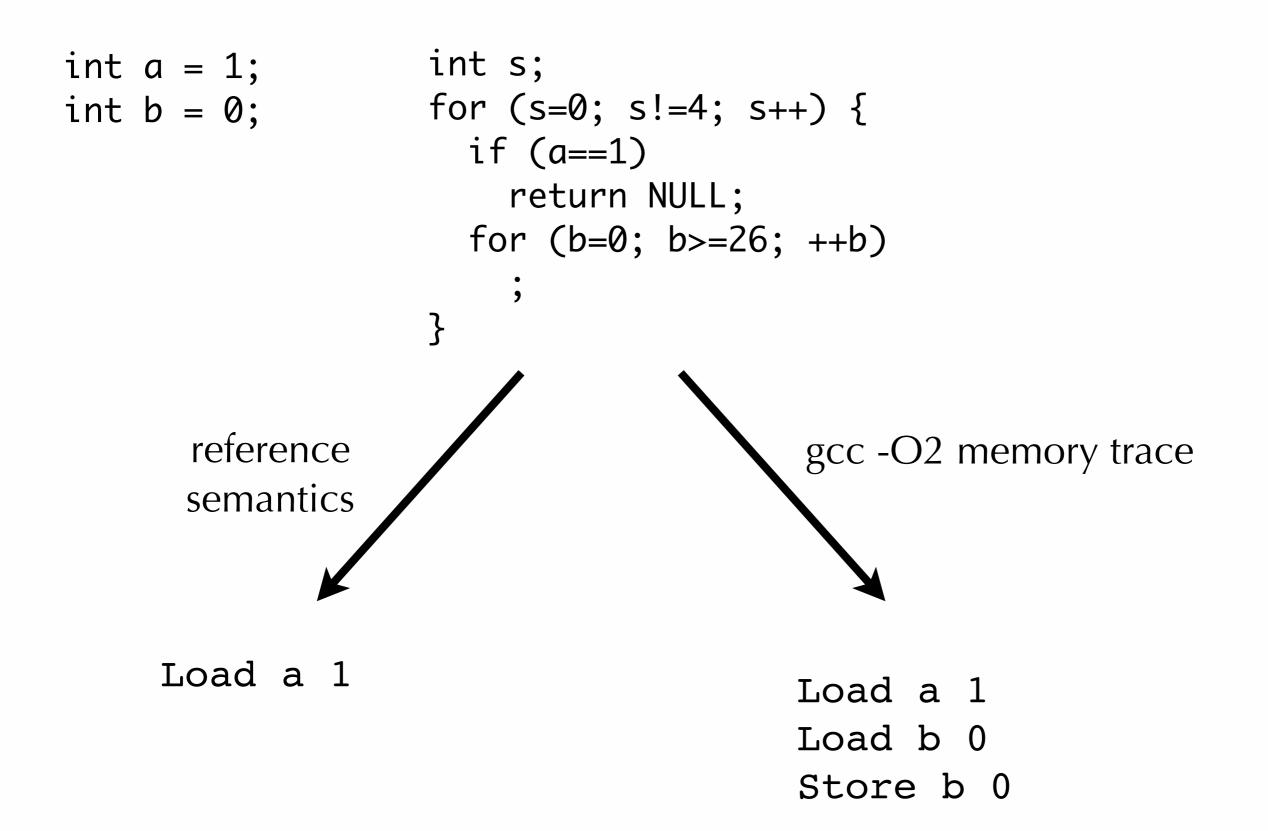
Hunt concurrency compiler bugs

_

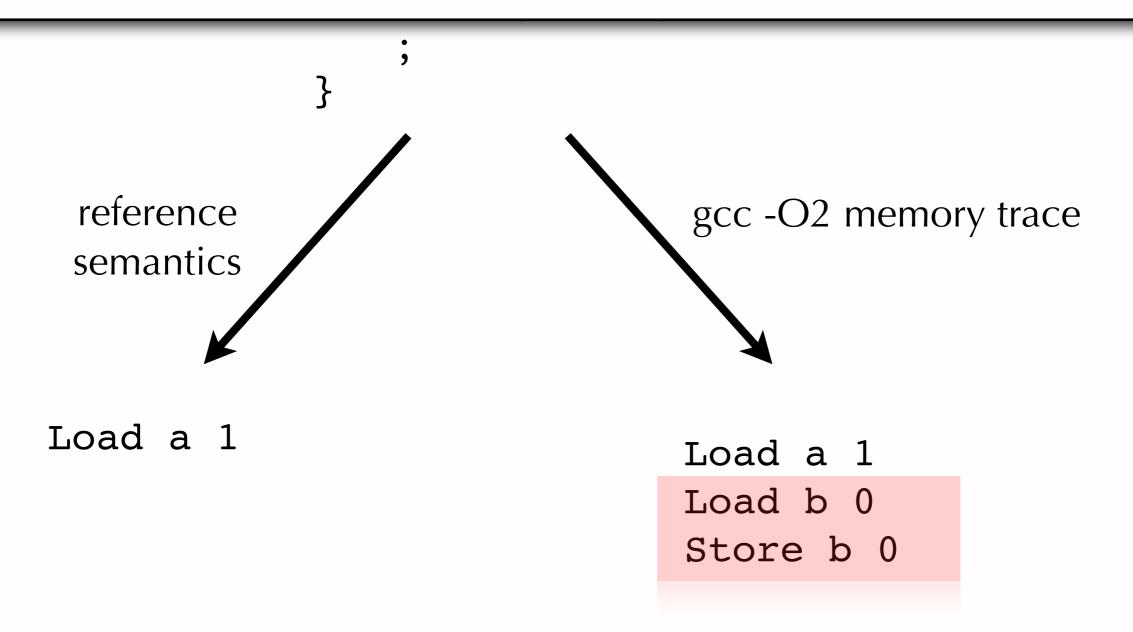
search for transformations of sequential code not sound in an arbitrary non-racy context



only transformations sound in any concurrent non-racy context?



Cannot match some events ——— 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

Compiler Writer



Semanticist



Compiler Writer



Sophisticated program analyses Fancy algorithms Source code or IR

Operations on AST

Semanticist



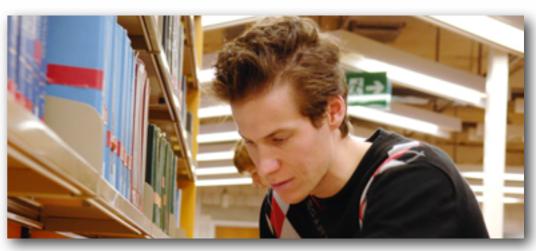
Compiler Writer



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;
}</pre>
```

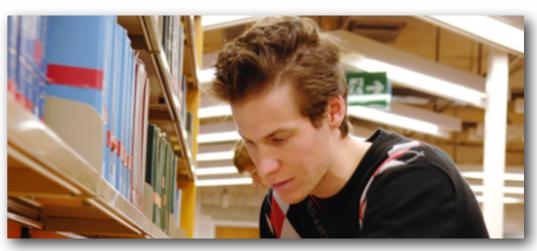
Compiler Writer



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;
}</pre>
```

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;
}</pre>
```

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;
}</pre>
```

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

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;
}</pre>
```

Semanticist



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

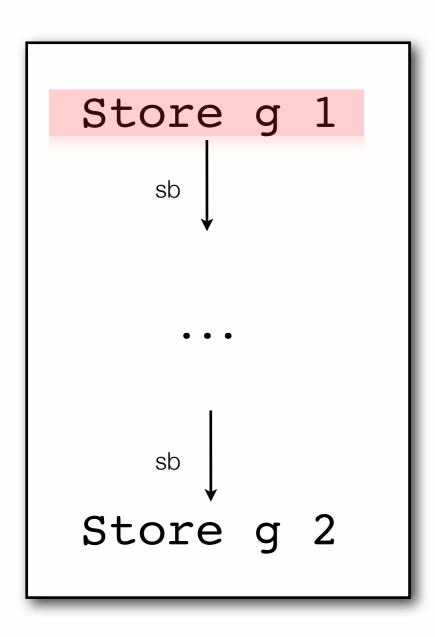
Operations on sets of events

```
Load y 42
Store z 0

Store x[0] 43
Store z 1
```

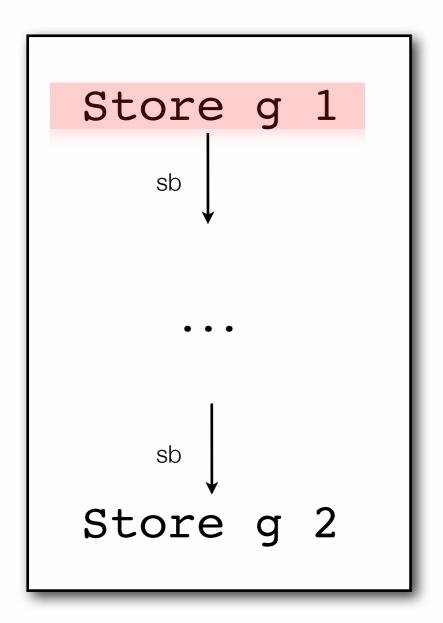
Store x[1] 43

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)
```

$$g = 0$$
; atomic $f = 0$;

Thread 1

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

Thread 2

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

$$g = 0$$
; atomic $f = 0$;

Thread 1

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

Thread 2

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

$$g = 0$$
; atomic $f = 0$;

Thread 1

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

Thread 2

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

$$g = 0$$
; atomic $f = 0$;

Thread 1

Thread 2

```
f.store(1,MO_RELEASE); while (f.load(MO_ACQUIRE)==0); printf ("%d" a)
```

$$g = 0$$
; atomic $f = 0$;

Thread 1

Thread 2

```
g = 42;
f.store(1,M0_RELEASE); while (f.load(M0_ACQUIRE)==0);
printf ("%d",g)
```

The release/acquire synchronisation guarantees that:

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

Remark: unlock ≃ release, lock ≃ acquire.

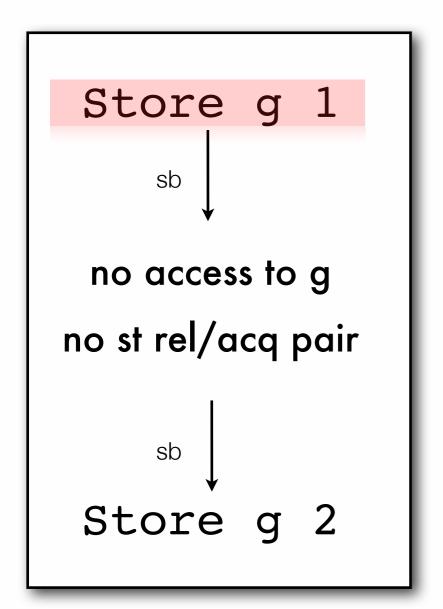
Same-thread release/acquire pairs

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

An action is a *release* if it is a possible source of a synchronisation *unlock mutex, release or seq_cst atomic write*

An action is an acquire if it is a possible target of a synchronisation lock mutex, acquire or seq_cst atomic read

Elimination of overwritten writes



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

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

Shared memory

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

Thread 1

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

Shared memory

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

Shared memory

$$g = 0$$
; atomic $f1 = f2 = 0$;

```
Thread 1 candidate overwritten write

g = 1;

f1.store(1,RELEASE);

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

same-thread release-acquire pair

g = 2;
```

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);
```

Shared memory

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

Thread 1

Thread 2

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

Thread 2 is non-racy

Shared memory

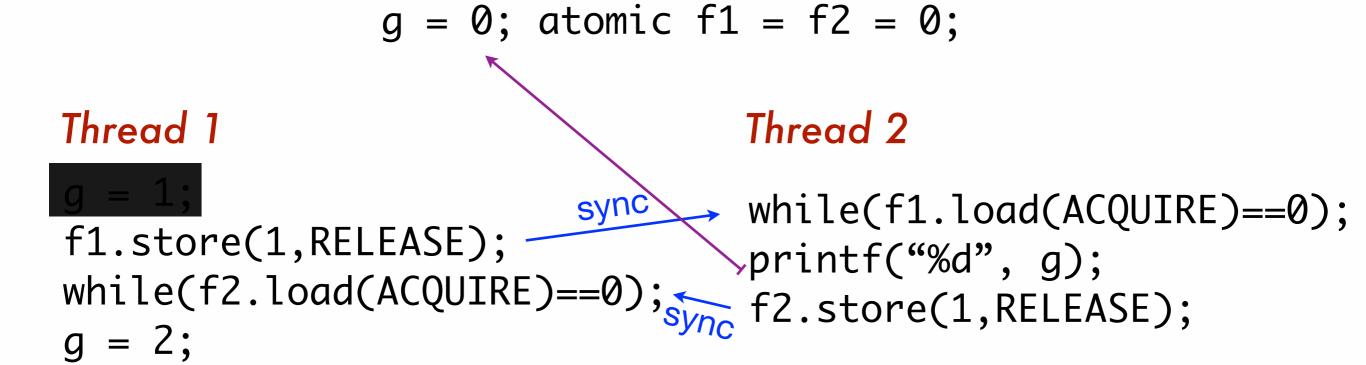
```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

Thread 2

Thread 2 is non-racy
The program should only print 1

Shared memory



Thread 2 is non-racy
The program should only print 1

If we perform overwritten write elimination it prints 0

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);
```

Shared memory

$$g = 0$$
; atomic $f1 = f2 = 0$;

Thread 1

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

```
g = 2;
```

Thread 2

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

Shared memory

$$g = 0$$
; atomic $f1 = f2 = 0$;

Thread 1

g = 1;

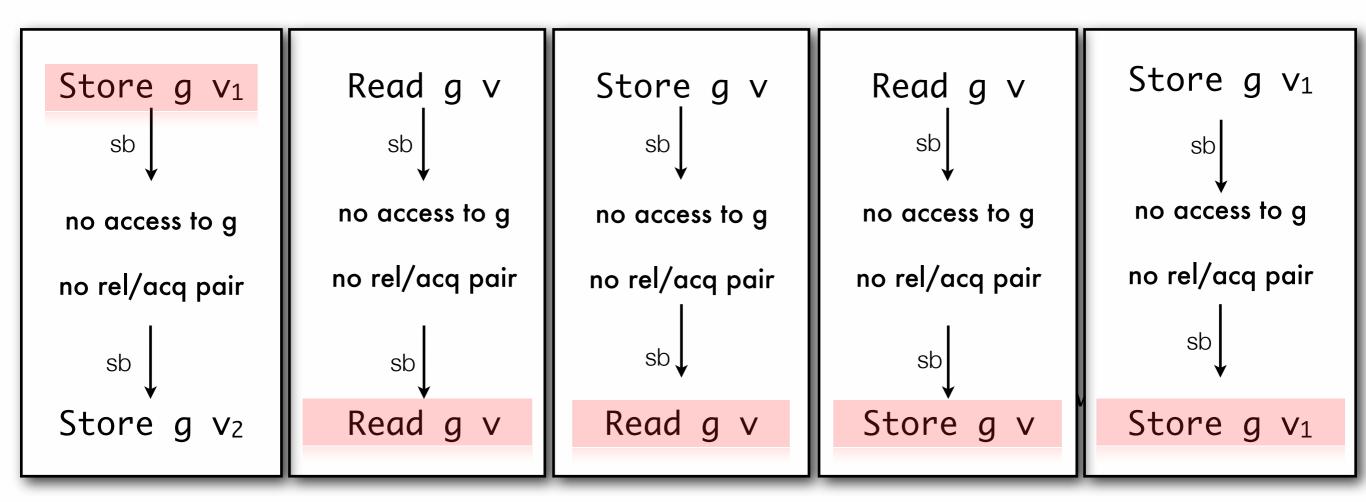
f1.store(1,RELEASE);

Thread 2

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

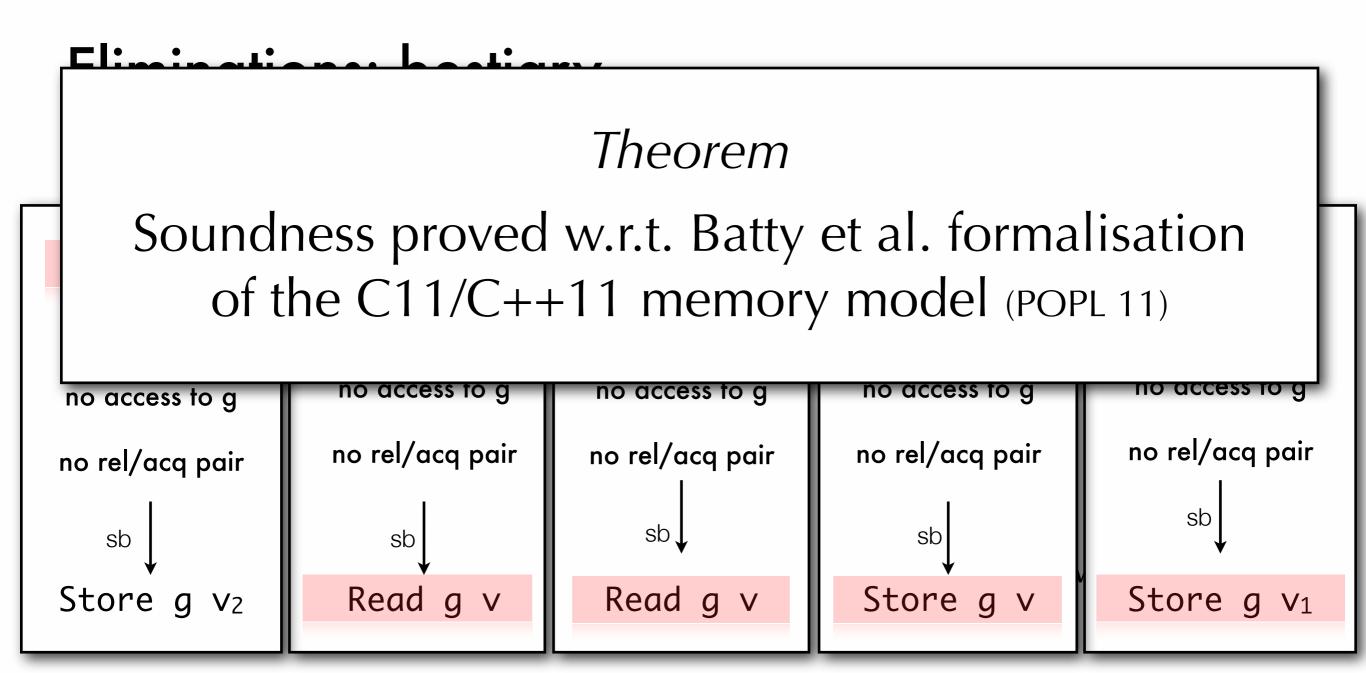
If only a release (or acquire) is present, then all discriminating contexts *are racy*. It is sound to optimise the overwritten write.

Eliminations: bestiary



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

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



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:

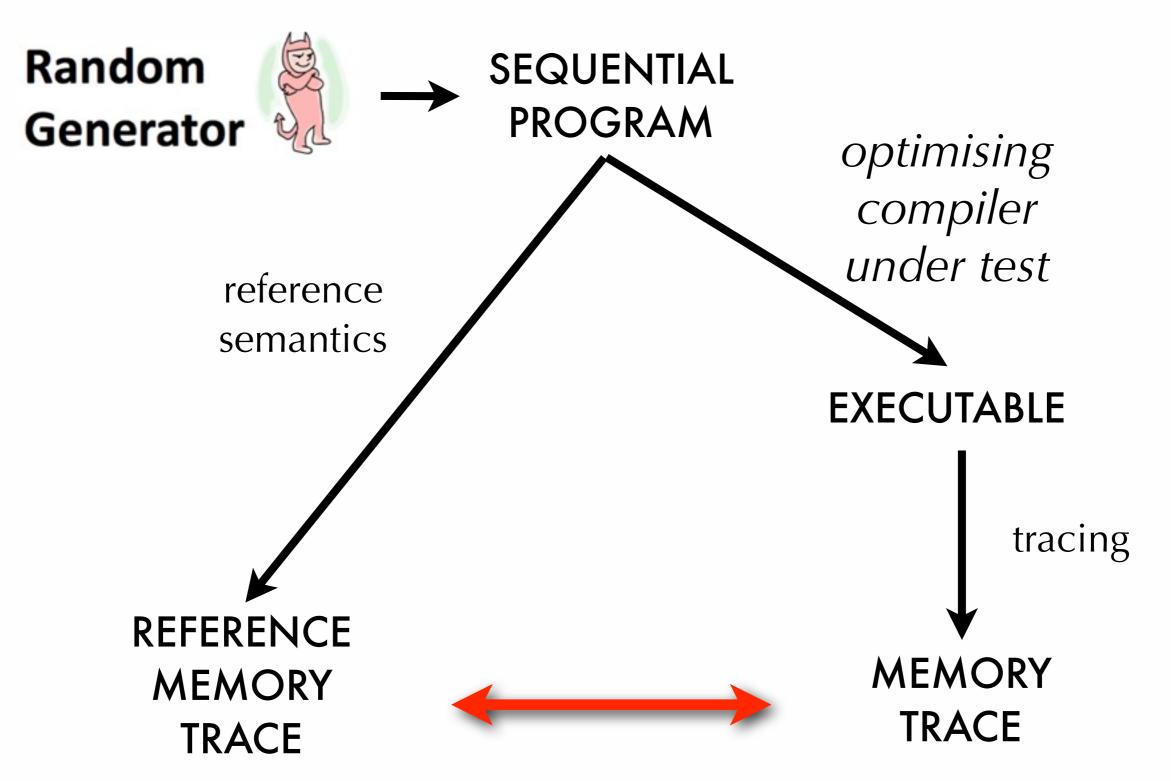
- different addresses
- no synchronisations in-between

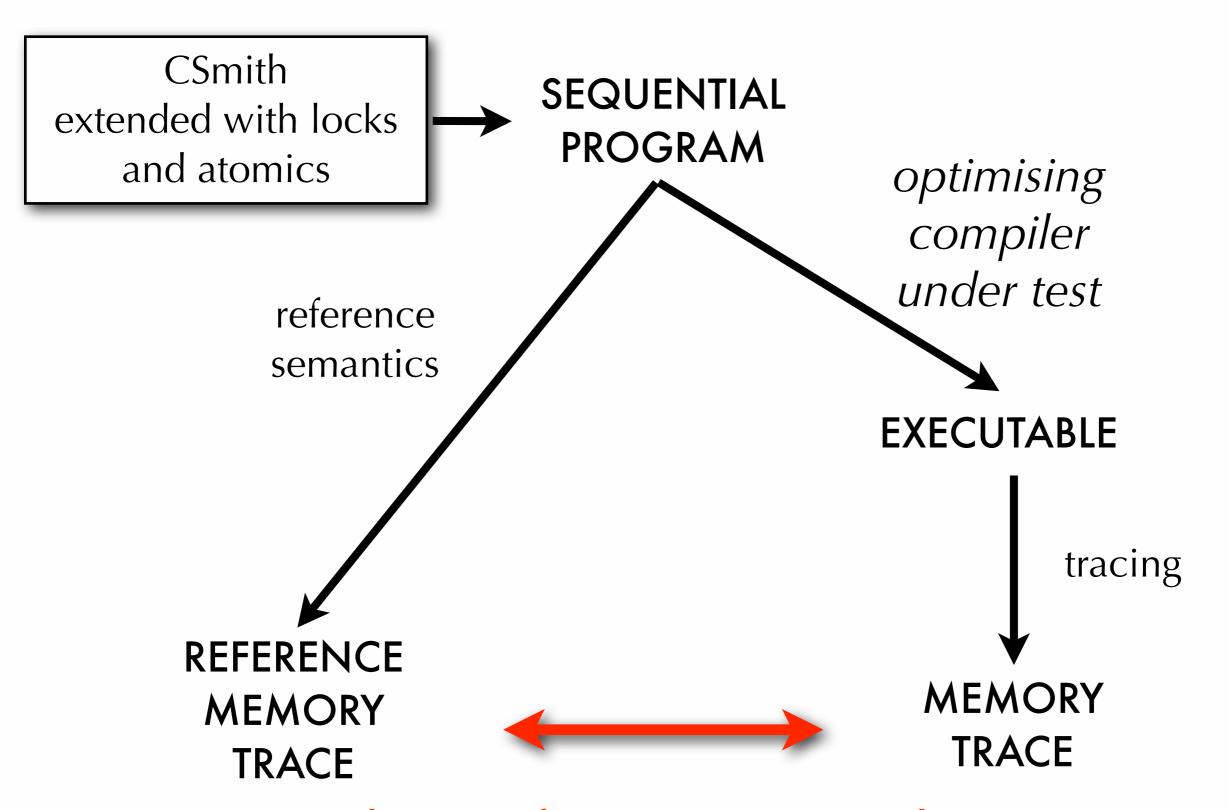
Roach-motel reordering (reordering across locks) not observed in practice

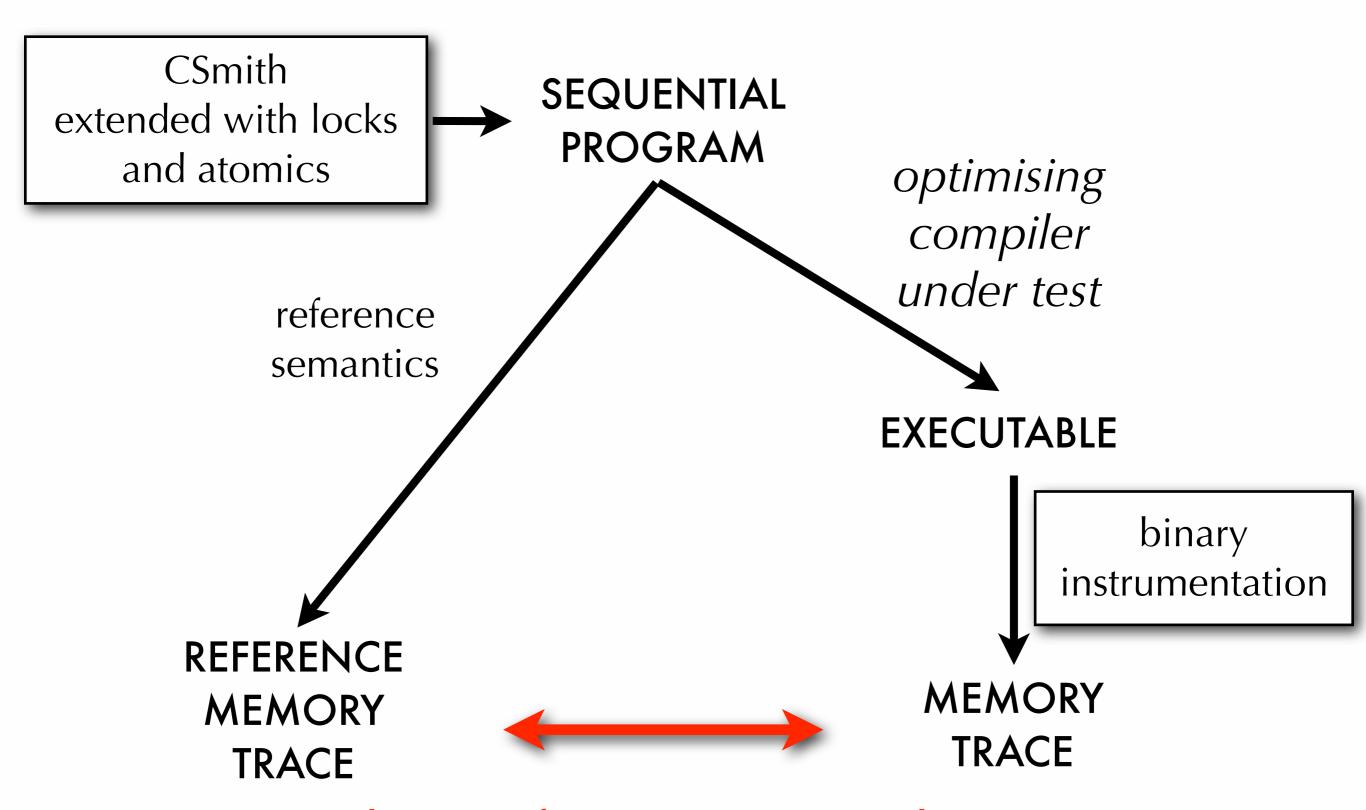
Read introductions observed in practice (gcc, clang).

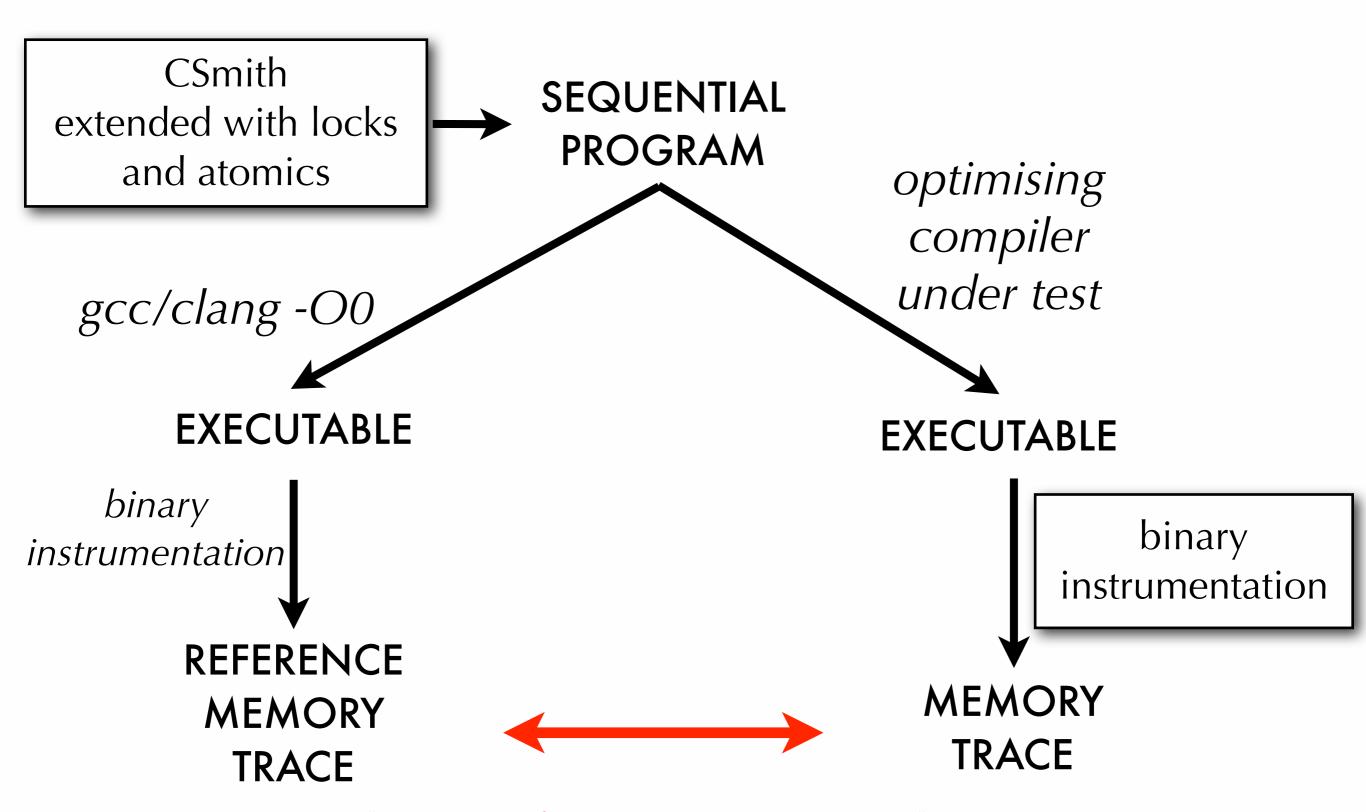
Introduction of eliminable reads proved correct. Introduction of irrelevant reads does not introduce new behaviours, but cannot be proved correct in a DRF model.

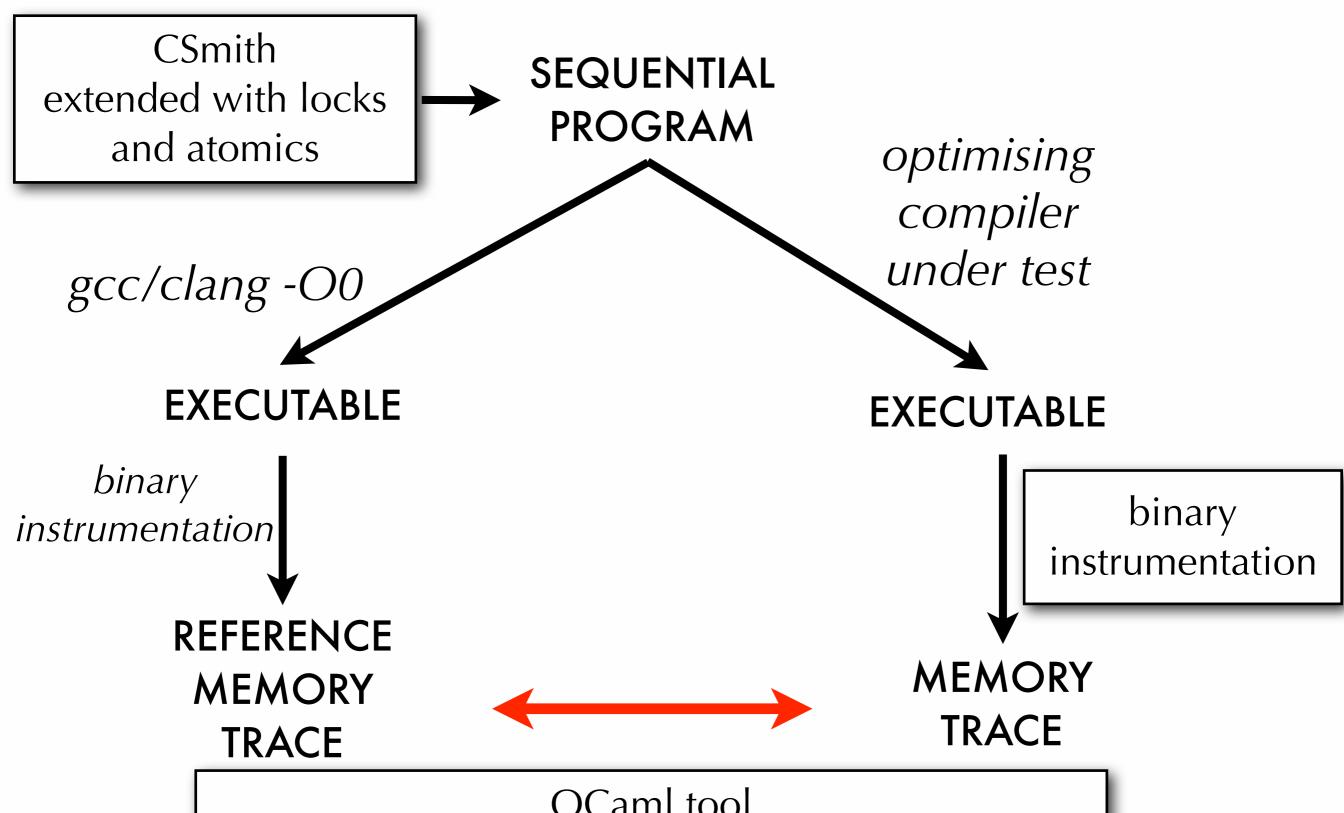
The CMMTEST Tool











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

TIMAL

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

```
int main (int, char *[]) {
atomic_uint a;
int32_t g1, g2;
                        a.load() & a.load ();
                        g2 = g1 != 0;
                      }
ALoad
                                          g1
ALoad
                                  ALoad
Load g1 0
                                  ALoad
Store
        g2
                                 Store
                                         g2
```

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:

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

Memory access synchronisation

Thread 1
$$x = y = 0$$

$$y = 1$$

$$x.store(1,MO_RELEASE)$$

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

$$r2 = y$$

$$\stackrel{happens-before}{\longrightarrow} = \\ (\underbrace{\hspace{1cm} \begin{array}{c} \text{sequenced-before} \\ \end{array}} \begin{array}{c} \text{synchronizes-with} \\ \end{array})^{+}$$

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

Understanding MO_RELAXED

$$x = y = 0$$

Thread 1

Thread 2

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 Thread 2

WELL DEFINED

but r2 = 0 is possible

Understanding MO_RELAXED

$$x = y = 0$$

Thread 1 Thread 2

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

Thread 2

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

$$x = y = 0$$

Thread 2

$$r1 = x$$

$$y = r1$$

$$r2 = y$$

 $x = 42$

$$x = 42$$

Out-of-thin-air

Thread 1

$$x = y = 0$$

Thread 2

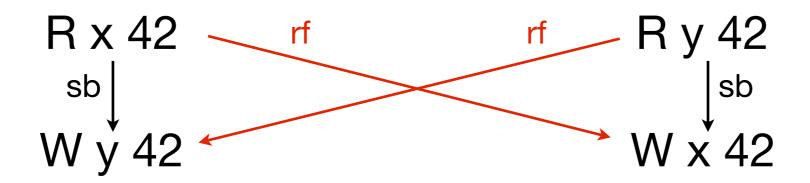
$$r1 = x$$

$$y = r1$$

$$r2 = y$$

$$x = 42$$

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



Intuition

the compiler (or hardware) can reorder independent accesses

Thread 1

$$x = y = 0$$

Thread 2

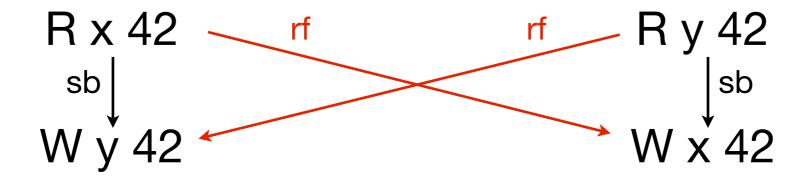
$$r1 = x$$

$$y = r1$$

$$r2 = y$$

$$x = 42$$

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



Out-of-thin-air reads

Thread 1

$$x = y = 0$$

Thread 2

$$r1 = x$$

y = r1

$$r2 = y$$

$$x = r2$$

Out-of-thin-air reads

Thread 1

$$x = y = 0$$

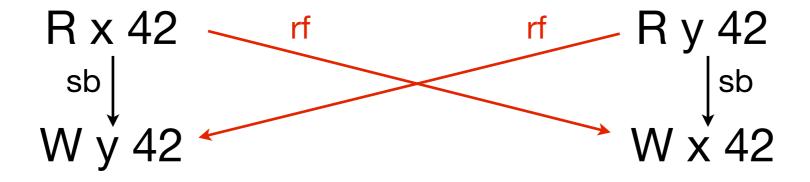
Thread 2

$$r1 = x$$
 $y = r1$

$$r2 = y$$
 $x = r2$

$$r1 = r2 = 42$$

is also an allowed execution



the value 42 appears out-of-thin-air

Thread 1

$$x = y = 0$$

Thread 2

$$r1 = x$$

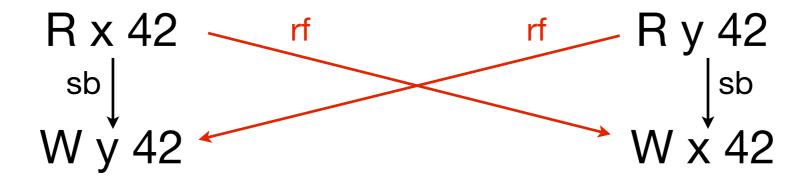
 $y = r1$

$$r2 = y$$

$$x = r2$$

$$r1 = r2 = 42$$

is also an allowed execution



Speculation can justify out-of-thin-air reads

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

Speculation can justify out-of-thin-air reads

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

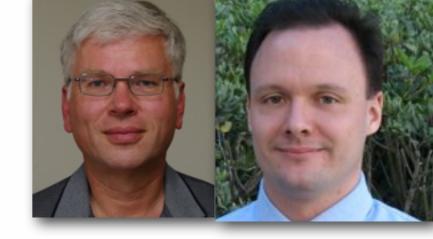
$$y := 42$$

 $r1 := x$
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!

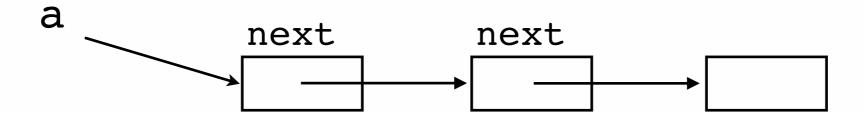


```
struct foo {
  atomic<struct foo *> next;
}
struct foo *a;
```

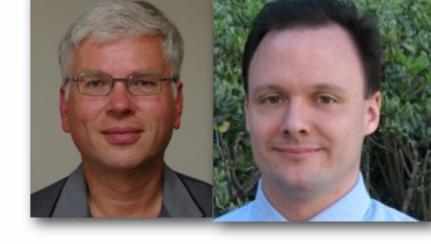


$$r1 = a->next$$

 $r1->next = a$

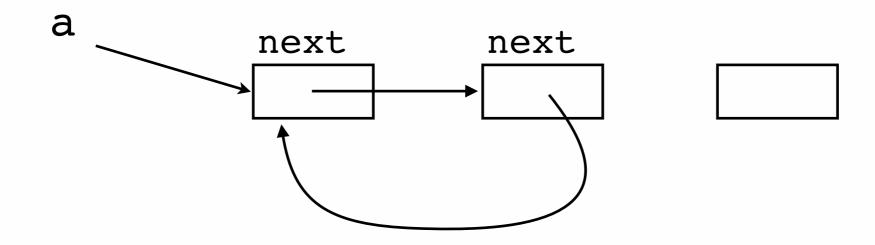


```
struct foo {
  atomic<struct foo *> next;
}
struct foo *a;
```



$$r1 = a->next$$

 $r1->next = a$



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



$$r1 = a->next$$

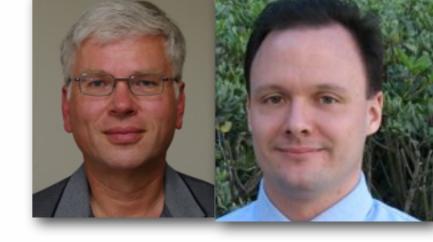
$$r1->next = a$$

Thread 2

$$r2 = b->next$$

 $r2->next = b$

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

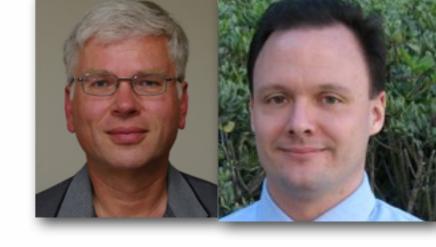


Thread 1 Thread 2

$$r1 = a-$$
next $r2 = b-$ next $r1-$ next = a $r2-$ next = b

If a and b initially reference disjoint data-structures we expect a and b to remain disjoint

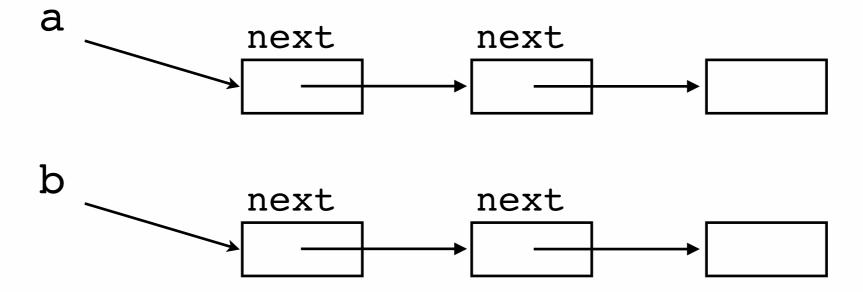
```
struct foo {
  atomic<struct foo *> next;
}
struct foo *a, *b;
```



r1 = a->nextr1->next = a Thread 2

$$r2 = b->next$$

 $r2->next = b$

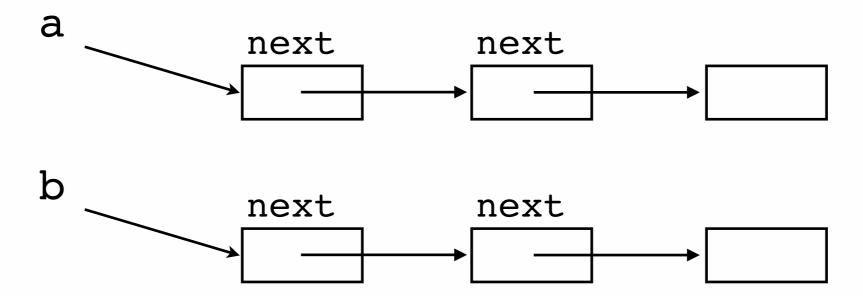


If the compiler speculates r1=b and r2=a, then the store r1->next=a justifies r2=b->next assigning r2=a (and symmetrically to justify r1=b)

Thread 1 Thread 2

$$r2 = b->next$$

 $r2->next = b$



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

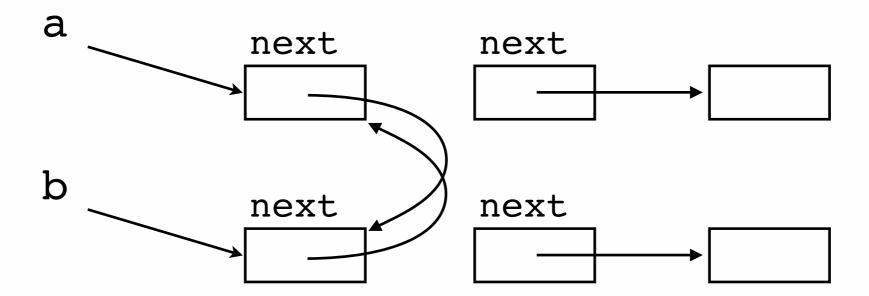
Thread 1 Thread 2

$$r1 = a->next$$

 $r1->next = a$

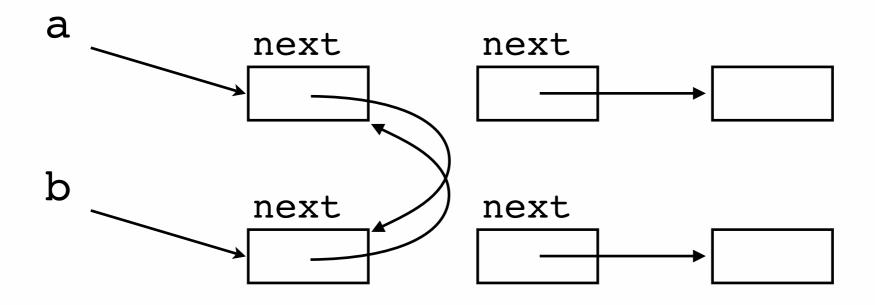
$$r2 = b->next$$

 $r2->next = b$



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

Break our basic intuitions about memory and sharing!





Common compiler optimisations are unsound in C11

$$x = y = a = 0$$

$$x = y = a = 0$$

Remark 1

This code is not racy!

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

$$x = y = a = 0$$

Remark 2

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

is the only possible final state

$$x = y = a = 0$$

Consider sequentialisation:

$$C \mid \mid D \implies C ; D$$

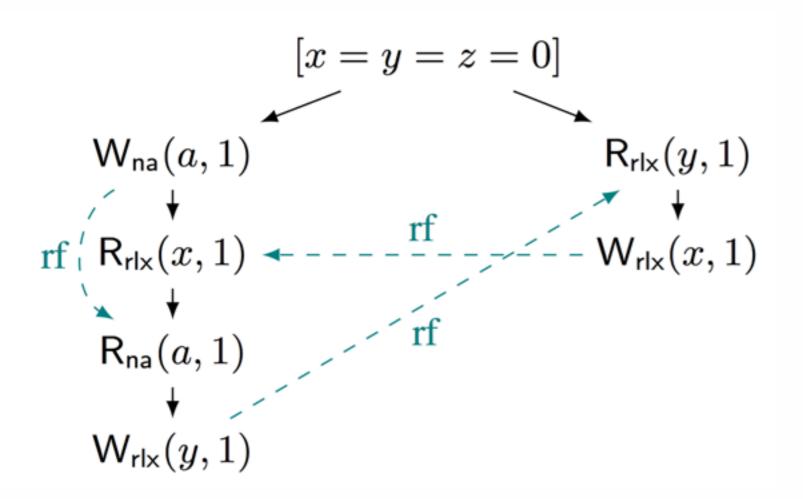
(ought to be correct)

$$x = y = a = 0$$



$$x = y = a = 0$$

$$\begin{vmatrix} a = 1 \\ if (x.load(rlx)==42) & if (y.load(rlx)==42) \\ y.write(42,rlx) & if (a==1) \end{vmatrix}$$



$$a = 1$$

$$x = y = 42$$
is also possible

x.write(42,rlx)

```
x = y = a = 0
\begin{vmatrix}
a = 1 \\
if (x.load(rlx)==42) \\
y.write(42,rlx)
\end{vmatrix}
if (y.load(rlx)==42)
if (a==1) \\
x.write(42,rlx)
```

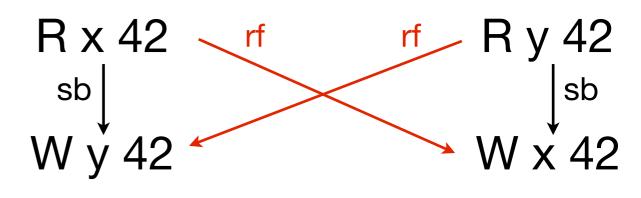
Break common source-to-source (or LLVM IR - to - LLVM IR) compiler optimisations

including expression linearisation and roach-motel reorderings

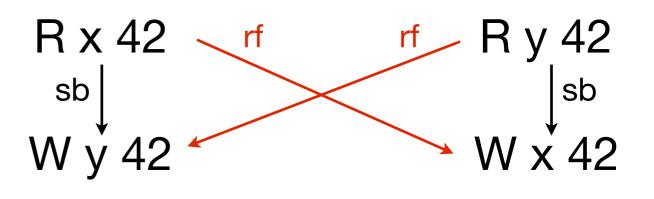


Are there any solutions?

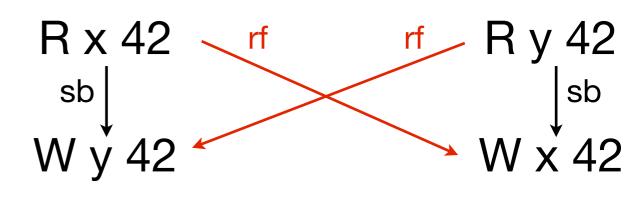
Thread 0	Thread 1
r1 = x	r2 = y
y = r1	x = 42



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

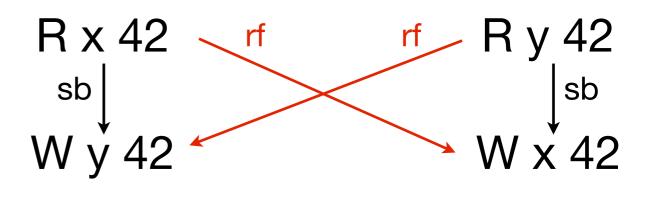


Thread 0	Thread 1
r1 = x	r2 = y
y = r1	x = 42

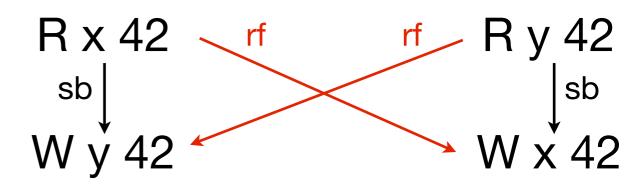


r1 = r2 = 42. Can you spot the difference?

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

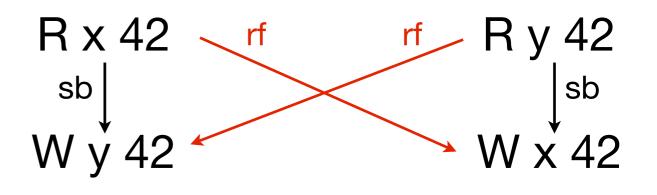


Thread 0	Thread 1
r1 = x	r2 = y
y = r1	x = 42



The "bad" example has a cycle of dependencies.

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

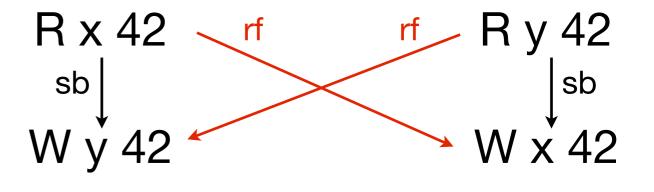


Solution 1.

Prohibit executions with dependency cycles

The "bad" example has a cycle of dependencies.

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



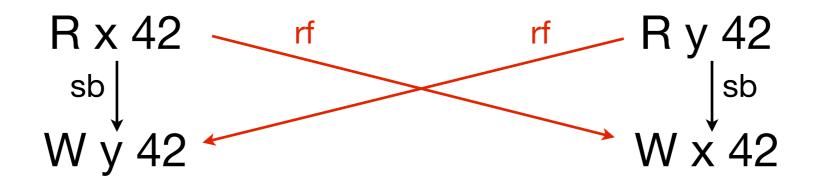
Compiler writers do not want to track all dependencies

Compiler writers do not want to track all dependencies

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

Solution 2. Brute force

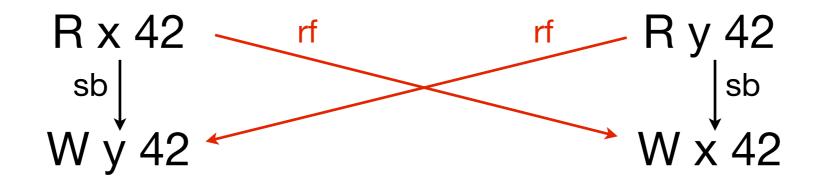
Disallow cycles altogether



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

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

Disallow cycles altogether



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

Solution 3. less brute force

Allow cycles but make this racy

by allowing a to read 1

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

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 highlevel programming languages.

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

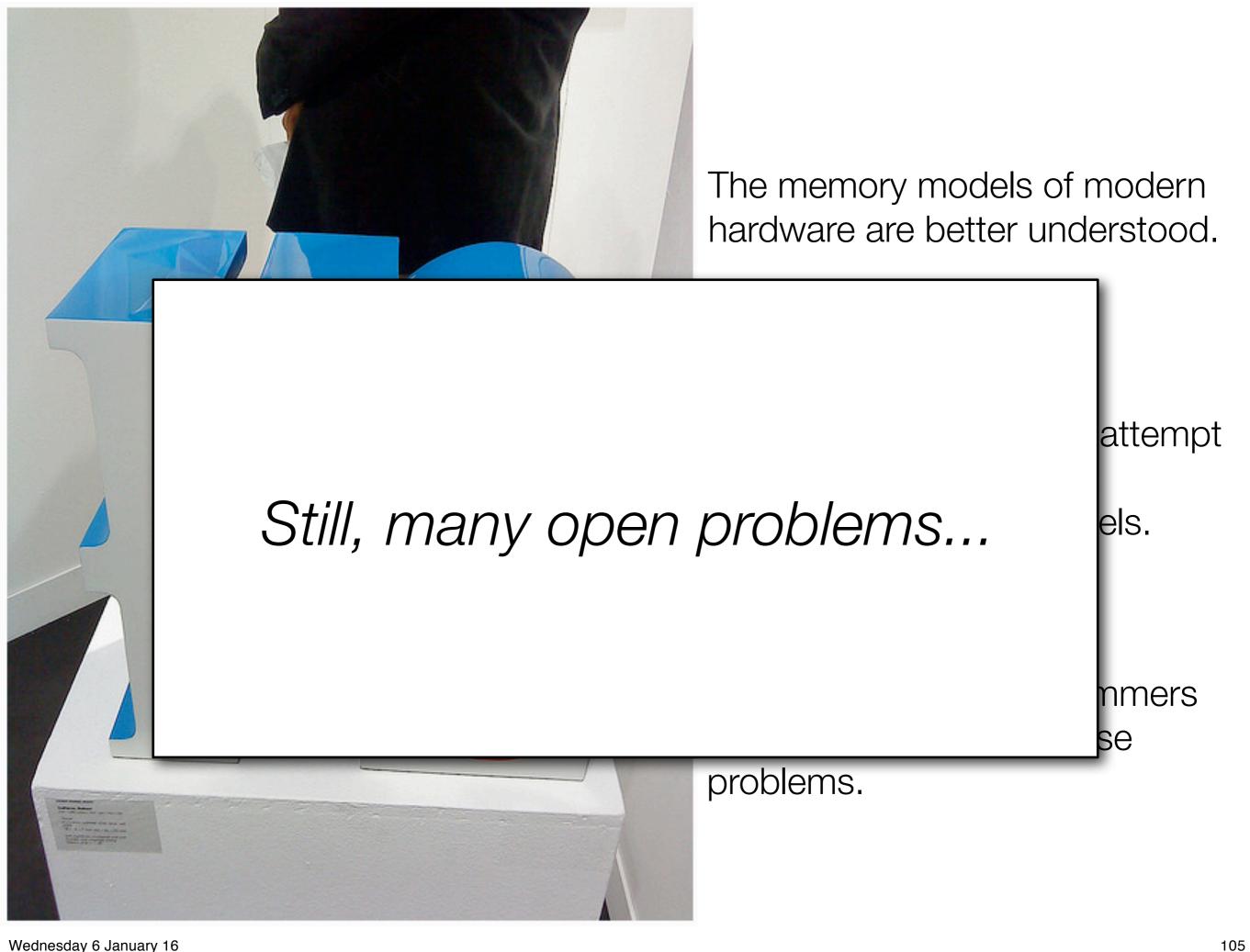
After these lectures, you might have the feeling that multicore programming is a mess and things can't just work.

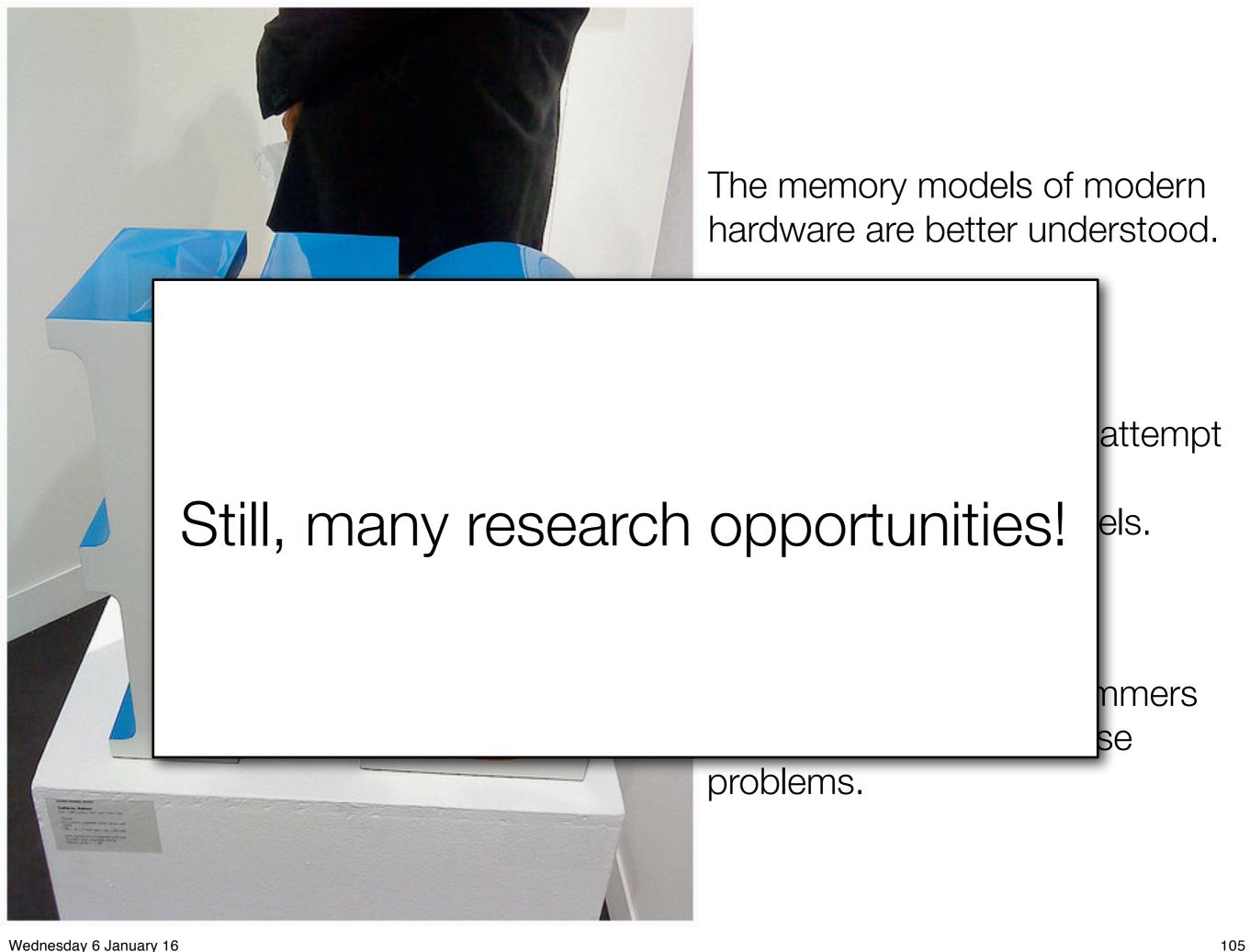


The memory models of modern hardware are better understood.

Programming languages attempt to specify and implement reasonable memory models.

Researchers and programmers are now interested in these problems.

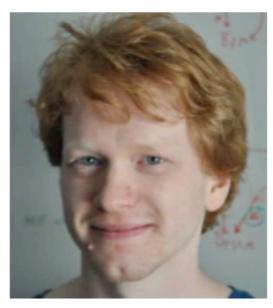






All these lectures are based on work done with/by my colleagues. Thank you!







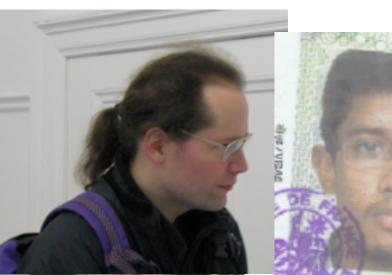










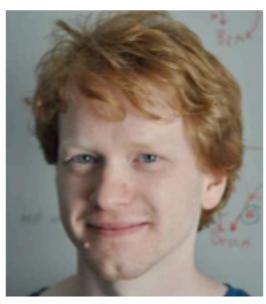




And thank you all for attending these lectures!

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









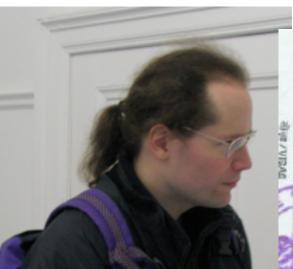




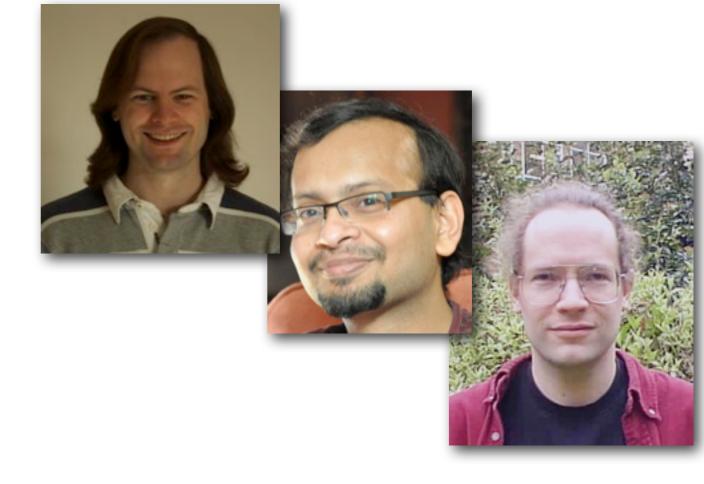










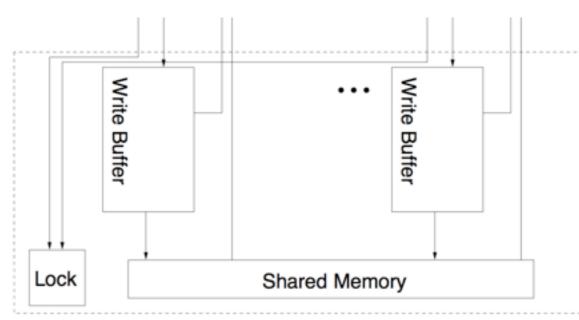


4. Sketch of an operational formalisation of x86-TSO

...starting with a formalisation of SC

Separate language and memory semantics

```
class ArrayWrapper
    public:
        ArrayWrapper (int n)
            : p vals( new int[ n ] )
            , size(n)
        // copy constructor
                                                                                  Write Buffer
        ArrayWrapper (const ArrayWrapper& other)
            : p vals( new int[ other. size ] )
            , _size( other._size )
            for ( int i = 0; i < size; ++i )</pre>
                p vals[ i ] = other. p vals[ i ];
        ~ArrayWrapper ()
                                                                          Lock
            delete [] p vals;
    private:
    int * p vals;
    int size;
```



program semantics defined via an LTS

memory semantics defined via an LTS

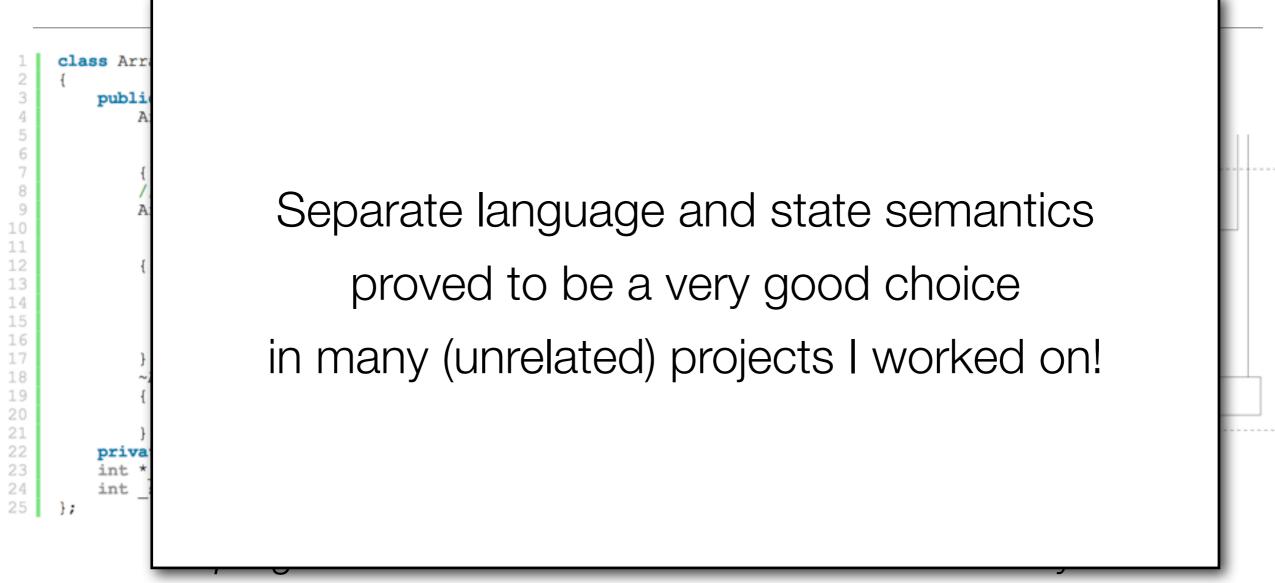
Labels for interaction:

W_t[a]v: a write of value v to address a by thread t

R_t[a]v: a read of v from a by t by thread t

+ other events for barriers and locked instructions

Separate language and memory semantics



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

```
address (or pointer value)
location, x, m
         integer
integer, n
thread_id, t thread id
k, i, j
                              expression
expression, e
                                integer literal
                                read from pointer
                    *x
                   *x = e write to pointer
                |e;e'| sequential composition
                  e + e'
                                plus
                              process
process, p
                                thread
                                parallel composition
```

What can a thread do in isolation?

$$e \xrightarrow{l} e'$$

 $e \stackrel{l}{\rightarrow} e'$ | e does l to become e'

$$\xrightarrow[*x \xrightarrow{\mathsf{R}\,x=n} n]{\mathsf{READ}}$$

$$x = n \xrightarrow{\mathsf{W} \, x = n} n$$
 WRITE

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

$$\overline{n;e \xrightarrow{\tau} e}$$
 SEQ

$$rac{e_1 \stackrel{l}{
ightarrow} e_1'}{e_1; e_2 \stackrel{l}{
ightarrow} e_1'; e_2}$$
 SEQ_CONTEXT

$$rac{e_1 \stackrel{l}{
ightarrow} e_1'}{e_1 + e_2 \stackrel{l}{
ightarrow} e_1' + e_2}$$
 PLUS_CONTEXT_1

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

$$\frac{n=n_1+n_2}{n_1+n_2\stackrel{ au}{
ightarrow} n}$$
 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:

$$(*x = *y); *x \xrightarrow{Ry=7} \xrightarrow{Wx=7} \xrightarrow{\tau} \xrightarrow{Rx=9} 9$$

Lifting to processes

$$p \xrightarrow{l_t} p'$$

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

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

Actions are labelled by the thread that performed the action.

$$\frac{p_1 \xrightarrow{l_t} p_1'}{p_1|p_2 \xrightarrow{l_t} p_1'|p_2} \quad \mathsf{PAR_CONTEXT_LEFT}$$

$$\frac{p_2 \xrightarrow{l_t} p_2'}{p_1|p_2 \xrightarrow{l_t} p_1|p_2'} \quad \mathsf{PAR_CONTEXT_RIGHT}$$

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'

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

$$\overline{M \xrightarrow{\mathsf{W}\, x = n} M \oplus (x \mapsto n)}$$
 MWRITE

SC semantics: whole system transitions

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

$$\begin{array}{c} p \xrightarrow{\mathsf{R}_t \ x=n} \ p' \\ \hline M \xrightarrow{\mathsf{R} \ x=n} \ M' \\ \hline \langle p, \ M \rangle \xrightarrow{\mathsf{R}_t \ x=n} \ \langle p', \ M' \rangle \end{array} \quad \mathsf{SREAD} \\ p \xrightarrow{\mathsf{W}_t \ x=n} \ p' \\ \hline M \xrightarrow{\mathsf{W} \ x=n} \ M' \\ \hline \langle p, \ M \rangle \xrightarrow{\mathsf{W}_t \ x=n} \ \langle p', \ M' \rangle} \quad \mathsf{SWRITE} \\ \hline \frac{p \xrightarrow{\tau_t} \ p'}{\langle p, \ M \rangle \xrightarrow{\tau_t} \ \langle p', \ M \rangle} \quad \mathsf{STAU} \end{array}$$

Synchronising between the processes and the memory.

SC semantics, example

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

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.

$$M \xrightarrow{l} M'$$

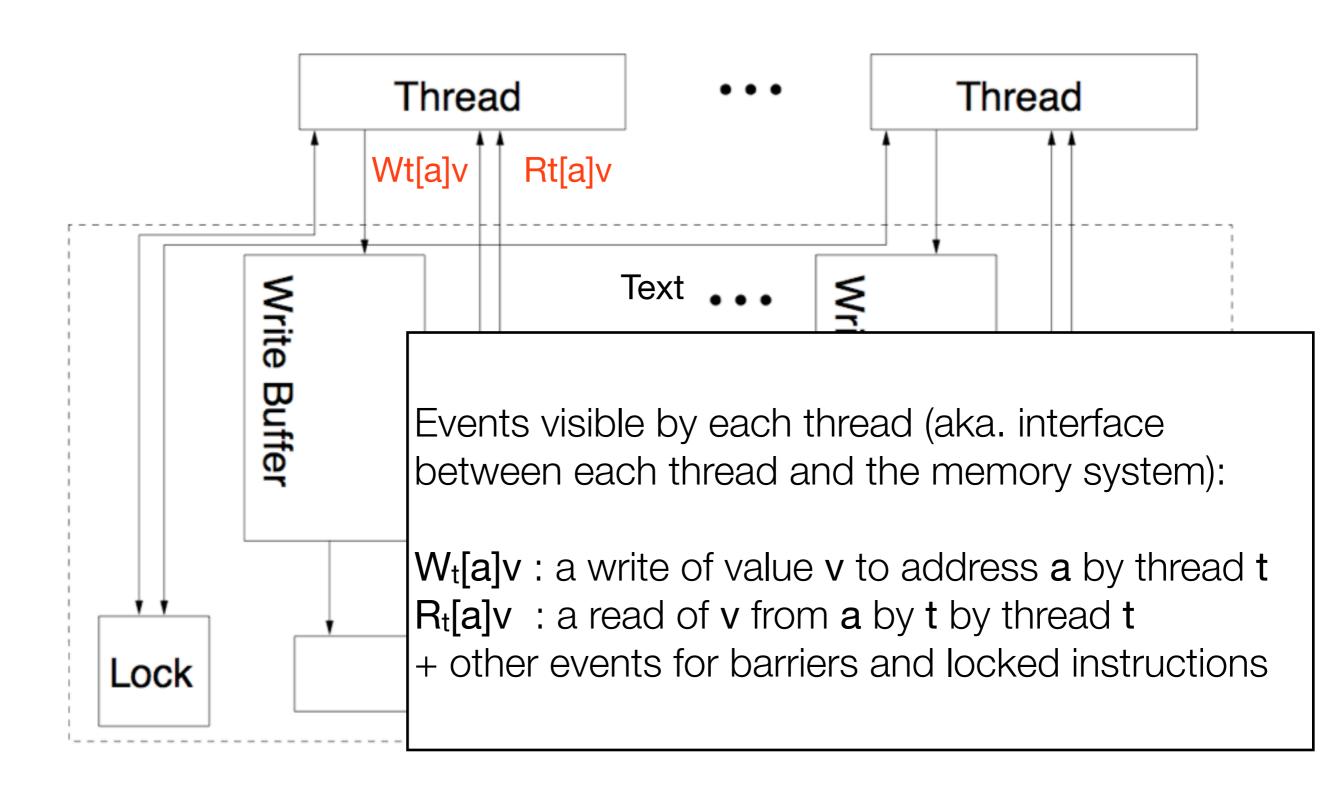
 $M \xrightarrow{l} M'$ M does l to become M'

$$\frac{M(x) = n}{M \xrightarrow{\mathsf{R} \, x = n} M}$$
 MREAD

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

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 -> value option
```

B : tid -> (address * value) list

L : tid option

where:

M is the shared memory, mapping addresses to values

B gives the store buffer for each thread

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

x86-tso abstract machine: selected transition rules

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

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

RM: Read from memory

$$\text{not_blocked}(s, t)$$
 $s.M(x) = v$
 $\text{no_pending}(s.B(t), x)$
 $\xrightarrow{R_t x = v} s$

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

$$\operatorname{not_blocked}(s,t)$$
 $\exists b_1 \ b_2. \ s.B(t) = b_1 ++ [(x,v)] ++ b_2$
 $\operatorname{no_pending}(b_1,x)$
 $s \xrightarrow{\mathsf{R}_t \ x=v} s$

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

x86-tso abstract machine: selected transition rules

WB: Write to write buffer

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

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

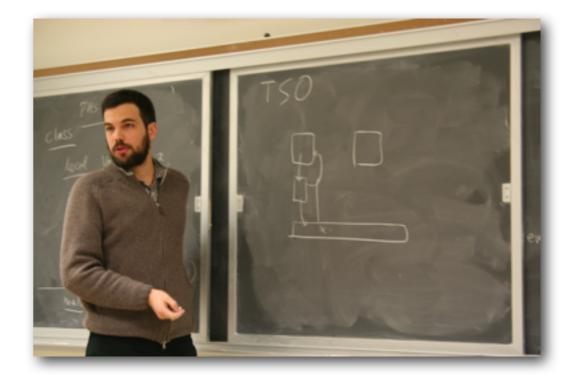
WM: Write from write buffer to memory

$$not_blocked(s, t)$$

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

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

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. Veryfing fence elimination optimisations

aka reasoning on the x86TSO operational memory model and compiler correctness

CompCertTSO

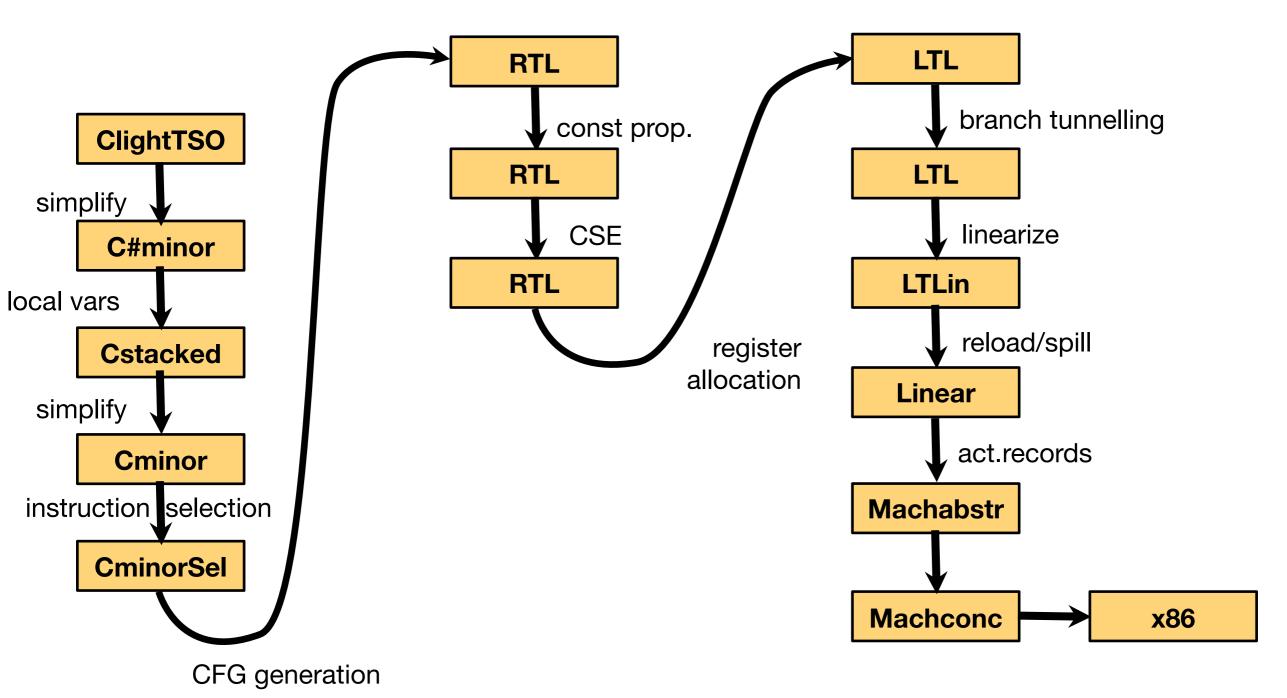










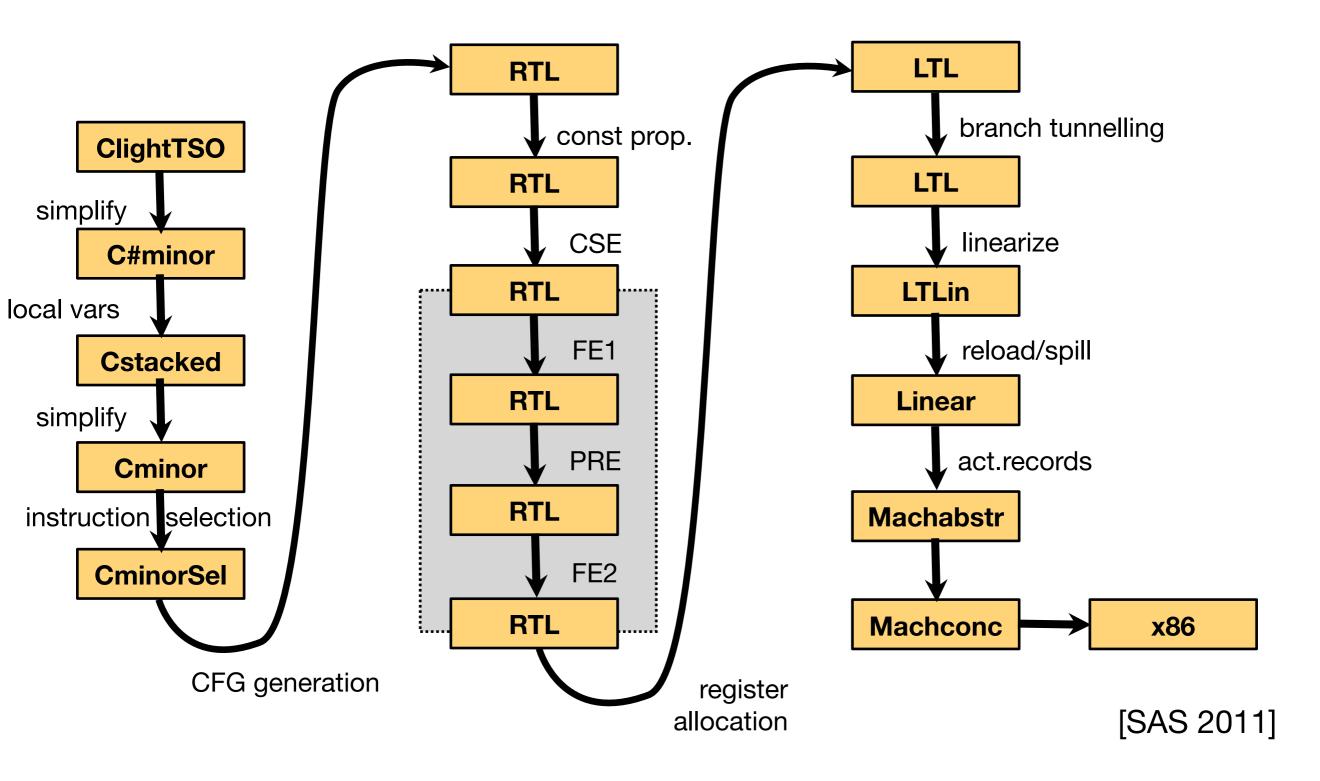


[POPL 2011]









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 11vm 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, init $\in prg \to \mathbb{P}(states)$
- a transition relation, $\rightarrow \in \mathbb{P}(states \times event) \times states)$ 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

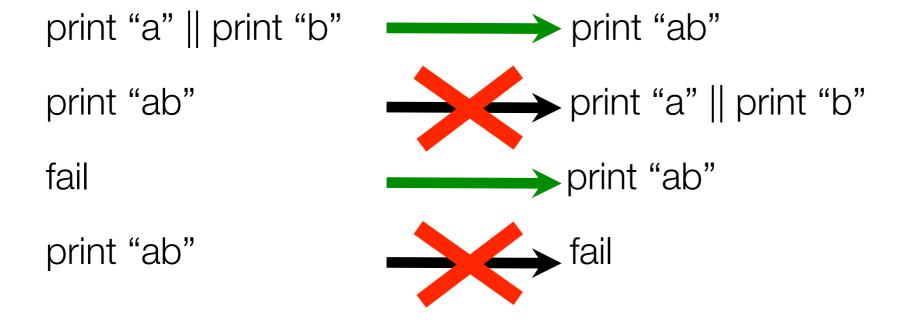
- Infinite sequences of call & return events;

NB: Erroneous computations become undefined after the first error.

Compiler correctness



traces(source_program) ⊇ traces(target_program)



Fence instructions prevent hardware reorderings

E.g., on x86-TSO:

$$[x] = [y] = 0$$

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

$$EAX = EBX = 0$$
 allowed

$$[x] = [y] = 0$$

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

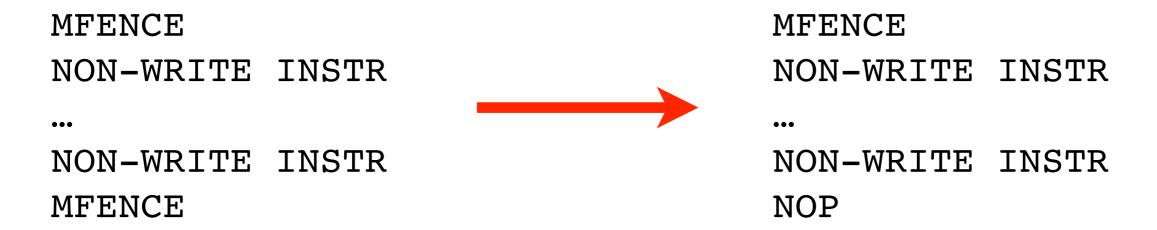
2. Fences can be expensive

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



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

Generalisation:



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 $\{\bot, \top\}$

Associate to each program point:

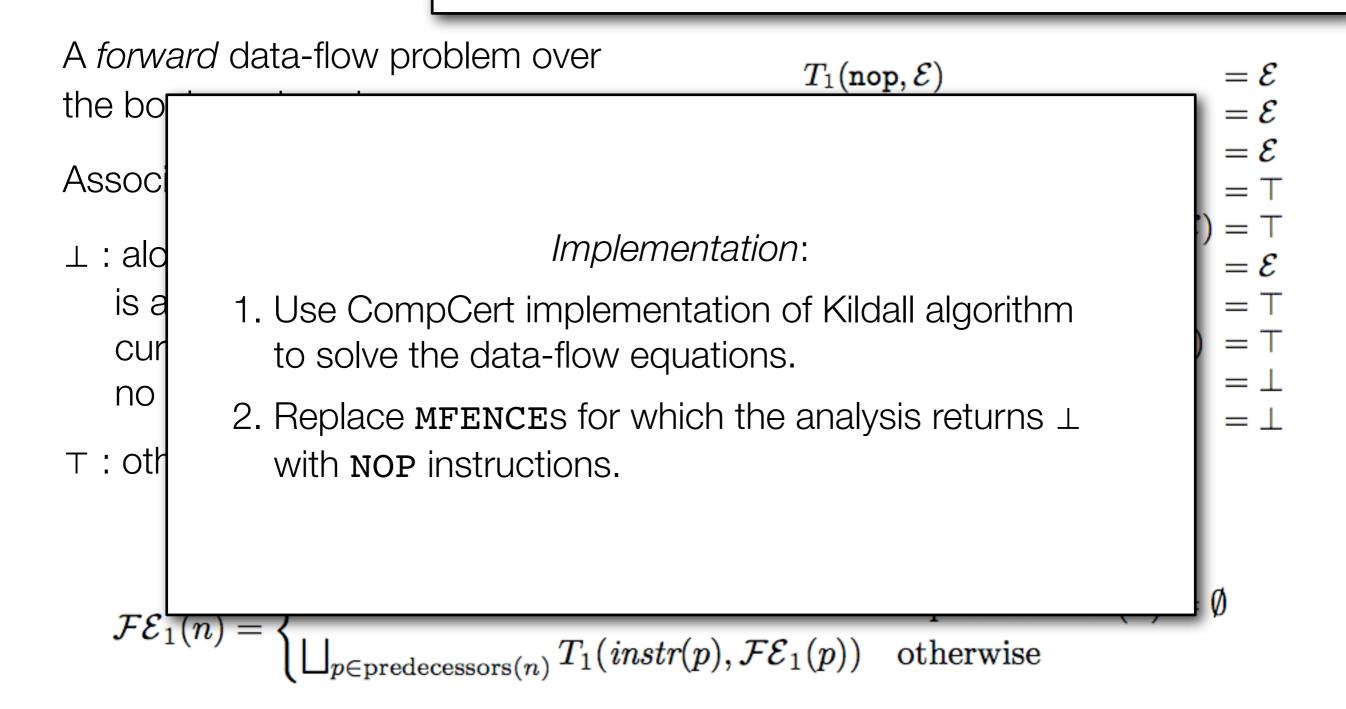
- ⊥ : along all execution paths there
 is an atomic instruction before the
 current program point, with
 no intervening writes;
- T: otherwise.

 $egin{array}{lll} T_1(\mathsf{nop},\mathcal{E}) &= \mathcal{E} \ T_1(\mathsf{op}(op,ec{r},r),\mathcal{E}) &= \mathcal{E} \ T_1(\mathsf{load}(\kappa,addr,ec{r},r),\mathcal{E}) &= \mathcal{E} \ T_1(\mathsf{store}(\kappa,addr,ec{r},src),\mathcal{E}) &= \top \ T_1(\mathsf{call}(sig,ros,args,res),\mathcal{E}) &= \top \ T_1(\mathsf{cond}(cond,args),\mathcal{E}) &= \mathcal{E} \ T_1(\mathsf{return}(optarg),\mathcal{E}) &= \mathcal{E} \ T_1(\mathsf{threadcreate}(optarg),\mathcal{E}) &= \top \ T_1(\mathsf{atomic}(aop,ec{r},r),\mathcal{E}) &= \bot \ T_1(\mathsf{fence},\mathcal{E}) &= \bot \ \end{array}$

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

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.



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:

MFENCE INSTRUCTION 1	???	NOP INSTRUCTION	1
•••		•••	
INSTRUCTION n		INSTRUCTION	n
MFENCE		MFENCE	

If there are reads in between the fences...

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

$$EAX = EBX = 0$$
 forbidden

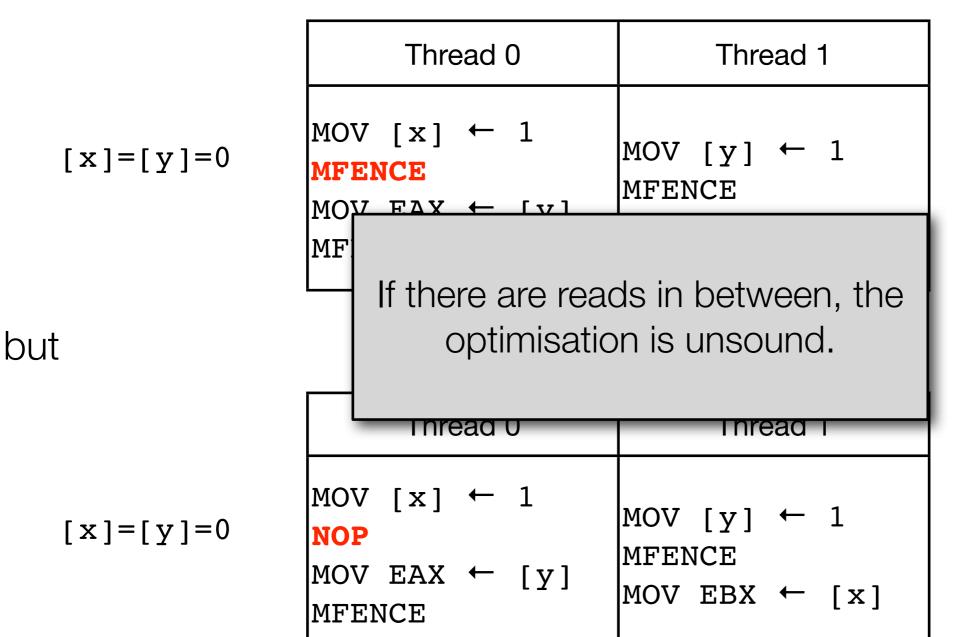
but

$$[x]=[y]=0$$

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

EAX = EBX = 0 allowed

If there are reads in between the fences...



EAX = EBX = 0 forbidden

EAX = EBX = 0 allowed

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

MFENCE; STORE STORE; MFENCE

- transformed program's behaviours ⊆ 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 $\{\bot, \top\}$

Associate to each program point:

- ⊥ : along all execution paths there
 is an atomic instruction after the
 current program point, with
 no intervening reads;
- T: otherwise.

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

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

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);

Basic simulations

```
A pair of relations
```

```
\sim \in \mathbb{P}(src.states \times tgt.states) > \in \mathbb{P}(tgt.states \times tgt.states)
```

is a basic simulation for compile : $src.prg \rightarrow tgt.prg$ if:

```
\begin{array}{l} sim\_init: \forall p \ p'. \ \mathsf{compile}(p) = p' \implies \forall t \in \mathsf{init}(p'). \ \exists s \in \mathsf{init}(p). \ s \sim t \\ sim\_end: \forall s \ t. \ s \sim t \land t \not \to \bot \implies s \not \to \bot \\ sim\_step: \forall s \ t \ t' \ ev. \ s \sim t \land t \xrightarrow{ev} t' \land ev \neq \mathsf{oom} \implies \\ (s \xrightarrow{\tau}^* \xrightarrow{\mathsf{fail}} \bot) \qquad \qquad -s \ \textit{reaches a failure} \\ \lor (\exists s'. \ s \xrightarrow{\tau}^* \xrightarrow{ev} s' \land s' \sim t') \qquad -s \ \textit{does matching step sequence} \\ \lor (\textit{ev} = \tau \land t > t' \land s \sim t'). \qquad -s \ \textit{stutters (only allowed if } t > t') \end{array}
```

Exhibiting a basic simulation implies:

 $traces(compile(p)) \setminus \{t \cdot inftau \mid t \text{ trace}\} \subseteq traces(p)$ "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, traces(compile(p)) \subseteq 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 \bot 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 $t \to t'$ by a thread executing a NOP, OP, STORE OF MFENCE (and t's buffer being non-empty)

Simulation for FE2

```
s = i t iff thread i of s and t have identical pc. local states and buffers s \sim_i s' iff the MFE 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.

- t's CFG
- s and t
- t thread

the analysis for t's pc returned t and t and t and t are t is some instructions behind and can catch up"
```

Stutter condition:

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

Simulation for FE2

(an

 $s \equiv_i t$ iff thread i of s and t have identical pc. local states and buffers But if (1) all threads have non-empty buffers, and $s \sim i s' \text{ iff th}$ (2) are stuck executing infinite loops, and MFE (3) no writes are ever propagated to memory, then we can stutter forever. s ~ t - t's CFG -s and t**Solution 1:** Assume this case never arises (fairness) - ∀ thread **Solution 2:** Do a case split. If this case does not arise, we are done. If it does, use a different (weaker) simulation to Stutter condition construct an infinite trace for the source t > t' iff

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, $\simeq \in \mathbb{P}(src.states \times tgt.states)$ satisfying the following properties:

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

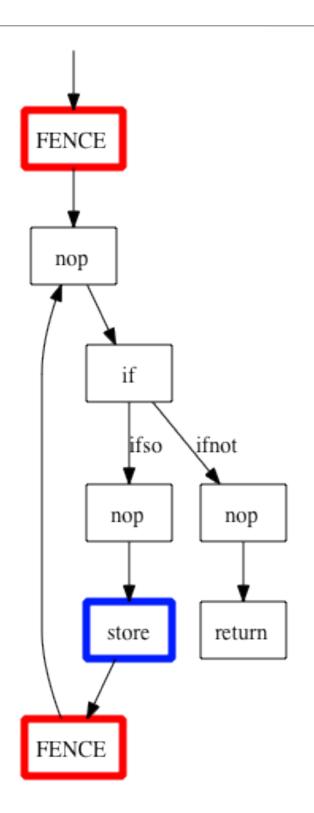
 $s \sim t$, t > t' as before.

 $s \simeq t$ iff

- t's CFG is the optimised version of s's CFG; and
- $-\forall i, \exists s' \text{ s.t. } s \sim_i s' \equiv_i t.$

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

A closer look at the RTL

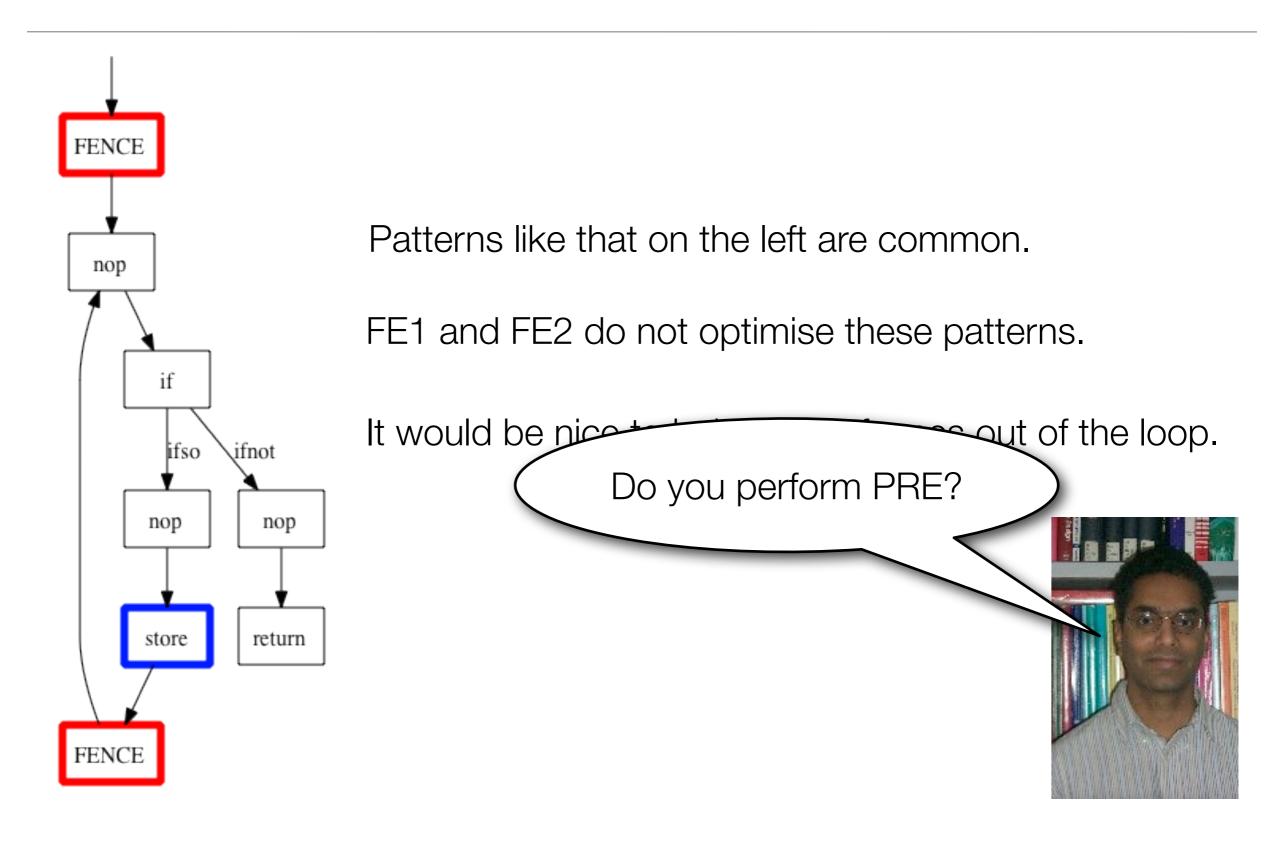


Patterns like that on the left are common.

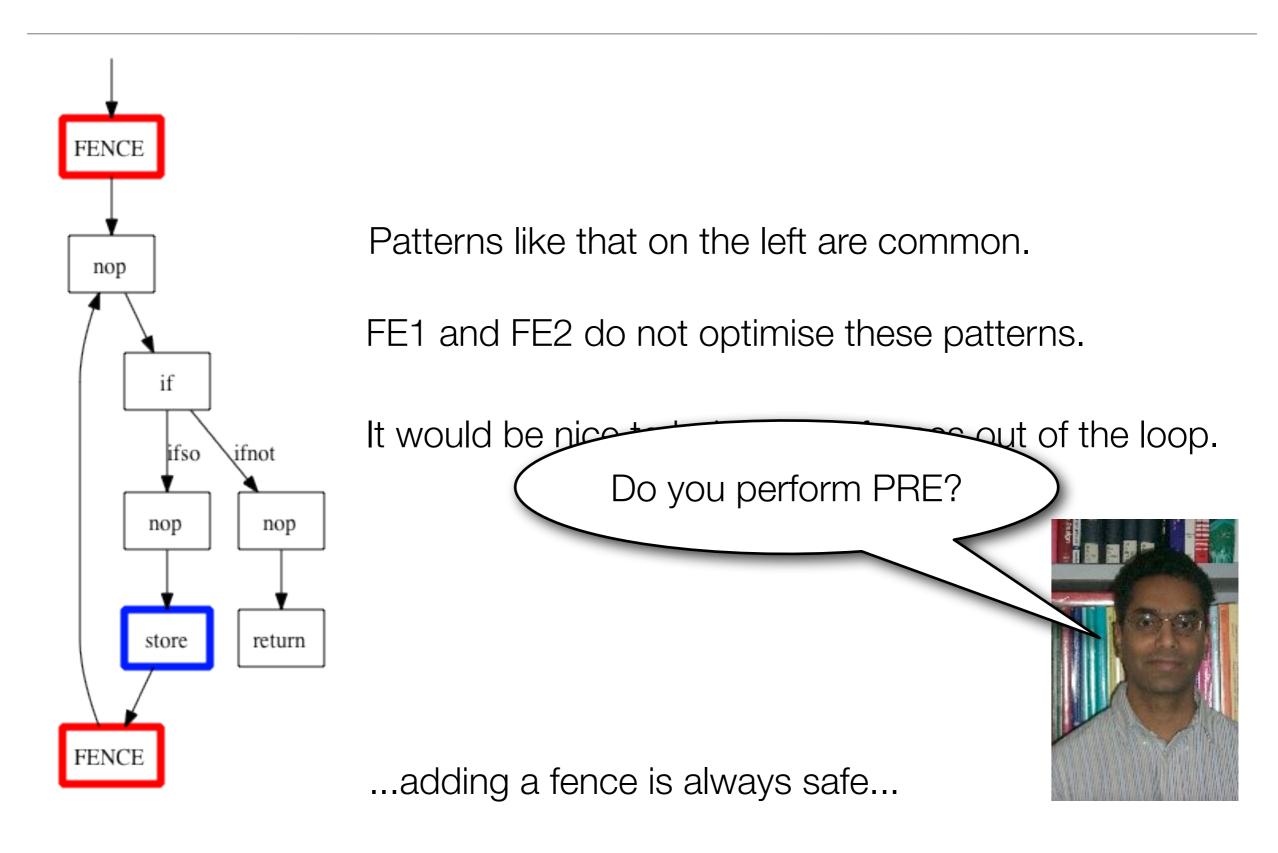
FE1 and FE2 do not optimise these patterns.

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

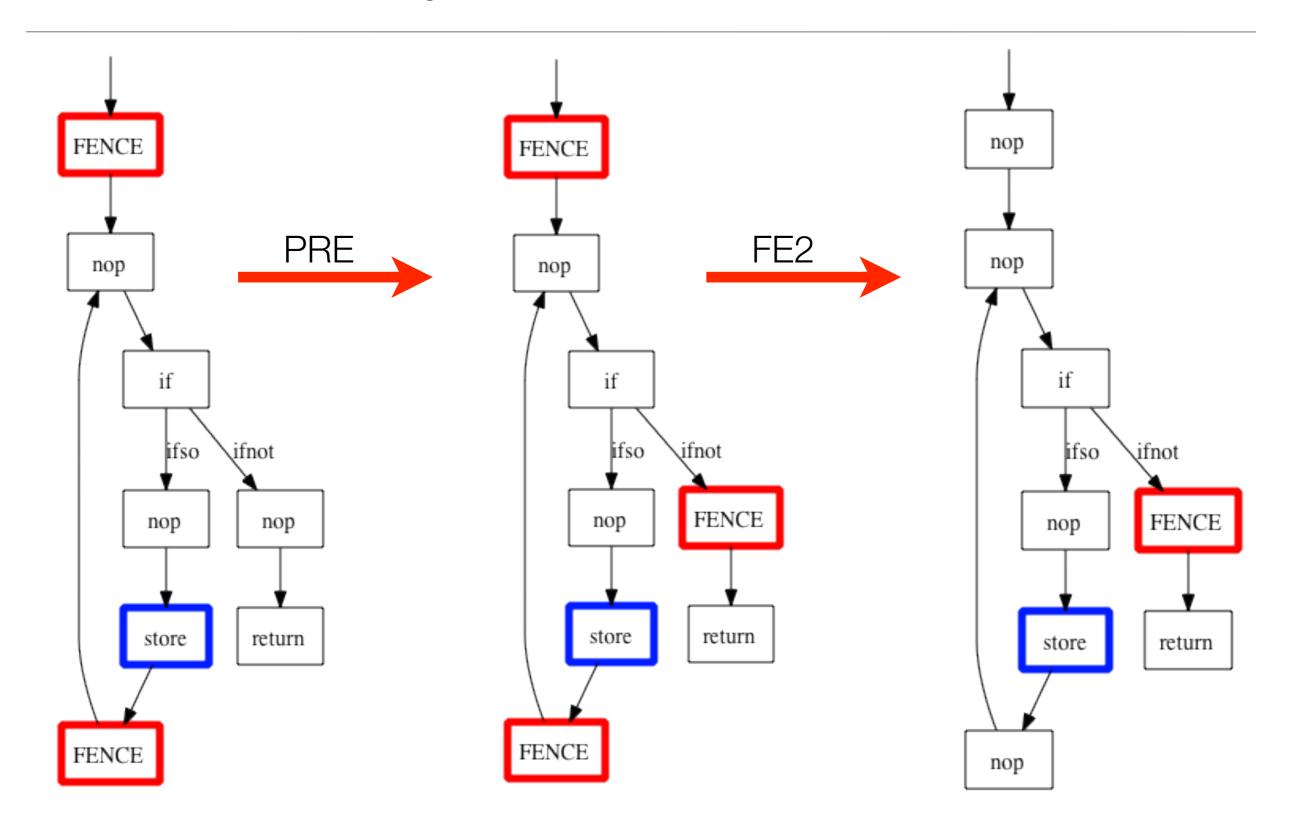
A closer look at the RTL



A closer look at the RTL



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).

