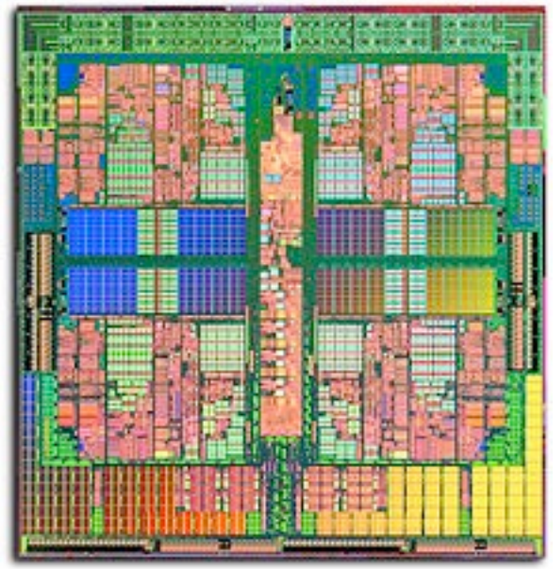
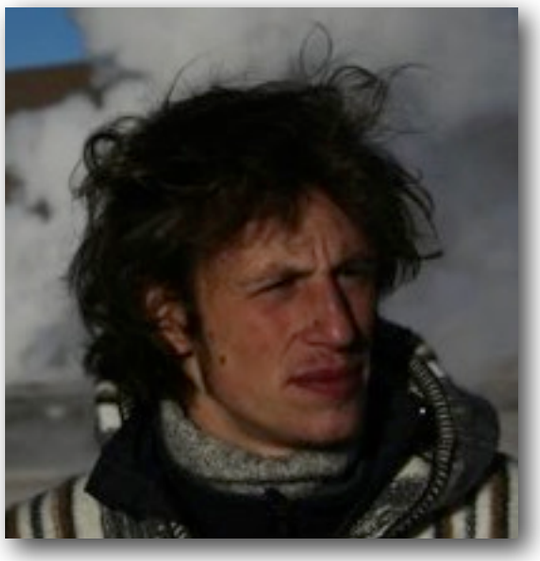


Semantics, languages and algorithms for multicore programming

Albert Cohen

Luc Maranget

Francesco Zappa Nardelli



Vote: topics for my next lecture

1. The lwarx and stwcx Power instructions 3
2. Hunting compiler concurrency bugs 5
3. Operational and axiomatic formalisation of x86-TSO 4
4. Fence optimisations for x86-TSO 2
5. The Java memory model 5
6. **The C11/C++11 memory model 9**
7. **Static and dynamic techniques for data-race detection 11**
8. **The Linux memory model (?!) 6**


1. The C++11 memory model

a good example of an axiomatic memory model

The C++11 memory model

1300 page prose specification defined by the ISO.

The design is a detailed compromise:

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

Welcome to the official home of

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

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

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

The syntactic divide

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

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

where *memory_order* is one of the following:

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

How may a program execute?

Two layer semantics:

1) a denotational semantics processes programs, identifying memory actions, and constructs candidate executions (*E*opsem);

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

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

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

3) searches the consistent executions for races and uncostrained reads

is there an X_{ij} with a race?

Relations

An E_{opsem} part containing:

sb sequenced before, program order

asw additional synchronizes with, inter-thread ordering

An $X_{witness}$ part containing:

rf relates a write to any reads that take its value

sc a total order over mo_seq_cst and mutex actions

mo modification order, per location total order of writes

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

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

Full details in Batty et al., POPL 11.

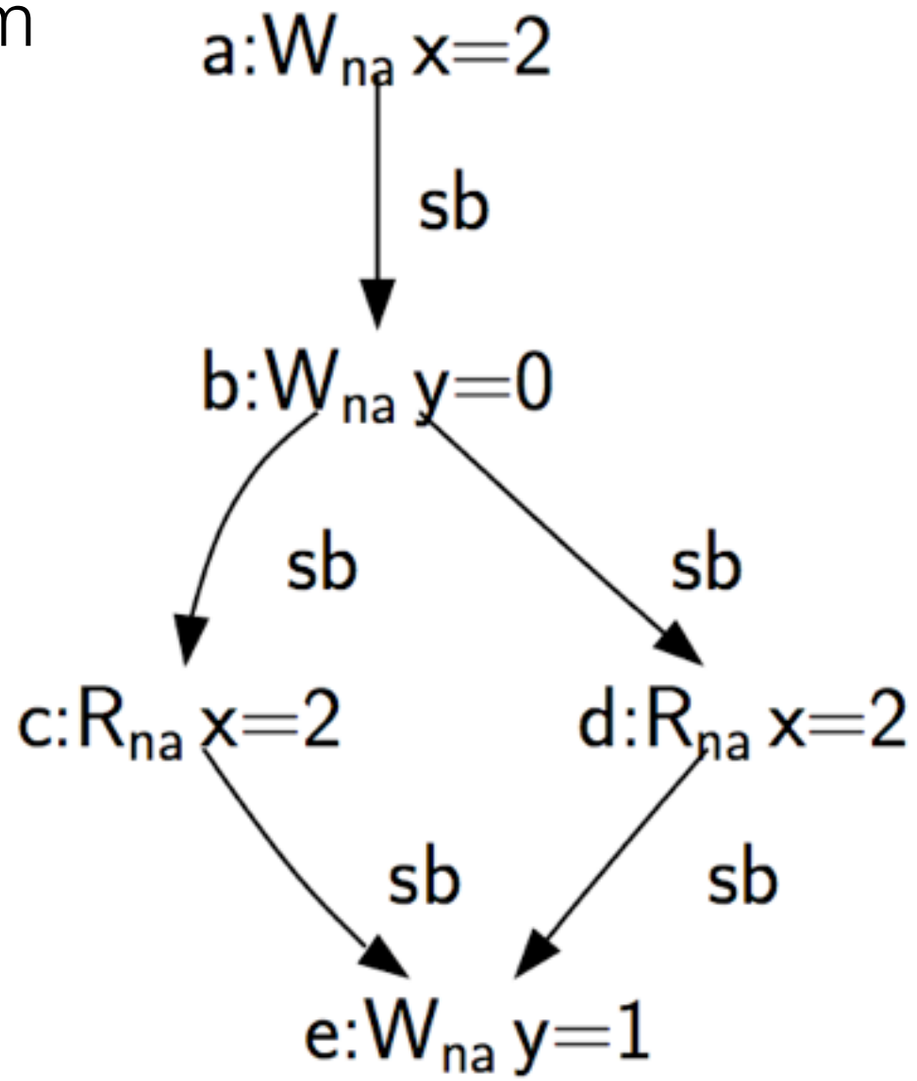
Formally

```
cpp_memory_model_opsem (p : program) =
let pre_executions =
  {(Eopsem,Xwitness).  opsem p Eopsem ∧
    consistent_execution (Eopsem, Xwitness)}
in
if ∃X ∈ pre_executions.
   (indeterminate_reads X = {}) ∨
   (unsequenced_races X = {}) ∨
   (data_races X = {})
then NONE
else SOME pre_executions
```

A single-threaded example

1. sequenced before (sb) - given by opsem

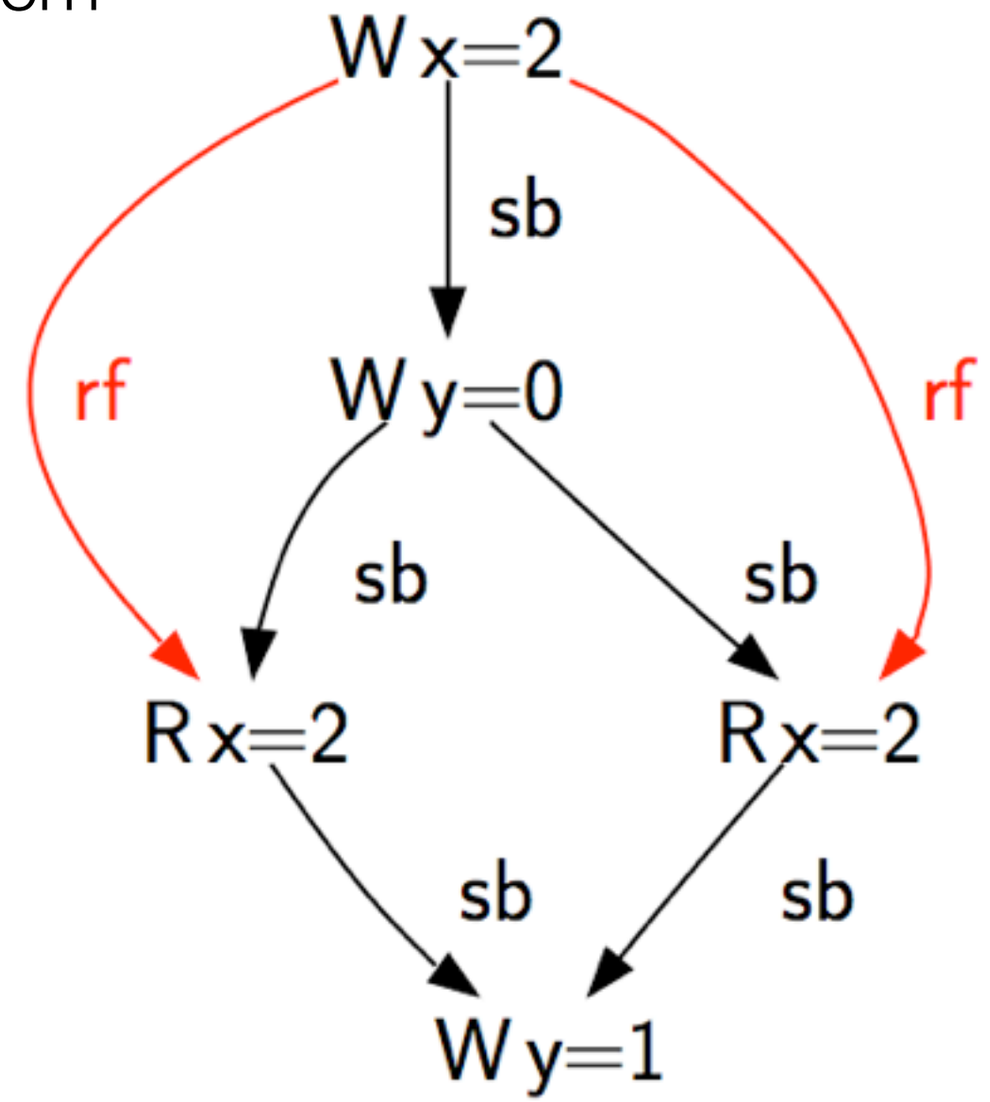
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



A single-threaded example

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

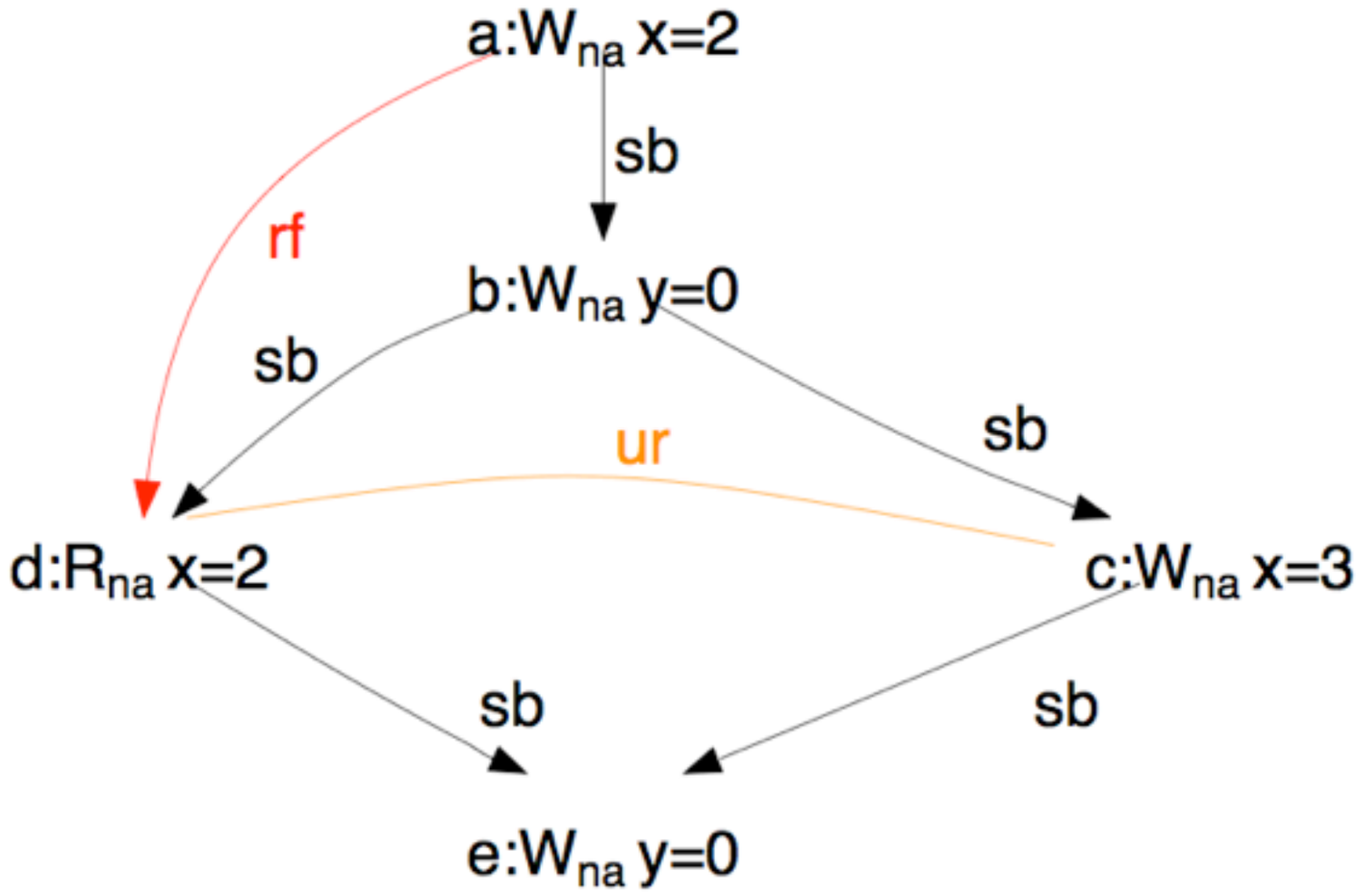
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



A single-threaded ex. with undefined behaviour

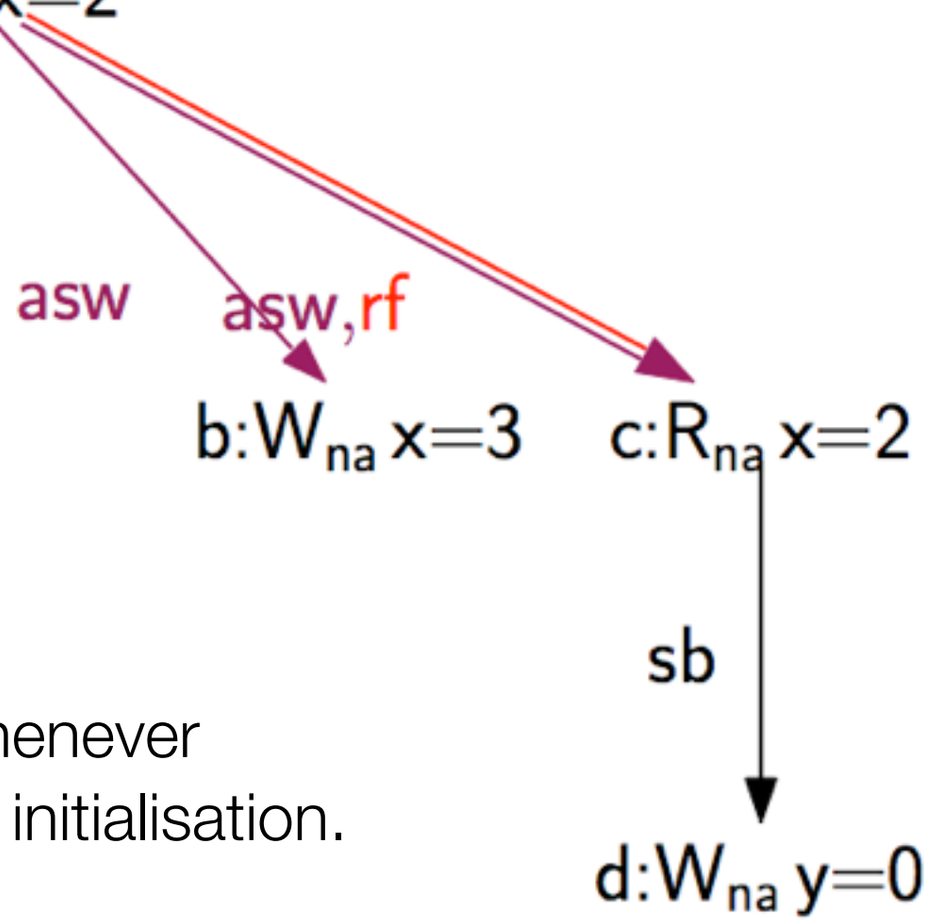
An unsequenced race.

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



A simple concurrent program

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



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

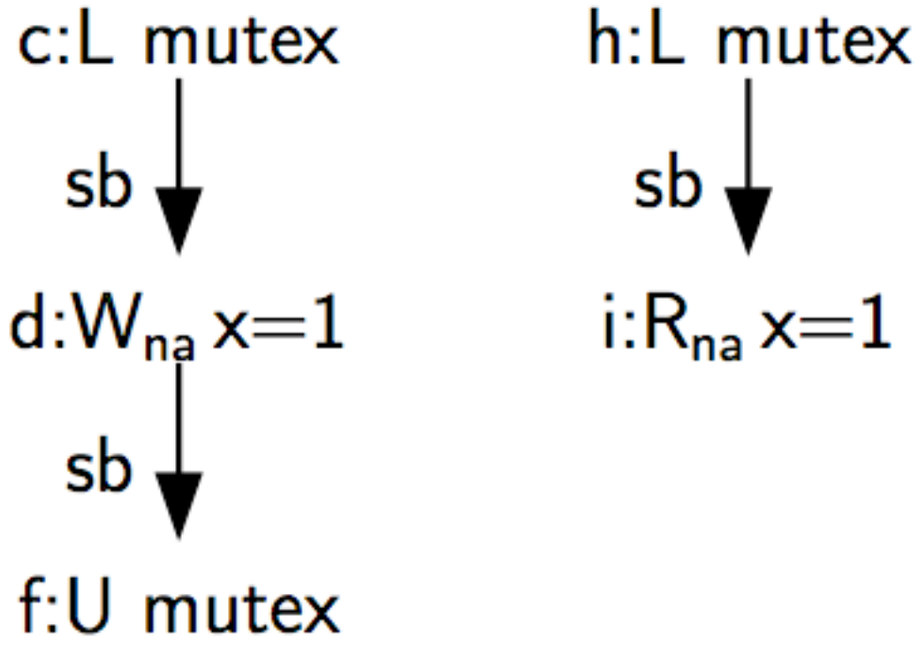
Locks and unlocks

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

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

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

1. the operational semantics defines the sb arrows

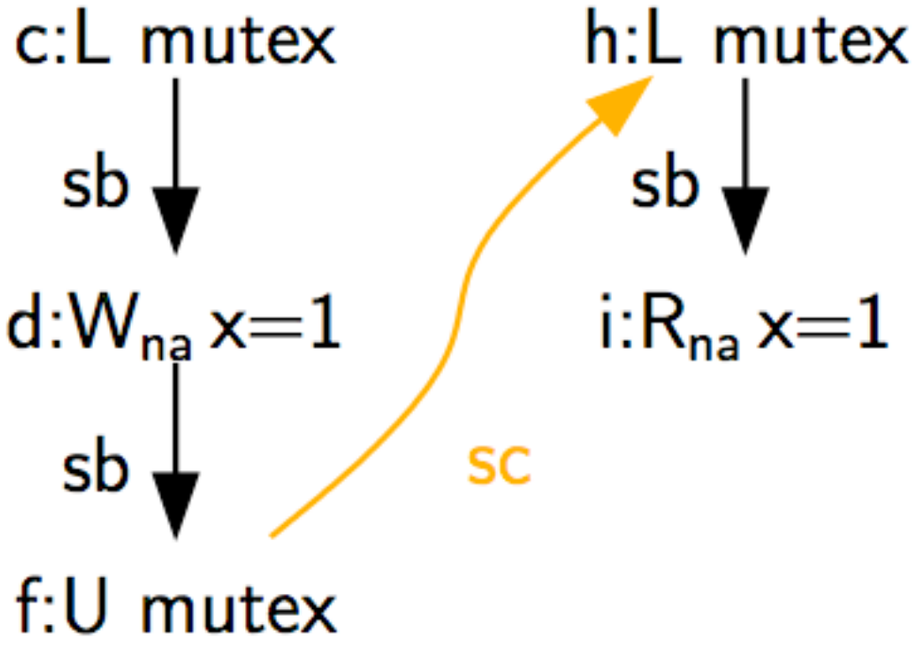


Locks and unlocks

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

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

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

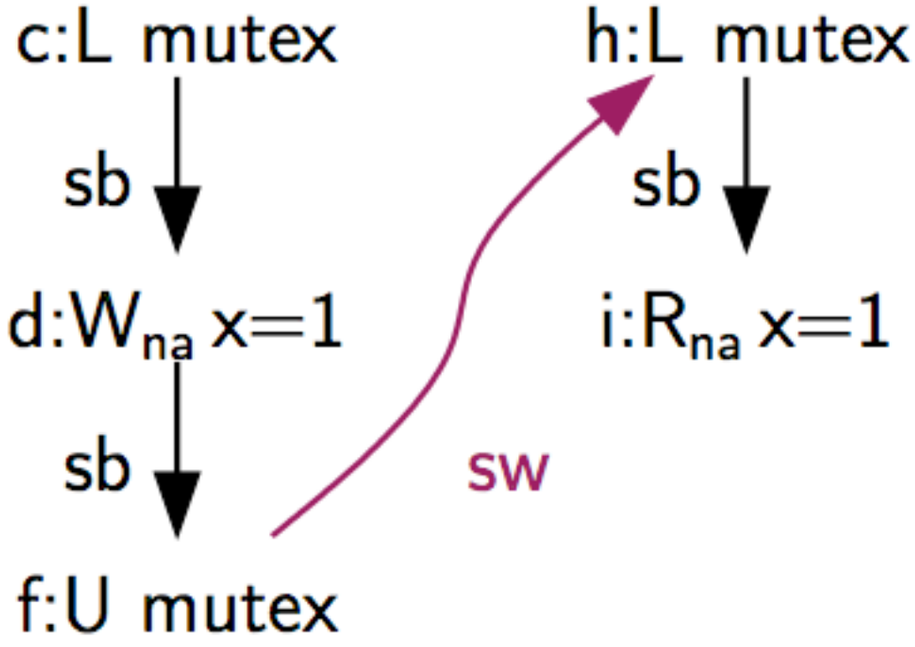


Locks and unlocks

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

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

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



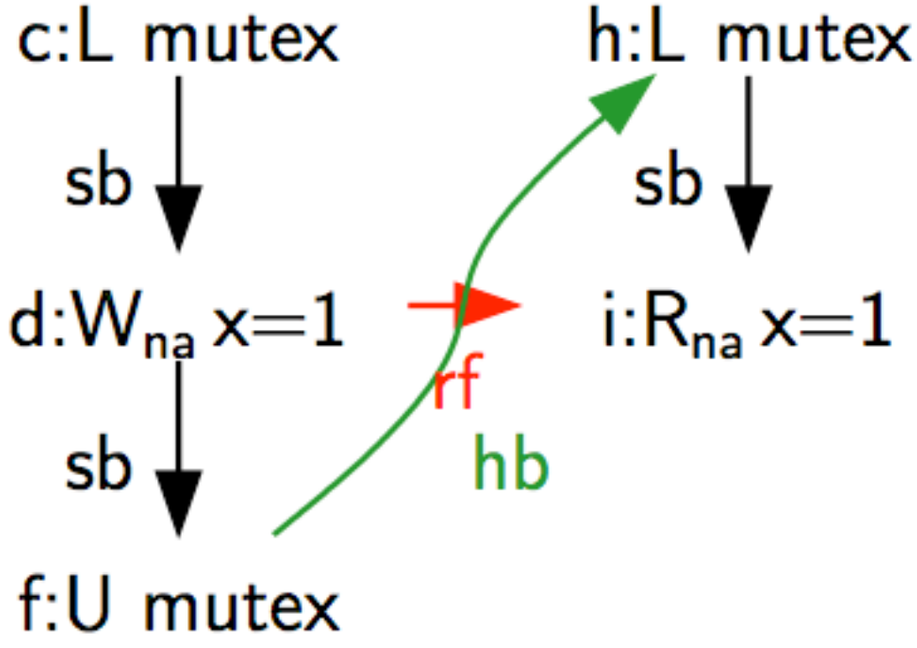
Locks and unlocks

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

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

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

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



Happens before

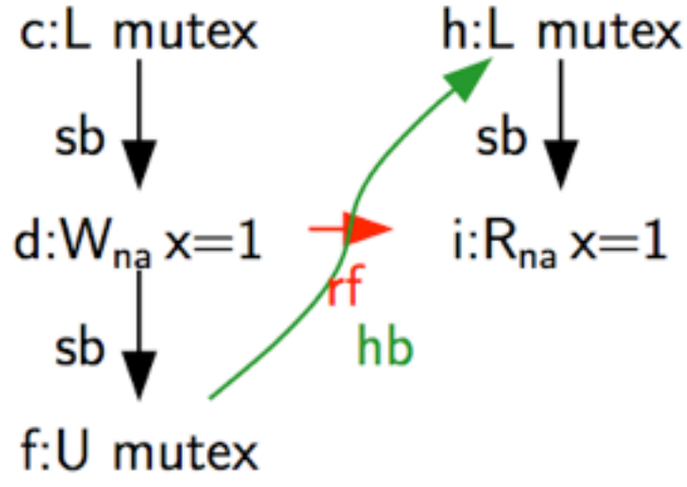
The *happens before* relation is key to the model:

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

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

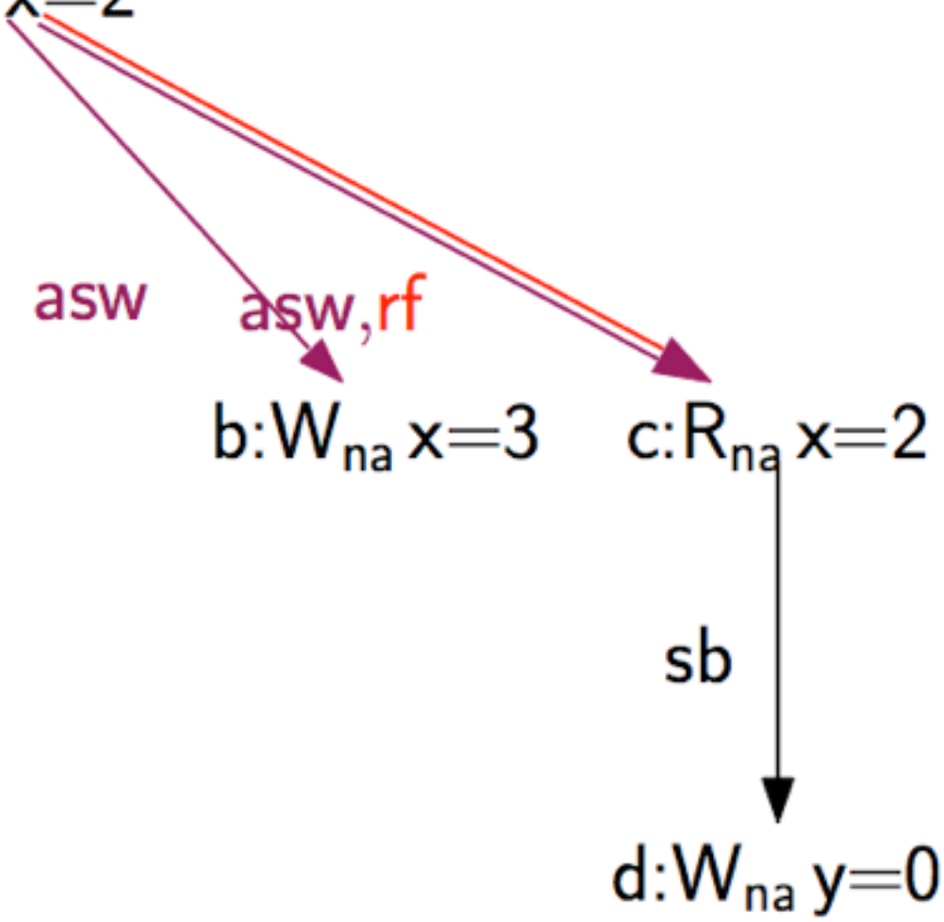
3. data races are defined as an absence of happens before
   between conflicting actions.

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



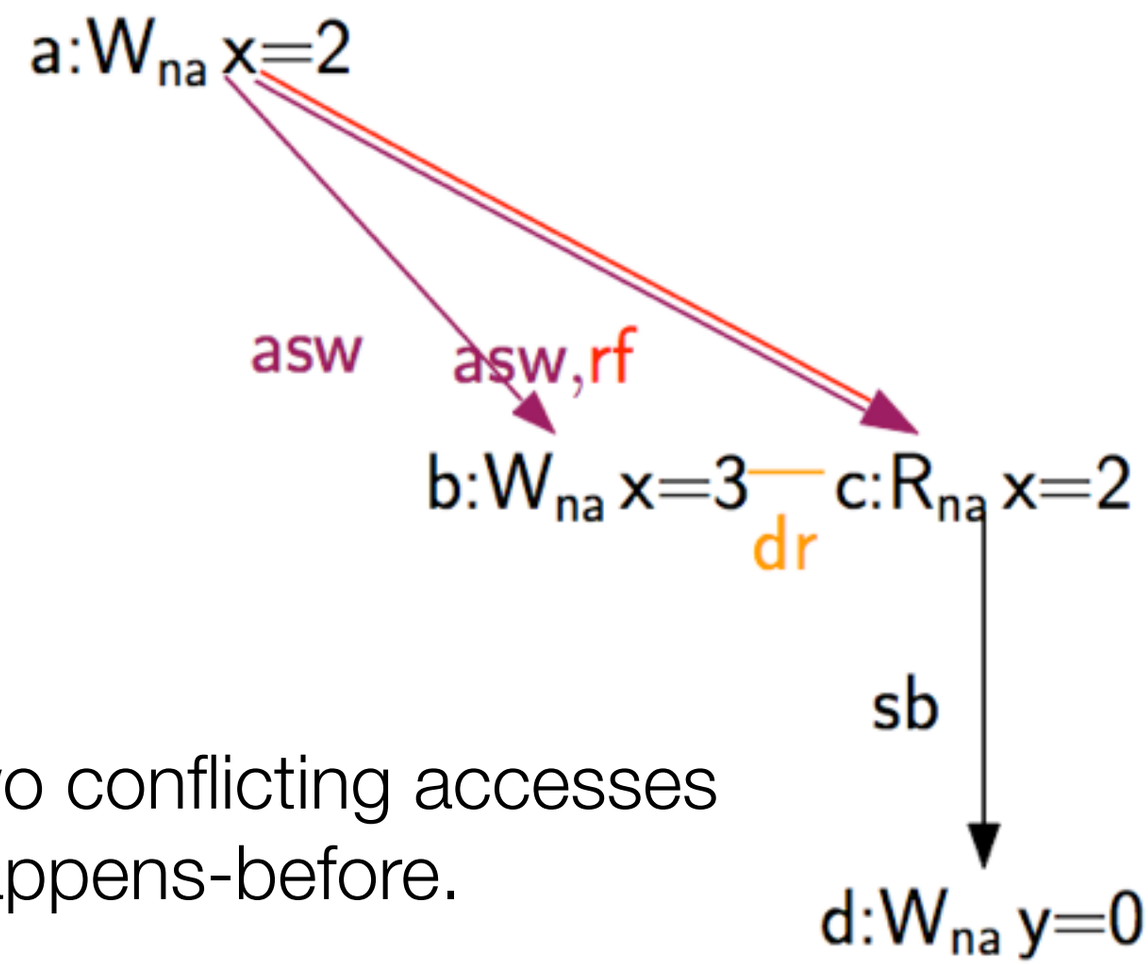
A data race

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



A data race

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



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

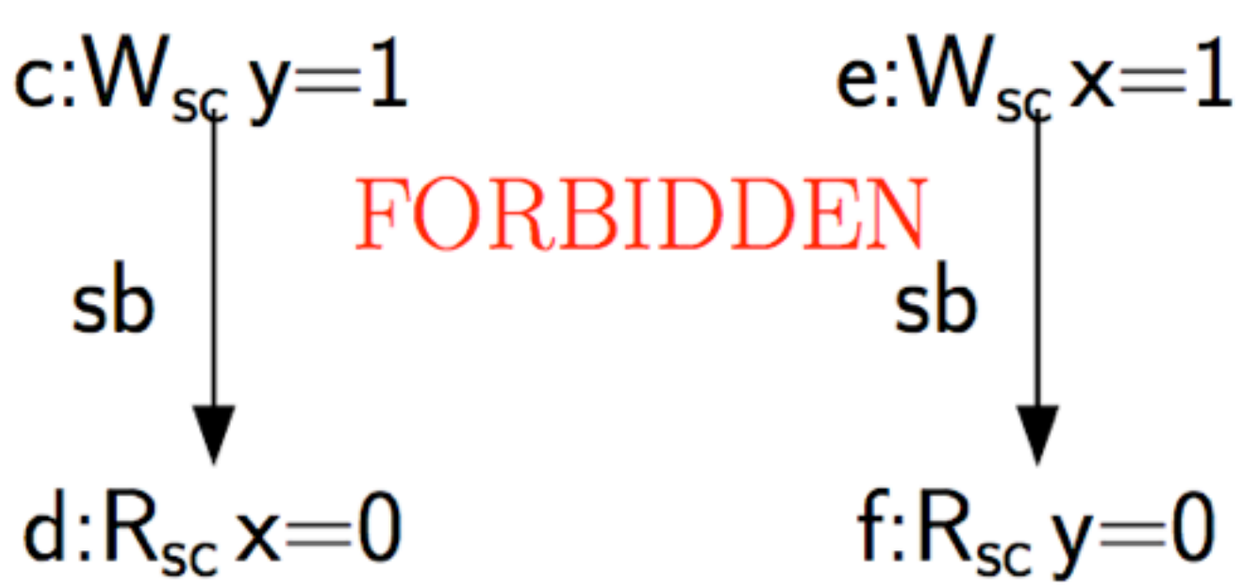
Data race definition

$$
\begin{aligned}
&\text{let } \mathit{data\_races}\ \mathit{actions}\ \mathit{hb} = \\
&\quad \{\ (a,\ b) \mid \forall\ a{\in}\mathit{actions}\ b{\in}\mathit{actions} \mid \\
&\qquad \neg\ (a = b) \wedge \\
&\qquad \text{same\_location}\ a\ b \wedge \\
&\qquad (\text{is\_write}\ a\ \vee\ \text{is\_write}\ b) \wedge \\
&\qquad \neg\ (\text{same\_thread}\ a\ b) \wedge \\
&\qquad \neg\ (\text{is\_atomic\_action}\ a\ \wedge\ \text{is\_atomic\_action}\ b) \wedge \\
&\qquad \neg\ ((a,\ b) \in \mathit{hb}\ \vee\ (b,\ a) \in \mathit{hb})\ \}
\end{aligned}
$$

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

Simple concurrency: Dekker's example and SC

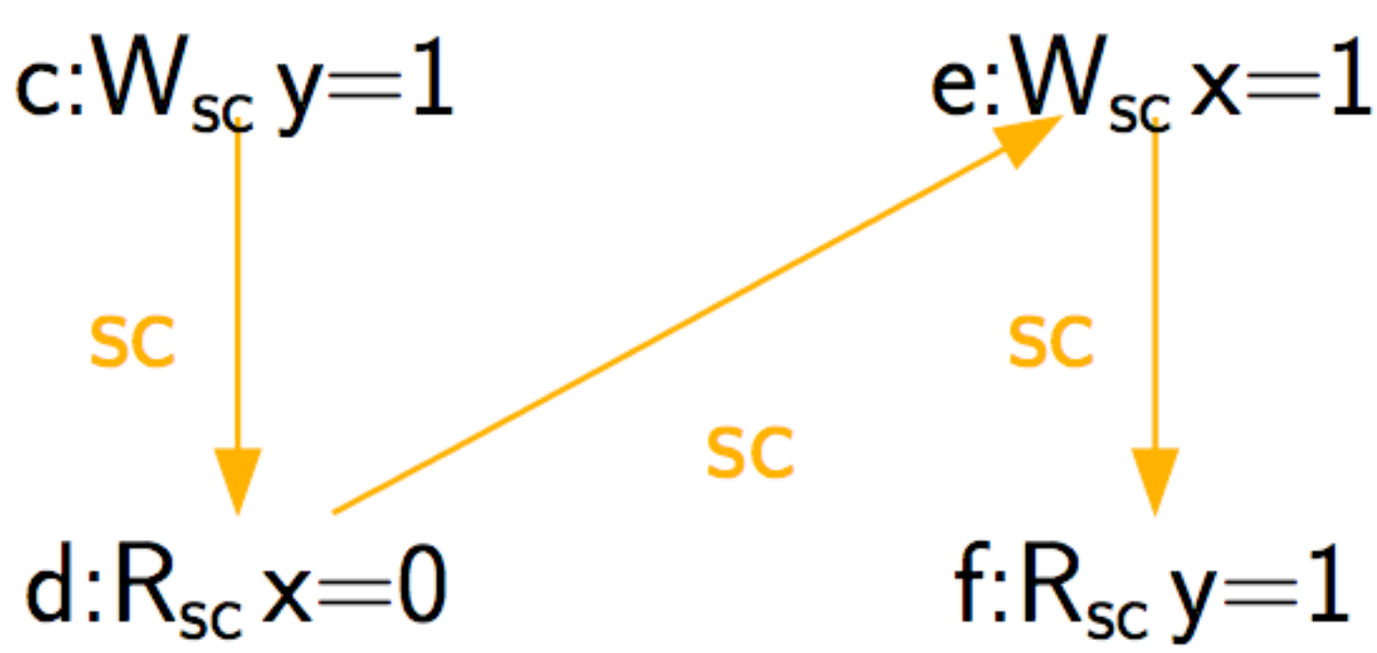
```
atomic_int x = 0;
atomic_int y = 0;

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



Why is this behaviour forbidden?

Simple concurrency, Dekker's example and SC

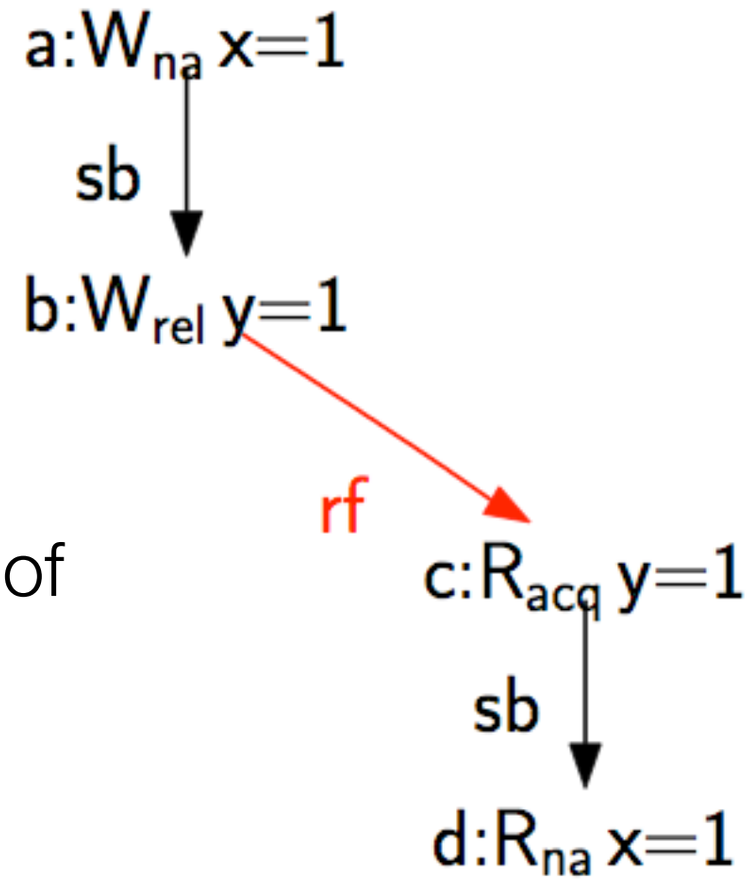
```
atomic_int x = 0;
atomic_int y = 0;

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



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

Expert concurrency: the release-acquire idiom

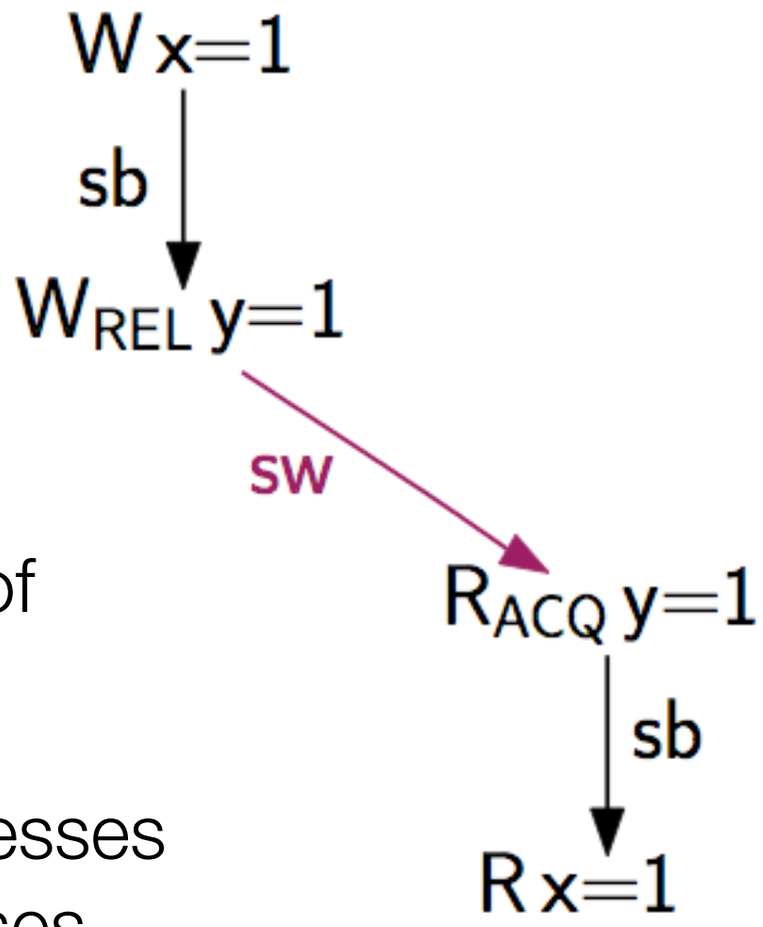
```
// sender
x = ...
y.store(1, release);

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



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

Expert concurrency: the release-acquire idiom

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

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

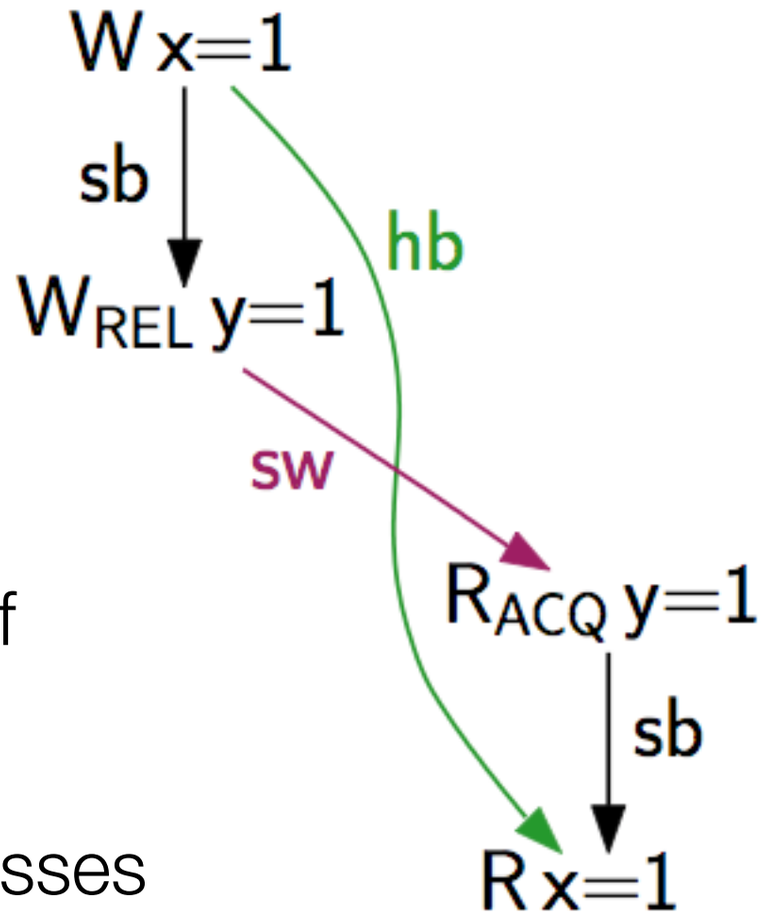


Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

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

Expert concurrency: the release-acquire idiom

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

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



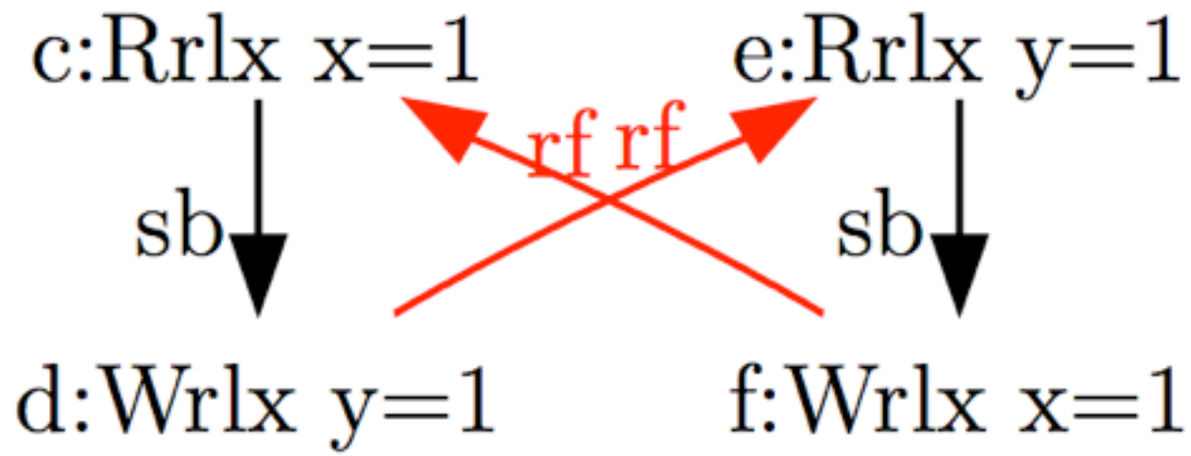
Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

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

And in turn defines an **`hb`** constraint. $\xrightarrow{\textit{simple-happens-before}} = (\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}})^+$

Relaxed writes

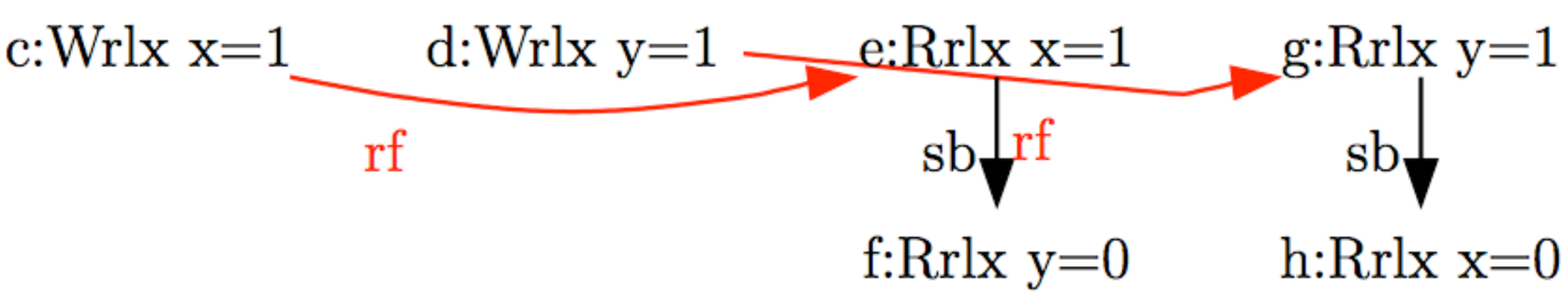
```
x.load(relaxed);          | y.load(relaxed);
y.store(1, relaxed);      | x.store(1, relaxed);
```



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

Relaxed writes, ctd.

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



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

Expert concurrency: fences avoid excess sync.

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

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

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

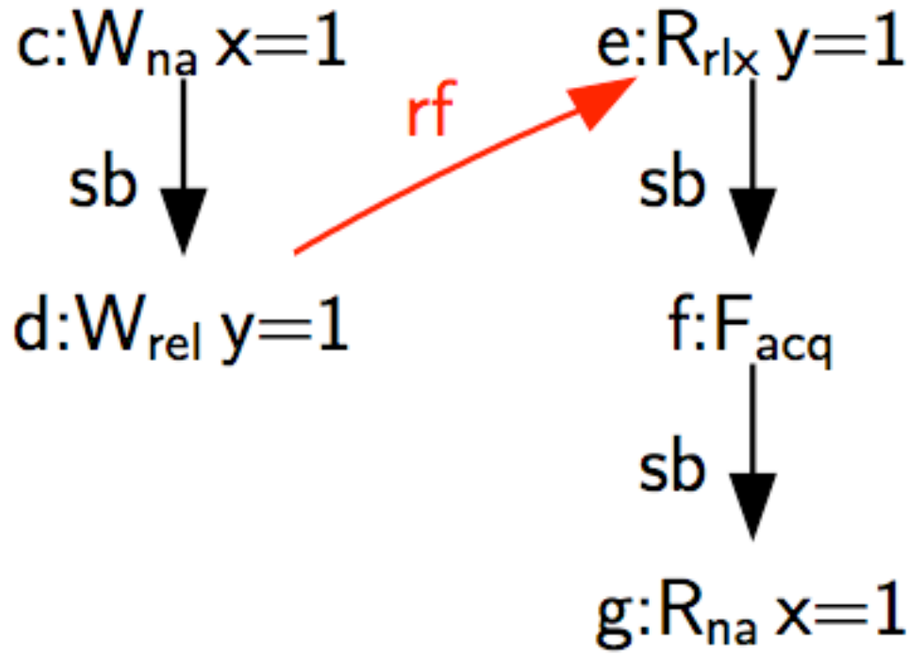
```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```

Expert concurrency: fences avoid excess sync.

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

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

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

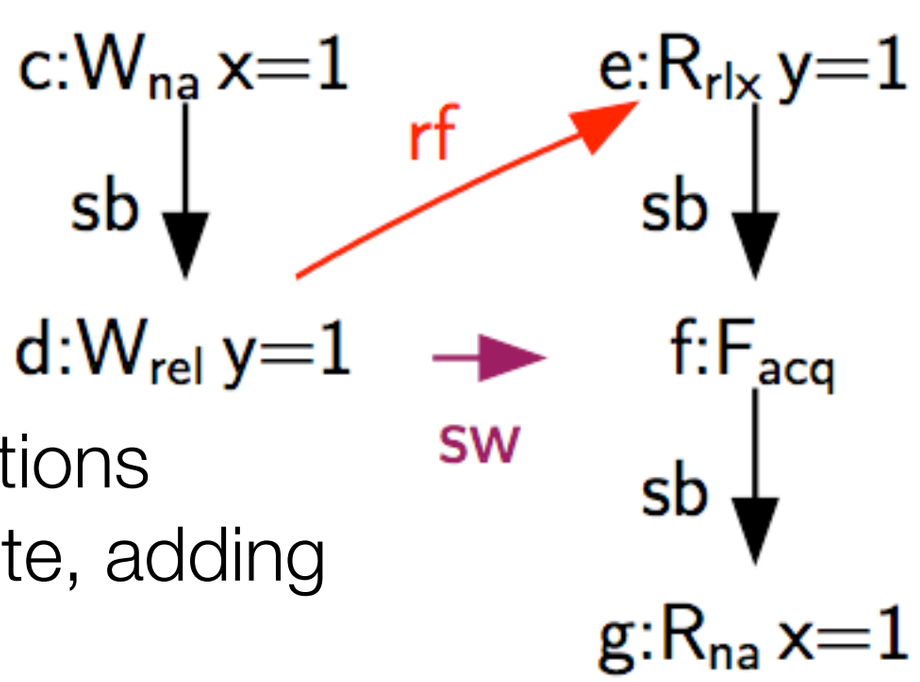


Expert concurrency: fences avoid excess sync.

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

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

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



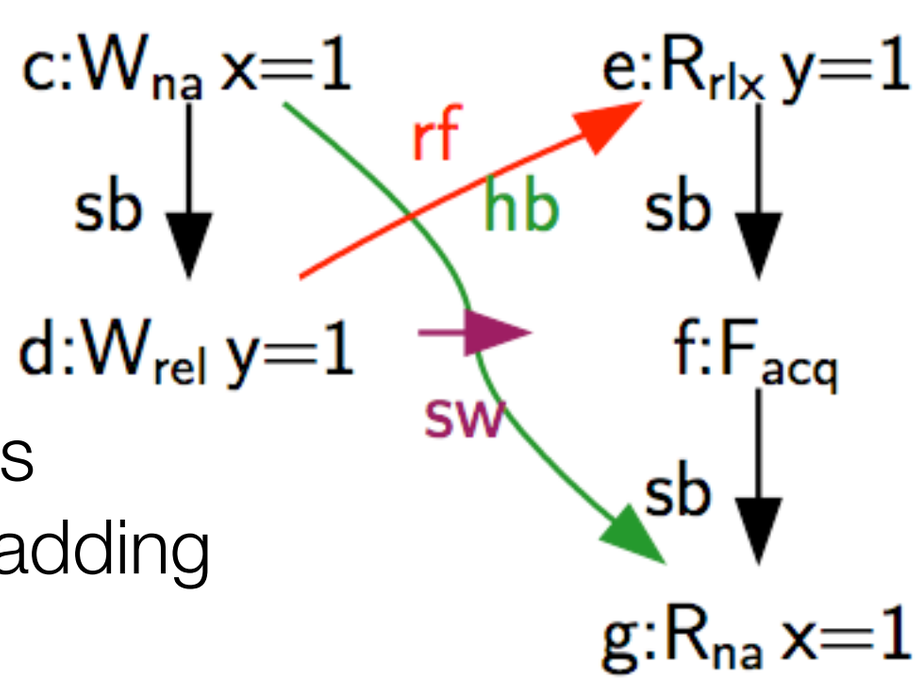
The acquire fence follows the **`sb`**/**`rf`** relations looking for the corresponding release write, adding a **`sw`** arrow.

Expert concurrency: fences avoid excess sync.

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

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

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

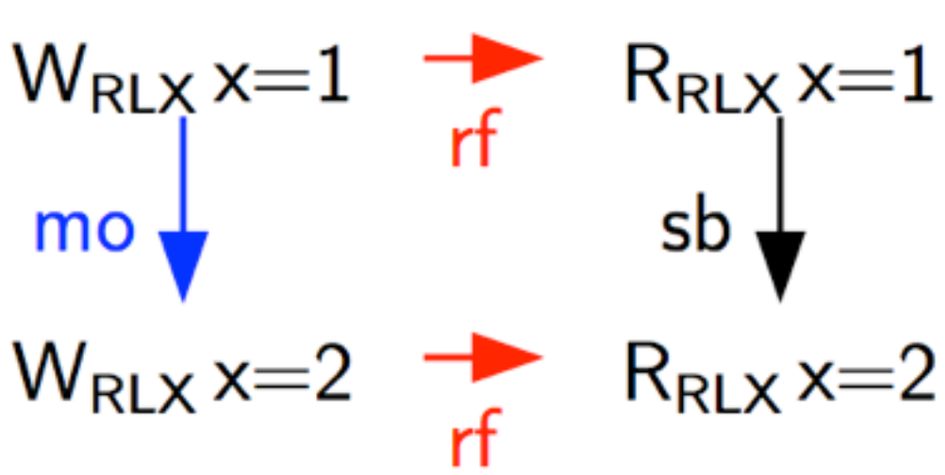


The acquire fence follows the **`sb`**/**`rf`** relations looking for the corresponding release write, adding a **`sw`** arrow.

Happens-before follows as usual...

Modification order

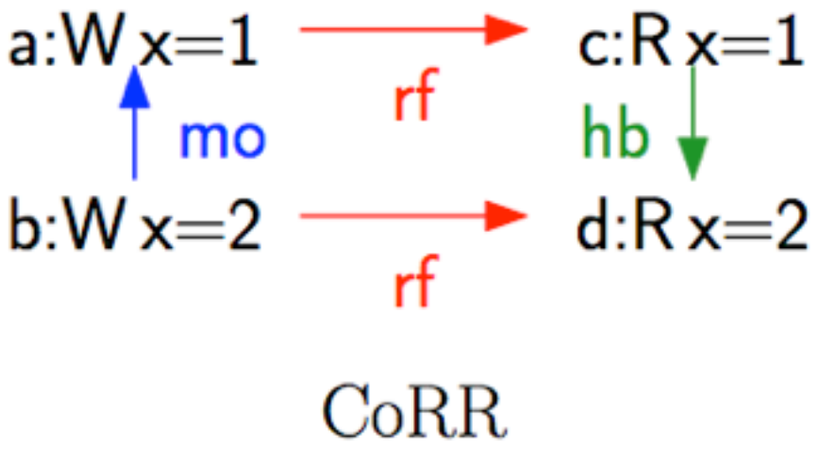
```
   atomic_int x = 0;
x.store(1, relaxed); ||  x.load(relaxed);
x.store(2, relaxed); ||  x.load(relaxed);
```



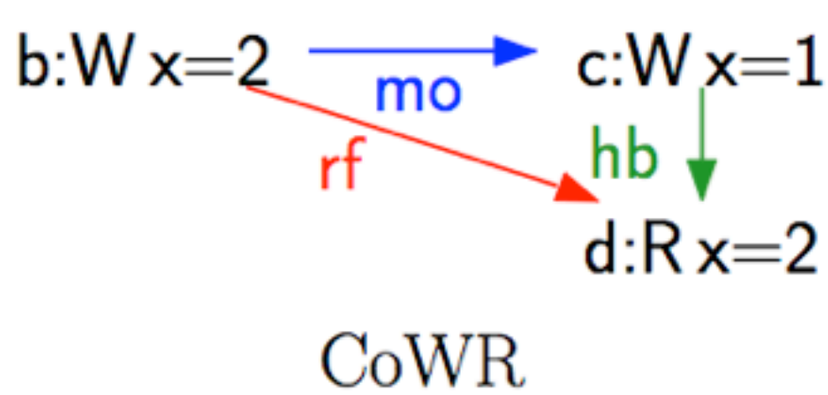
Modification order is a total order over atomic writes of any memory order.

Coherence and atomic reads

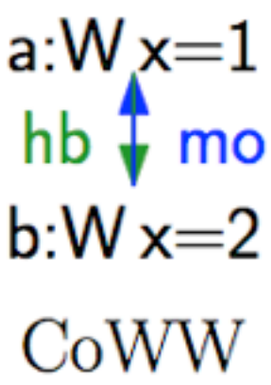
All forbidden:



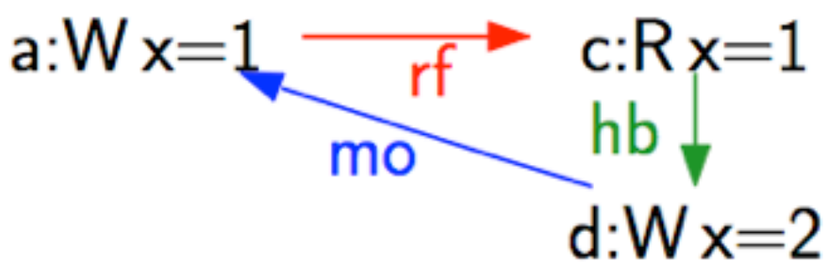
CoRR



CoWR



CoWW



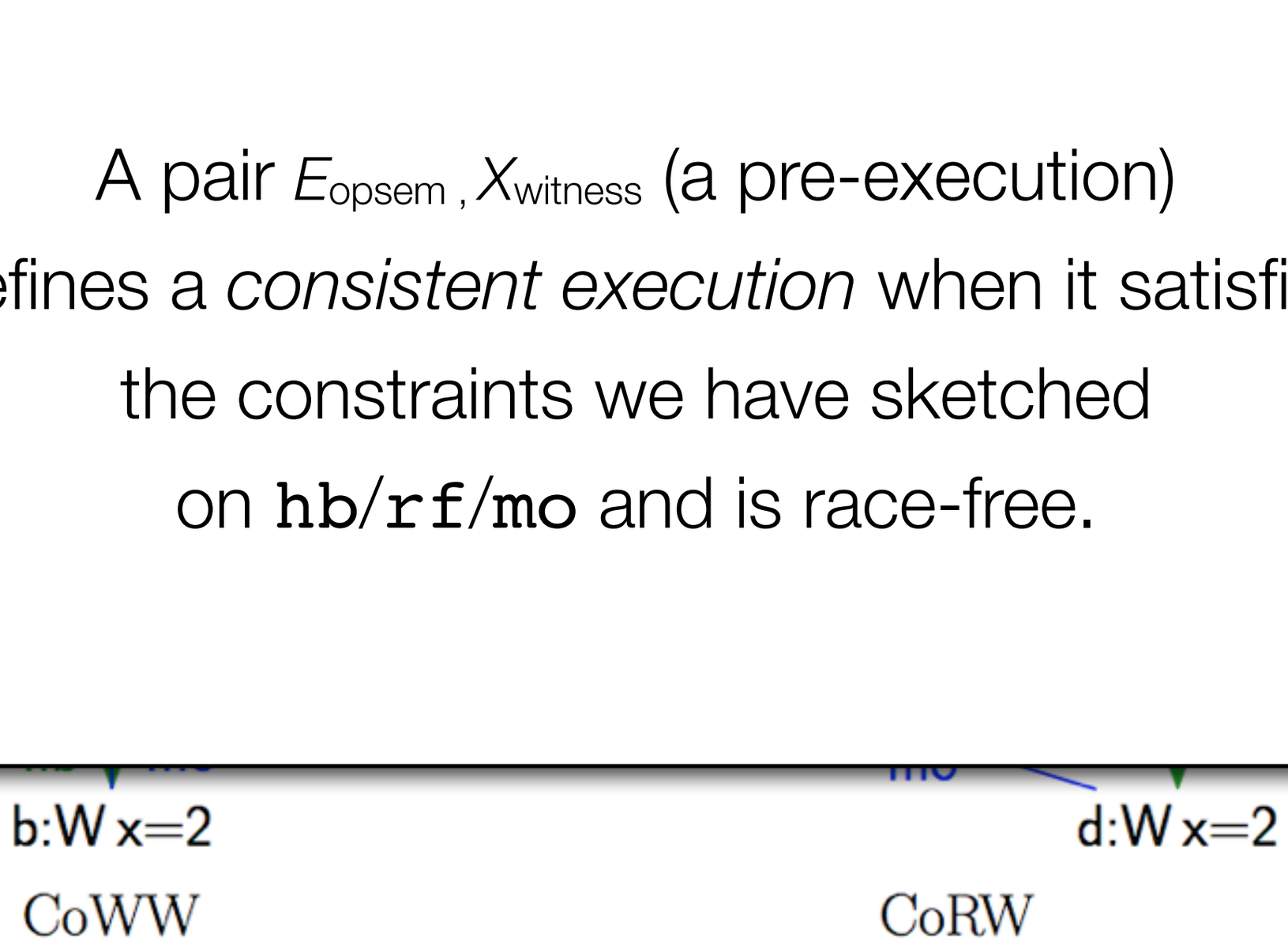
CoRW

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

Coherence and atomic reads

All forb

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



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

The full model

```
a -sb-> b = (a, b) ∈ sb
```

```
a r b = (a, b) ∈ r
```

```
a -/sb-> b = (a, b) ∉ sb
```

```
-sb-> = sb
```

```
a -sb-> b -sb-> c = a -sb-> b ∧ b -sb-> c
```

```
relation_over s rel = domain rel ⊆ s ∧ range rel ⊆ s
```

```
sb|s = rel ∩ (s × s)
```

```
rel|s = rel ∩ (s × s)
```

```
sb-1|s = rel ∩ (s × s)
```

```
rel|s = rel ∩ (s × s)
```

```
strict_preorder ord = irreflexive ord ∧ trans ord
```

```
total_over s ord =
  relation_over s ord ∧
  (∀x ∈ s. ∀y ∈ s. x -ord-> y ∨ y -ord-> x ∨ (x = y))
```

```
strict_total_order_over s ord =
  strict_preorder ord ∧ total_over s ord
```

```
x -ord-pred-> y =
  pred x ∧ x -ord-> y ∧ ¬(∃z. pred z ∧ x -ord-> z -ord-> y)
```

```
x -ord?-> y =
  x -ord-> y ∧ ¬(∃z. x -ord-> z -ord-> y)
```

```
well_founded r = wf r
```

```
type_abbrev action_id : string
```

```
type_abbrev thread_id : string
```

```
type_abbrev location : string
```

```
type_abbrev val : string
```

```
memory_order_enum =
    Mo_seq_cst
  | Mo_relaxed
  | Mo_release
  | Mo_acquire
  | Mo_consume
  | Mo_acq_rel
```

```
action =
    Lock of action_id thread_id location
  | Unlock of action_id thread_id location
  | Atomic_load of action_id thread_id memory_order_enum location val
  | Atomic_store of action_id thread_id memory_order_enum location val
  | Atomic_rmw of action_id thread_id memory_order_enum location val val
  | Load of action_id thread_id location val
  | Store of action_id thread_id location val
  | Fence of action_id thread_id memory_order_enum
```

```
(action_id_of (Lock aid _ _) = aid) ∧
(action_id_of (Unlock aid _ _) = aid) ∧
(action_id_of (Atomic_load aid _ _ _) = aid) ∧
(action_id_of (Atomic_store aid _ _ _ _) = aid) ∧
(action_id_of (Atomic_rmw aid _ _ _ _) = aid) ∧
(action_id_of (Load aid _ _ _) = aid) ∧
(action_id_of (Store aid _ _ _) = aid) ∧
(action_id_of (Fence aid _ _) = aid)
```

```
(thread_id_of (Lock _ tid _) = tid) ∧
(thread_id_of (Unlock _ tid _) = tid) ∧
(thread_id_of (Atomic_load _ tid _ _ _) = tid) ∧
(thread_id_of (Atomic_store _ tid _ _ _ _) = tid) ∧
(thread_id_of (Atomic_rmw _ tid _ _ _ _) = tid) ∧
(thread_id_of (Load _ tid _ _ _) = tid) ∧
(thread_id_of (Store _ tid _ _ _) = tid) ∧
(thread_id_of (Fence _ tid _) = tid)
```

```
(memory_order (Atomic_load _ _ mem_ord _ _) =
  Some mem_ord) ∧
(memory_order (Atomic_store _ _ mem_ord _ _) =
  Some mem_ord) ∧
(memory_order (Atomic_rmw _ _ mem_ord _ _ _) =
  Some mem_ord) ∧
(memory_order (Fence _ _ mem_ord) =
  Some mem_ord) ∧
(memory_order _ = None)
```

```
(location (Lock _ _ l) = Some l) ∧
(location (Unlock _ _ l) = Some l) ∧
(location (Atomic_load _ _ _ l _) = Some l) ∧
(location (Atomic_store _ _ _ l _) = Some l) ∧
(location (Atomic_rmw _ _ _ l _ _) = Some l) ∧
(location (Load _ _ l _) = Some l) ∧
(location (Store _ _ l _) = Some l) ∧
(location (Fence _ _ _) = None)
```

```
(value_read (Atomic_load _ _ _ v) = Some v) ∧
(value_read (Atomic_rmw _ _ _ _ v _) = Some v) ∧
(value_read (Load _ _ _ v) = Some v) ∧
(value_read _ = None)
```

```
(value_written (Atomic_store _ _ _ v) = Some v) ∧
(value_written (Atomic_rmw _ _ _ _ v) = Some v) ∧
(value_written (Store _ _ _ v) = Some v) ∧
(value_written _ = None)
```

```
is_lock a =
  case a of Lock _ _ _ → T | _ → F
```

```
is_unlock a =
  case a of Unlock _ _ _ → T | _ → F
```

```
is_atomic_load a =
  case a of Atomic_load _ _ _ _ _ → T | _ → F
```

```
is_atomic_store a =
  case a of Atomic_store _ _ _ _ _ → T | _ → F
```

```
is_atomic_rmw a =
  case a of Atomic_rmw _ _ _ _ _ _ → T | _ → F
```

```
is_load a = case a of Load _ _ _ _ → T | _ → F
```

```
is_store a = case a of Store _ _ _ _ → T | _ → F
```

```
is_fence a = case a of Fence _ _ _ → T | _ → F
```

```
is_lock_or_unlock a = is_lock a ∨ is_unlock a
```

```
is_atomic_action a =
  is_atomic_load a ∨ is_atomic_store a ∨ is_atomic_rmw a
```

```
is_load_or_store a = is_load a ∨ is_store a
```

```
is_read a =
  is_atomic_load a ∨ is_atomic_rmw a ∨ is_load a
```

```
is_write a =
  is_atomic_store a ∨ is_atomic_rmw a ∨ is_store a
```

```
is_acquire a =
  (case memory_order a of
    Some mem_ord →
      (mem_ord ∈
        {Mo_acquire, Mo_acq_rel, Mo_seq_cst} ∧
      (is_read a ∨ is_fence a)) ∨
      (* 29.8:5 states that consume fences are acquire fences. *)
      ((mem_ord = Mo_consume) ∧ is_fence a)
  | None → is_lock a)
```

```
is_consume a =
  is_read a ∧ (memory_order a = Some Mo_consume)
```

```
is_release a =
  (case memory_order a of
    Some mem_ord →
      mem_ord ∈ {Mo_release, Mo_acq_rel, Mo_seq_cst} ∧
    (is_write a ∨ is_fence a)
  | None → is_unlock a)
```

```
is_seq_cst a = (memory_order a = Some Mo_seq_cst)
```

```
location_kind =
    Mutex
  | Non_atomic
  | Atomic
```

```
actions_respect_location_kinds = actions_respect_location_kinds =
  ∀a.
    case location a of Some l →
      (case location_kind l of
        Mutex → is_lock_or_unlock a
      | Non_atomic → is_load_or_store a
      | Atomic → is_load_or_store a ∨ is_atomic_action a)
    | None → T
```

```
is_at_location_kind = is_at_location_kind =
  case location a of
    Some l → (location_kind l = lk)
  | None → F
```

```
is_at_mutex_location a =
  is_at_location_kind a Mutex
```

```
is_at_non_atomic_location a =
  is_at_location_kind a Non_atomic
```

```
is_at_atomic_location a =
  is_at_location_kind a Atomic
```

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

```
threadwise_relation_over s rel =
  relation_over s rel ∧ (∀(a, b) ∈ rel. same_thread a b)
```

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

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

```
well_formed_action a =
  case a of
    Atomic_load _ mem_ord _ _ → mem_ord ∈
      {Mo_relaxed, Mo_acquire, Mo_seq_cst, Mo_consume}
  | Atomic_store _ _ mem_ord _ _ → mem_ord ∈
      {Mo_relaxed, Mo_release, Mo_seq_cst}
  | Atomic_rmw _ _ mem_ord _ _ _ → mem_ord ∈
      {Mo_relaxed, Mo_release, Mo_acquire, Mo_acq_rel, Mo_seq_cst, Mo_consume}
  | _ → T
```

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

```
well_formed_reads_from_mapping = well_formed_reads_from_mapping =
  relation_over actions (rf^-1) ∧
  (∀a. ∀a'. ∀b. a -rf-> b ∧ a' -rf-> b ⟹ (a = a')) ∧
  (∀(a, b) ∈ rf.
    same_location a b ∧
    (value_read b = value_written a) ∧
    (a ≠ b) ∧
    (is_at_mutex_location a ⟹
      is_unlock a ∧ is_lock b) ∧
    (is_at_non_atomic_location a ⟹
      is_store a ∧ is_load b) ∧
    (is_at_atomic_location a ⟹
      (is_atomic_store a ∨ is_atomic_rmw a ∨ is_store a)
      ∧ (is_atomic_load b ∨ is_atomic_rmw b ∨ is_load b)))
```

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

```
consistent_locks = consistent_locks =
∀l ∈ locations_of actions. (location_kind l = Mutex) ⟹ (
  let lock_unlock_actions =
    all_lock_or_unlock_actions_at (Some l) actions in
  let lock_order = sb|lock_unlock_actions in
  (* 30.4.1:5 - The implementation shall serialize those (lock and unlock) operations. *)
  strict_total_order_over lock_unlock_actions lock_order ∧

  (* 30.4.1:1 A thread owns a mutex from the time it successfully calls one of the lock functions until it calls unlock.*)
  (* 30.4.1:2 Requires: The calling thread shall own the mutex. *)
  (* 30.4.1:21 Effects: Releases the calling threads ownership of the mutex.*)
  (∀a_l ∈ lock_unlock_actions. is_unlock a_l ⟹
    (∃a_l ∈ lock_unlock_actions.
      a_l -lock_order-> a_l ∧ same_thread a_l a_l ∧ is_lock a_l)) ∧

  (* 30.4.1:7 Effects: Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.*)
  (* 30.4.1:8 Postcondition: The calling thread owns the mutex. *)
  (∀a_l ∈ lock_unlock_actions. is_lock a_l ⟹
    (∀a_l ∈ lock_unlock_actions.
      a_l -lock_order-> a_l ⟹ is_unlock a_l)))
```

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

```
release_sequence = a_rel -release-sequence-> b =
  is_at_atomic_location b ∧
  is_release a_rel ∧ (
    (b = a_rel) ∨
    (rs_element a_rel b ∧ a_rel -modification-order-> b ∧
      (∀c. a_rel -modification-order-> c -modification-order-> b ⟹
        rs_element a_rel c)))
```

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

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

```
hypothetical_release_sequence = a -hypothetical-release-sequence-> b =
  is_at_atomic_location b ∧ (
  (b = a) ∨
  (rs_element a b ∧ a -modification-order-> b ∧
    (∀c. a -modification-order-> c -modification-order-> b ⟹
      rs_element a c)))
```

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

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

```
synchronizes_with = a -synchronize-with-> b =
  (* – additional synchronization, from thread create etc. – *)
  a -additional-synchronized-with-> b ∨

  (same_location a b ∧ a ∈ actions ∧ b ∈ actions ∧ (
    (* – mutex synchronization – *)
    (is_unlock a ∧ is_lock b ∧ a -sb-> b) ∨

    (* – release/acquire synchronization – *)
    (is_release a ∧ is_acquire b ∧ ¬same_thread a b ∧
    (∃c. a -release-sequence-> c -rf-> b)) ∨

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

    (is_fence a ∧ is_release a ∧
    is_atomic_action b ∧ is_acquire b ∧
    (∃x. same_location x b ∧
      is_atomic_action x ∧ is_write x ∧
      a -sequenced-before-> x ∧
      (∃z. x -hypothetical-release-sequence-> z -rf-> b))) ∨

    (is_atomic_action a ∧ is_release a ∧
    is_fence b ∧ is_acquire b ∧
    (∃x. same_location a x ∧ is_atomic_action x ∧
      x -sequenced-before-> b ∧
      (∃z. a -release-sequence-> z -rf-> x))))))
```

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

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

```
carries_a_dependency_to = a -carries-a-dependency-to-> b =
  a ((-rf-> ∩ -sequenced-before->) ∪ -data-dependency->)^+ b
```

```
carries_a_dependency_to_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf =

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

```
dependency_ordered_before = a -dependency-ordered-before-> d =
  a ∈ actions ∧ d ∈ actions ∧
  (∃b. is_release a ∧ is_consume b ∧
    (∃a'. a -release-sequence-> a' -rf-> b) ∧
    (b -carries-a-dependency-to-> d ∨ (b = d)))
```

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

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

```
simple_happens_before = -simple-happens-before-> =
  (-sequenced-before-> ∪ -synchronize-with->)^+
```

```
consistent_simple_happens_before shb =
  irreflexive (shb^+)
```

```
inter_thread_happens_before = -inter-thread-happens-before-> =
  let r = -synchronize-with-> ∪
    (-dependency-ordered-before-> ∪
      (-synchronize-with-> ∘ -sequenced-before->)) in
  (r^+ ∪ (-sequenced-before-> ∘ r^+))^+
```

```
consistent_inter_thread_happens_before = consistent_inter_thread_happens_before =
  irreflexive (-inter-thread-happens-before->^+)
```

```
happens_before = -happens-before-> =
  -sequenced-before-> ∪ -inter-thread-happens-before->
```

```
all_sc_actions = all_sc_actions =
  {a. (is_seq_cst a ∨ is_lock a ∨ is_unlock a)}
```

```
consistent_sc_order = consistent_sc_order =
  let sc_happens_before = -happens-before->|all_sc_actions in
  let sc_mod_order = -modification-order->|all_sc_actions in
  strict_total_order_over all_sc_actions (sc^+) ∧
  sc_happens_before ⊆ sc ∧
  sc_mod_order ⊆ sc
```

```
consistent_modification_order = consistent_modification_order =
  (∀a. ∀b. a -modification-order-> b ⟹ same_location a b) ∧
  (∀l ∈ locations_of actions. case location_kind l of
      Atomic → (
        let actions_at_l = {a. (location a = Some l)} in
        let writes_at_l = {a_at_l. (is_store a ∨
          is_atomic_store a ∨ is_atomic_rmw a)} in
          strict_total_order_over writes_at_l
          (-modification-order->|actions_at_l) ∧
          (* happens-before at the writes of l is a subset of mo for l *)
          (-happens-before->|writes_at_l ⊆ -modification-order-> ∧
          (* Mo_seq_cst fences impose modification order *)
          (-sequenced-before->|(is_fence)
            ∘ -sequenced-before->|writes_at_l)
            ⊆ -modification-order->)
    | _ → (
        let actions_at_l = {a. (location a = Some l)} in
        (-modification-order->|actions_at_l) = {})))
```

```
visible_side_effect = a -visible-side-effect-> b =
  a -happens-before-> b ∧
  is_write a ∧ is_read b ∧ same_location a b ∧
  ¬(∃c. (c ≠ a) ∧ (c ≠ b) ∧
      is_write c ∧ same_location c b ∧
      a -happens-before-> c -happens-before-> b)
```

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

```
visible_sequence_of_side_effects_tail = visible_sequence_of_side_effects_tail vsse_head b =
  {c. vsse_head -modification-order-> c ∧
    ¬(b -happens-before-> c) ∧
    (∀a. vsse_head -modification-order-> a -modification-order-> c
      ⟹ ¬(b -happens-before-> a))}
```

```
myimage f s = {y. ∃x ∈ s. (y = f x)}
```

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

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

```
consistent_reads_from_mapping = consistent_reads_from_mapping =
  (∀b. (is_read b ∧ is_at_atomic_location b) ⟹
    (if (∃a_se. a_se -visible-side-effect-> b)
    then (∃a_se. a_se -visible-side-effect-> b ∧ a_se -rf-> b)
    else ¬(∃a. a -rf-> b))) ∧

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

  (∀(x, a) ∈ -rf->.
    ∀(y, b) ∈ -sc->.
    a -happens-before-> b ∧
    same_location a b ∧ is_at_atomic_location b
      ⟹ (x = y) ∨ x -modification-order-> y) ∧
(* new CoWR *)
  (∀(a, b) ∈ -happens-before->.
    ∀c.
      c -rf-> b ∧
      is_write a ∧ same_location a b ∧ is_at_atomic_location b
        ⟹ (c = a) ∨ a -modification-order-> c) ∧
(* new CoRW *)
  (∀(a, b) ∈ -happens-before->.
    ∀c.
      c -rf-> a ∧
      is_write b ∧ same_location a b ∧ is_at_atomic_location a
        ⟹ c -modification-order-> b) ∧

  (∀(a, b) ∈ -rf->. is_atomic_rmw b
    ⟹ a -modification-order-> b) ∧

  (∀(a, b) ∈ -rf->. is_seq_cst b
    ⟹ (¬is_seq_cst a ∧ (∀x. x -sc-> a ∧ is_write x ∧ same_location b x ⟹ x -modification-order-> a)) ∨
      a -sc-> b ∧ is_write x ∧ same_location b x ∧ x ≠ b) ∧

  (* -Fence restrictions- *)

  (* 29.3:3 *)
  (∀a. ∀(x, b) ∈ -sequenced-before->. ∀y.
    (is_fence x ∧ is_seq_cst x ∧ is_atomic_action b ∧
      is_write a ∧ same_location a b ∧
      a -sc-> x ∧ y -rf-> b)
      ⟹ (y = a) ∨ a -modification-order-> y) ∧

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

  (* 29.3:5 *)
  (∀(a, x) ∈ -sequenced-before->. ∀(y, b) ∈ -sequenced-before->. ∀z.
    (is_atomic_action a ∧ is_fence x ∧ is_seq_cst x ∧
      is_write a ∧ is_fence y ∧ is_seq_cst y ∧
      is_atomic_action b ∧ same_location a b ∧
      x -sc-> y ∧ z -rf-> b)
      ⟹ (z = a) ∨ a -modification-order-> z)
```

```
all_data_dependency = -all_data_dependency-> =
  (-rf-> ∪ -carries-a-dependency-to->)^+
```

```
consistent_control_dependency = consistent_control_dependency =
  irreflexive ((-control-dependency-> ∪ -all_data_dependency->)^+)
```

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

```
indeterminate_reads actions threads = indeterminate_reads =
  {b. is_read b ∧ ¬(∃a. a -rf-> b)}
```

```
unsequenced_races = unsequenced_races = {(a, b).
  (a ≠ b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
  same_thread a b ∧
  ¬(a -sequenced-before-> b ∨ b -sequenced-before-> a)}
```

```
data_races = data_races = {(a, b).
  (a ≠ b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
  ¬same_thread a b ∧
  ¬(is_atomic_action a ∧ is_atomic_action b) ∧
  ¬(a -happens-before-> b ∨ b -happens-before-> a)}
```

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

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

Is C++11 hopelessly complicated?

Programmers cannot be given this model.

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

- Can we compile to x86?

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

- Can we compile to Power?

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

Is C++11 hopelessly complicated?

Simplifications:

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

Derivative models:

- without consume, happens-before is transitive

The current state of the standard

Fixed:

- in some cases, happens-before was cyclic

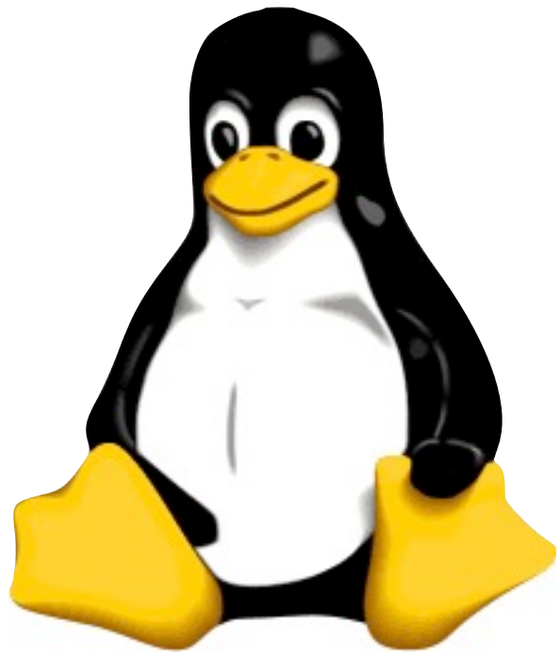
- coherence

- `seq_cst` atomics were more broken

Not fixed:

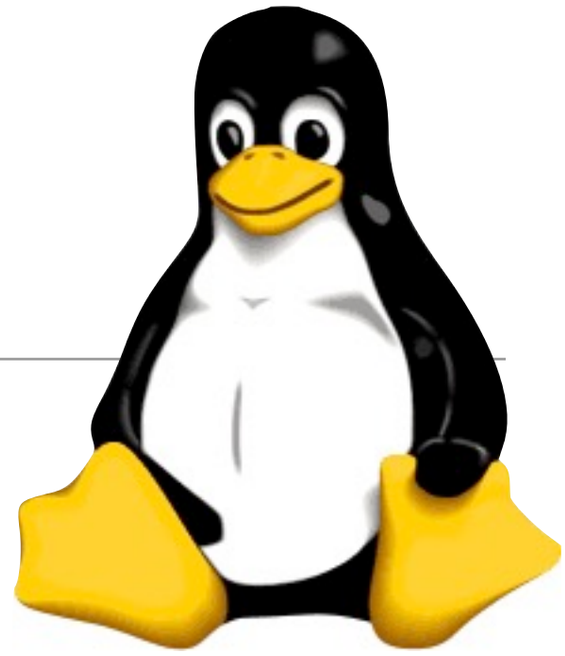
- out of thin air reads (and self satisfying conditionals)

- `seq_cst` atomics do not guarantee SC



2. The Linux memory model

The Linux memory model



Facts:

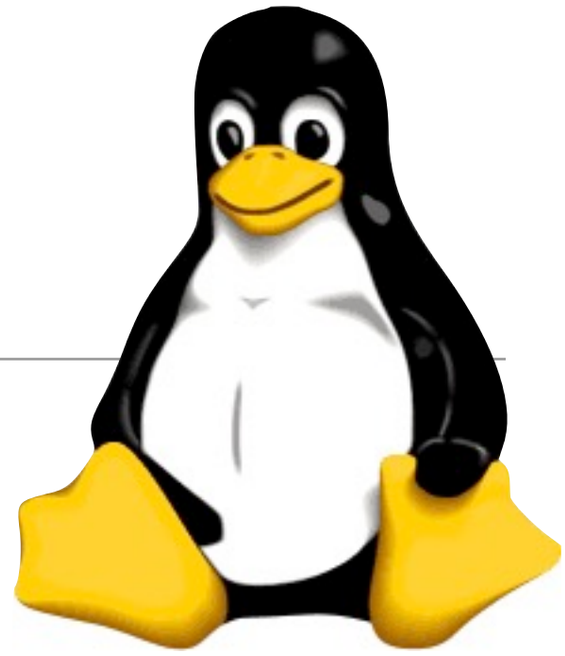
- abstraction layer over hardware and compilers

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

- documented by a text file:

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

The Linux memory model



Facts:

- abstraction layer over hardware and compilers

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

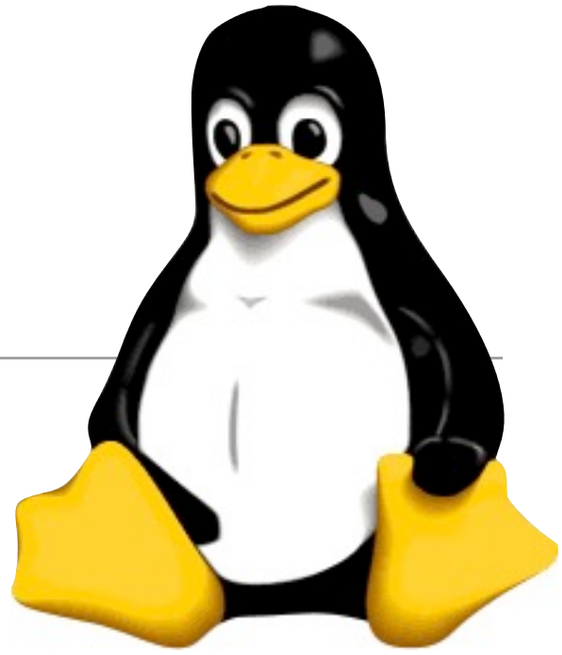
- documented by a text file:

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

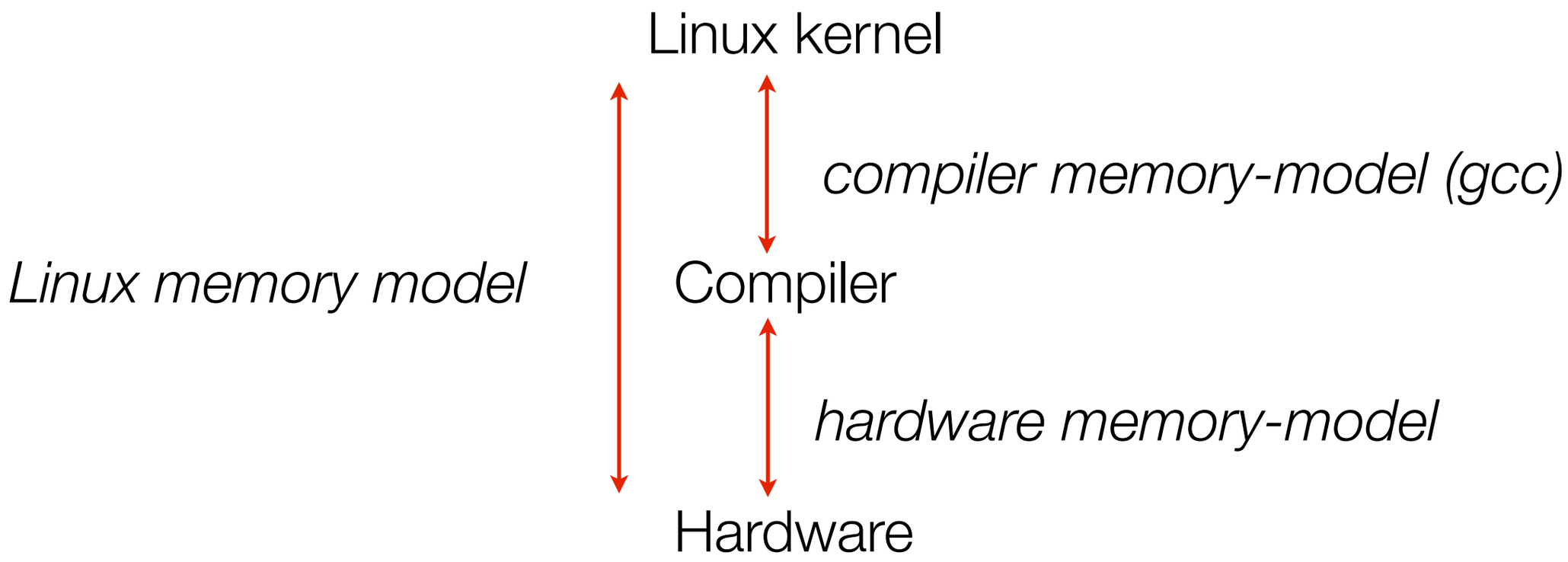
More facts: ...some time ago...

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

The Linux memory model



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



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

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

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

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

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

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

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

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

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

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

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

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

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

parisc: TSO, similar to x86.

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

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

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

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

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

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

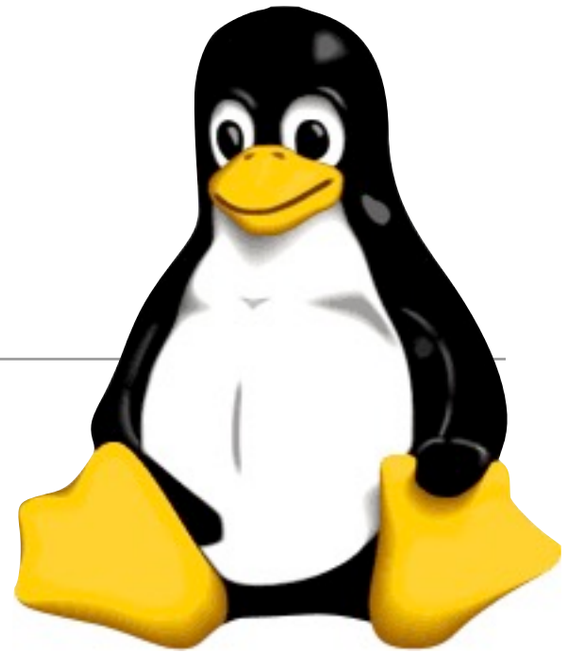
um: Looks like an x86 knockoff judging by the system.h file.

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

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

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

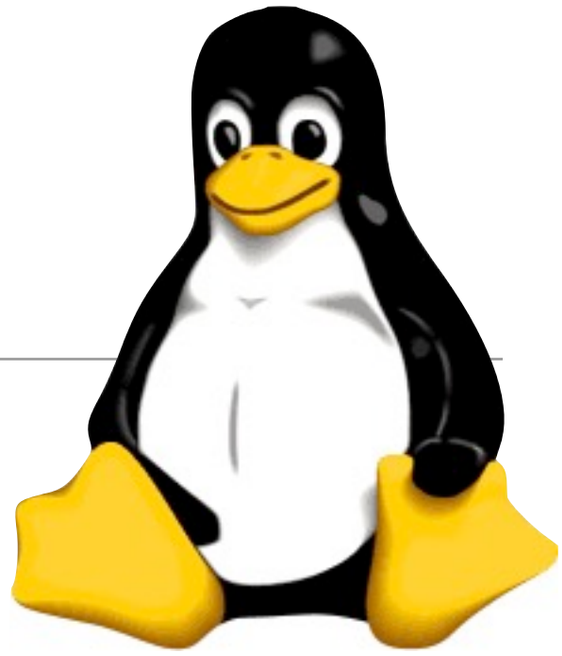
The Linux memory model



My intuition:

Annoying facts:

The Linux memory model



My intuition:

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

Annoying facts:

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

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

ACCESS_ONCE

Compilers can remove (or duplicate) accesses.

```
for (;;) {
  struct task_struct *owner;

  owner = ACCESS_ONCE(lock->owner);
  if (owner && !mutex_spin_on_owner(lock, owner))
      break;
  /* ... */
```

```
owner = ACCESS_ONCE(lock->owner);
    for (;;) {
     if (owner && !mutex_spin_on_owner(lock, owner))
       break;
```

ACCESS_ONCE

Compilers can remove (or duplicate) accesses.

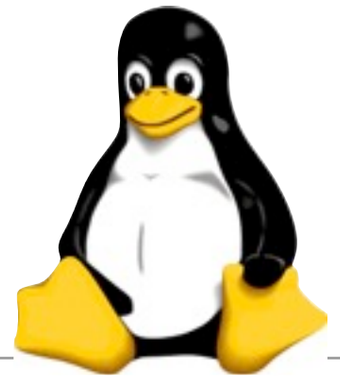
```
for (;;) {
  struct task_struct *owner;

  owner = ACCESS_ONCE(lock->owner);
  if (owner && !mutex_spin_on_owner(lock, owner))
      break;
  /* ... */
```

ACCESS_ONCE prevents this optimisation

```
owner = ACCESS_ONCE(lock->owner);
    for (;;) {
     if (owner && !mutex_spin_on_owner(lock, owner))
       break;
```

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

The Linux memory model: macros



```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
```

on x86:

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

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

on Power:

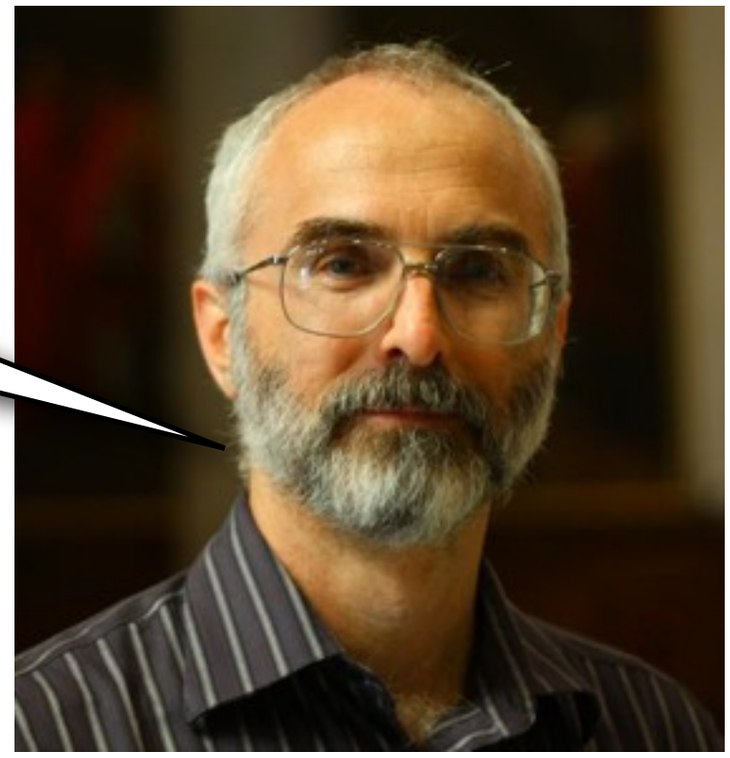
```
#define mb()   __asm__ __volatile__ ("sync" : : : "memory")
#define rmb()  __asm__ __volatile__ ("sync" : : : "memory")
#define wmb()  __asm__ __volatile__ ("sync" : : : "memory")
#define read_barrier_depends()  do { } while(0)
```

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

Challenging research direction:

Sort out what the REAL Linux memory model is

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



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

Out of thin-air reads

Memory access synchronisation

```
x = y = 0
```

Thread 1

```
y = 1

x.store(1,MO_RELEASE)
```

Thread 2

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

   r2 = y
```

Memory access synchronisation

```
x = y = 0
```

Thread 1

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

Thread 2

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

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

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

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1 | Thread 2

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

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1 | Thread 2

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

DATA RACE

Two conflicting accesses not related by happens-before

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1

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

Thread 2

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

WELL DEFINED

but r2 = 0 is possible

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1

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

Thread 2

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

Allow a RELAXED load to see any store that:

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

Intuition
the compiler (or hardware) can reorder independent accesses

`x = y = 0`

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

Allow a RELAXED load to see any store that:

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

Shorthand

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

Out-of-thin-air

```
x = y = 0
```

Thread 1	Thread 2
`r1 = x`	`r2 = y`
`y = r1`	`x = 42`

Out-of-thin-air

```
x = y = 0
```

Thread 1

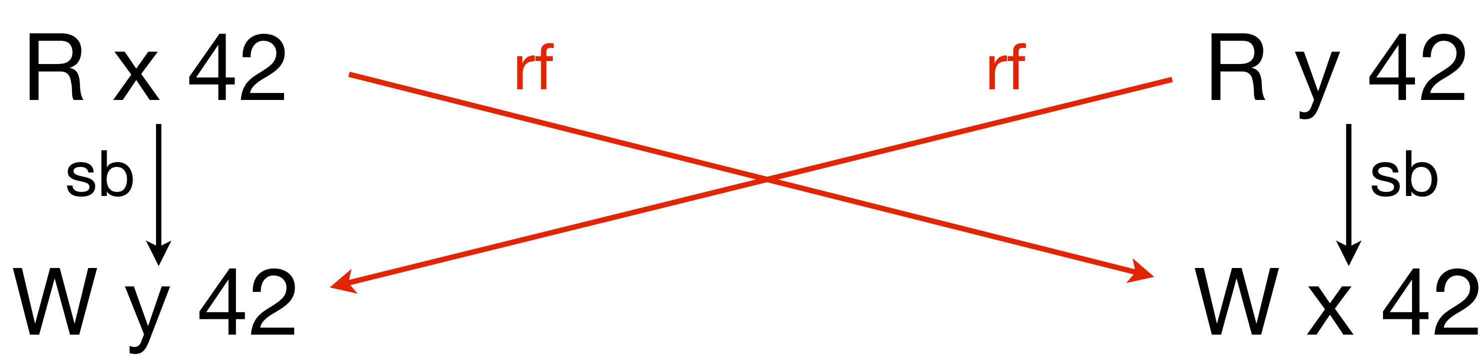
```
r1 = x
y = r1
```

Thread 2

```
r2 = y
x = 42
```

r1 = r2 = 42

is a valid execution.



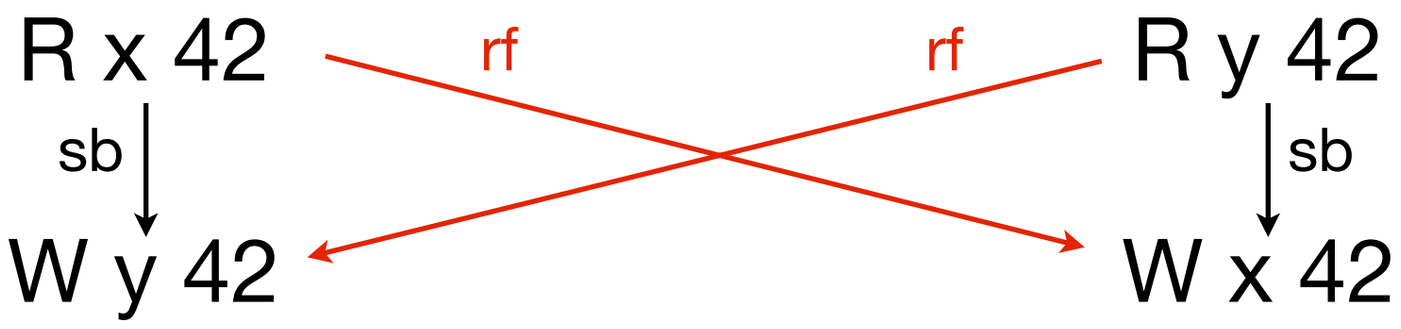
Intuition
the compiler (or hardware) can reorder independent accesses

`x = y = 0`

Thread 1	Thread 2
`r1 = x`	`r2 = y`
`y = r1`	`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

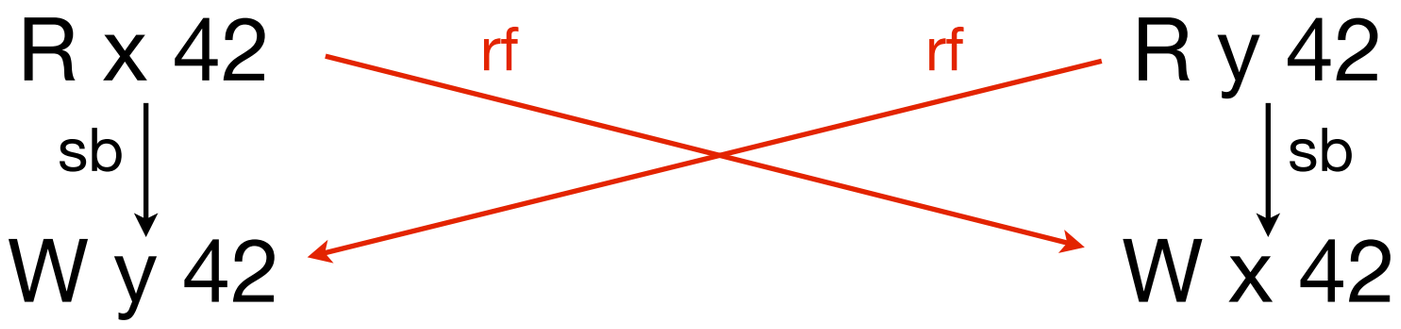
$$x = y = 0$$

Thread 2

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

r1 = r2 = 42

is also an allowed execution



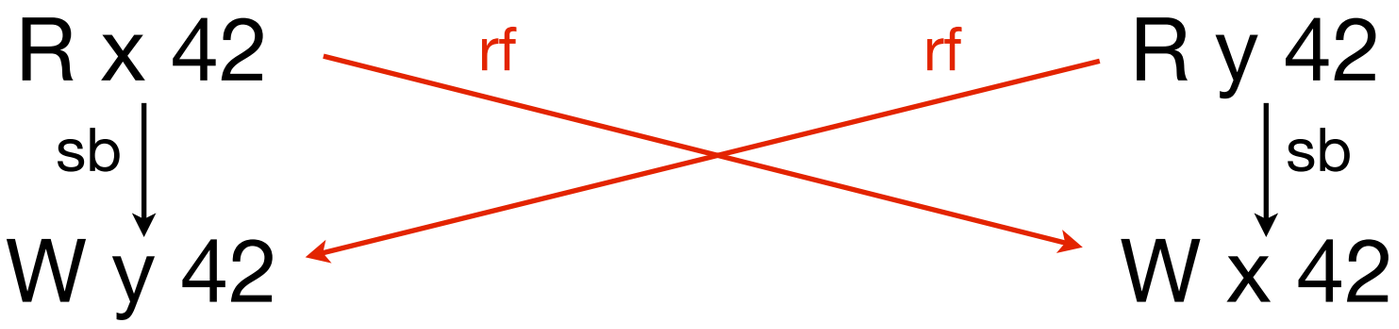
the value 42 appears *out-of-thin-air*

`x = y = 0`

Thread 1	Thread 2
`r1 = x`	`r2 = y`
`y = r1`	`x = r2`

r1 = r2 = 42

is also an allowed execution



Speculation can justify out-of-thin-air reads

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

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

initially x = y = 0

r1 := x	r2 := y
y := r1	x := r2
print r1	print r2

Speculation can justify out-of-thin-air reads

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

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

initially x = y = 0

r1 := x	r2 := y
y := r1	x := r2
print r1	print r2

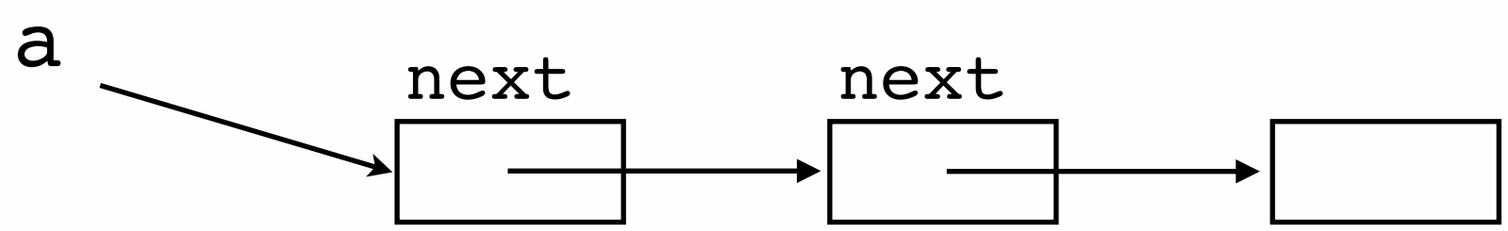
It does not happen in practice... even if it might!

Consequences of out-of-thin-air reads

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

Thread 1

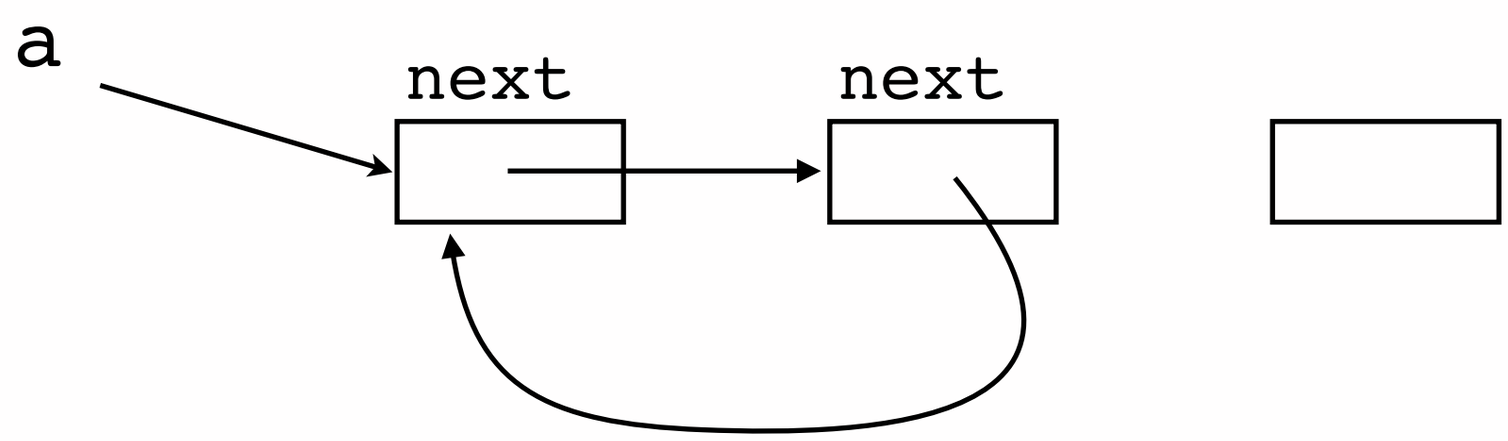
```
r1 = a->next
r1->next = a
```



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

Thread 1

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



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

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

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

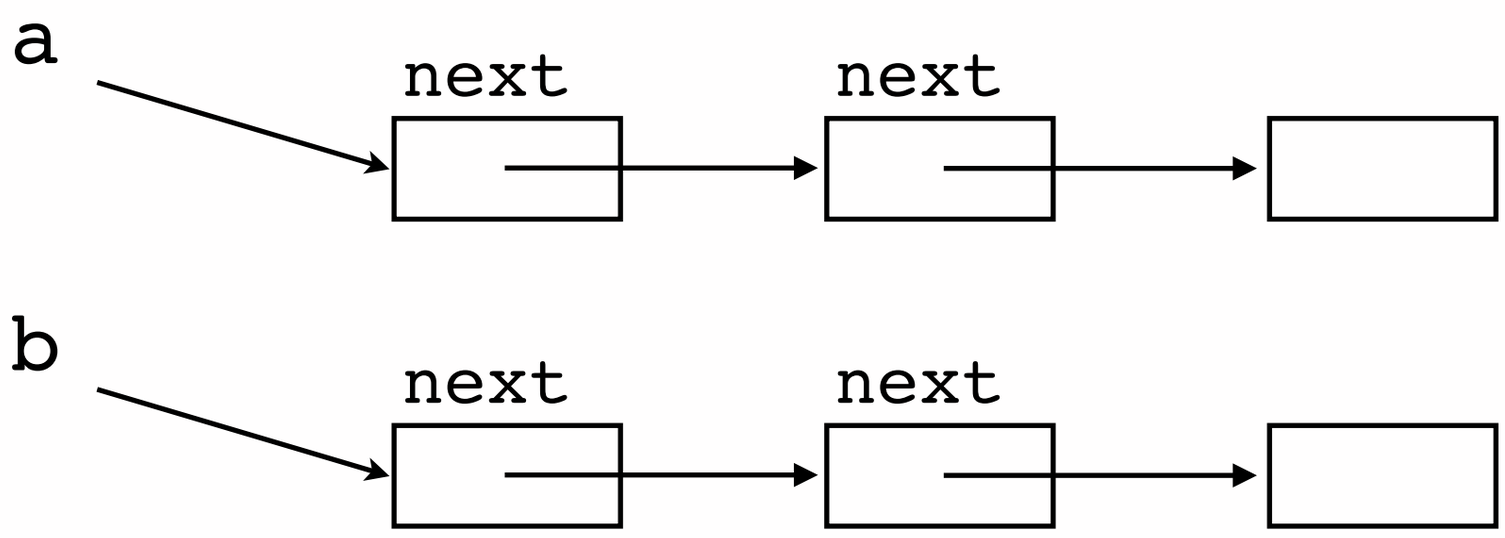
Thread 1	Thread 2
`r1 = a->next`	`r2 = b->next`
`r1->next = a`	`r2->next = b`

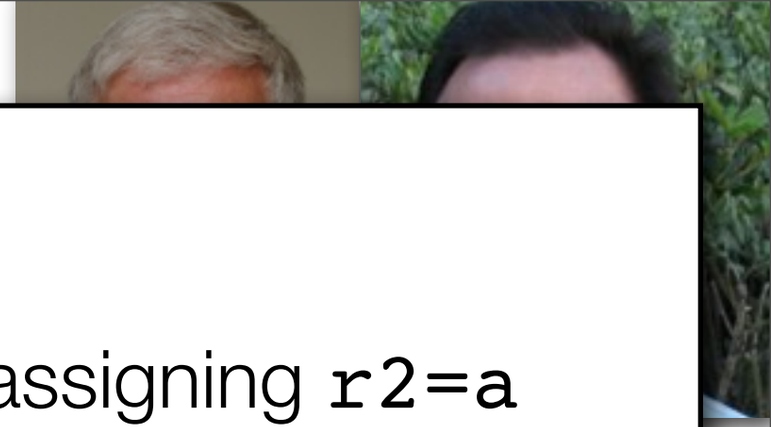
If **a** and **b** initially reference disjoint data-structures

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

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

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



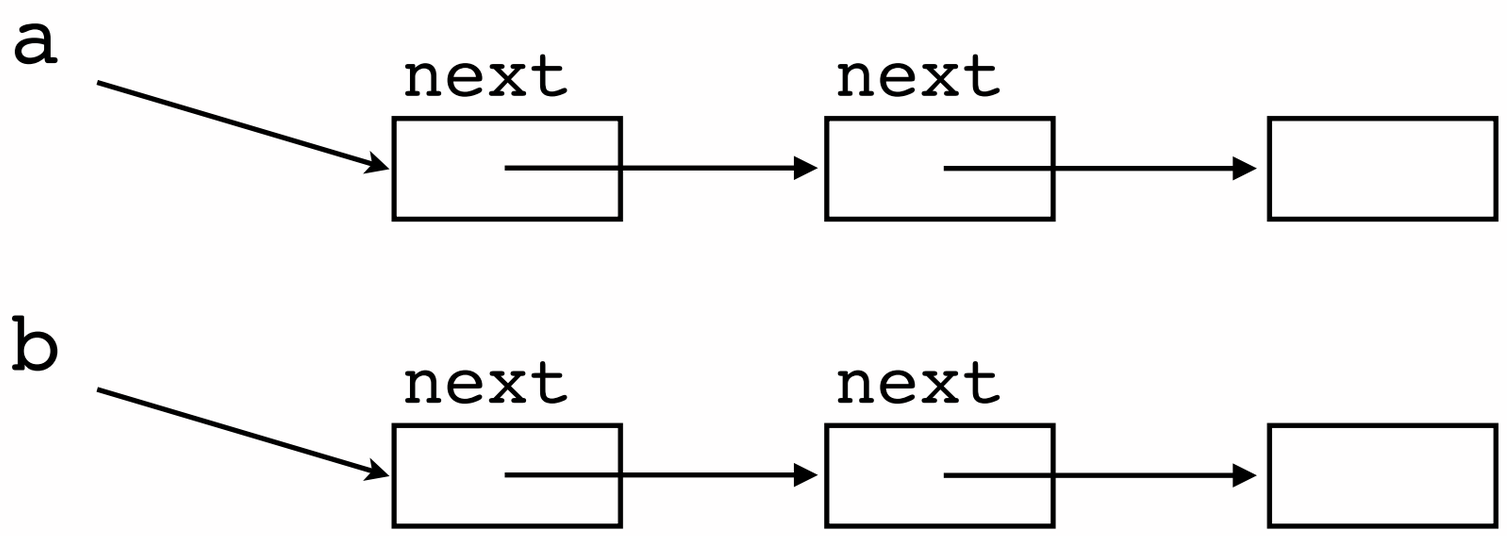


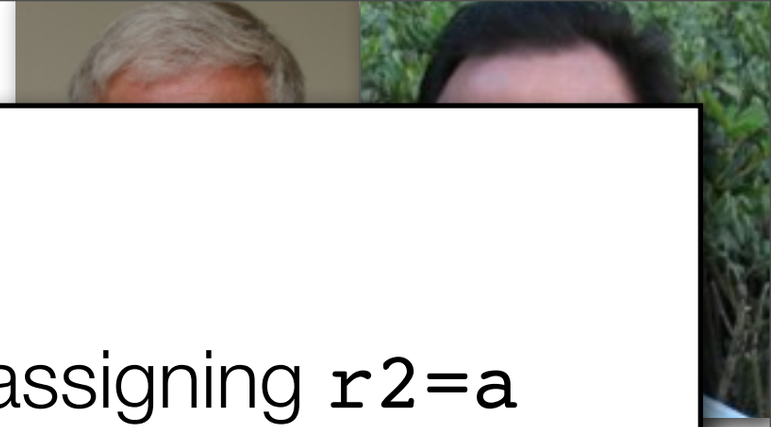
If the compiler speculates `r1=b` and `r2=a`, then

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

(and symmetrically to justify `r1=b`)

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





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

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

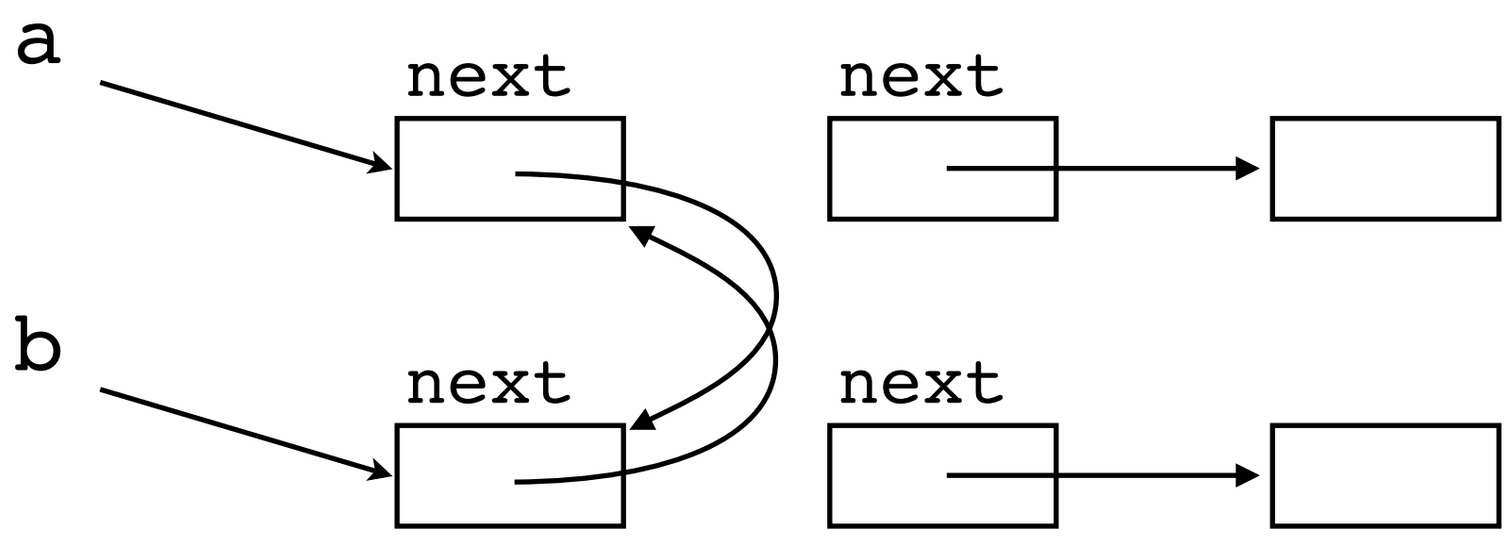
(and symmetrically to justify `r1=b`)

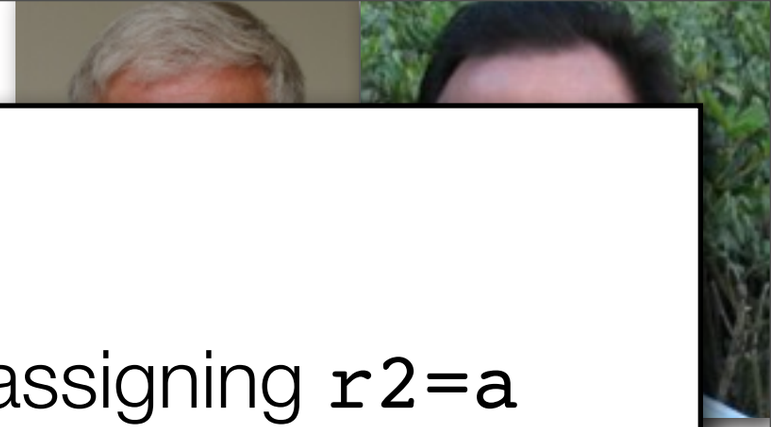
Thread 1

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

Thread 2

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



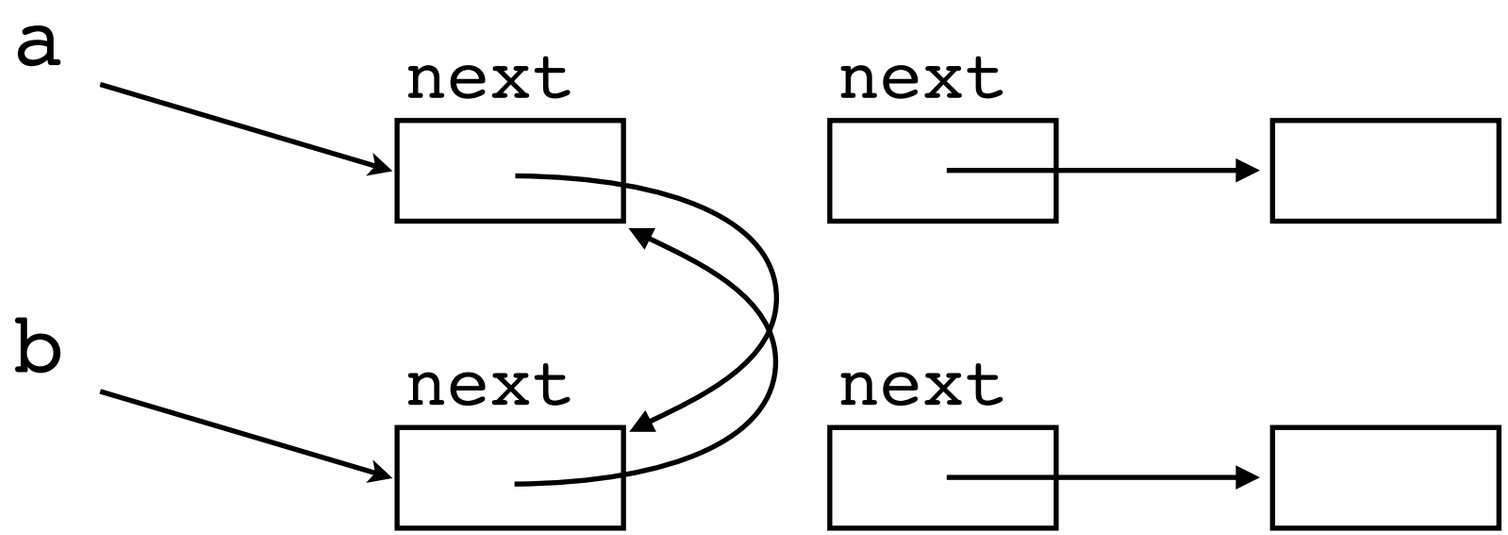


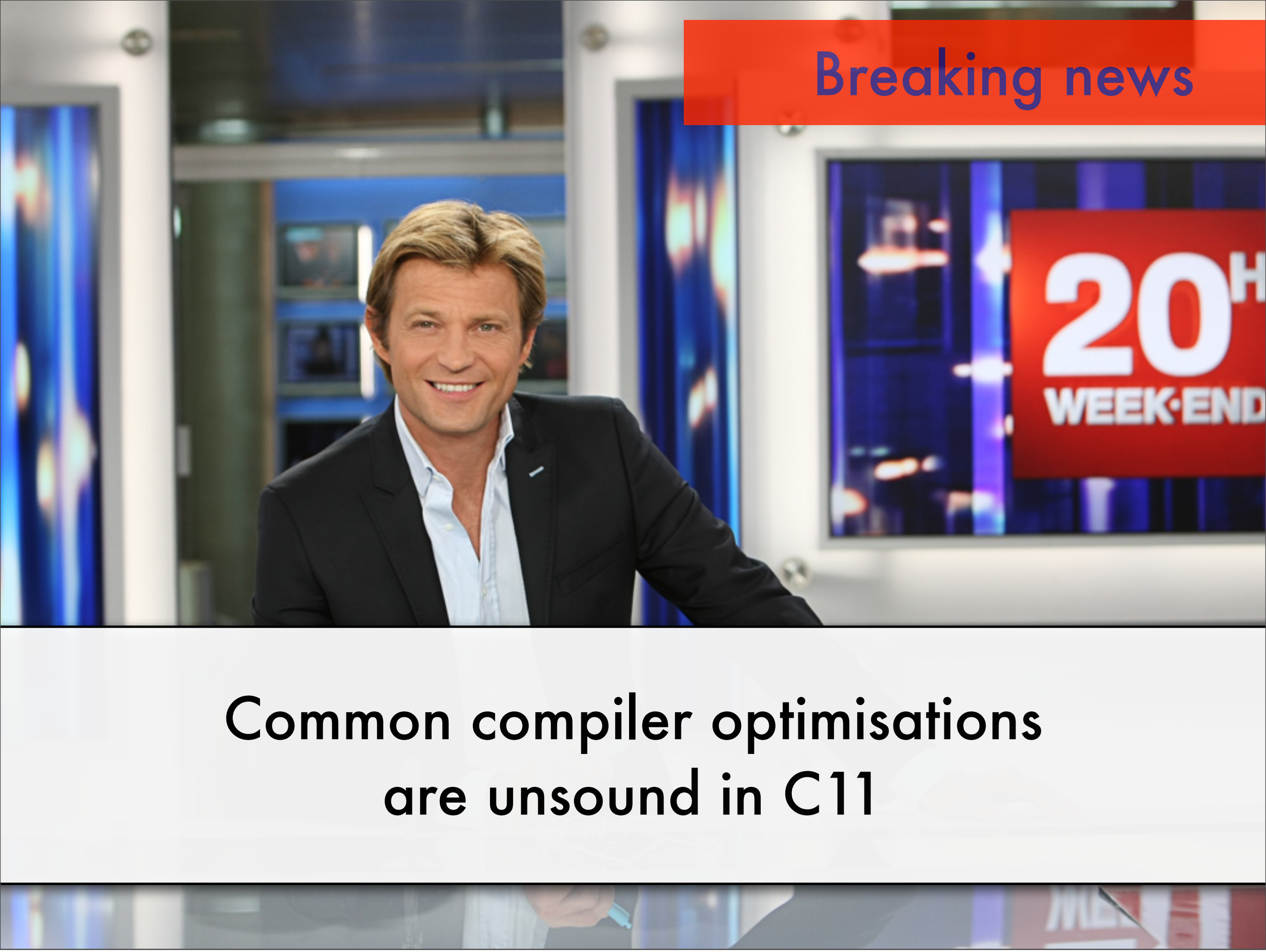
If the compiler speculates `r1=b` and `r2=a`, then

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

(and symmetrically to justify `r1=b`)

Break our basic intuitions about memory and sharing!





Common compiler optimisations are unsound in C11

x = y = a = 0

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

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

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

Remark 1

This code is not racy!

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

```
x = y = a = 0

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

Remark 2

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

is the only possible final state

$$x = y = a = 0$$

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

Consider sequentialisation:

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

(ought to be correct)

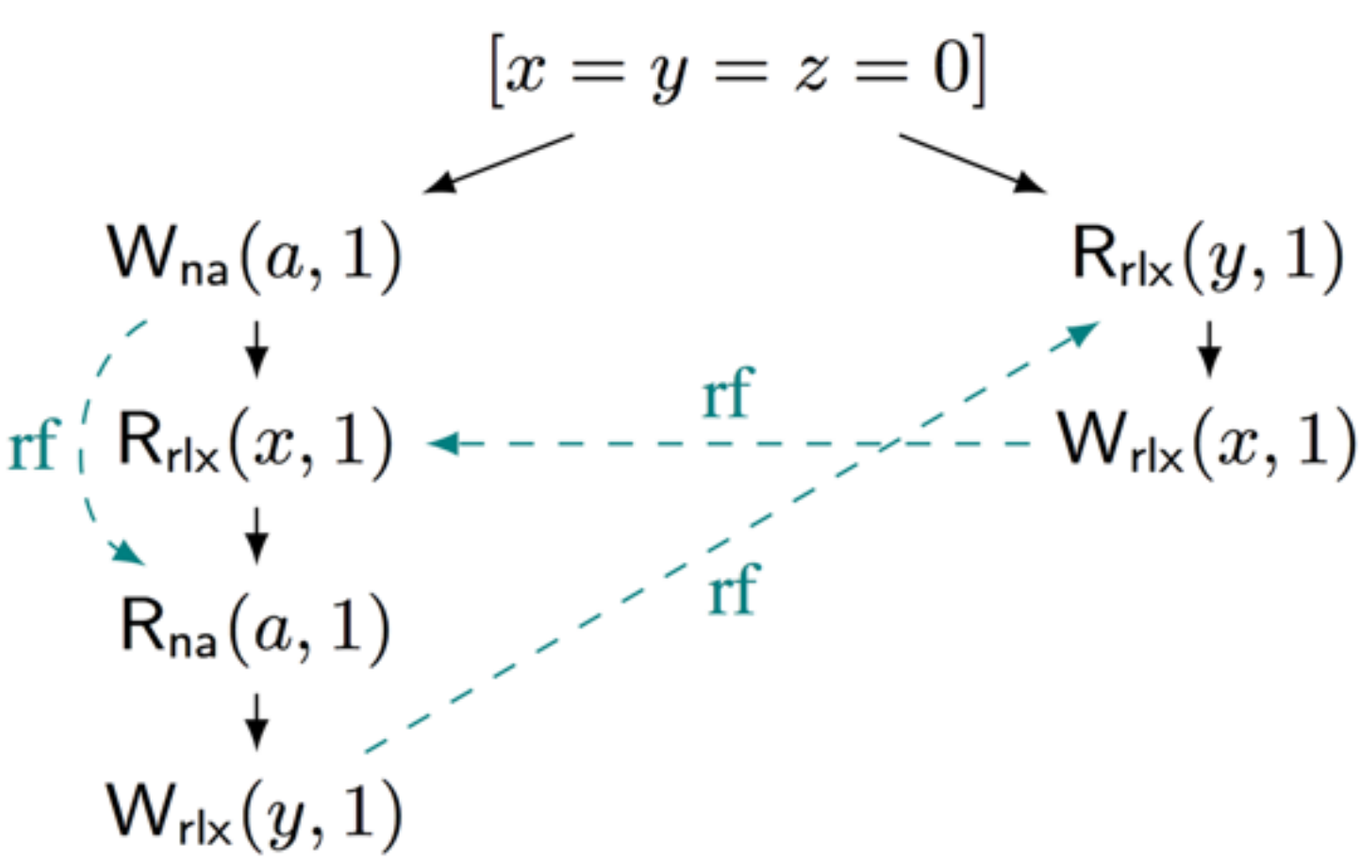
```
x = y = a = 0
```

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

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

```
                    x = y = a = 0

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

```
                    x = y = a = 0

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



a = 1

x = y = 42

is also possible

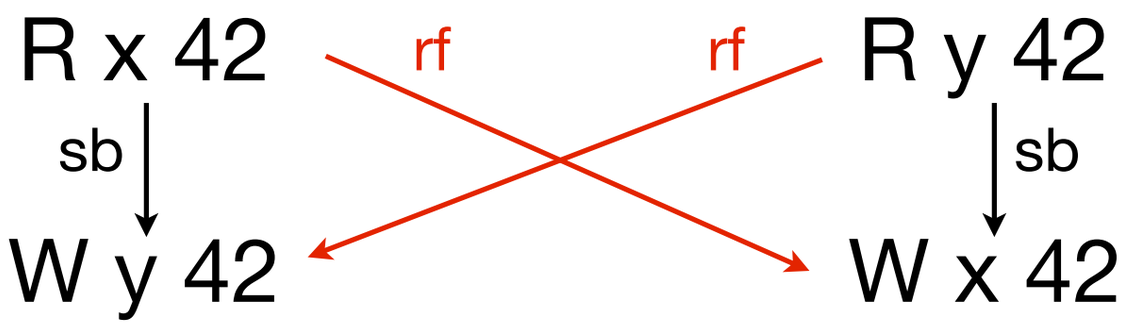
```
                    x = y = a = 0

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

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

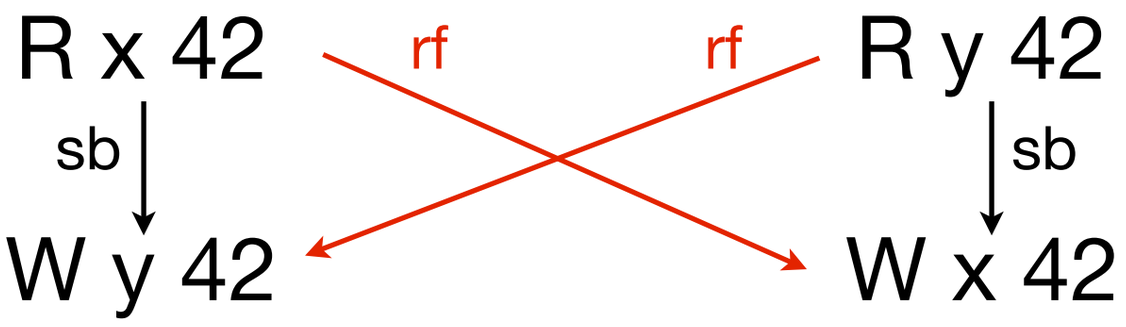
including *expression linearisation* and *roach-motel reorderings*

Are there any solutions?

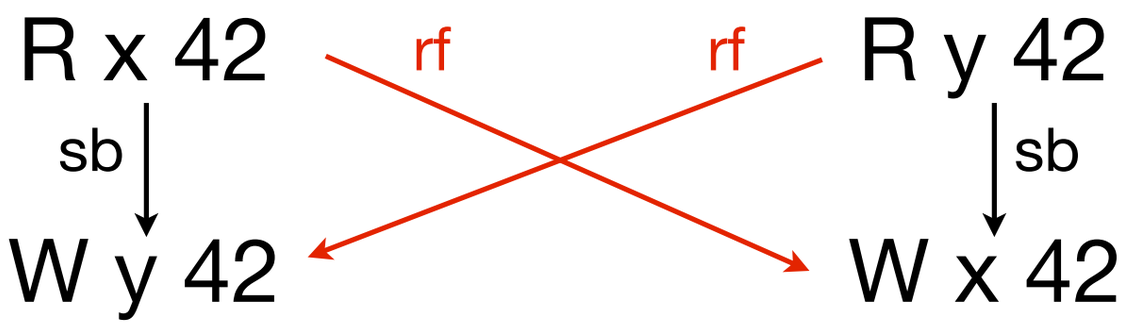
Thread 0	Thread 1
`r1 = x`	`r2 = y`
`y = r1`	`x = 42`



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

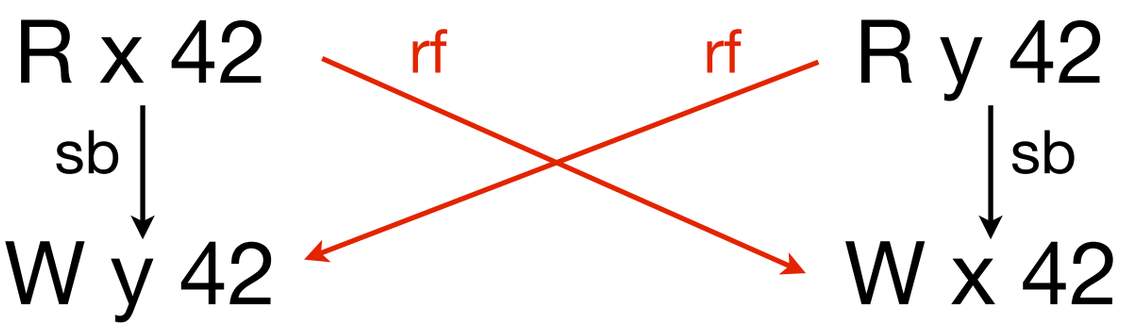


Thread 0	Thread 1
`r1 = x`	`r2 = y`
`y = r1`	`x = 42`

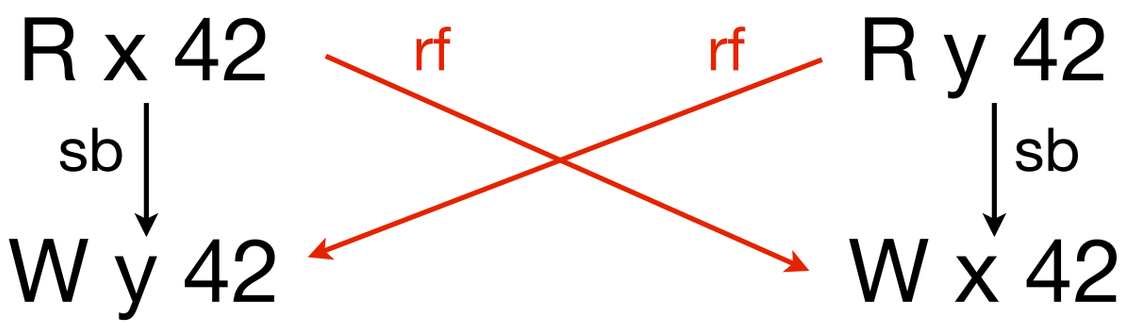


r1 = r2 = 42. Can you spot the difference?

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

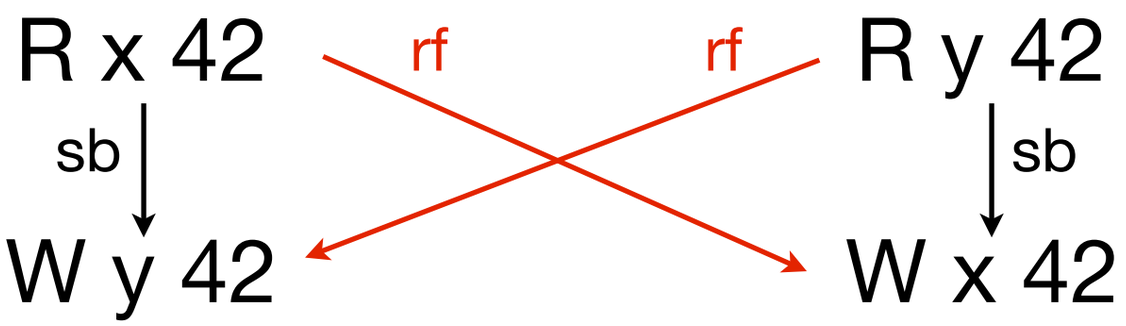


Thread 0	Thread 1
`r1 = x`	`r2 = y`
`y = r1`	`x = 42`



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

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

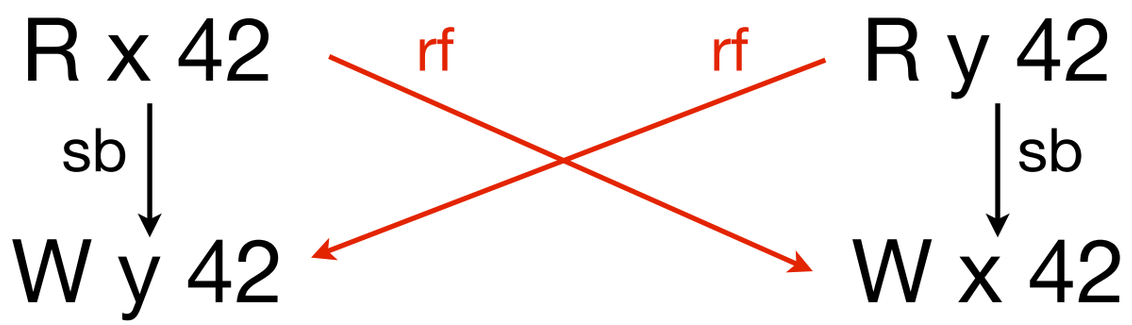


Solution 1.

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

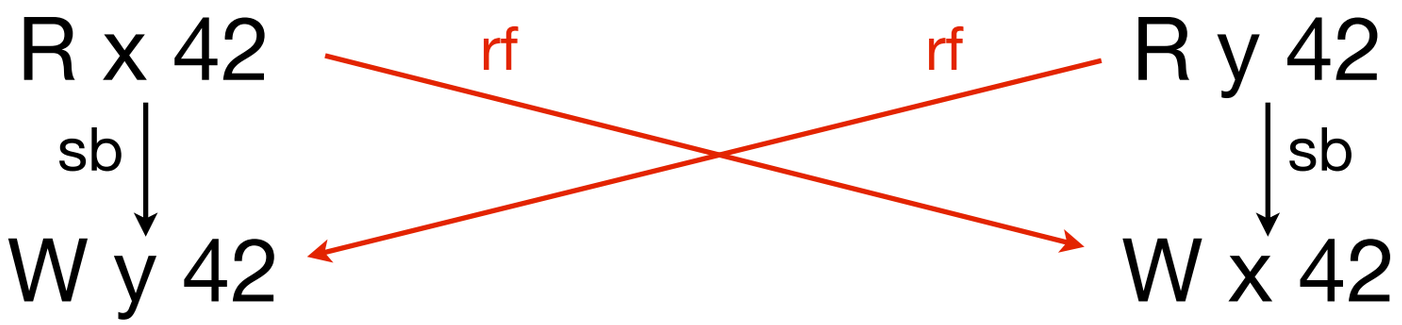
Compiler writers
do not want to track all dependencies

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

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

Solution 2. Brute force

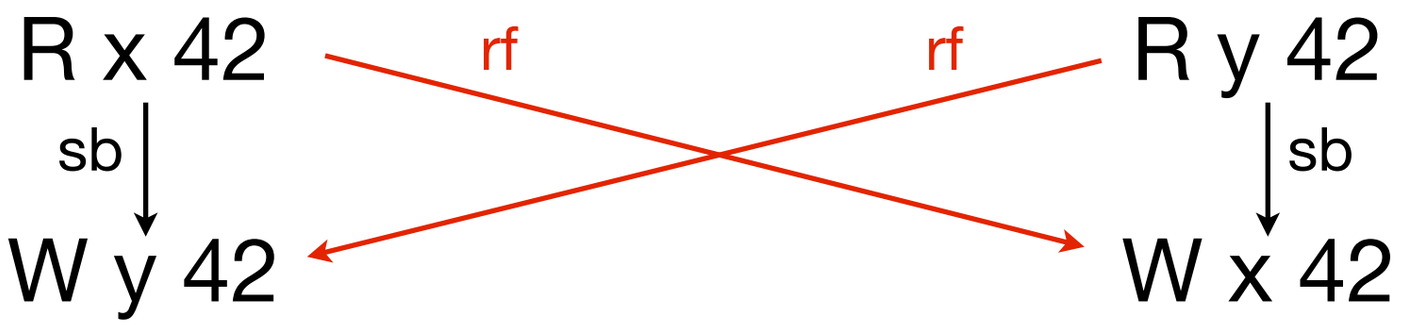
Disallow cycles altogether



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

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

Disallow cycles altogether



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

Solution 3. less brute force

Allow cycles but make this racy

by allowing a to read 1

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

Efficient implementation of atomics on ARM/GPUs

but all R/W reorderings are unsound

Allow cycles but make this racy

by allowing a to read 1

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

State of the art

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

Current proposal for C14

3. A word on techniques for data-race detection

Data race detection: dynamic approaches

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

happens-before ordering

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

no false positives

drawback: misses races
occurring on rare executions

lockset computation

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

popularised by Eraser (Savage et al.) ’97

can detect races not observed in
the execution being monitored

drawback: unsound (false positives)

Examples of lockset computation

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

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

```
locks held:  1:b     1:b,a                    2:a
C(x):                x:a,b                    x:a
```

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

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

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

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

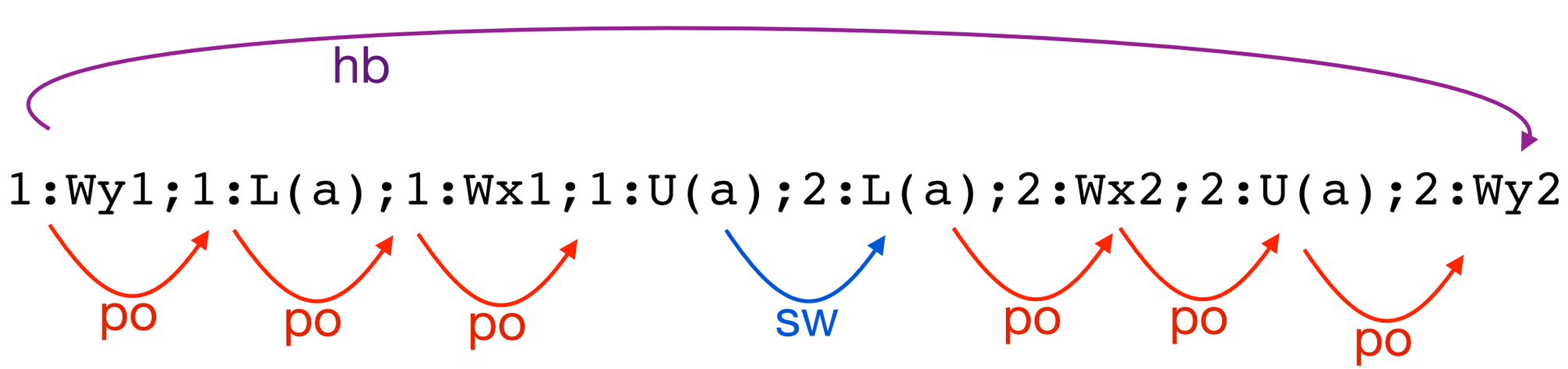
lockset for x empty at the end, possible data-race

lockset vs happens-before

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

This program has a race on y

If only the execution below is observed:



happens-before computation does not report a race.

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

lockset vs happens-before (2)

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

This program instead is DRF.

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

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

Data race detection

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

happens-before ordering

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

sound

drawback: misses races
occurring on rare executions

lockset computation

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

popularised by Eraser (Savage et al.) ’97

can detect races not observed in
the execution being monitored

drawback: unsound (false positives)

Data race detection

Mode er on:

hap

recons
in
report all
conflicti y

al.) ’97

d in

ed

dr

occu ves)

Current state of the art:

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

Helgrind, RaceFuzzer, ThreadSanitizer...

Impressive:

tolerable slowdown on large applications
found thousands races

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

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

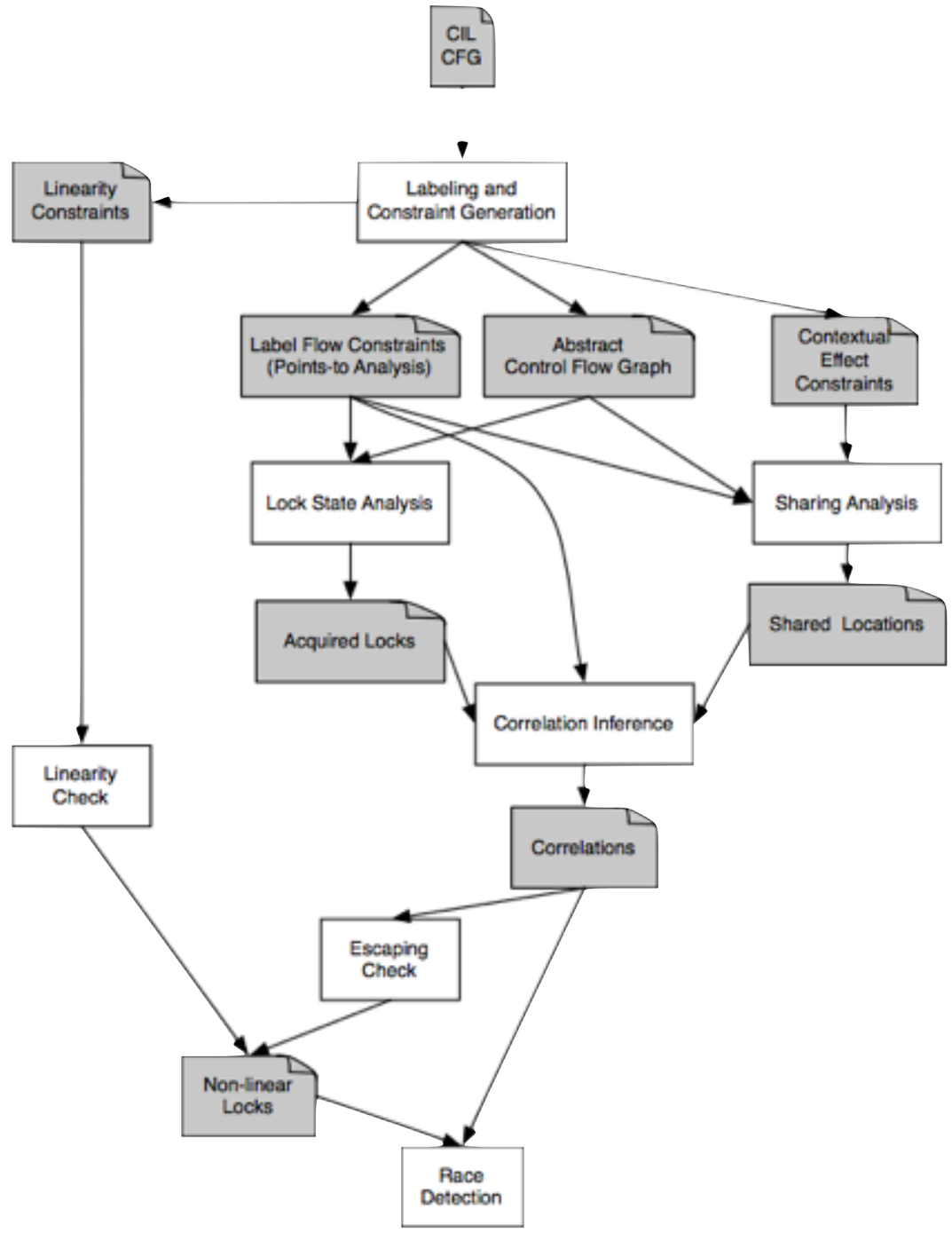
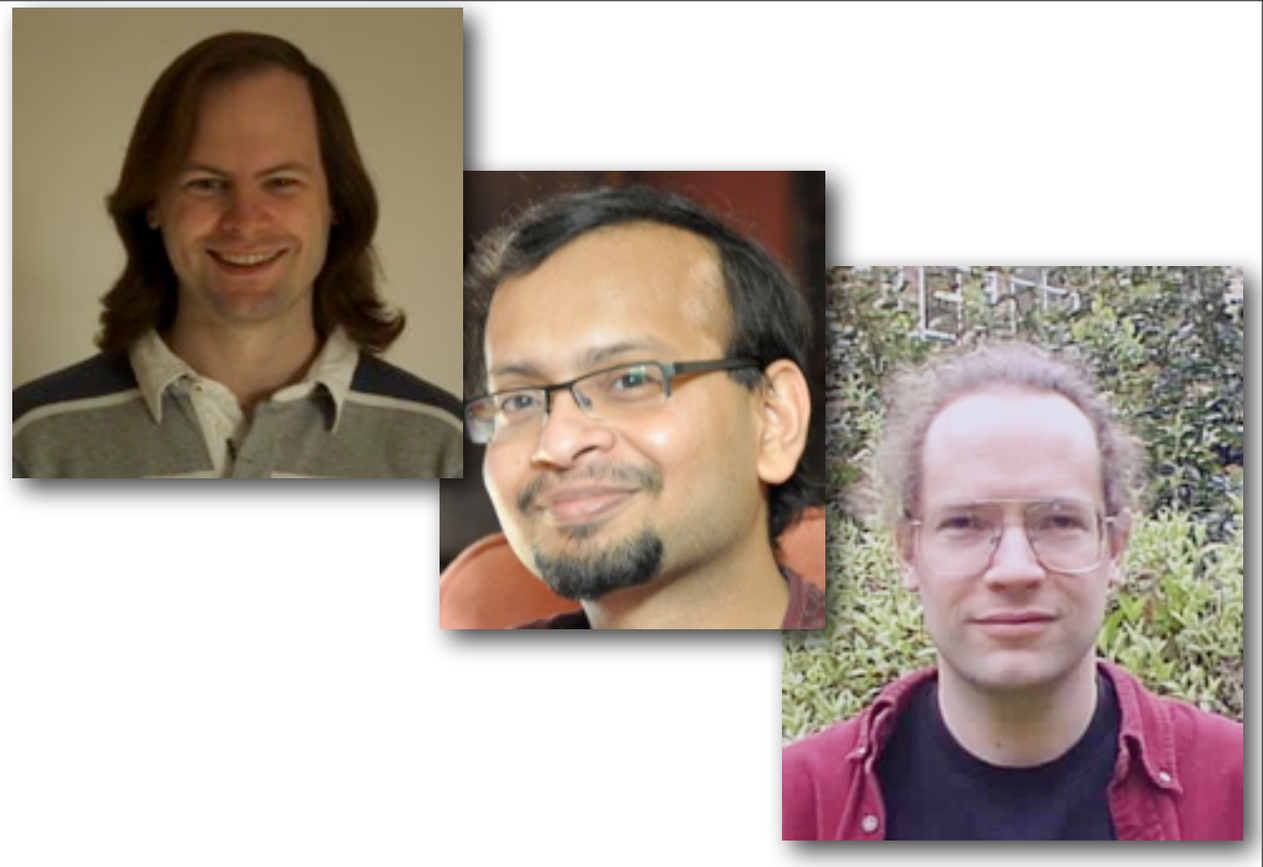


Fig. 1. LOCKSMITH architecture

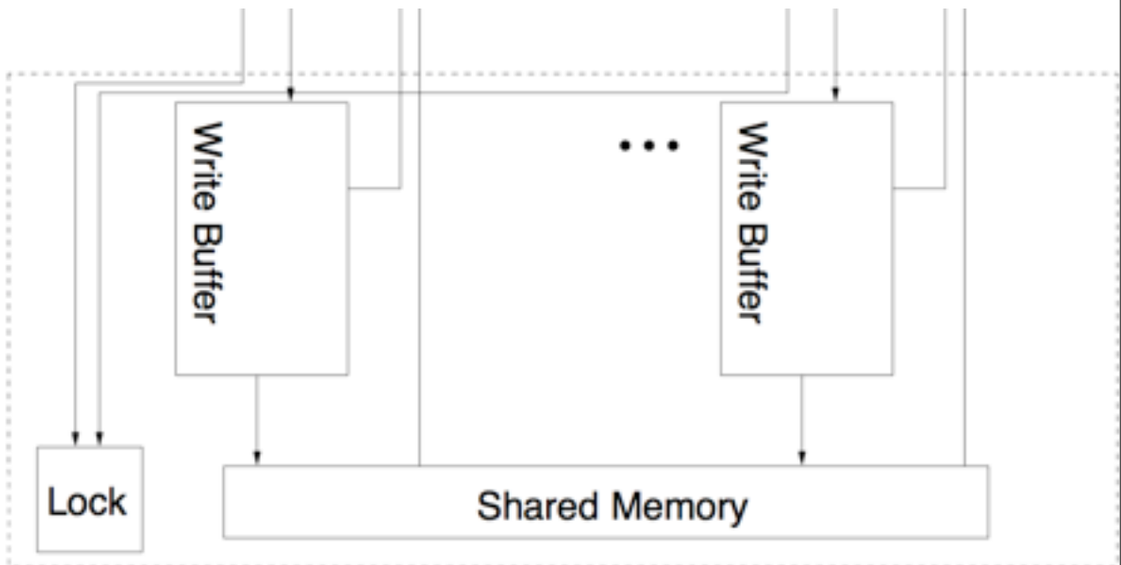


4. Sketch of an operational formalisation of x86-TSO

...starting with a formalisation of SC

Separate language and memory semantics

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



program
semantics defined via an LTS

memory
semantics defined via an LTS

Labels for interaction:

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

Separate language and memory semantics

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

Separate language and state semantics

proved to be a very good choice

in many (unrelated) projects I worked on!

semantics defined via an LTS

semantics defined via an LTS

Labels for interaction:

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

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

+ other events for barriers and locked instructions

A tiny language

$location,\ x,\ m$			address (or pointer value)
$integer,\ n$			integer
$thread\_id,\ t$			thread id
$k,\ i,\ j$			
$expression,\ e$	$::=$		expression
	$\mid$	n	integer literal
	$\mid$	$*x$	read from pointer
	$\mid$	$*x = e$	write to pointer
	$\mid$	$e; e'$	sequential composition
	$\mid$	$e + e'$	plus
$process,\ p$	$::=$		process
	$\mid$	$t{:}e$	thread
	$\mid$	$p \mid p'$	parallel composition

What can a thread do in isolation?

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

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

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

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

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

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

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

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

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

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

Example

Show that the expression:

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

can perform the following trace:

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

Lifting to processes

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

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

Actions are labelled by the thread that performed the action.

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

Free interleaving.

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

A sequentially consistent memory

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

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

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

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

SC semantics: whole system transitions

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

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

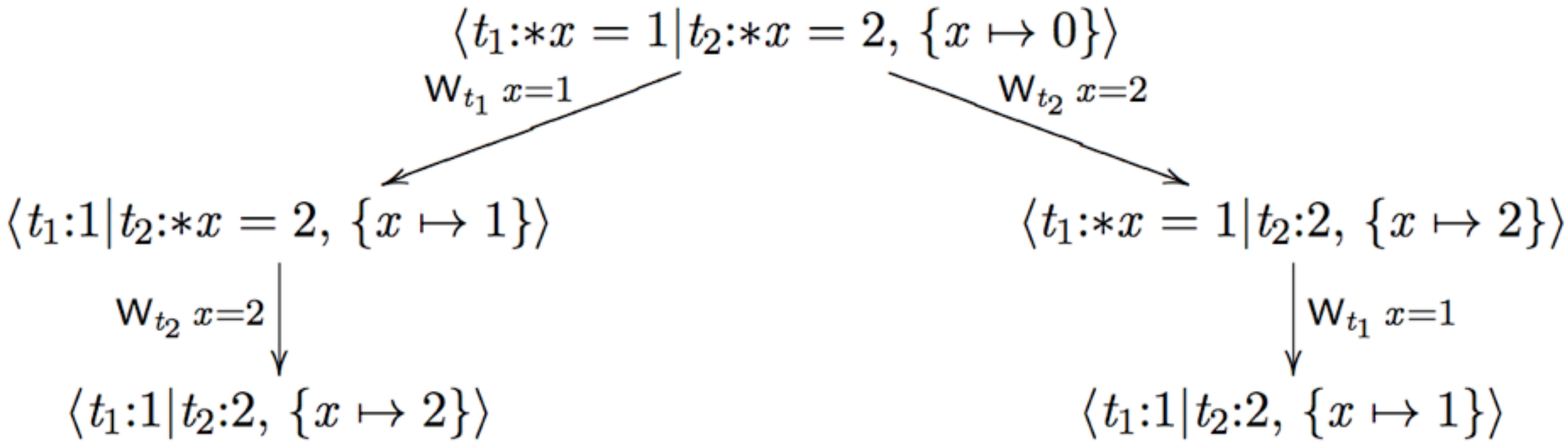
$$\frac{p \xrightarrow{\mathsf{W}_t\, x=n} p' \qquad M \xrightarrow{\mathsf{W}\, x=n} M'}{\langle p,\, M\rangle \xrightarrow{\mathsf{W}_t\, x=n} \langle p',\, M'\rangle}\ \textsc{Swrite}$$

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

Synchronising between the processes and the memory.

SC semantics, example

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



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

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

A sequentially consistent memory

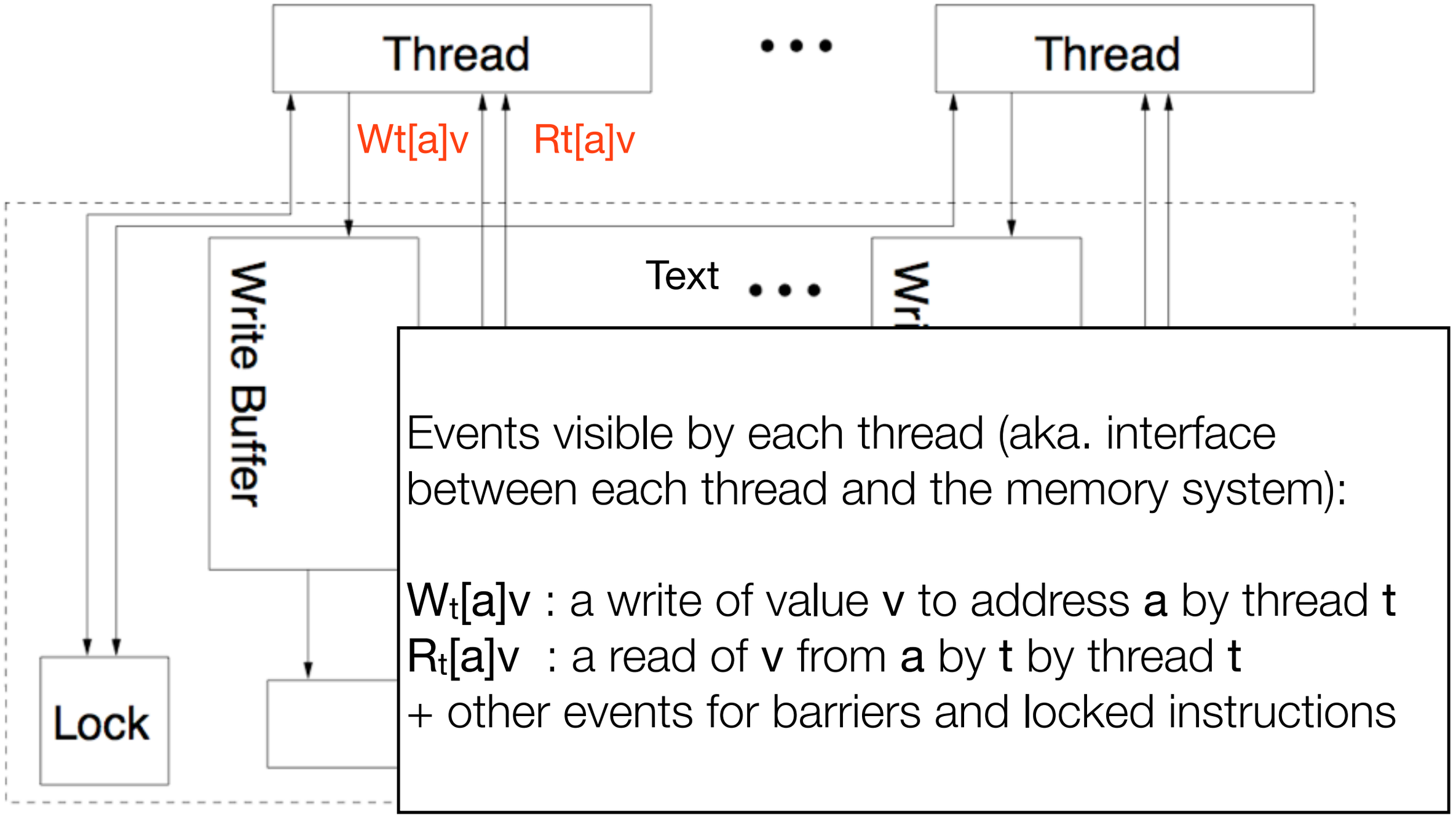
Take **M** to be a function from addresses to integers.

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

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

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

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

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

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

x86-tso abstract machine: selected transition rules

RB: Read from write buffer

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

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

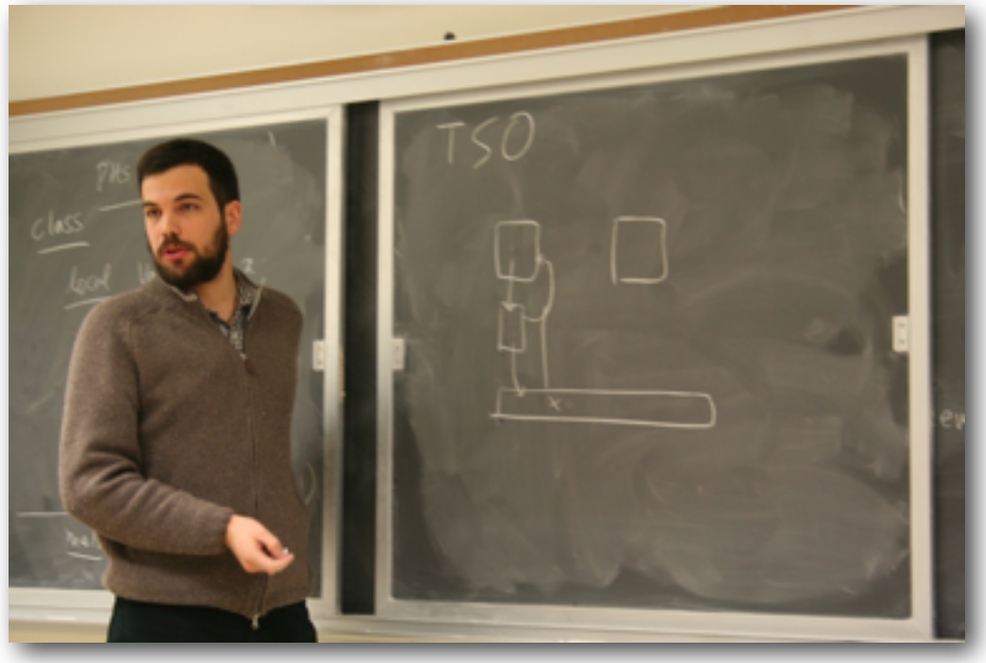
$$s \xrightarrow{\mathsf{W}_t\, x=v} s \oplus \langle\!\langle B := s.B \oplus (t \mapsto ([(x,v)] +\!\!+ s.B(t))) \rangle\!\rangle$$

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

WM: Write from write buffer to memory

$$\frac{\mathrm{not\_blocked}(s,t) \qquad s.B(t) = b +\!\!+ [(x,v)]}{s \xrightarrow{\tau_t\, x=v} s \oplus \langle\!\langle M := s.M \oplus (x \mapsto v) \rangle\!\rangle \oplus \langle\!\langle B := s.B \oplus (t \mapsto b) \rangle\!\rangle}$$

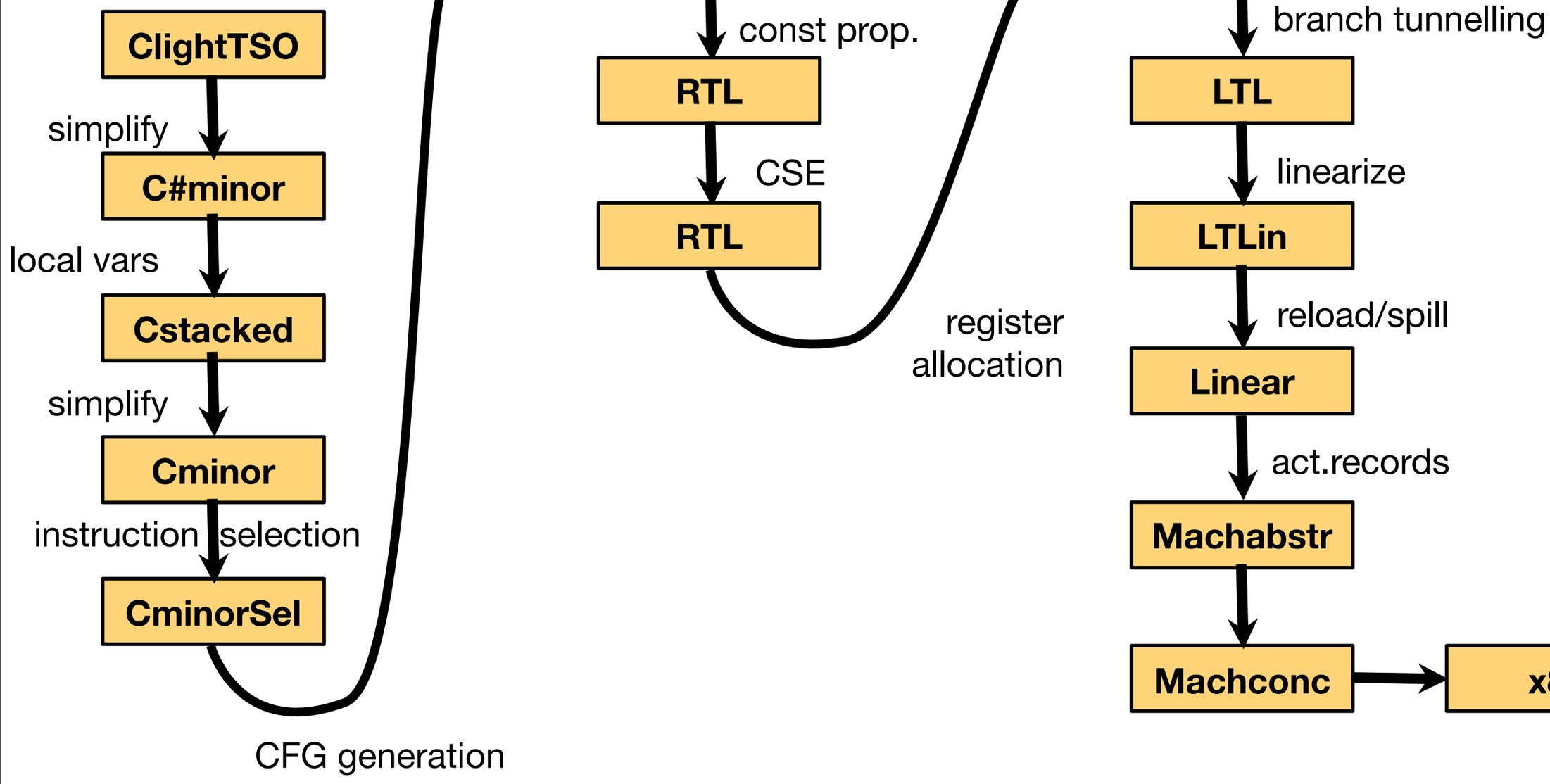
If t is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread



5. Veryfing fence elimination optimisations

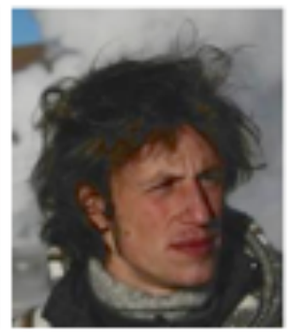
aka reasoning on the x86TSO operational memory model
and compiler correctness

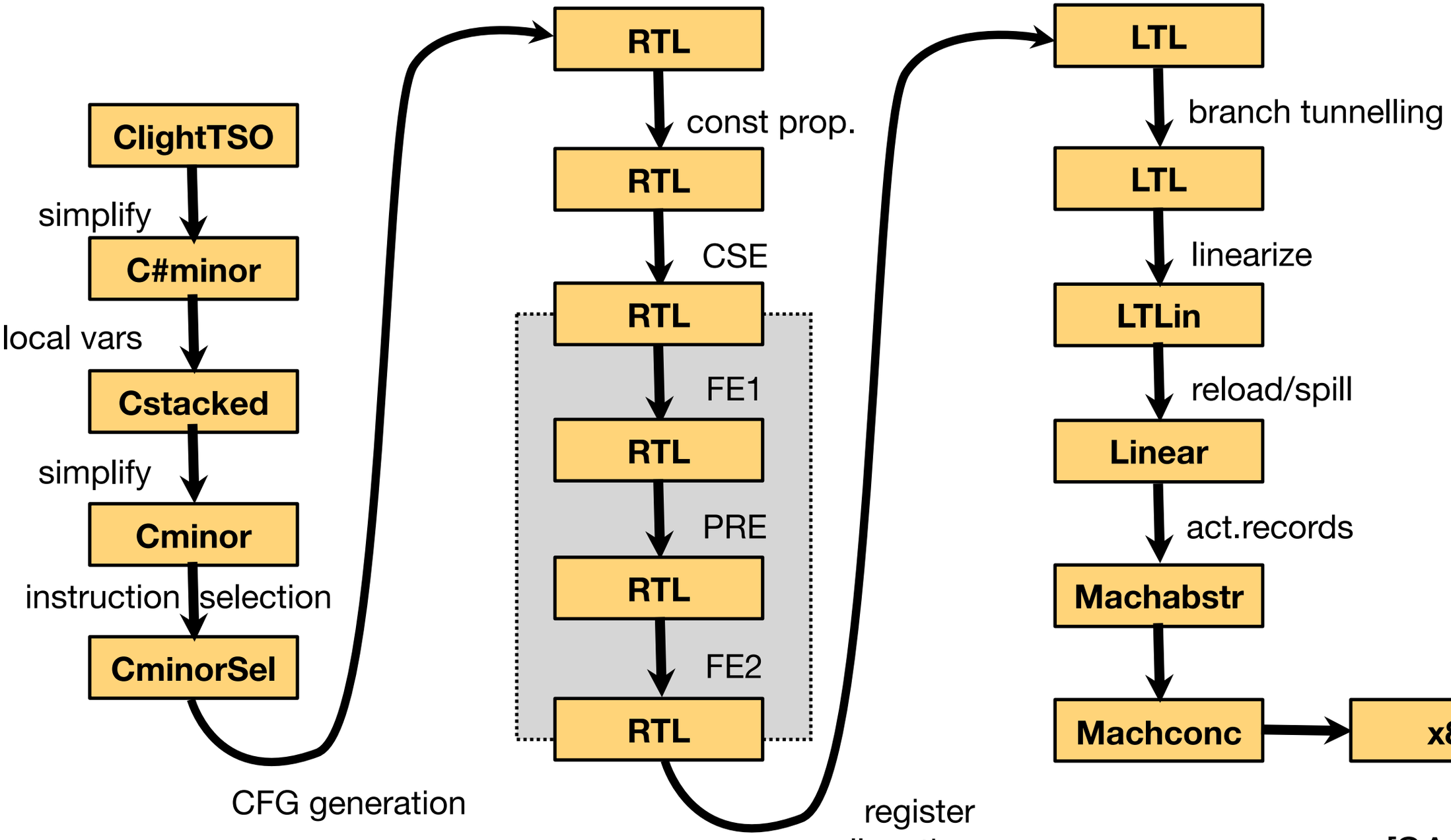
CompCertTSO



[POPL 2011]

CompCertTSO + fence optimisations



[SAS 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 `llvm` bugs

— "Specifiable": compiler correctness = preservation of behaviours

— Interesting: naturally higher-order, involve clever algorithms

— Big, but modular

Language semantics

The semantics of all the CompCertTSO languages is defined by:

– a type of programs, prg

– a type of states, $states$

– a set of initial states for each program, $\mathsf{init} \in prg \to \mathbb{P}(states)$

– a transition relation, $\to \in \mathbb{P}(states \times event \times states)$

`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

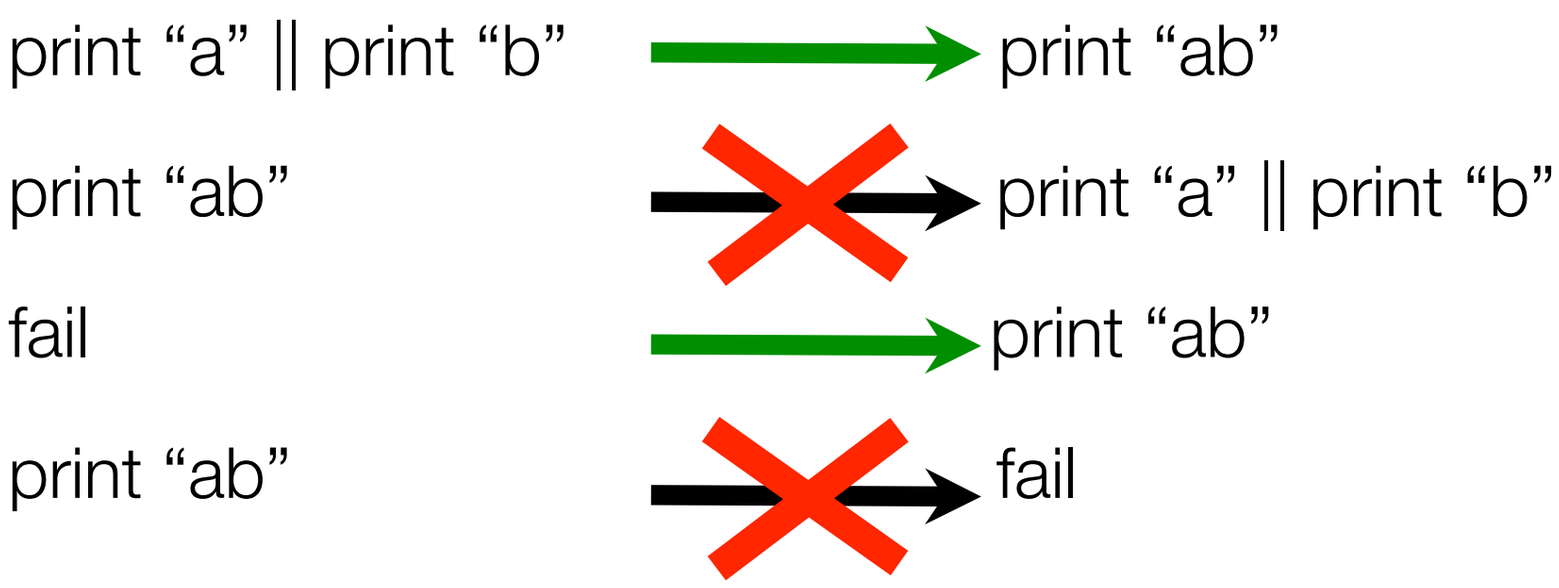
– *Infinite sequences* of call & return events;

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

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*

Redundant fences (1)

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

```
MFENCE
MFENCE
```

```
MFENCE
NOP
```

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

Generalisation:

```
MFENCE
NON-WRITE INSTR
…
NON-WRITE INSTR
MFENCE
```

```
MFENCE
NON-WRITE INSTR
…
NON-WRITE INSTR
NOP
```

FE1

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

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

Associate to each program point:

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

$\top$: otherwise.

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

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

FE1

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

A *forward* data-flow problem over the bo

Associ

⊥ : alo
is a
cur
no

⊤ : oth

Implementation:

1. Use CompCert implementation of Kildall algorithm to solve the data-flow equations.
2. Replace **`MFENCE`**s for which the analysis returns ⊥ with **`NOP`** instructions.

$$T_1(\mathbf{nop}, \mathcal{E}) = \mathcal{E}$$
$$= \mathcal{E}$$
$$= \mathcal{E}$$
$$= \top$$
$$= \top$$
$$= \top$$
$$= \bot$$
$$= \bot$$

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

Redundant fences (2)

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

```
MFENCE          ──────────►   NOP
MFENCE                        MFENCE
```

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                       …
MFENCE                              INSTRUCTION n
                                    MFENCE
```

Redundant fences (2)

If there are reads in between the fences…

[x]=[y]=0

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

EAX = EBX = 0
forbidden

but

[x]=[y]=0

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

EAX = EBX = 0
allowed

Redundant fences (2)

If there are reads in between the fences…

[x]=[y]=0

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

EAX = EBX = 0
forbidden

but

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

[x]=[y]=0

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

EAX = EBX = 0
allowed

Redundant fences (2)

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

`MFENCE; STORE` → **`STORE; MFENCE`**

1. 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:

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

$\top$: otherwise.

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

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

FE1 and FE2 are both useful

Removed by FE1 but not FE2:

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

Removed by FE2 but not FE1:

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

Informal correctness argument

Intuition: FE2 can be thought as iterating

```
MFENCE; STORE      →    STORE; MFENCE
MFENCE; non-mem    →    non-mem; MFENCE
```

and then applying

```
MFENCE; MFENCE     →    NOP; MFENCE
```

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

```
MFENCE;            →    NOP;
STORE;                  STORE;
WHILE(1);               WHILE(1);
MFENCE                  MFENCE
```

Basic simulations

A pair of relations

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

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

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

$$sim\_end : \forall s\, t.\ s \sim t \wedge t \not\to \_ \implies s \not\to \_$$

$$sim\_step : \forall s\, t\, t'\, ev.\ s \sim t \wedge t \xrightarrow{ev} t' \wedge ev \neq \mathsf{oom} \implies$$

$$(s \xrightarrow{\tau}{}^{*} \xrightarrow{\texttt{fail}} \_) \qquad \text{— } s \textit{ reaches a failure}$$

$$\vee\ (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s' \sim t') \qquad \text{— } s \textit{ does matching step sequence}$$

$$\vee\ (ev = \tau \wedge t > t' \wedge s \sim t')\,. \qquad \text{— } s \textit{ stutters (only allowed if } t > t')$$

Exhibiting a basic simulation implies:

traces(**compile**(*p*)) \ {*t* · **inftau** | *t* trace} ⊆ **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, $\mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$.

Simulation for FE2

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

$s \rightsquigarrow_i s'$ 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 $\exists s'$, $s \rightsquigarrow_i s'$ and $s' \equiv_i t$

"s is some instructions behind and can catch up"

Stutter condition:

$t > t'$ iff $t \rightarrow t'$ by a thread executing a `NOP`, `OP`, `STORE` or `MFENCE`

(and t's buffer being non-empty)

Simulation for FE2

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

$s \rightsquigarrow_i s'$ iff th

MFE

$s \sim t$ iff

– t's CFG

– s and t l

– ∀ thread

the analysis for i's pc returned $\bot$ and $\exists s'$, $s \rightsquigarrow_i s'$ and $s' \equiv_i t$

"s is some instructions behind and can catch up"

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

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

Stutter condition:

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

Simulation for FE2

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

$s \rightsquigarrow_i s'$ iff th

MFE

$s \sim t$ iff

– t's CFG

– s and t h

– ∀ thread

(i.

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

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

Solution 2: Do a case split.

— If this case does not arise, we are done.

— If it does, use a different (weaker) simulation to construct an infinite trace for the source

t

$\equiv_i t$

Stutter conditio

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

(and

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:*

$$sim\_weaken : \forall s, t.\ s \sim t \implies s \simeq t$$

$$sim\_wstep : \forall s\, t\, t'.\ s \simeq t \wedge t \xrightarrow{\tau} t' \wedge t > t' \implies$$

$$(s \xrightarrow{\tau}{}^{*} \xrightarrow{\texttt{fail}} \_) \qquad \text{— } s \textit{ reaches a failure}$$

$$\vee\ (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{\tau} s' \wedge s' \simeq t') \qquad \text{— } s \textit{ does a matching step sequence.}$$

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

Remarks:

— Once the simulation game moves from ~ to ≃, stuttering is forbidden;

— Can view difference between ~ and ≃ as a boolean prophecy variable.

Weaktau simulation for FE2

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

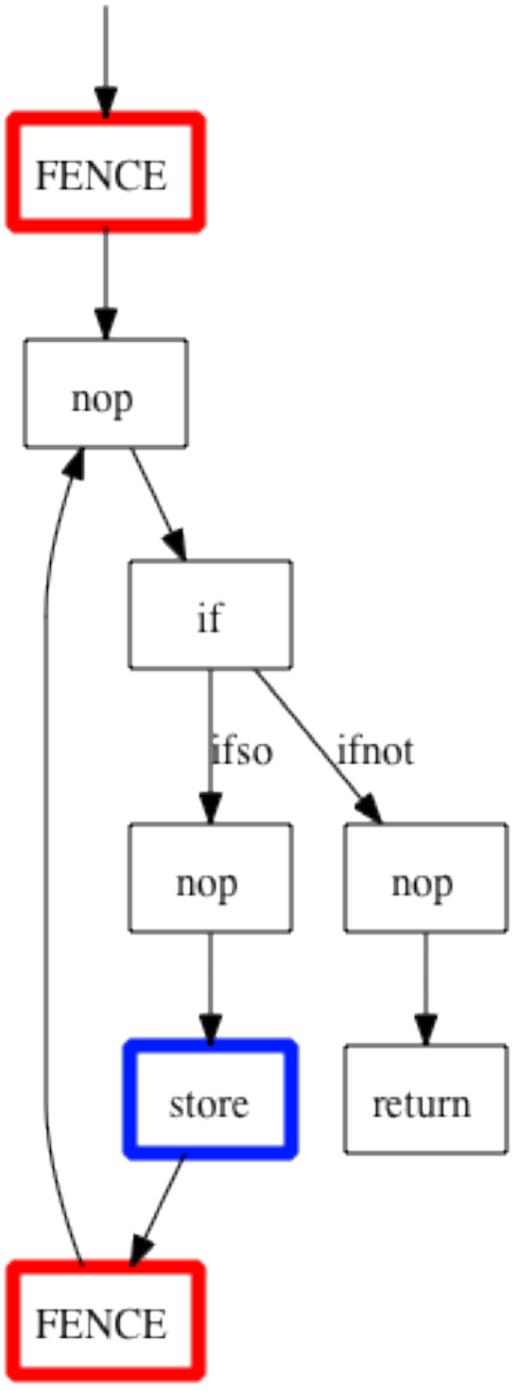
$s \simeq t$ iff

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

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

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

A closer look at the RTL

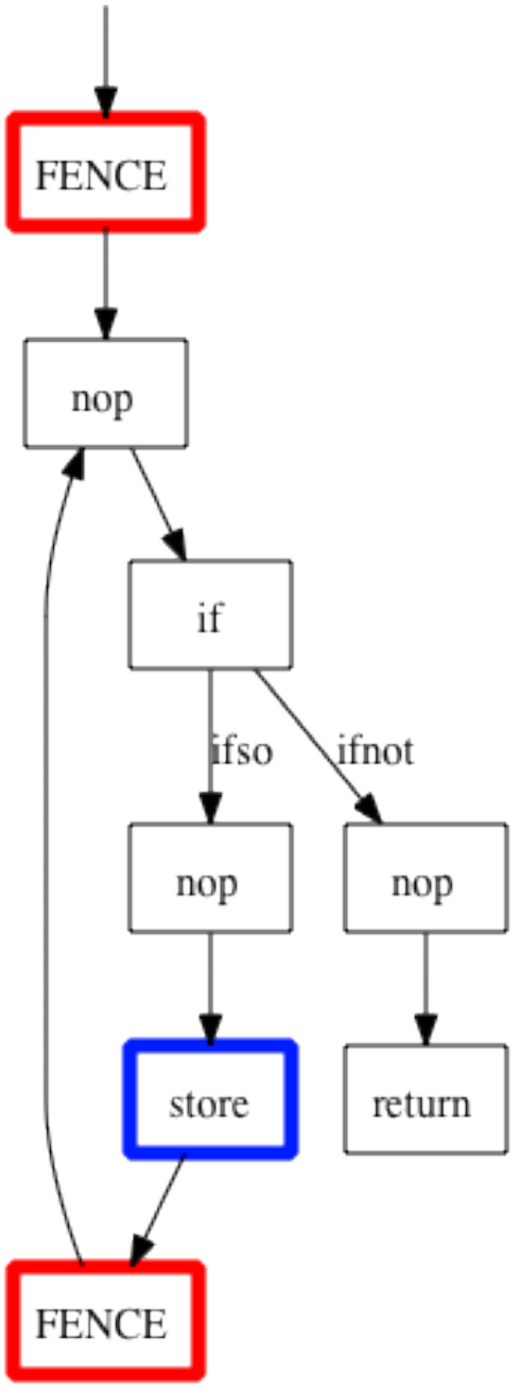


Patterns like that on the left are common.

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



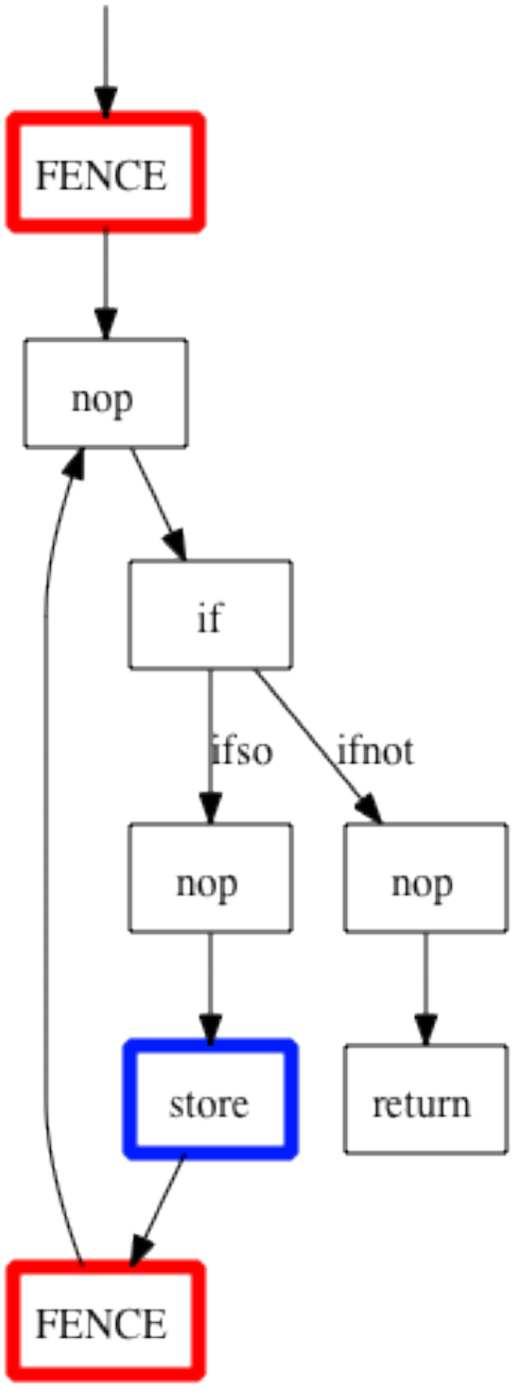
Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

It would be nice t[illegible] out of the loop.

Do you perform PRE?

A closer look at the RTL



Patterns like that on the left are common.

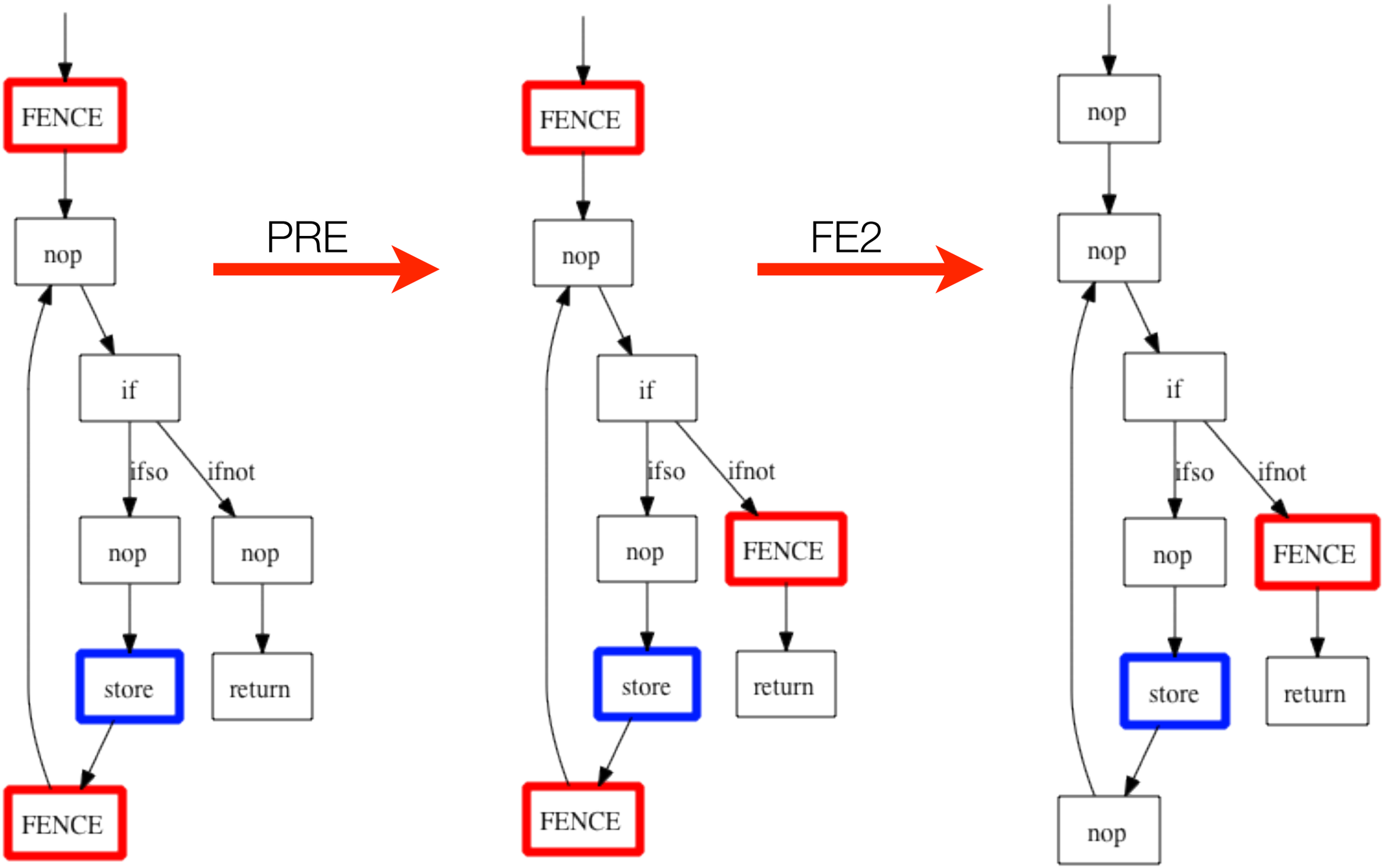
FE1 and FE2 do not optimise these patterns.

It would be nice t[illegible] out of the loop.

Do you perform PRE?

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

Partial redundancy elimination



Evaluation of the optimisations

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

– Cour

Important remark for your future work:

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

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

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`.

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`.

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

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`.

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

DRF programs must only exhibit sequentially consistent behaviours

C11/C++11 standard

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`.

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

DRF programs must only exhibit sequentially consistent behaviours

C11/C++11 standard

This program only prints 42.

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

gcc 4.7 -O2

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

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

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)
    ;
}
```

gcc 4.7 -O2

The outer loop can be (and is) optimised away

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

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, but...

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

- Read a (1) into edx

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

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

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

The compiled code saves and restores b

Correct in a sequential setting

Introduces unexpected behaviours in some concurrent context

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

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

The compiled code saves and restores b

Correct in a sequential setting

Introduces unexpected behaviours in some concurrent context

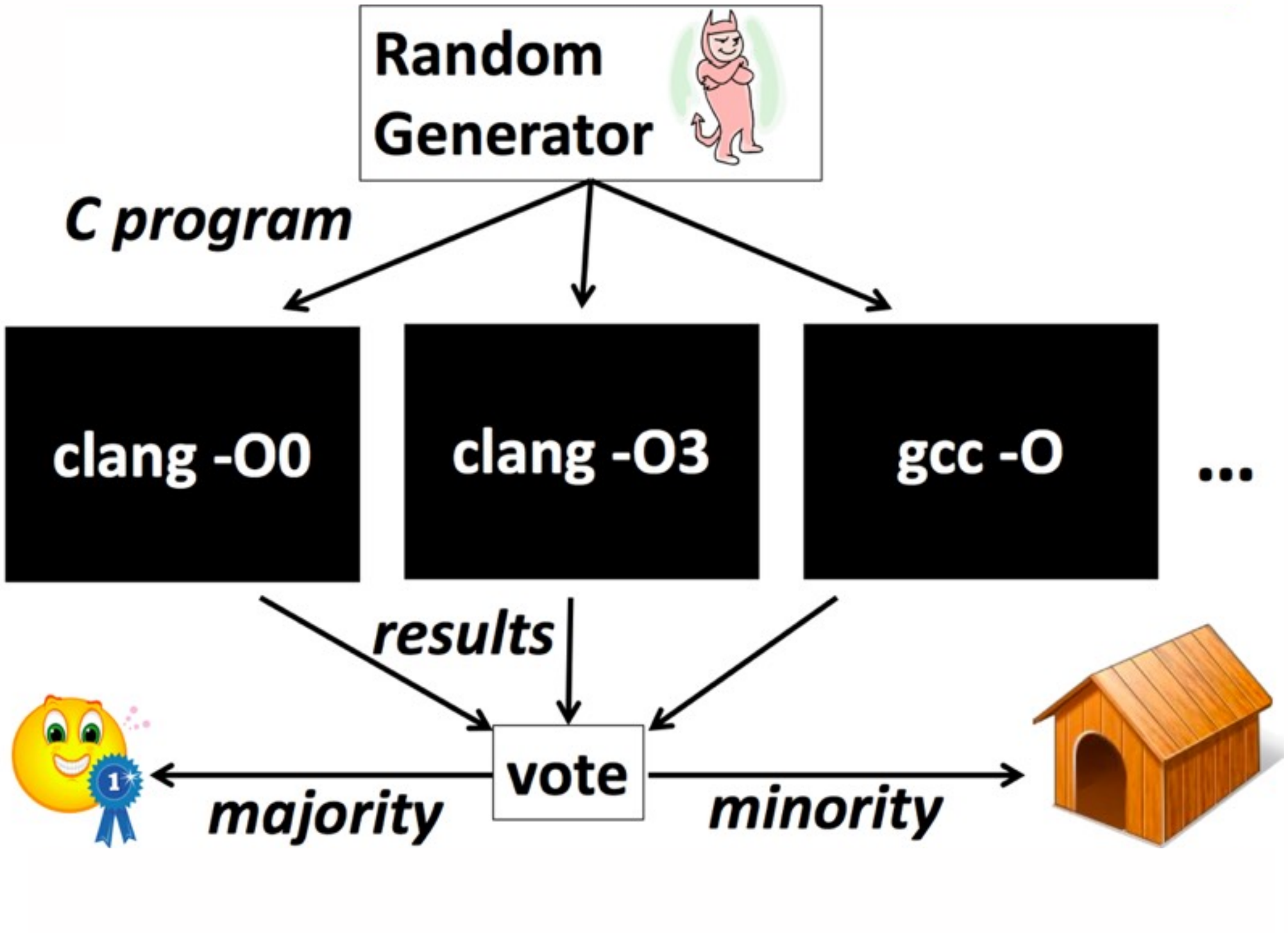
This is a *concurrency compiler bug*

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

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

Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

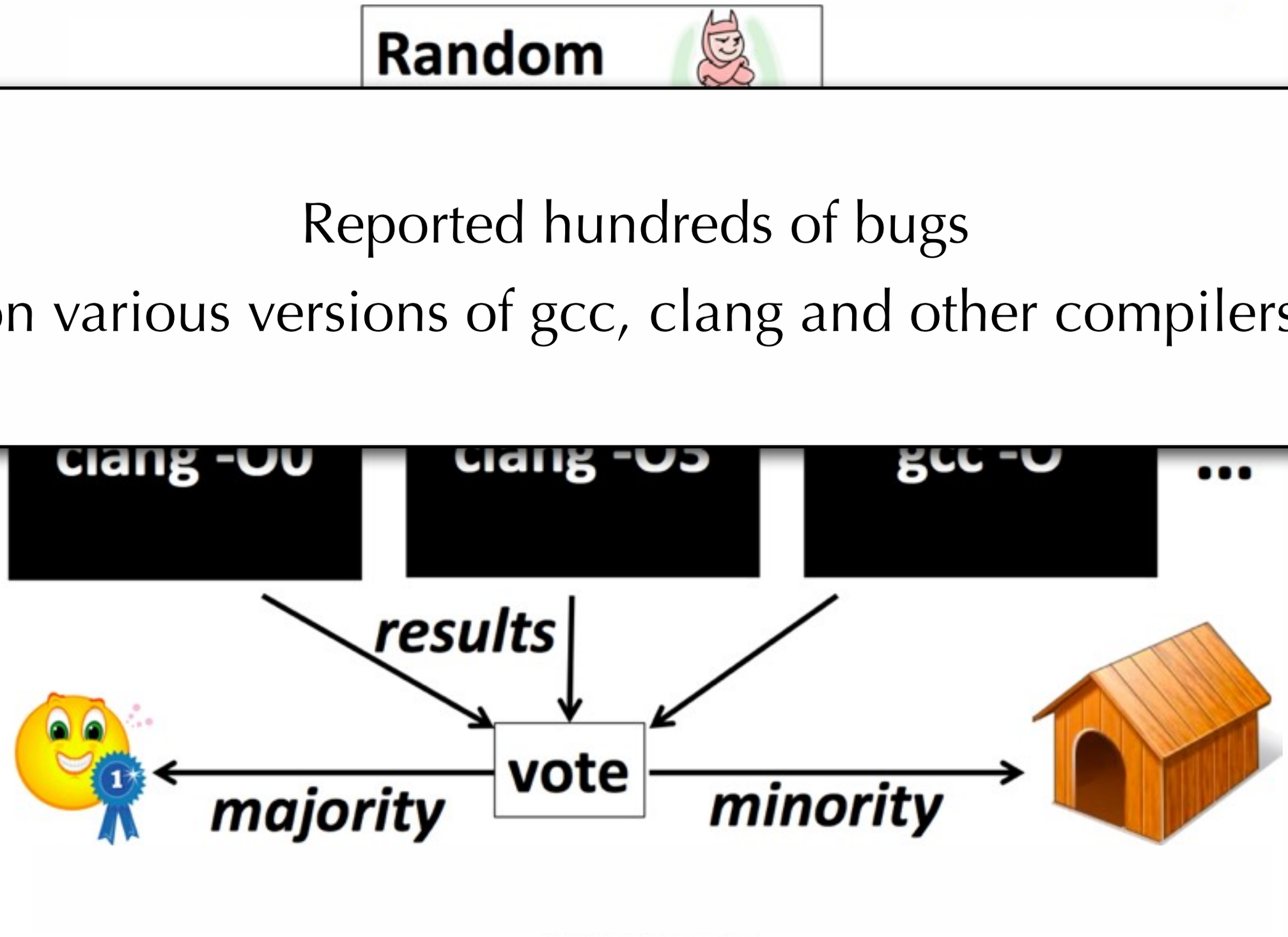


Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

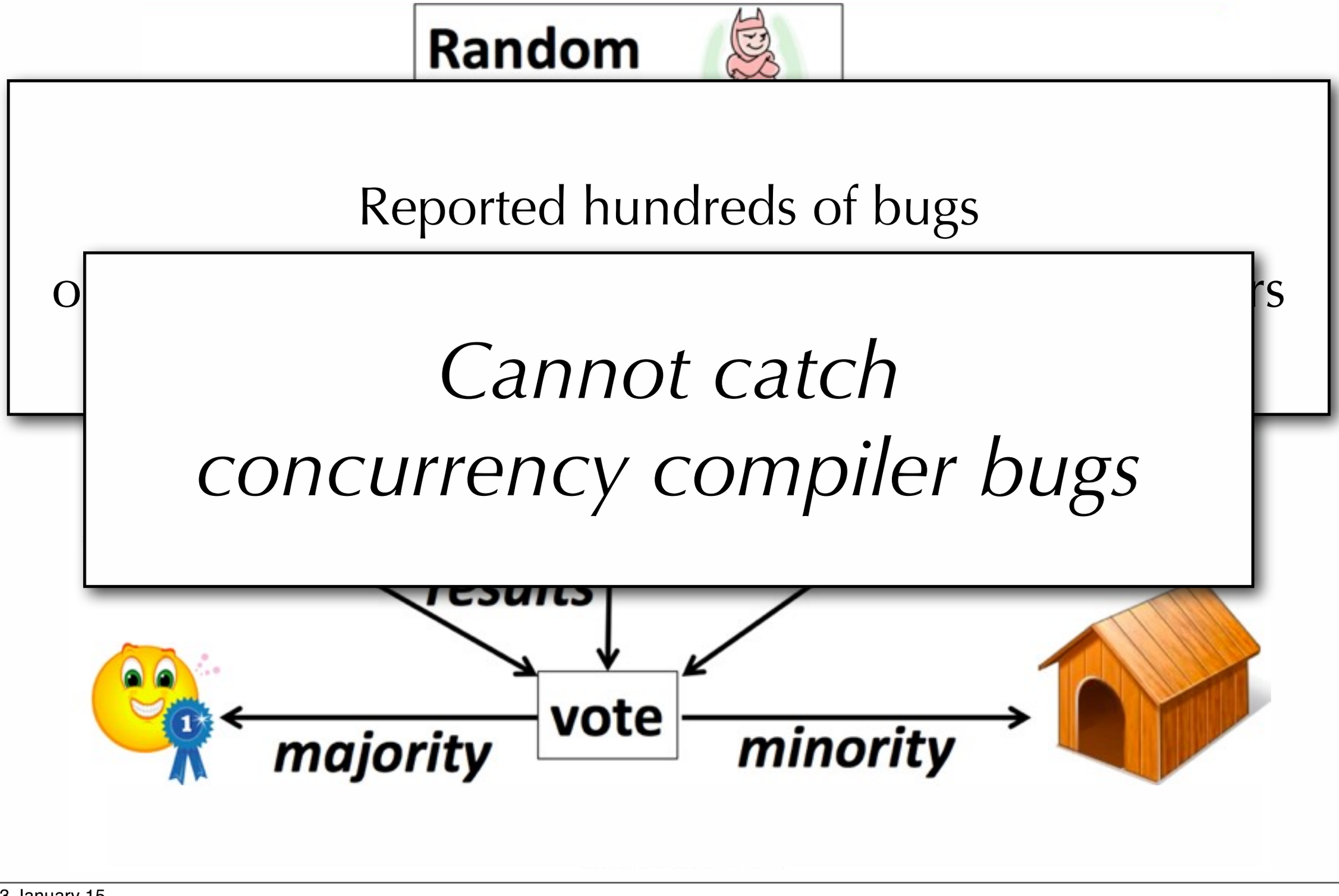
Reported hundreds of bugs

on various versions of gcc, clang and other compilers



Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



Hunting *concurrency* compiler bugs?

How to deal with non-determinism?

How to generate non-racy interesting programs?

How to capture all the behaviours of concurrent programs?

A compiler can optimise away behaviours:

how to test for correctness?

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

Idea

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

↓

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

Hunt concurrency compiler bugs

=

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

Random Generator

SEQUENTIAL PROGRAM

reference semantics

optimising compiler under test

EXECUTABLE

tracing

REFERENCE MEMORY TRACE

MEMORY TRACE

only transformations sound in any concurrent non-racy context?

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

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

reference semantics

```
Load a 1
```

gcc -O2 memory trace

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

Cannot match some events ⟶ detect compiler bug

```
;
}
```

reference semantics

```
Load a 1
```

gcc -O2 memory trace

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

Contributions

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

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

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

Interaction with GCC developers

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

Example: loop invariant code motion

Compiler Writer

Semanticist

Example: loop invariant code motion

Compiler Writer

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

Semanticist

Example: loop invariant code motion

Compiler Writer

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

Semanticist

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

Example: loop invariant code motion

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

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

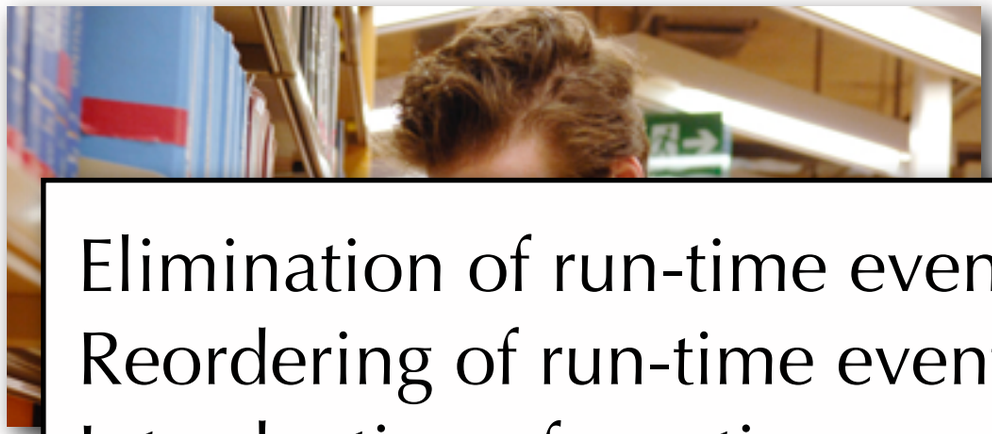
Example: loop invariant code motion

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist



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

Operations on sets of events

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

Example: loop invariant code motion

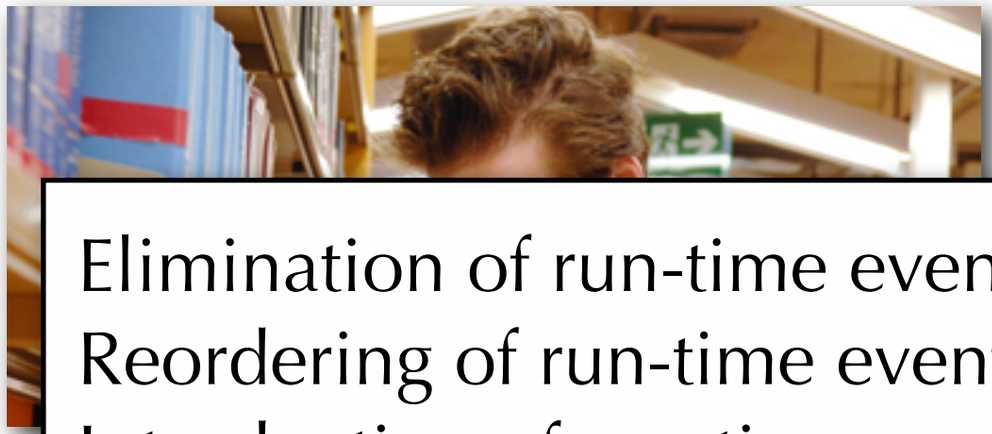
Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

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

Semanticist



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

Operations on sets of events

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

Example: loop invariant code motion

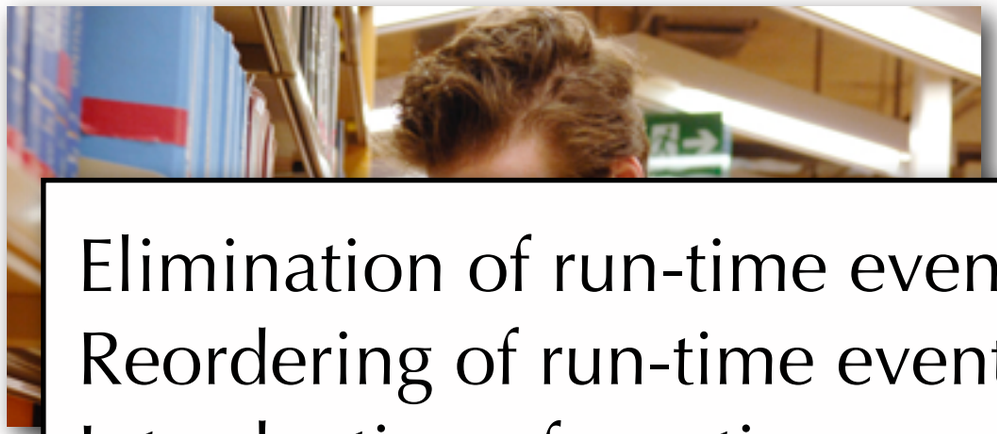
Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

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

Semanticist

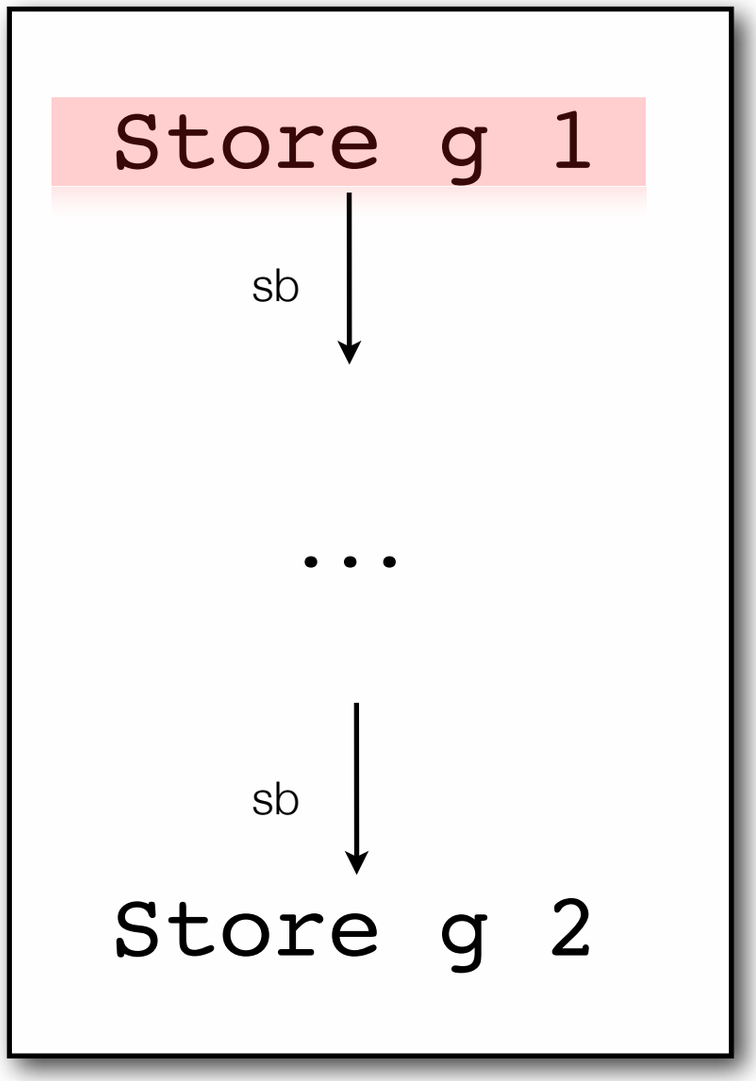


Elimination of run-time events
Reordering of run-time events
Introduction of run-time events

Operations on sets of events

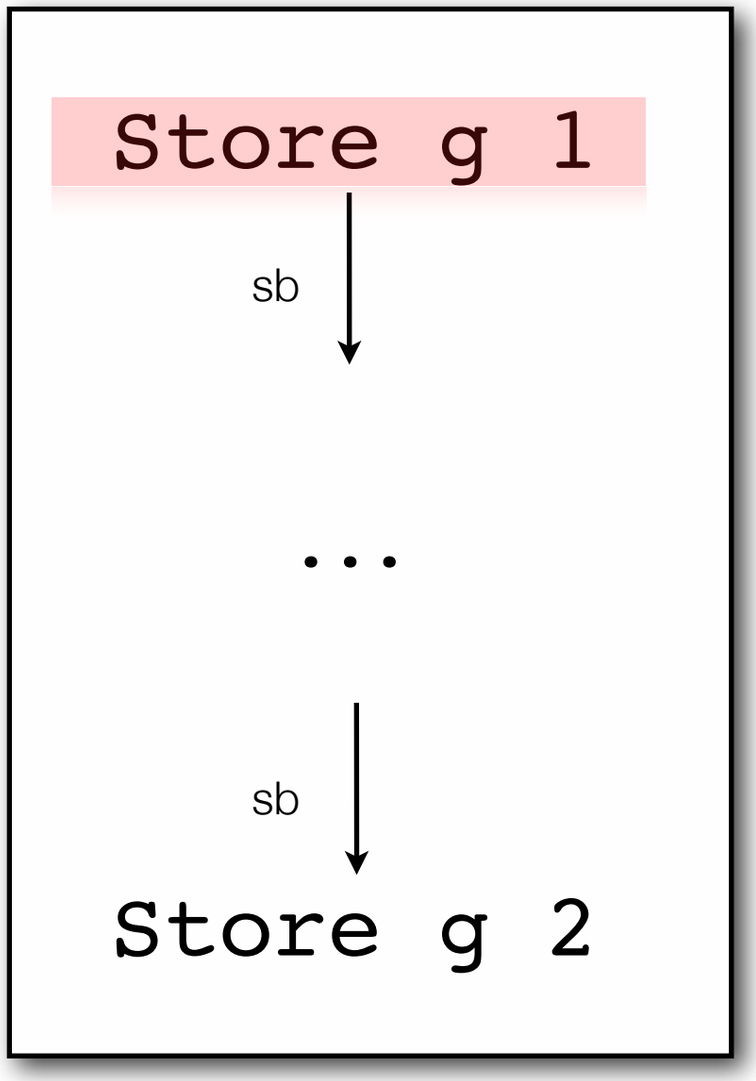
```
Load y 42
Store z 0

Store x[0] 43
Store z 1

Store x[1] 43
```

Elimination of *overwritten writes*



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

Elimination of *overwritten writes*



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

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

The C11/C++11 memory model

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

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

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

- races allowed
- attributes on accesses specify their semantics:

`MO_SEQ_CST` `MO_RELEASE/MO_ACQUIRE` `MO_RELAXED`

MO_RELEASE / MO_ACQUIRE

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

Thread 1

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

Thread 2

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

MO_RELEASE / MO_ACQUIRE

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

Thread 1

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

Thread 2

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

MO_RELEASE / MO_ACQUIRE

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

Thread 1

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

Thread 2

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

MO_RELEASE / MO_ACQUIRE

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

Thread 1

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

Thread 2

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

MO_RELEASE / MO_ACQUIRE

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

Thread 1

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

sync

Thread 2

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

MO_RELEASE / MO_ACQUIRE

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

Thread 1

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

Thread 2

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

The release/acquire synchronisation guarantees that:

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

Remark: unlock ≃ release, lock ≃ acquire.

Same-thread release/acquire pairs

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

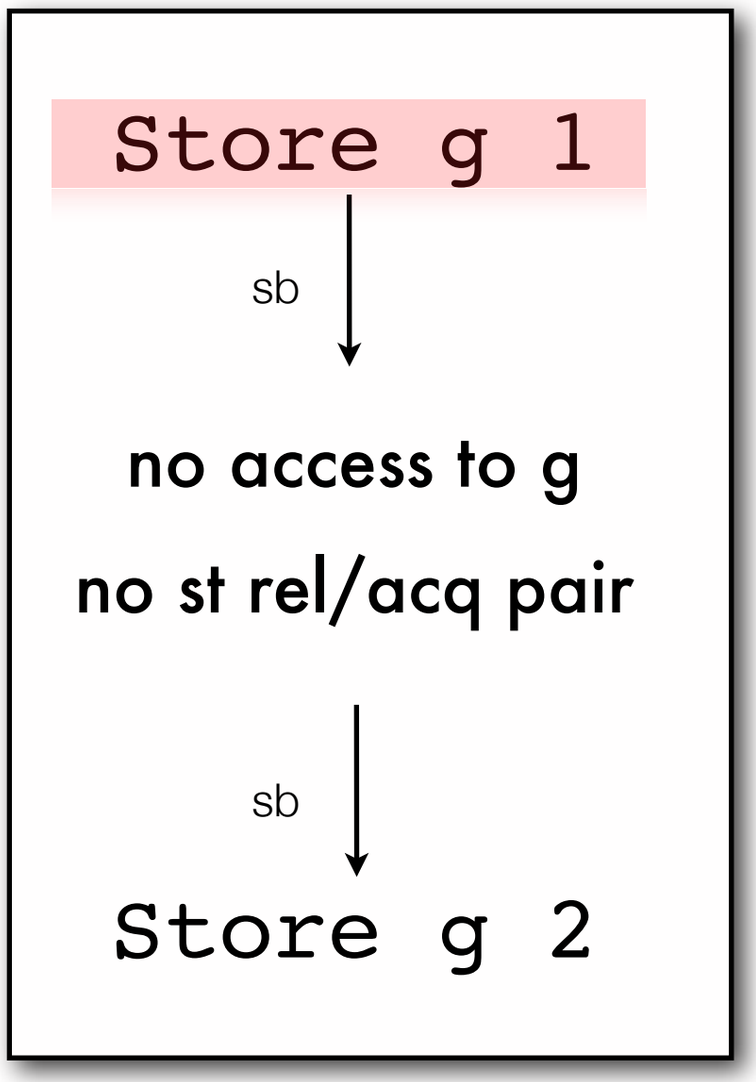
An action is a *release* if it is a possible source of a synchronisation

unlock mutex, release or seq_cst atomic write

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

lock mutex, acquire or seq_cst atomic read

Elimination of *overwritten writes*



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

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

The soundness condition

Shared memory

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

Thread 1

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

The soundness condition

Shared memory

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

Thread 1

candidate overwritten write

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

The soundness condition

Shared memory

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

Thread 1

candidate overwritten write

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

same-thread release-acquire pair

The soundness condition

Shared memory

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

Thread 1

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

Thread 2

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

The soundness condition

Shared memory

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

Thread 1

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

Thread 2

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

Thread 2 is non-racy

The soundness condition

Shared memory

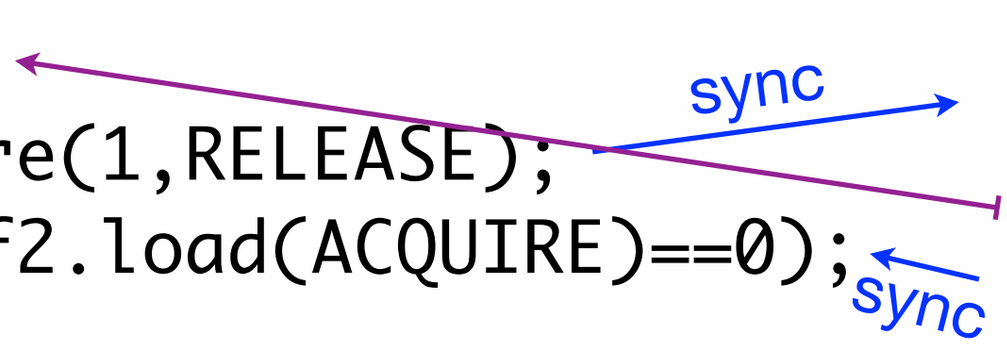
```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

Thread 2

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



Thread 2 is non-racy

The program should only print 1

The soundness condition

Shared memory

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

Thread 1

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

Thread 2

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

sync

sync

Thread 2 is non-racy

The program should only print 1

If we perform overwritten write elimination it prints 0

The soundness condition

Shared memory

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

Thread 1

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

sync →

Thread 2

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

The soundness condition

Shared memory

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

Thread 1

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

g = 2;
```

sync →

Thread 2

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

The soundness condition

Shared memory

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

Thread 1

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

g = 2;
```

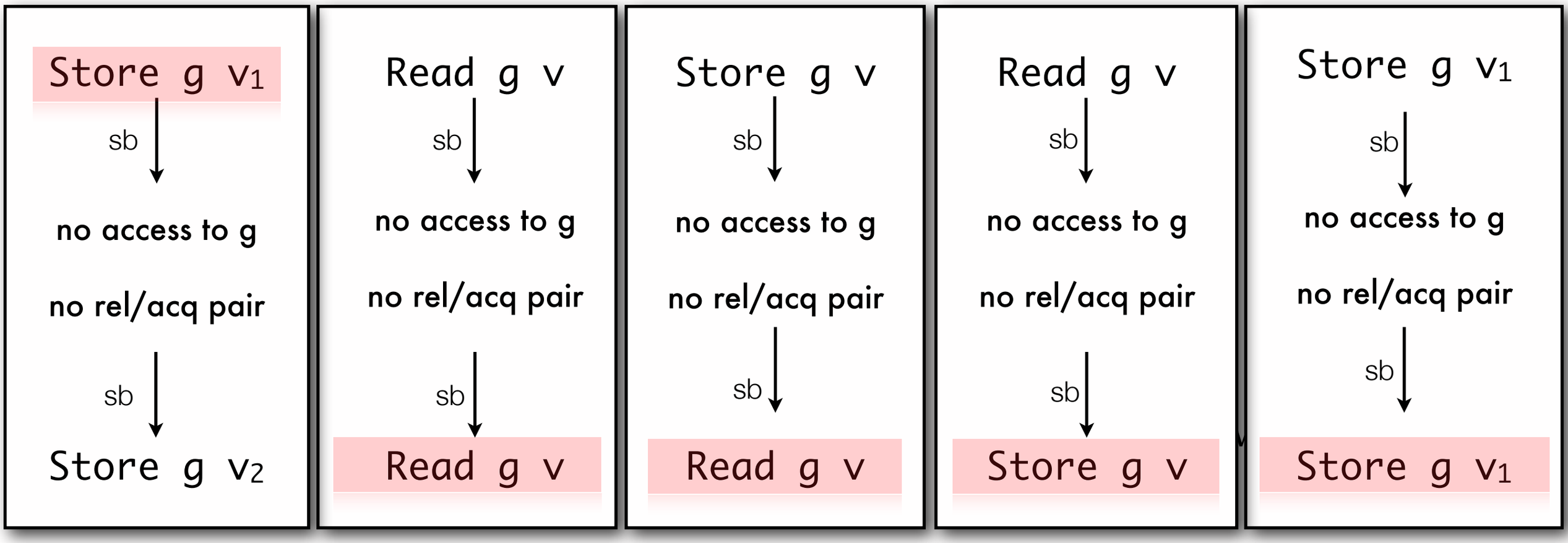
Thread 2

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

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

It is sound to optimise the overwritten write.

Eliminations: bestiary



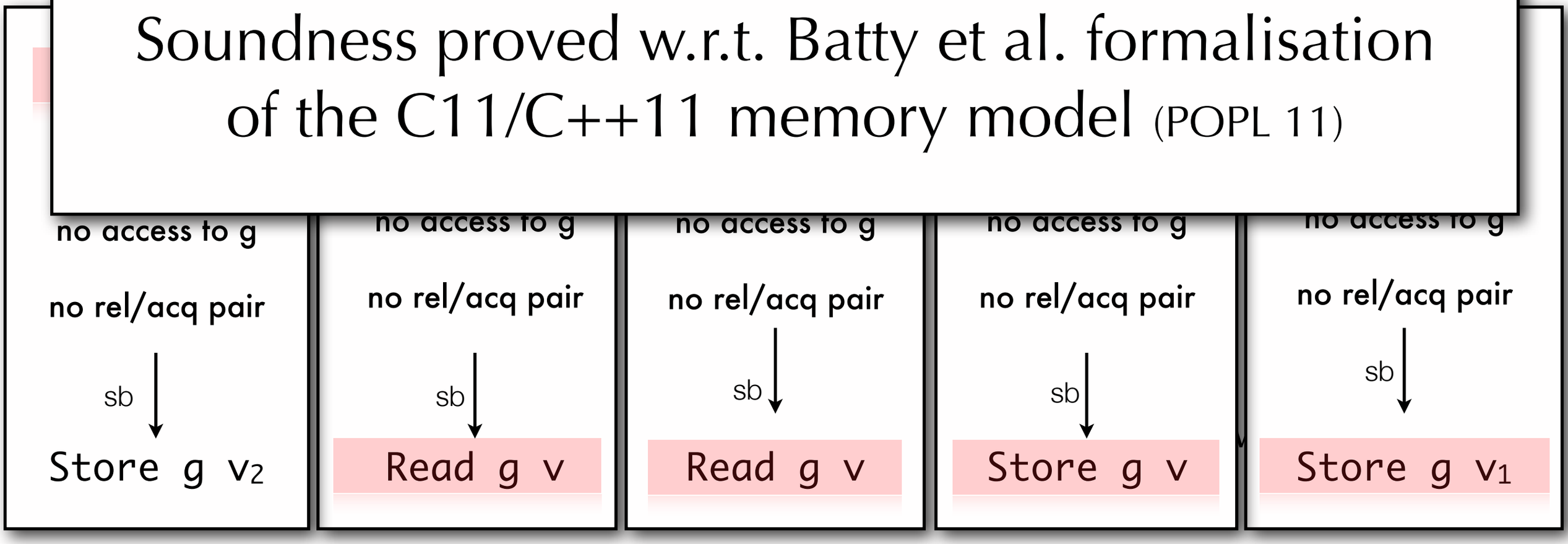
Overwritten-Write Read-after-Read Read-after-Write Write-after-Read Write-after-Write

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

Eliminations: bestiary

Theorem

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



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

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

Reorderings and introductions

Correctness criterion for reordering events:

— different addresses

— no synchronisations in-between

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

Read introductions observed in practice (gcc, clang).

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

The CMMTEST Tool

Random Generator

SEQUENTIAL PROGRAM

optimising compiler under test

reference semantics

EXECUTABLE

tracing

REFERENCE MEMORY TRACE

MEMORY TRACE

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

CSmith
extended with locks
and atomics

SEQUENTIAL PROGRAM

optimising compiler under test

reference semantics

EXECUTABLE

tracing

REFERENCE MEMORY TRACE

MEMORY TRACE

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

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

optimising
compiler
under test

reference
semantics

EXECUTABLE

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

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

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

gcc/clang -O0

*optimising
compiler
under test*

EXECUTABLE

EXECUTABLE

*binary
instrumentation*

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

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

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

gcc/clang -O0

optimising
compiler
under test

EXECUTABLE

EXECUTABLE

binary
instrumentation

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

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

CSmith
extended with locks
and atomics

→ SEQUENTIAL PROGRAM

optimising compiler

Subtleties:

- dependencies between eliminable events

- some optimisations (e.g. merging of accesses) cannot be expressed in the C11/C++11 formalisation

- the tool also ensures that the compilation of atomic accesses is preserved by the optimiser

ir

n

TRACE

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

Interaction with GCC developers

1. Some GCC bugs

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

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

All promptly fixed.

Remark: these bugs break the Posix thread model too.

2. Checking compiler invariants

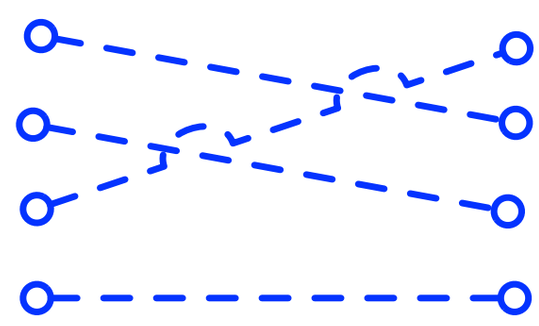
GCC internal invariant: never reorder with an atomic access

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

...not a bug, but fixed anyway

```
atomic_uint a;
int32_t g1, g2;

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

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



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

3. Detecting unexpected behaviours

```
                    uint16_t g

for (; g==0; g--);   ⟶   g=0;
```

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

```
Load  g  0   )——?——(   Store  g  0
```

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

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

Conclusion

Syllabus

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

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

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

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

The memory models of modern hardware are better understood.

Programming languages attempt to specify and implement reasonable memory models.

Researchers and programmers are now interested in these problems.

The memory models of modern hardware are better understood.

Still, many open problems...

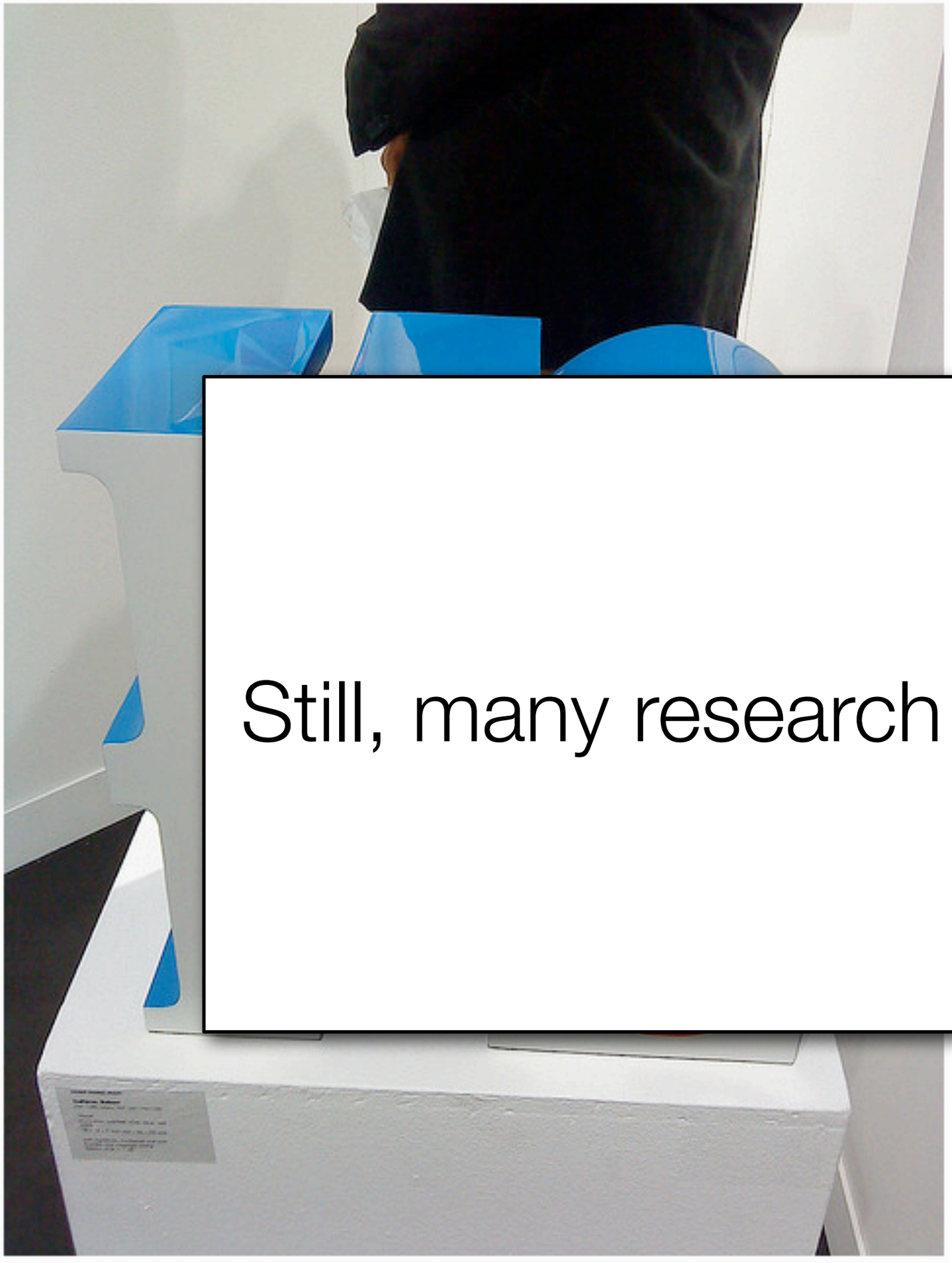
attempt

els.

mmers

se

problems.



The memory models of modern hardware are better understood.

Still, many research opportunities!

attempt

els.

mmers

se

problems.

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

And thank you all for attending these lectures!

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