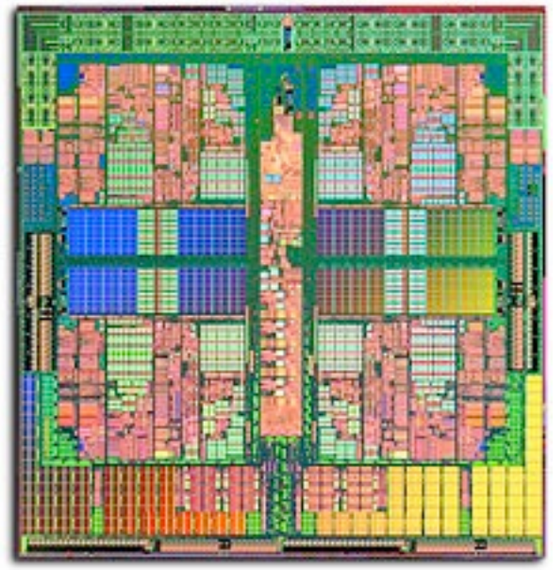


Semantics, languages and algorithms for multicore programming

Albert Cohen

Luc Maranget

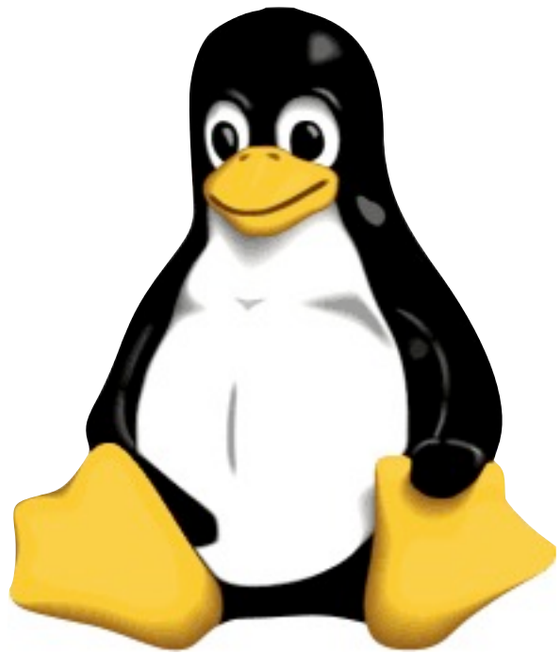
Francesco Zappa Nardelli

Vote: topics for my this lecture

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

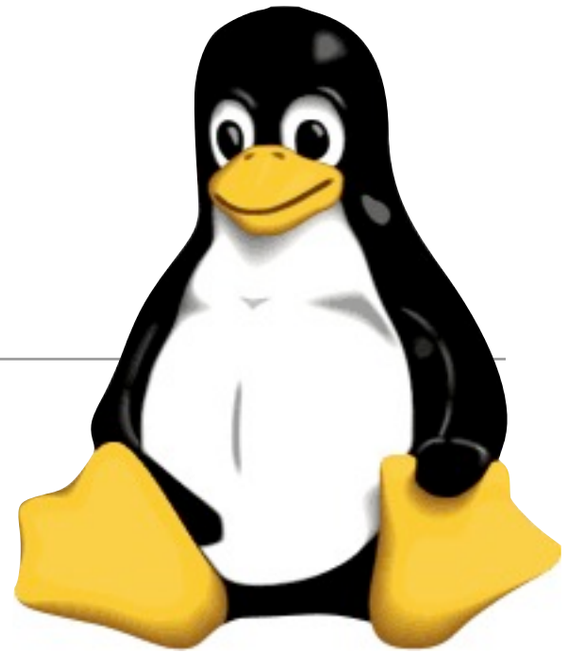
Vote: topics for my this lecture

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



1. The Linux memory model (ahem, kinda)

The Linux memory model



Facts:

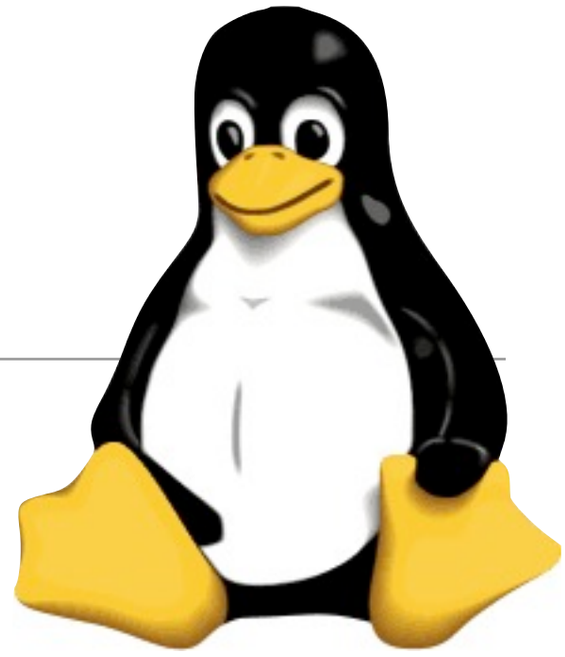
- abstraction layer over hardware and compilers

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

- documented by a text file:

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

The Linux memory model



Facts:

- abstraction layer over hardware and compilers

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

- documented by a text file:

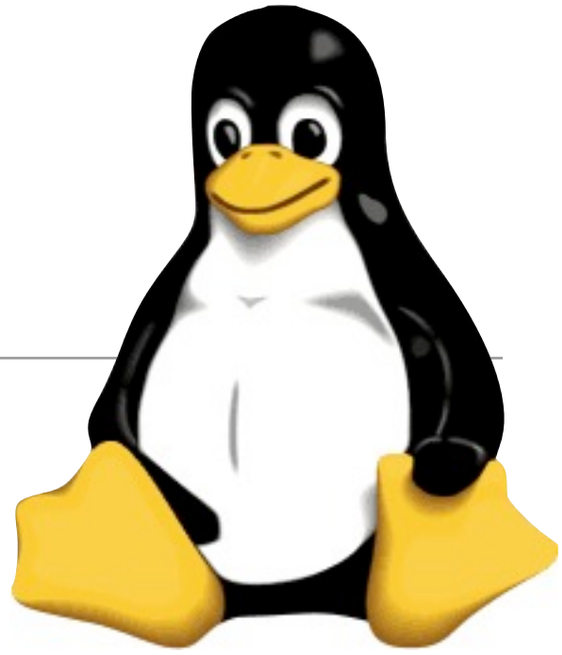
`http://www.kernel.org/doc/Documentation/memory-barriers.txt`

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

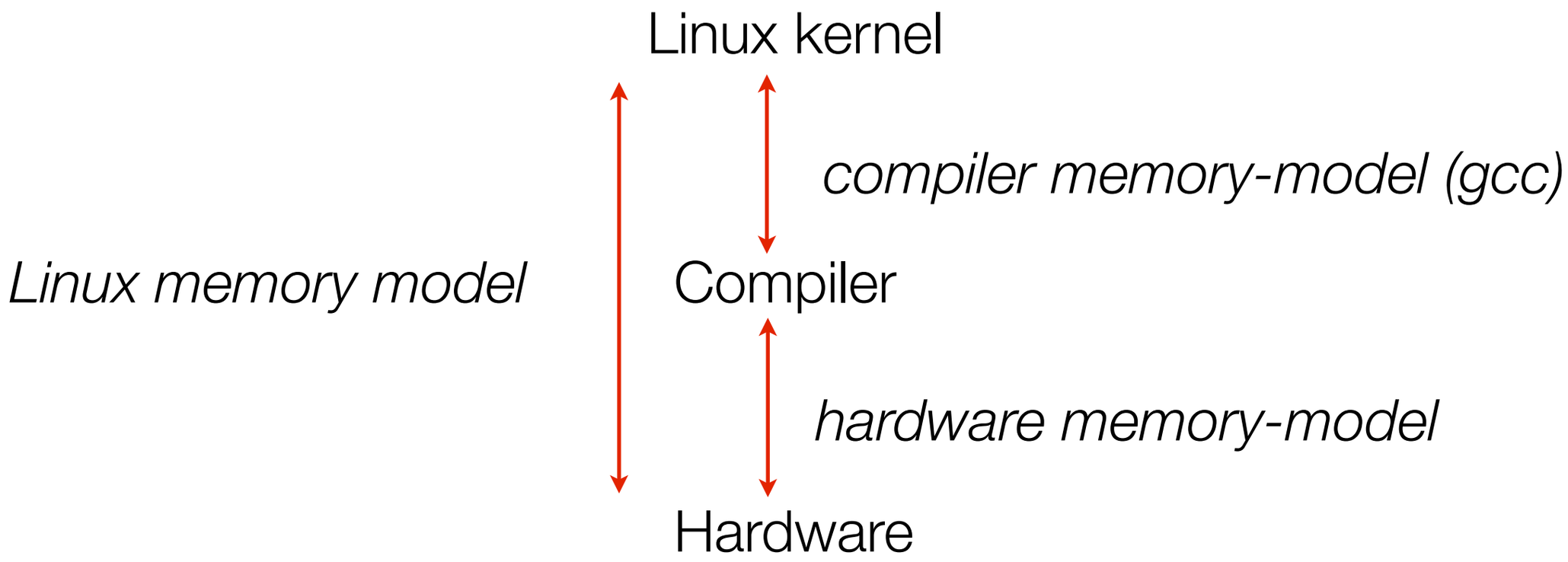
I attempted to understand the doc, and exchanged a few email with Paul Mc Kenney. However I don't understand much...

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

The Linux memory model



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



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

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

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

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

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

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

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

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

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

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

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

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

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

parisc: TSO, similar to x86.

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

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

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

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

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

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

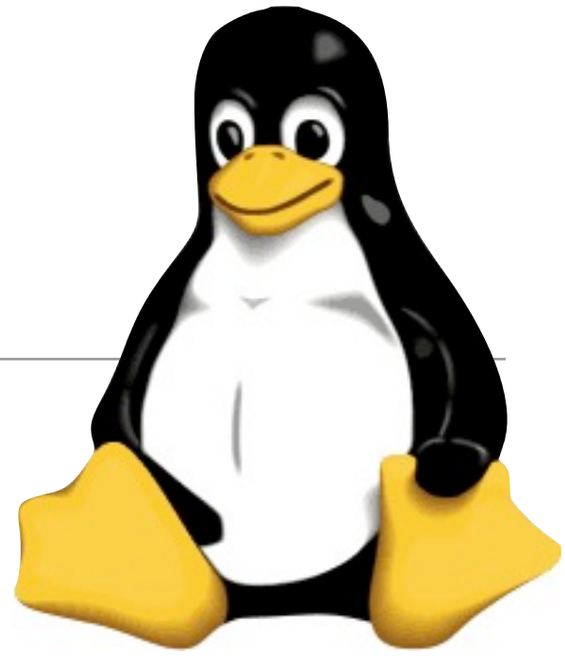
um: Looks like an x86 knockoff judging by the system.h file.

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

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

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

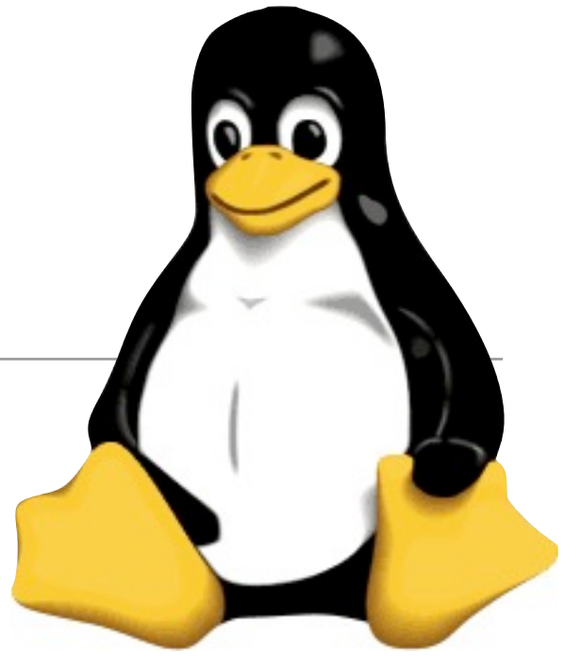
The Linux memory model



My intuition:

Annoying facts:

The Linux memory model



My intuition:

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

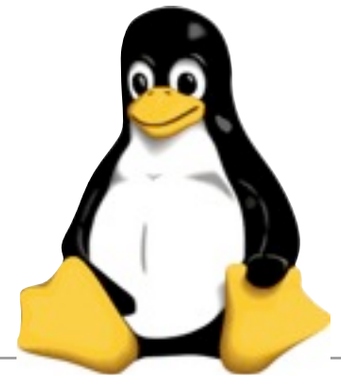
Annoying facts:

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

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

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

The Linux memory model: macros



on x86:

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

as far as we know, lfence and sfence are noop in x86TSO

on Power:

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

So I still stick with my earlier statements:

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

Question:

Initially x=
Thread 0:
Thread 1:
Thread 2:

to be forbi

And a few emails exchanged last year...

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

So I still stick with my earlier statements:

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

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

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

to be forbidden or not?

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

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

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

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

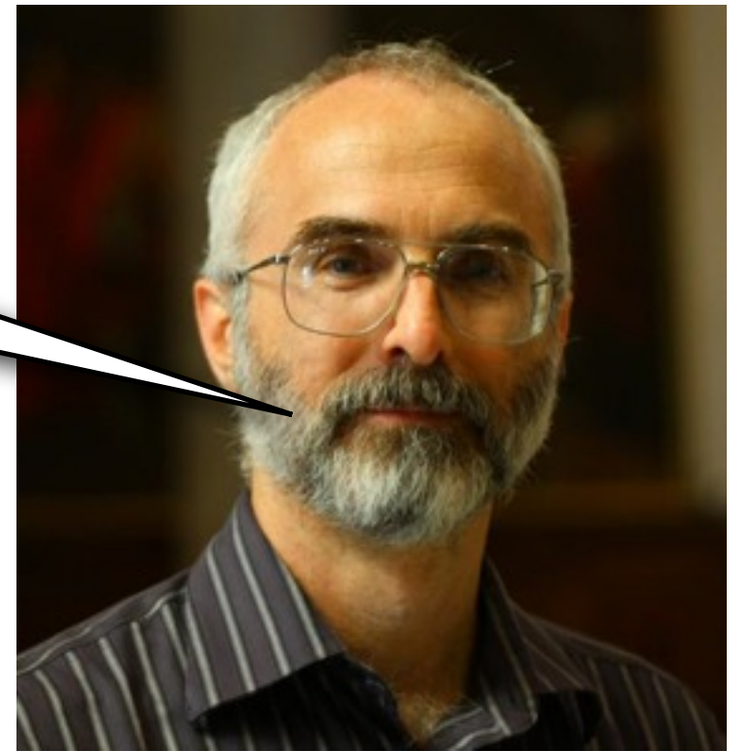
If this is the case, I suspect that the Linux kernel has a few possible failure modes when running on Itanium hardware. Which it might well have...

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

Internship proposal:

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.



Pros:

Challenging!

Can have a huge impact!

Collaboration with Paul Mc Kenney possible!

2. Load Reserve / Store Conditional

RISC-friendly synchronisation operations

Load-reserve/Store-conditional

(aka LL/SC, larx/stcx and lwarx/stwcx, LDREX/STREX).

- can be used to implement CAS, atomic add, spinlocks, ...

- universal (like CAS) [Herlihy'93] (but no ABA problem)

Informally, stwcx succeeds only if no other write to the same address since last lwarx, setting a flag iff it succeeds.

Atomic Addition
loop: lwarx r, x; add r,3,r; stwcx r, x; bne loop;

What is *no write since*...

Informally, stwcx succeeds only if no other write to the same address since last lwarx, setting a flag iff it succeeds.

- In machine time?

  (neither necessary, nor sufficient)

- Microarchitecturally: if cache-line ownership not lost since last lwarx

  (simplified, and we don't want to model the microarchitecture)

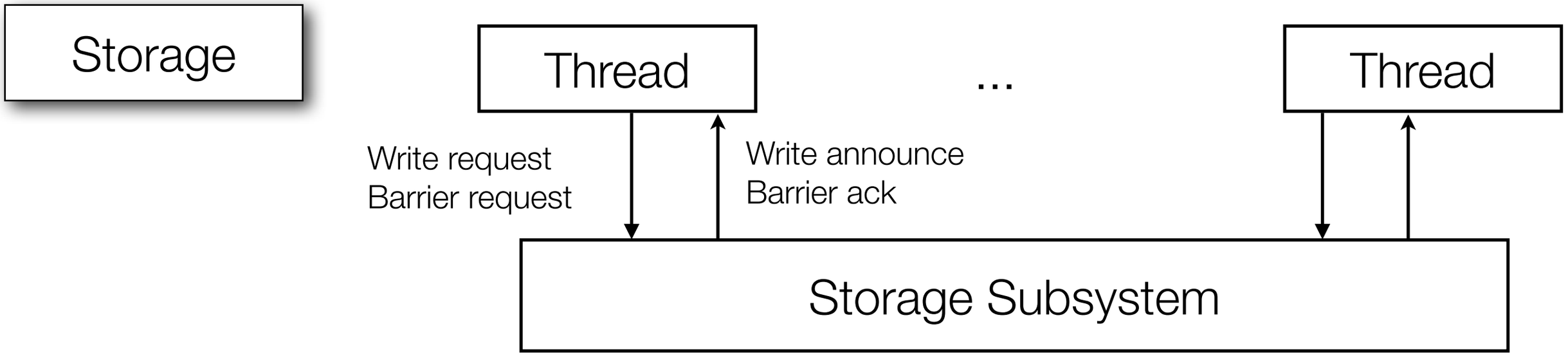
Modeling *no lost since*

- Abstractly: ownership chain modeled by building up coherence order
- Coherence: order relating stores to the same location (eventually linear)

> A stwcx succeeds only if it is (or at least, if it can become)
> *coherence-next-to* the write read from by lwarx,
> and no other write can later come in between.

Isolate key concept: *write reaching coherence point*

coherence is linear below this write, and no new edges will be added



The storage keeps: ...

2. for each location, a partial order of *coherence commitments*

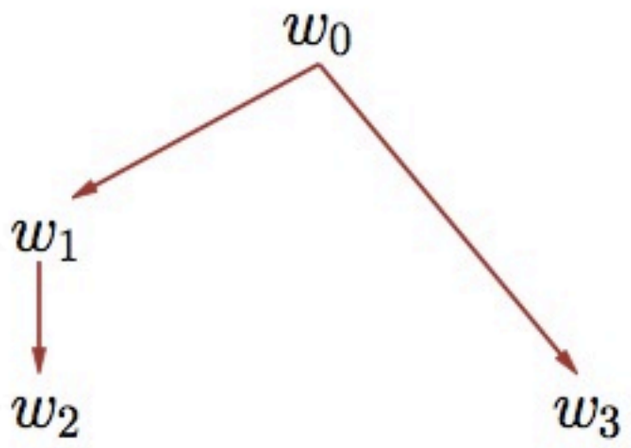
Idea 1: at the end of the execution, writes to each location are totally ordered.

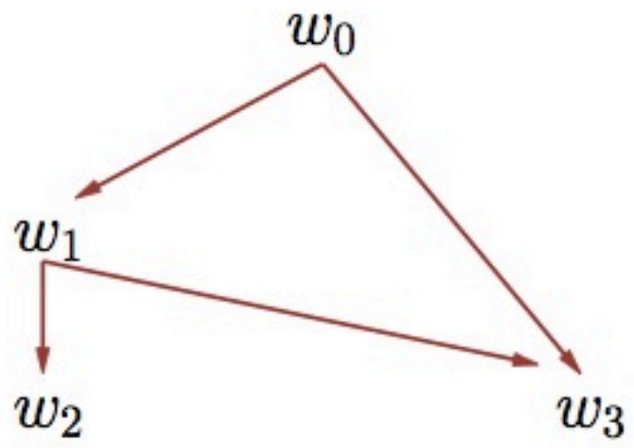
Idea 2: during computation, reads and propagation of writes must respect the coherence order (*reduce non-determism of previous rules).*

Intuition: if a thread executes `x=1` and then `x=2`, another thread cannot first read `2` and then `1`.

Coherence by Fiat

Suppose the storage subsystem has seen 4 writes to x:





Suppose just $[w_1]$ has propagated to tid and then tid reads x.

- it cannot be sent w_0, as w_0 is coherence-before the w_1 write that (because it is in the writes-propagated list) it might have read from;
- it could re-read from w_1, leaving the coherence constraint unchanged;
- it could be sent w_2, again leaving the coherence constraint unchanged, in which case w_2 must be appended to the events propagated to *tid*; or
- it could be sent w_3, again appending this to the events propagated to *tid*, which moreover entails committing to w_3 being coherence-after w_1, as in the coherence constraint on the right above. Note that this still leaves the relative order of w_2 and w_3 unconstrained, so another thread could be sent w_2 then w_3 or (in a different run) the other way around (or indeed just one, or neither).

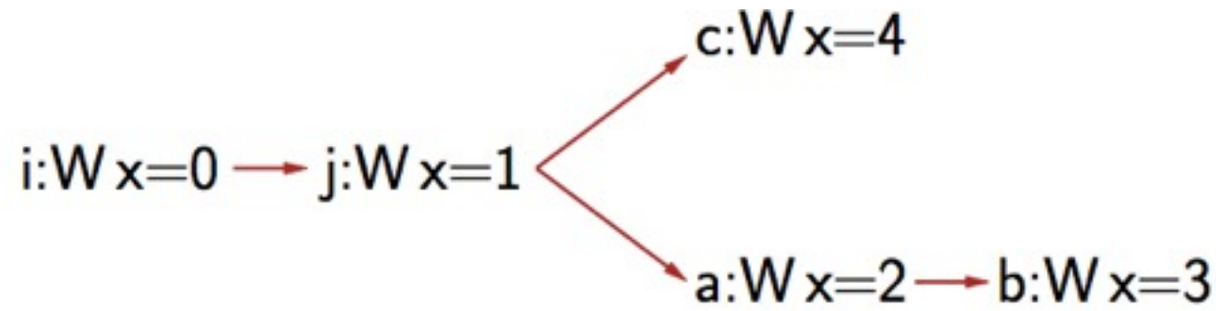
– p

Coherence points and a successful stwcx

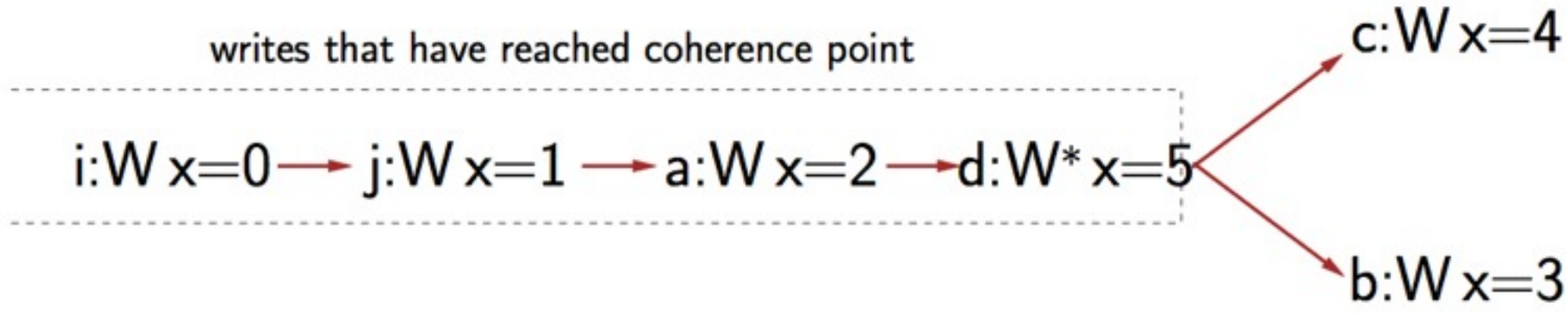
Atomic Addition

```
loop: lwarx r, x;
      add r,3,r;
      stwcx r, x;
      bne loop;
```

Coherence order for x:



Suppose lwarx reads from the a:W x=2. stwcx can succeed if this becomes possible:



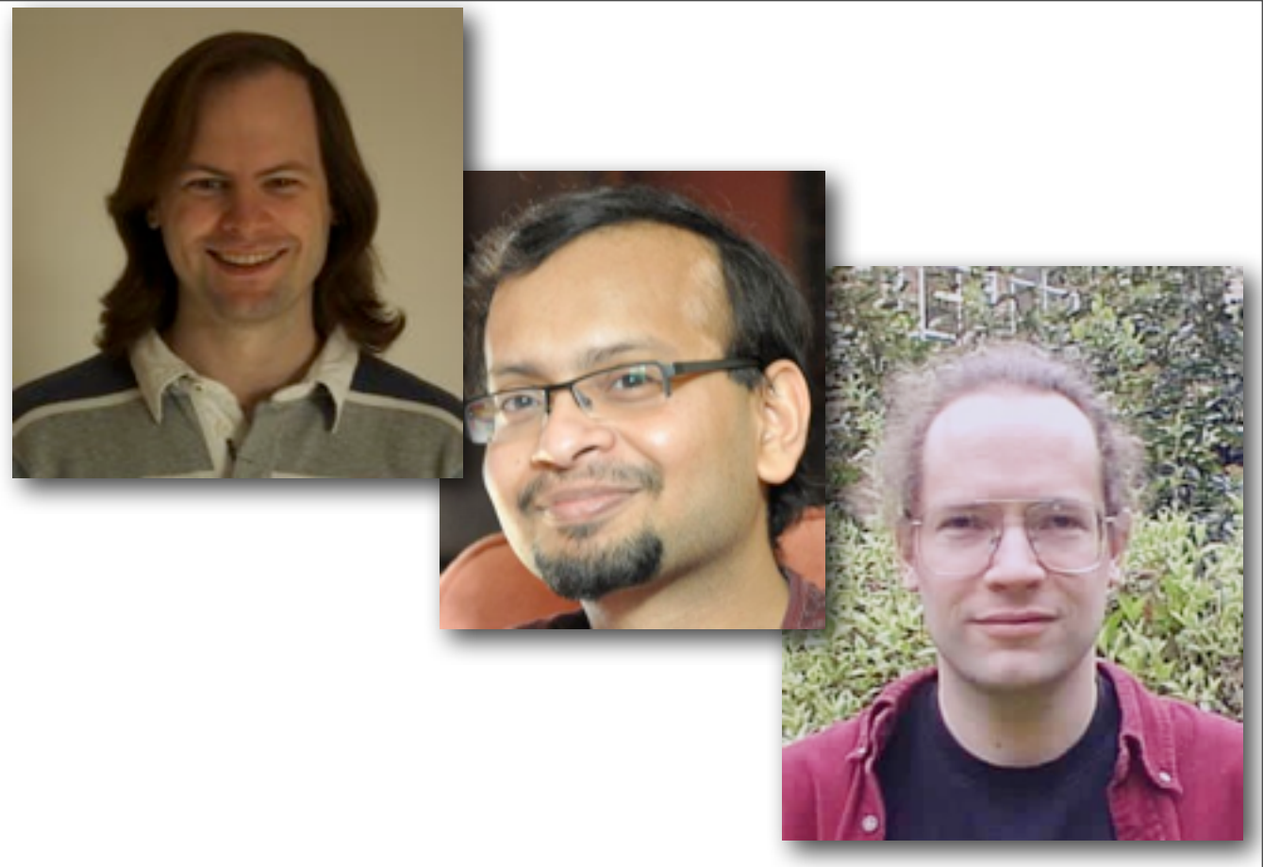
Warning: stwcx can fail spuriously.

Load-reserve/store-conditional and ordering

- Same-thread load-reserve/store-conditionals ordered by program order;
- if all memory accesses are l-r/s-c sequences, then only SC behaviour;
- but normal loads/stores (to different address) not ordered;
  the l-r/s-c do not act as a barrier.

Confusion led to a Linux bug: bad barrier placement in atomic-add-return.

Synchronising C/C++ and POWER
Sarkar, Memarian, Owens, Batty, Sewell, Maranget, Alglave, Williams

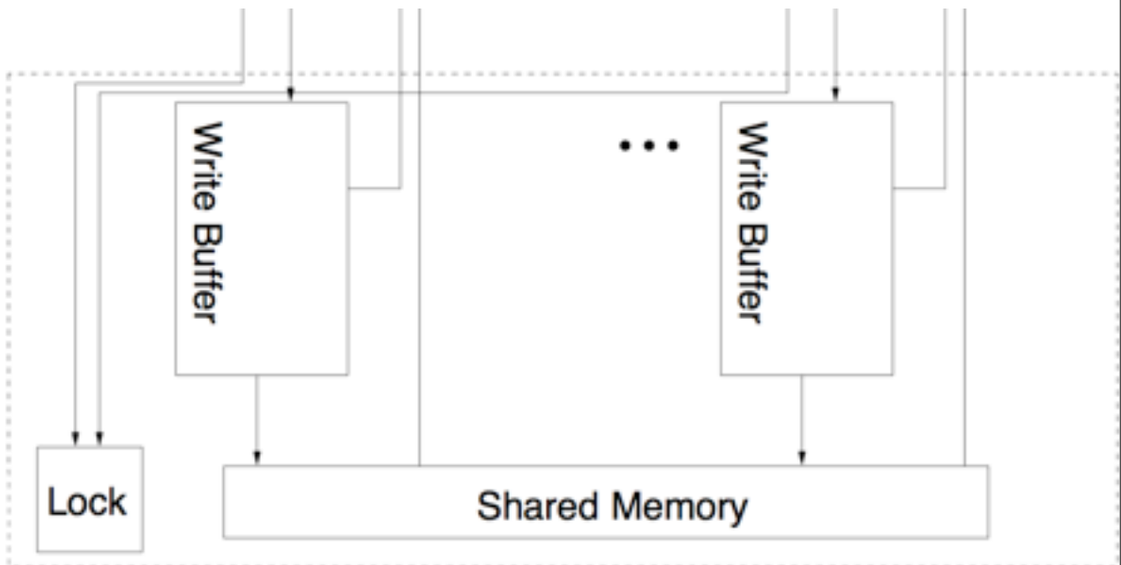


3. Sketch of an operational formalisation of x86-TSO

...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.

Lifting to processes

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

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

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

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

Actions are labelled by the thread that performed the action.

Free interleaving.

A sequentially consistent memory

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

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

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

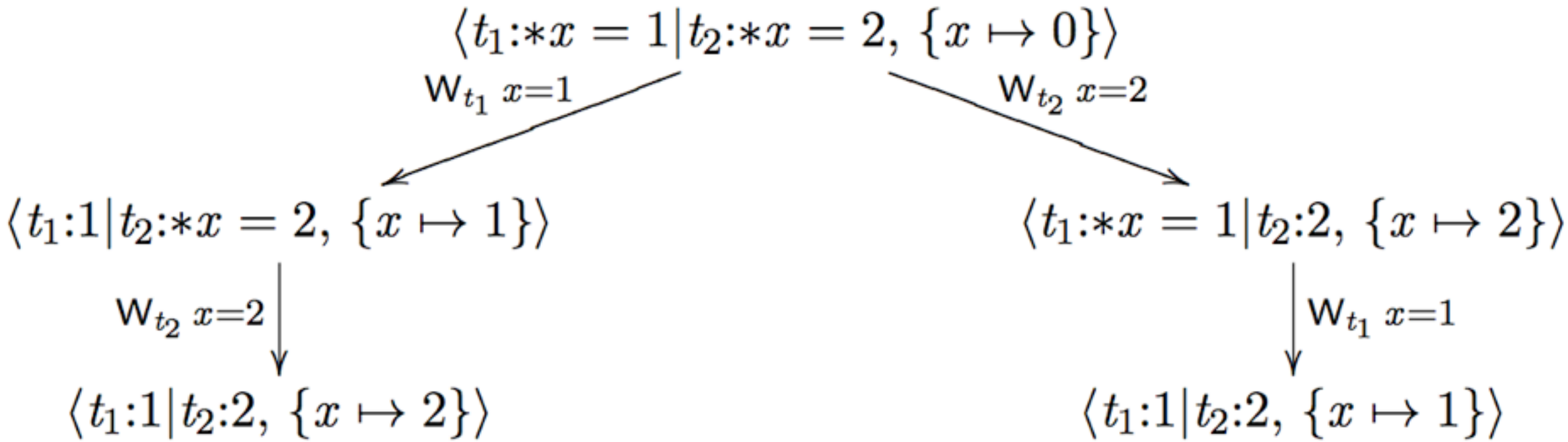
$$\frac{p \xrightarrow{\mathsf{W}_t\, x=n} p' \qquad M \xrightarrow{\mathsf{W}\, x=n} M'}{\langle p,\, M\rangle \xrightarrow{\mathsf{W}_t\, x=n} \langle p',\, M'\rangle}\ \textsc{Swrite}$$

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

Synchronising between the processes and the memory.

SC semantics, example

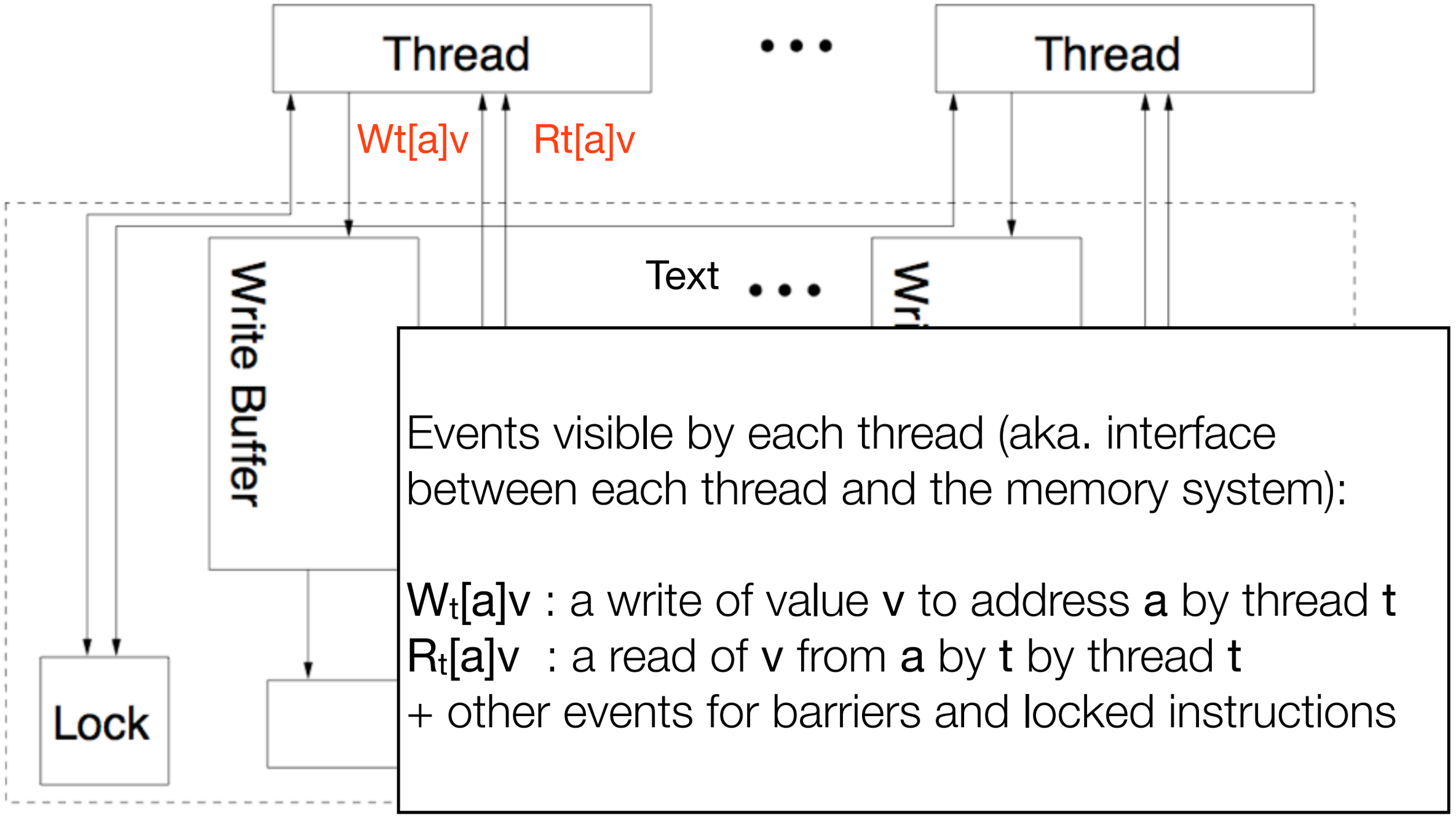
All threads read and write the shared memory. Threads execute asynchronously,the semantics allows any interleaving of the thread transitions.



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

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

x86-TSO abstract machine



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

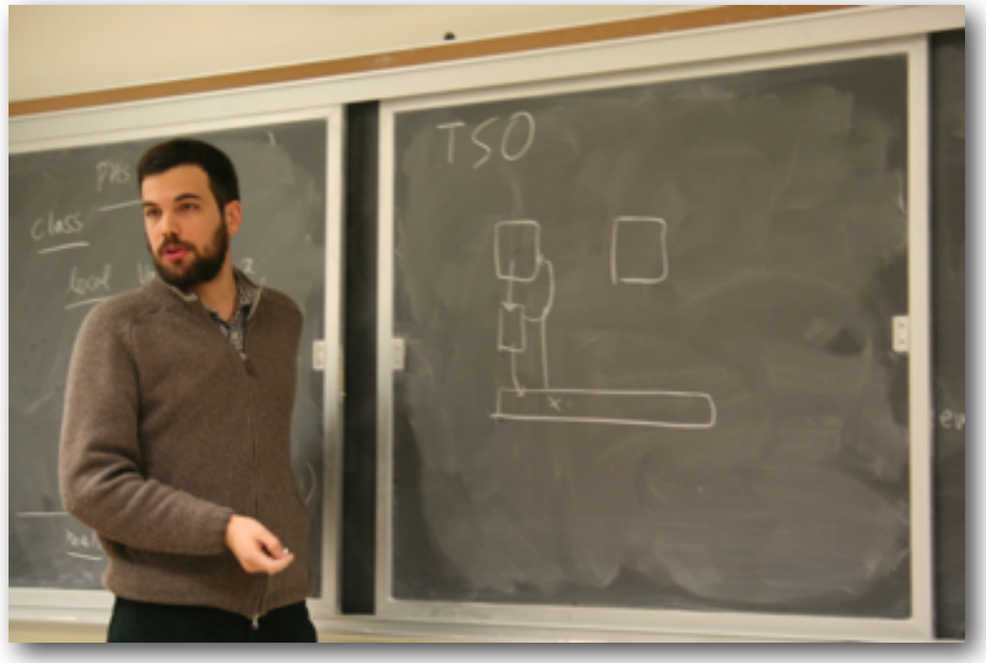
$$s \xrightarrow{\mathsf{W}_t\, x=v} s \oplus \langle\!\langle B := s.B \oplus (t \mapsto ([(x,v)] +\!\!+ s.B(t))) \rangle\!\rangle$$

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

WM: Write from write buffer to memory

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

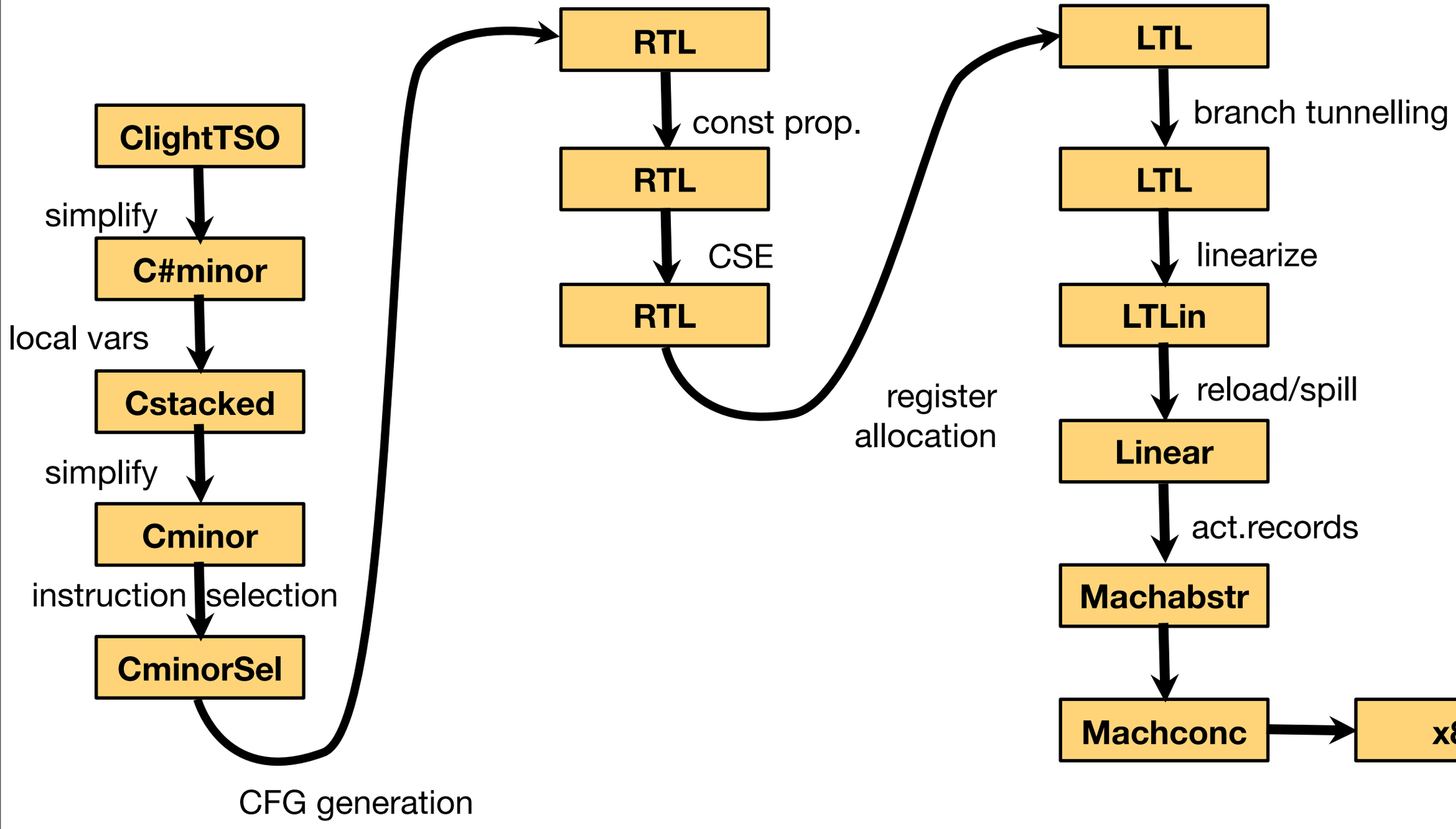
If t is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread



4. Veryfing fence elimination optimisations

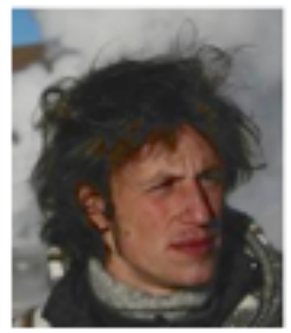
aka reasoning on the x86TSO operational memory model
and compiler correctness

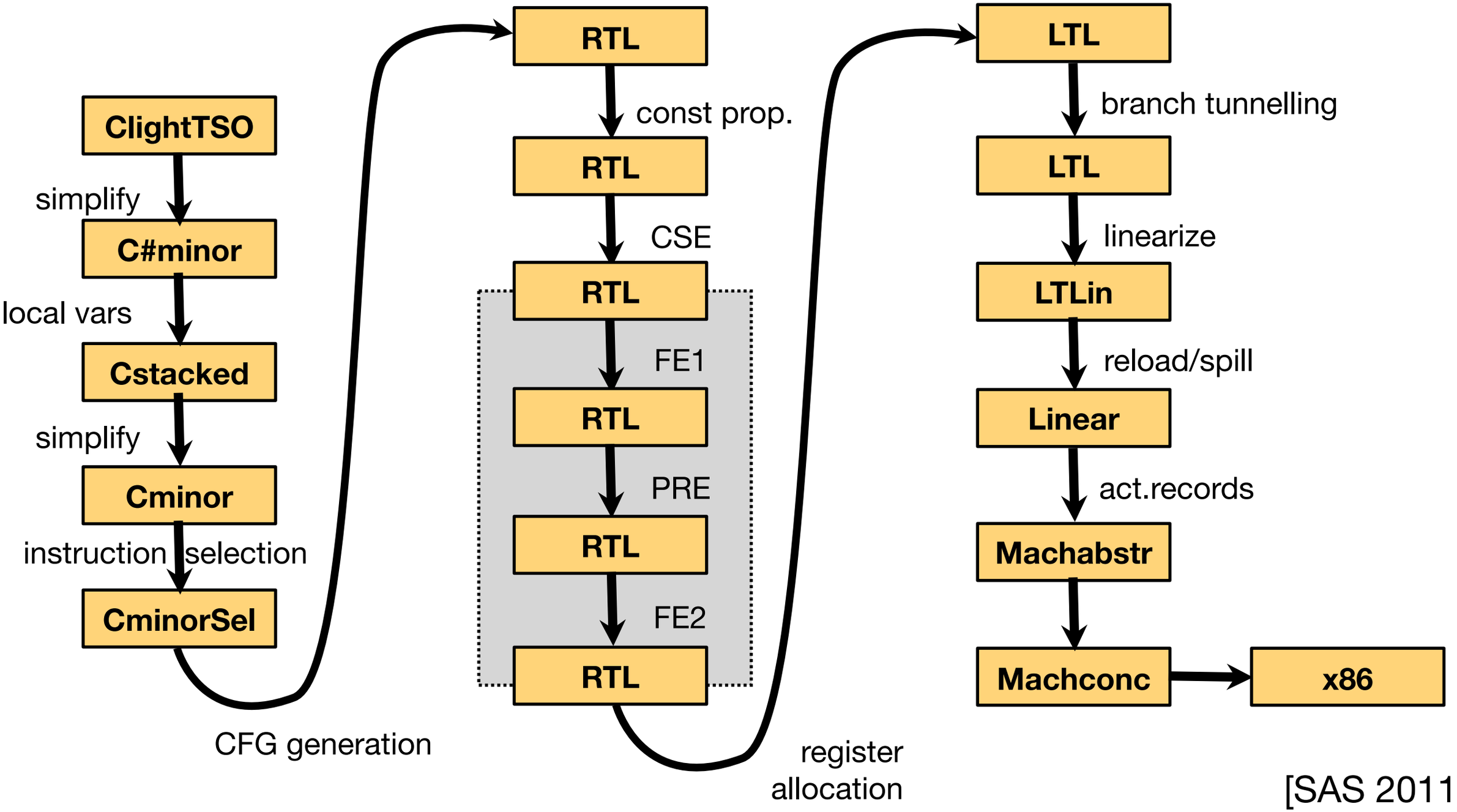
CompCertTSO



[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

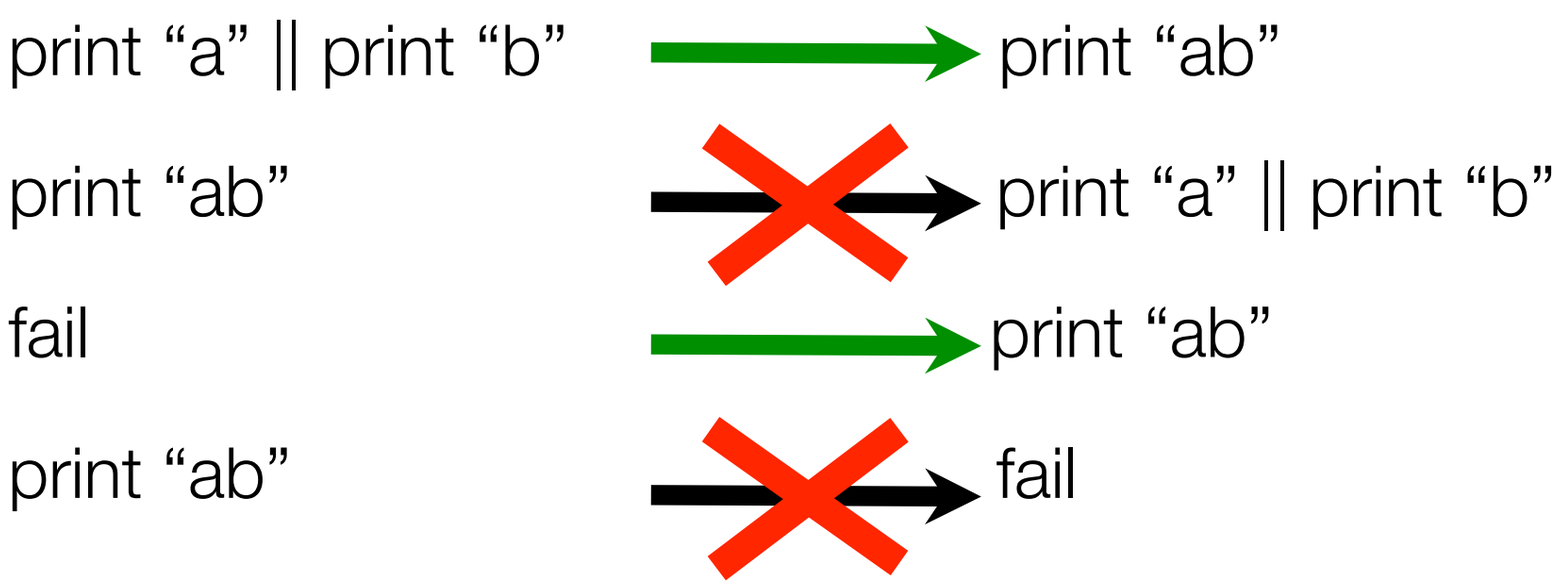
– *Infinite sequences* of call & return events;

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

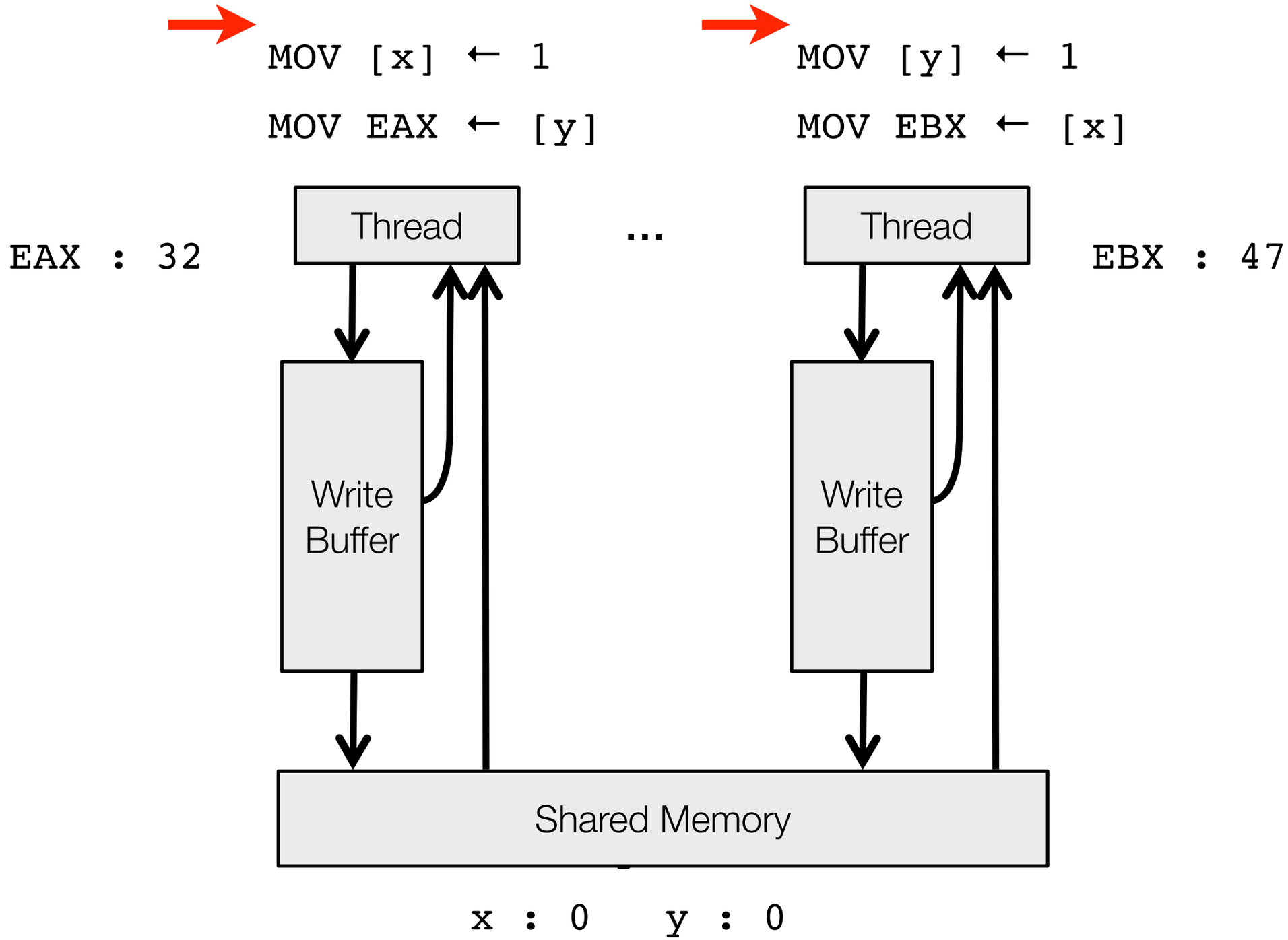
NB: Erroneous computations become undefined after the first error.

Compiler correctness

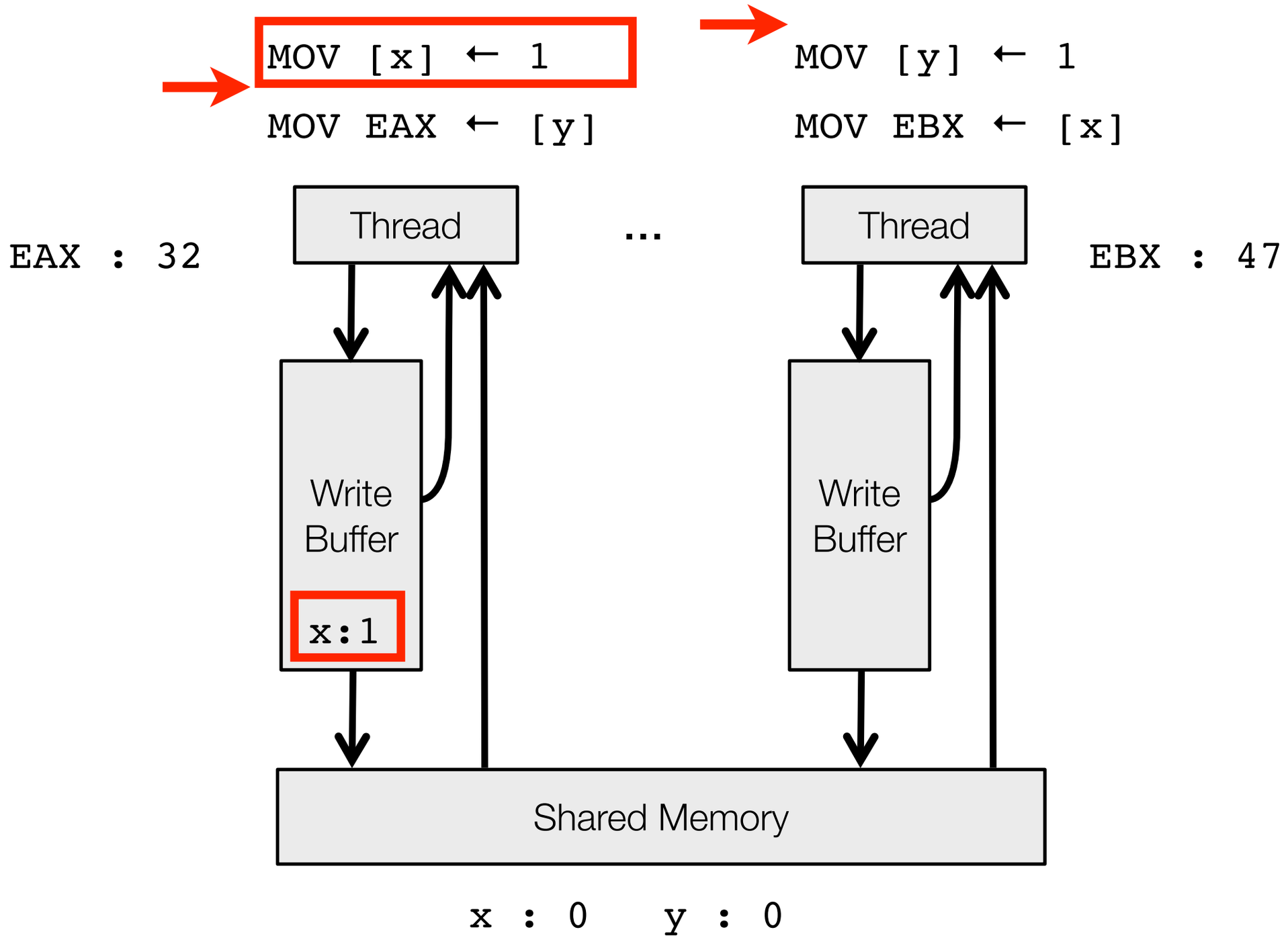
traces(source_program) ⊇ traces(target_program)



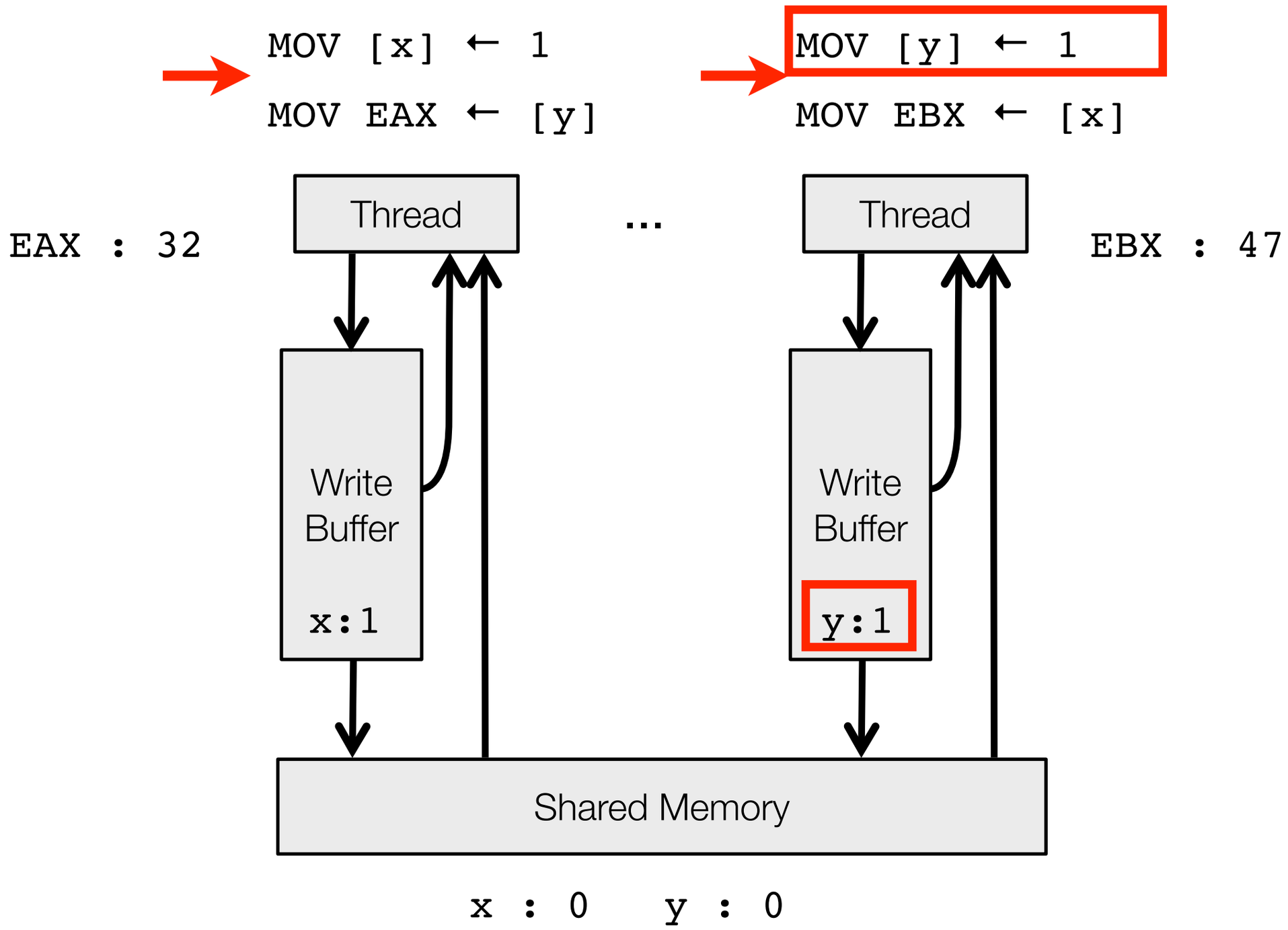
Store buffering



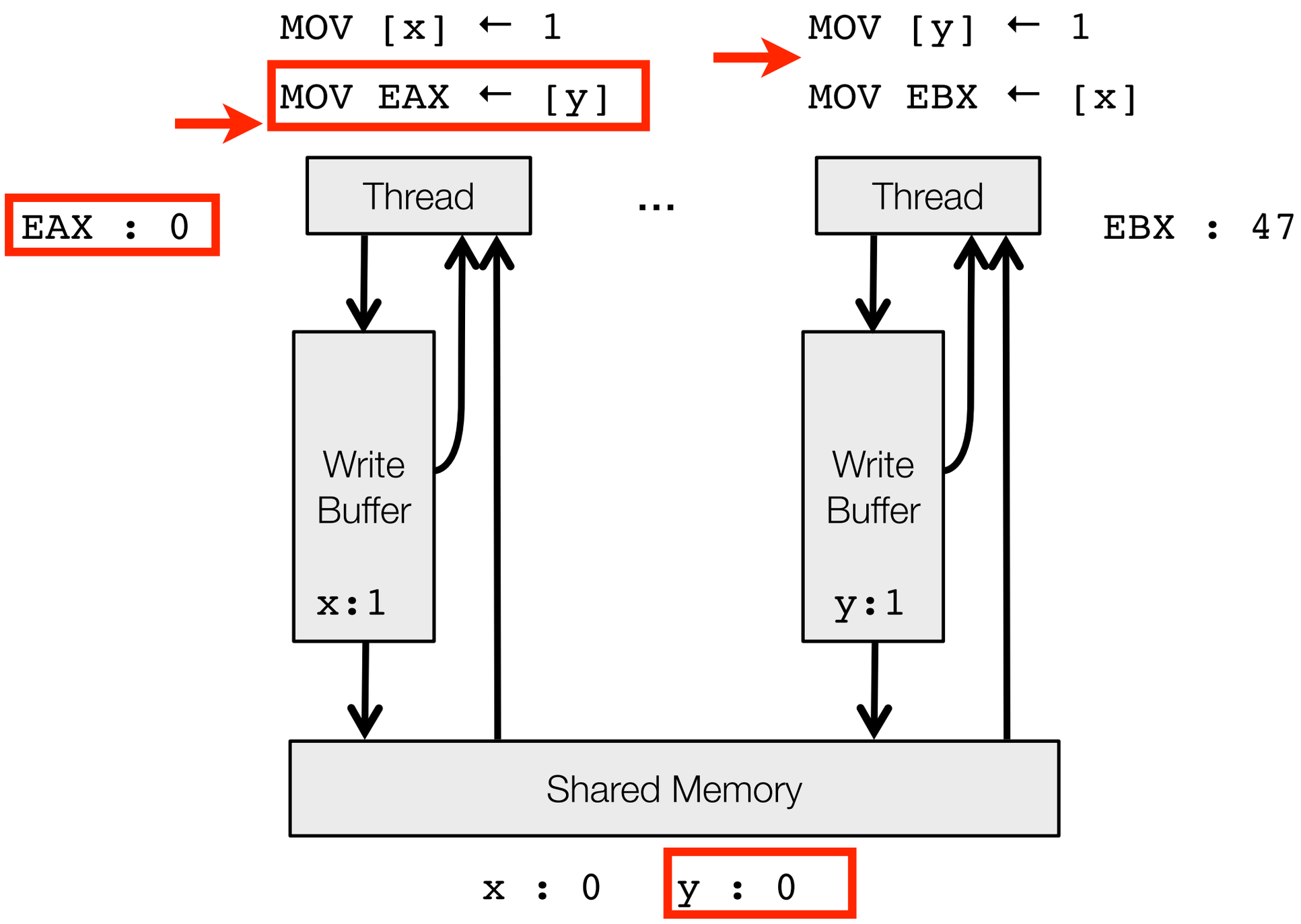
Store buffering



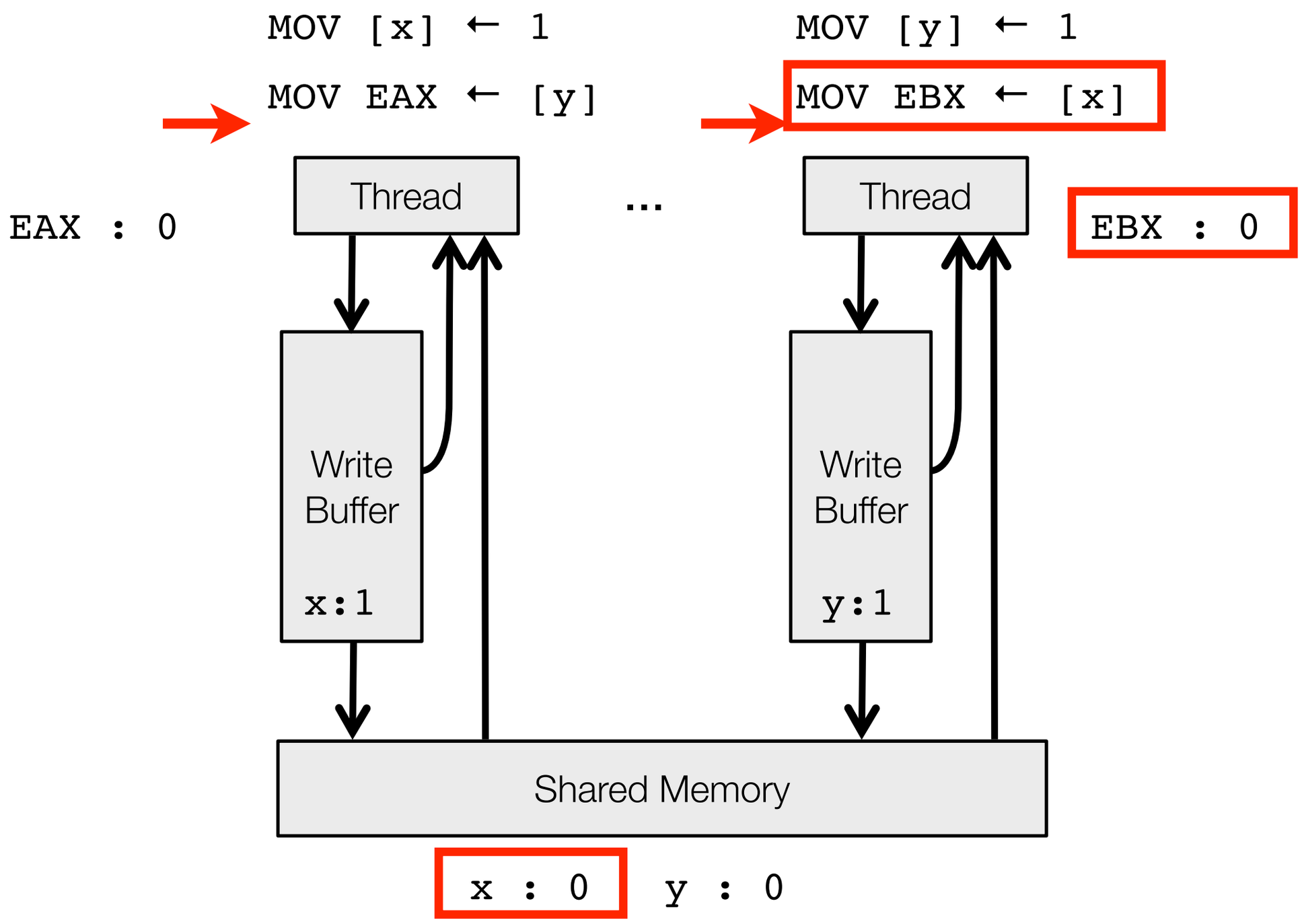
Store buffering



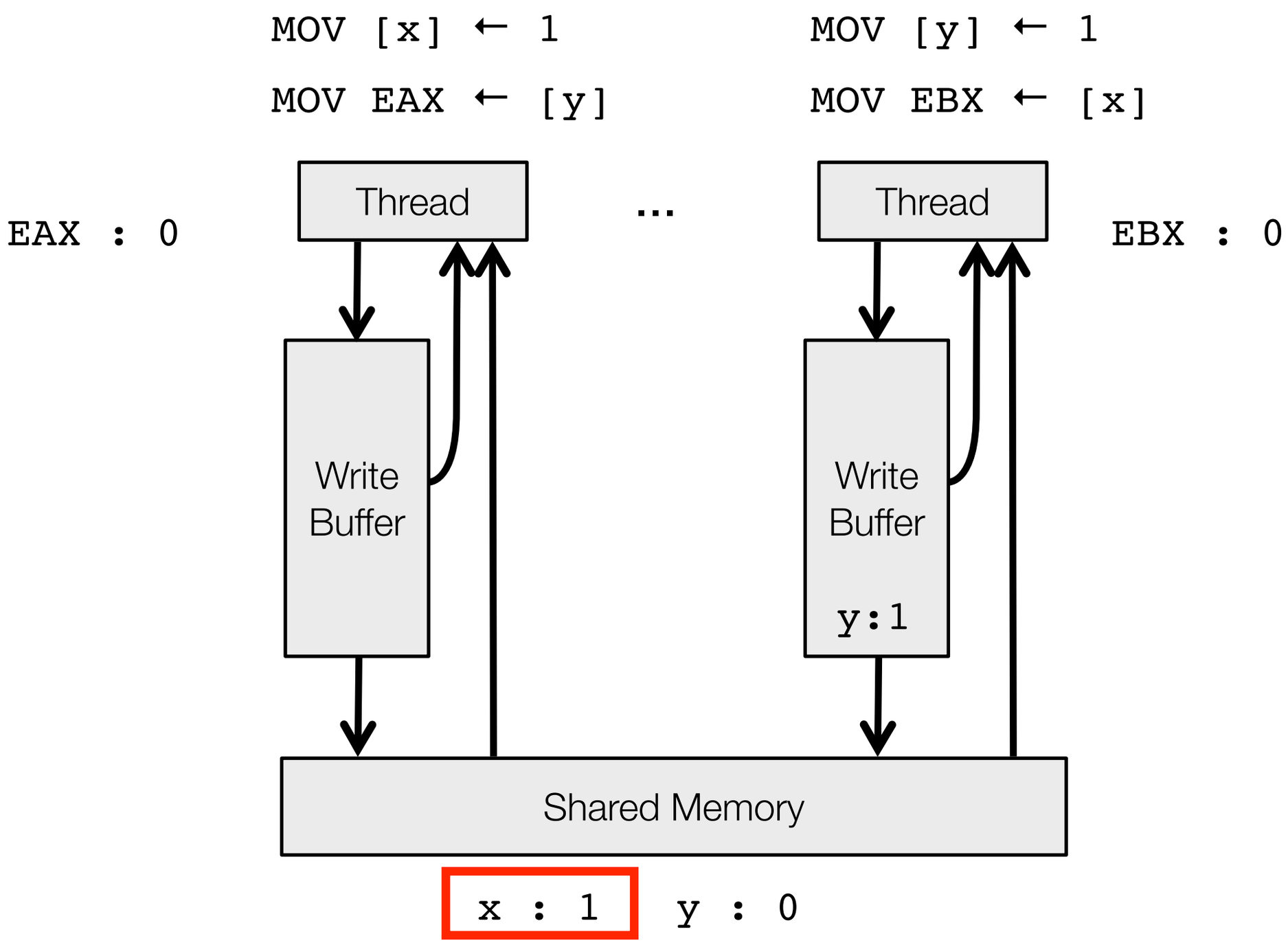
Store buffering



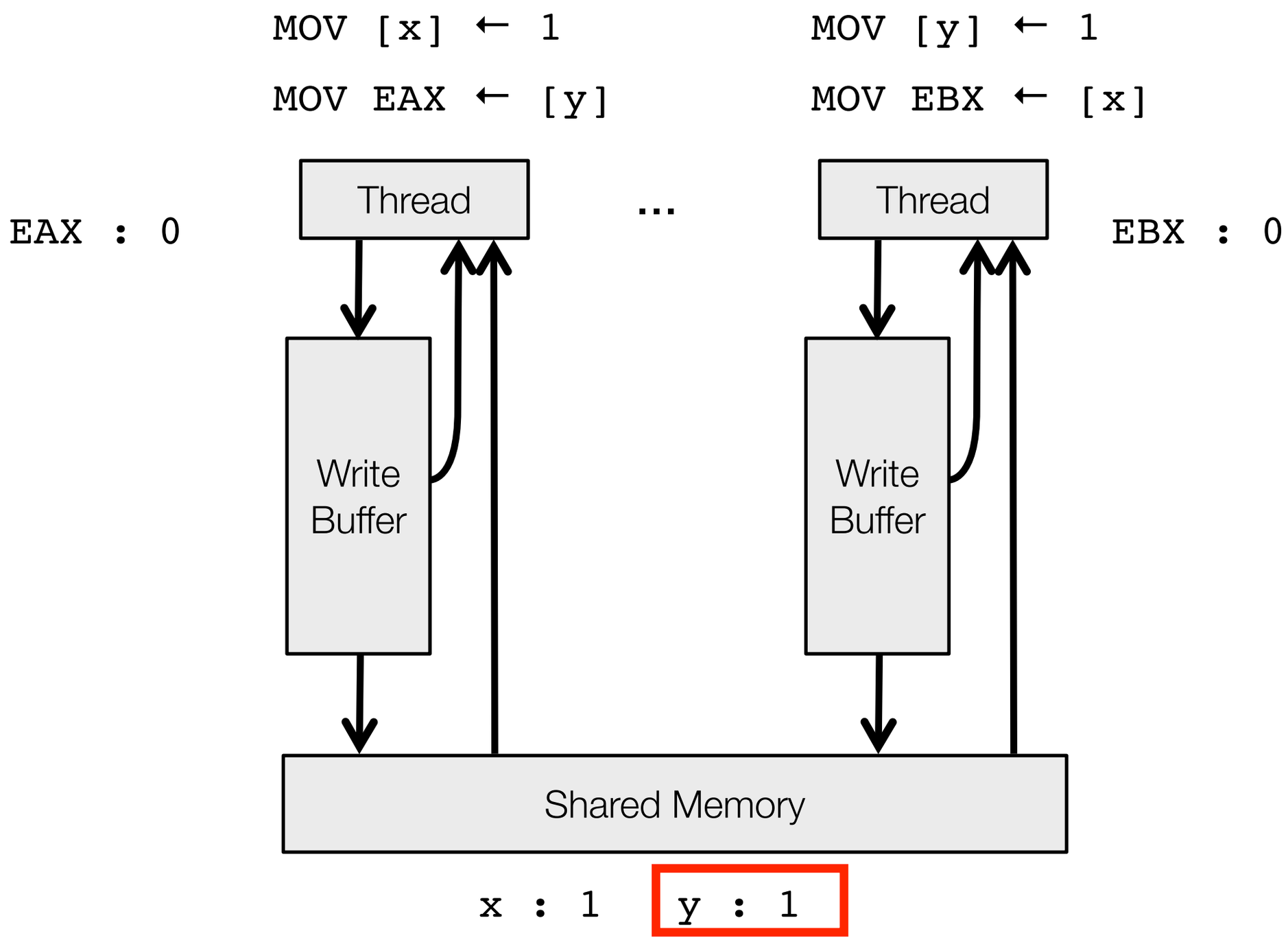
Store buffering



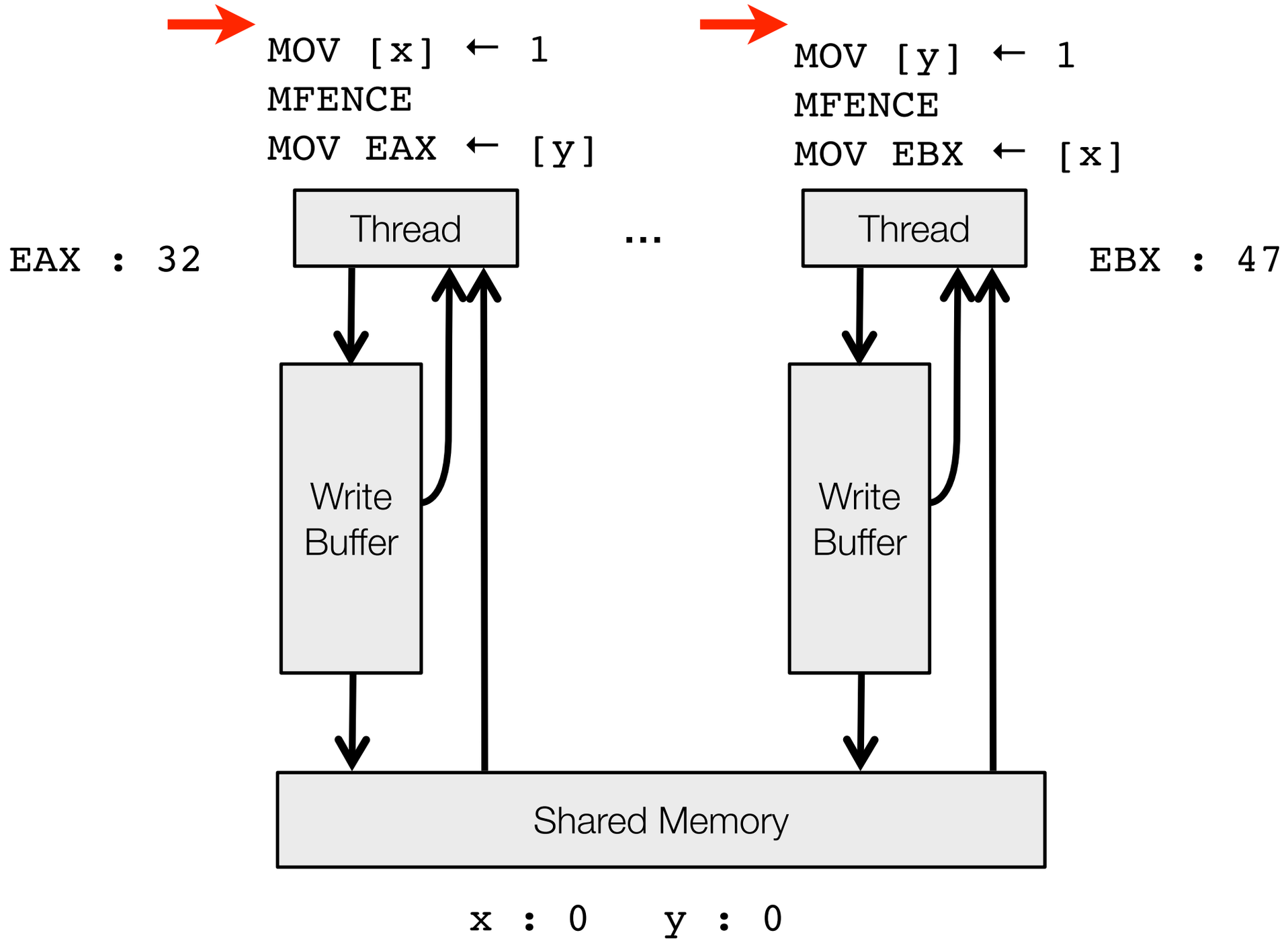
Store buffering



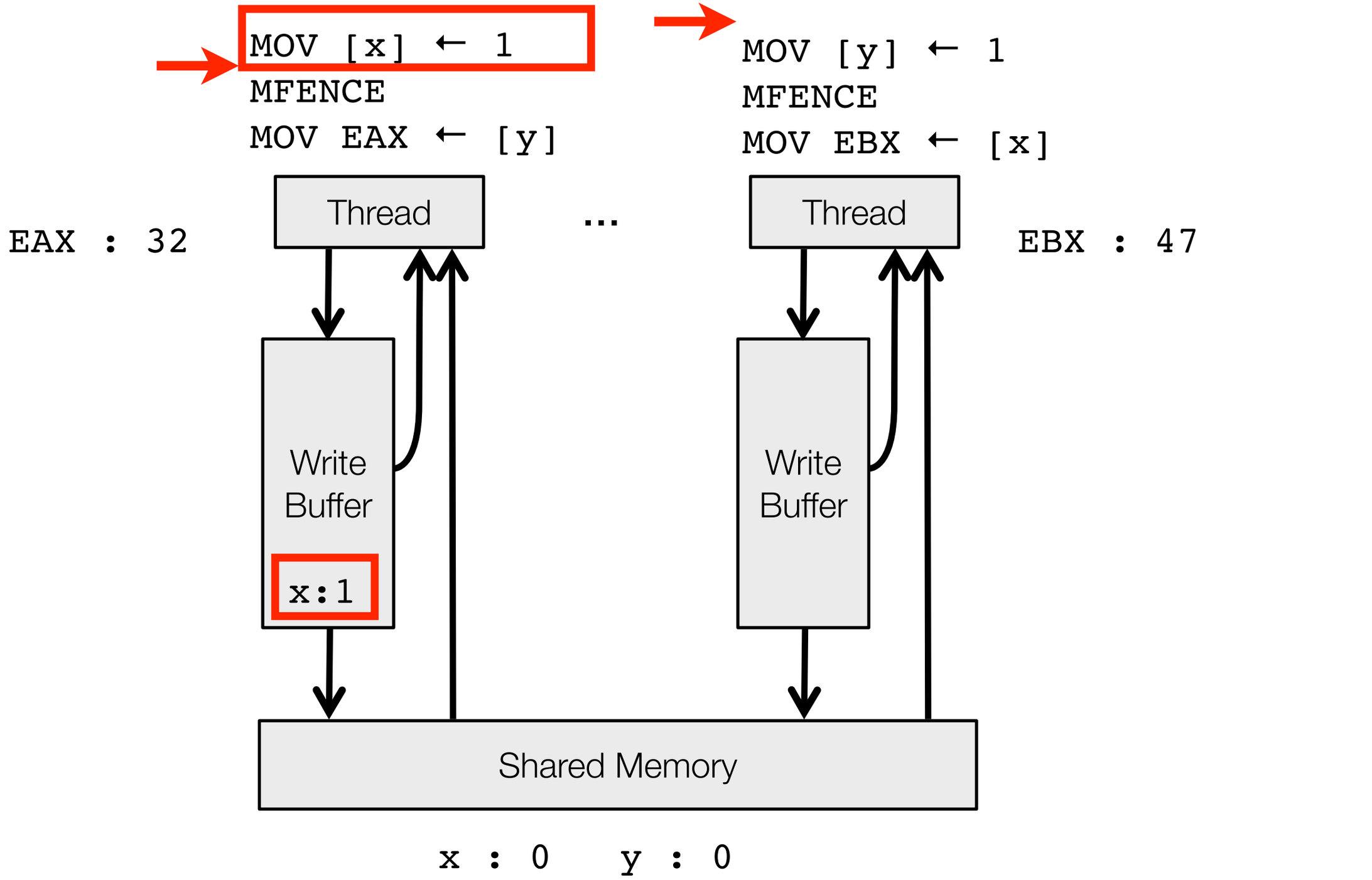
Store buffering



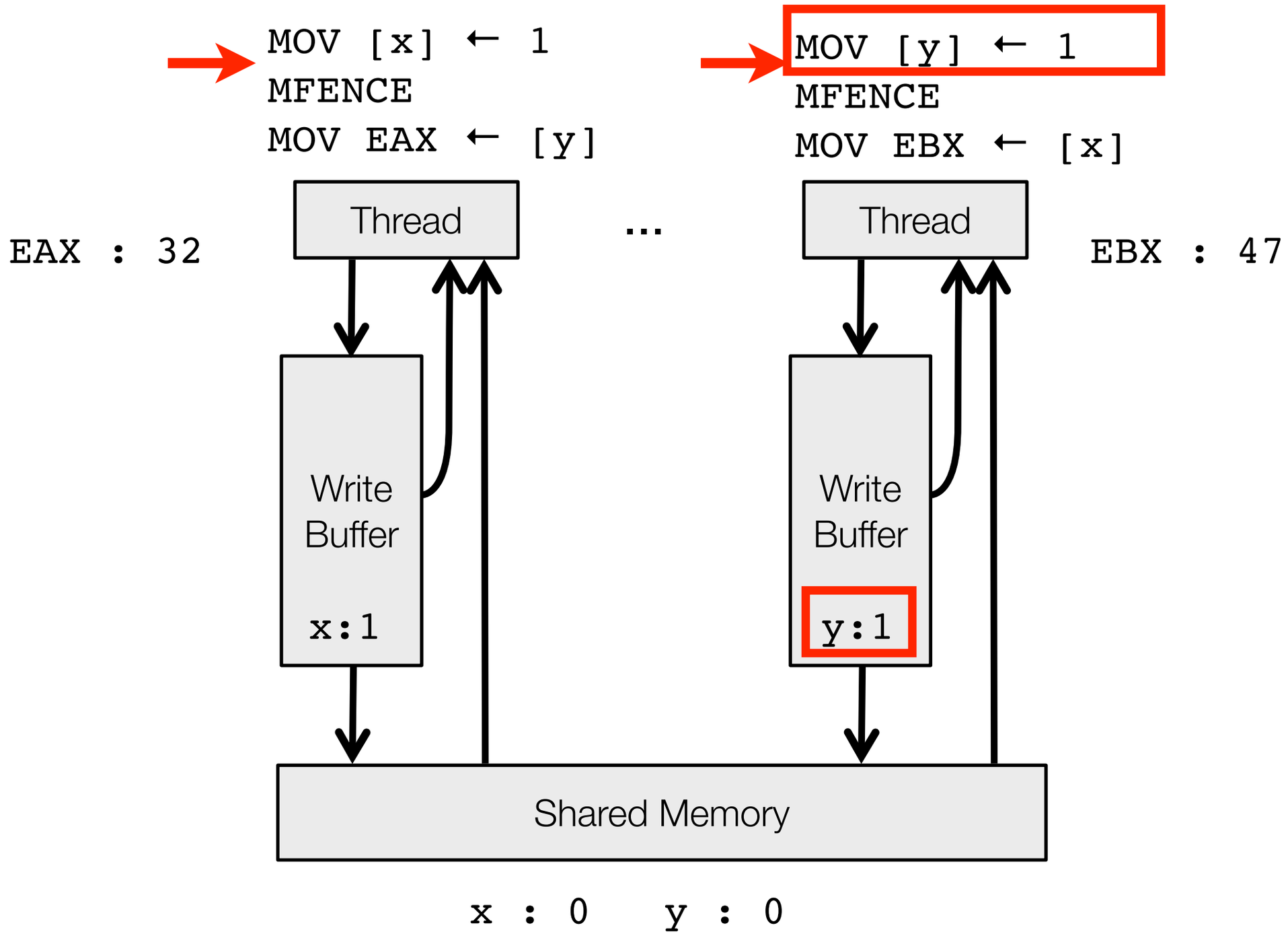
Store buffering + fences



Store buffering + fences

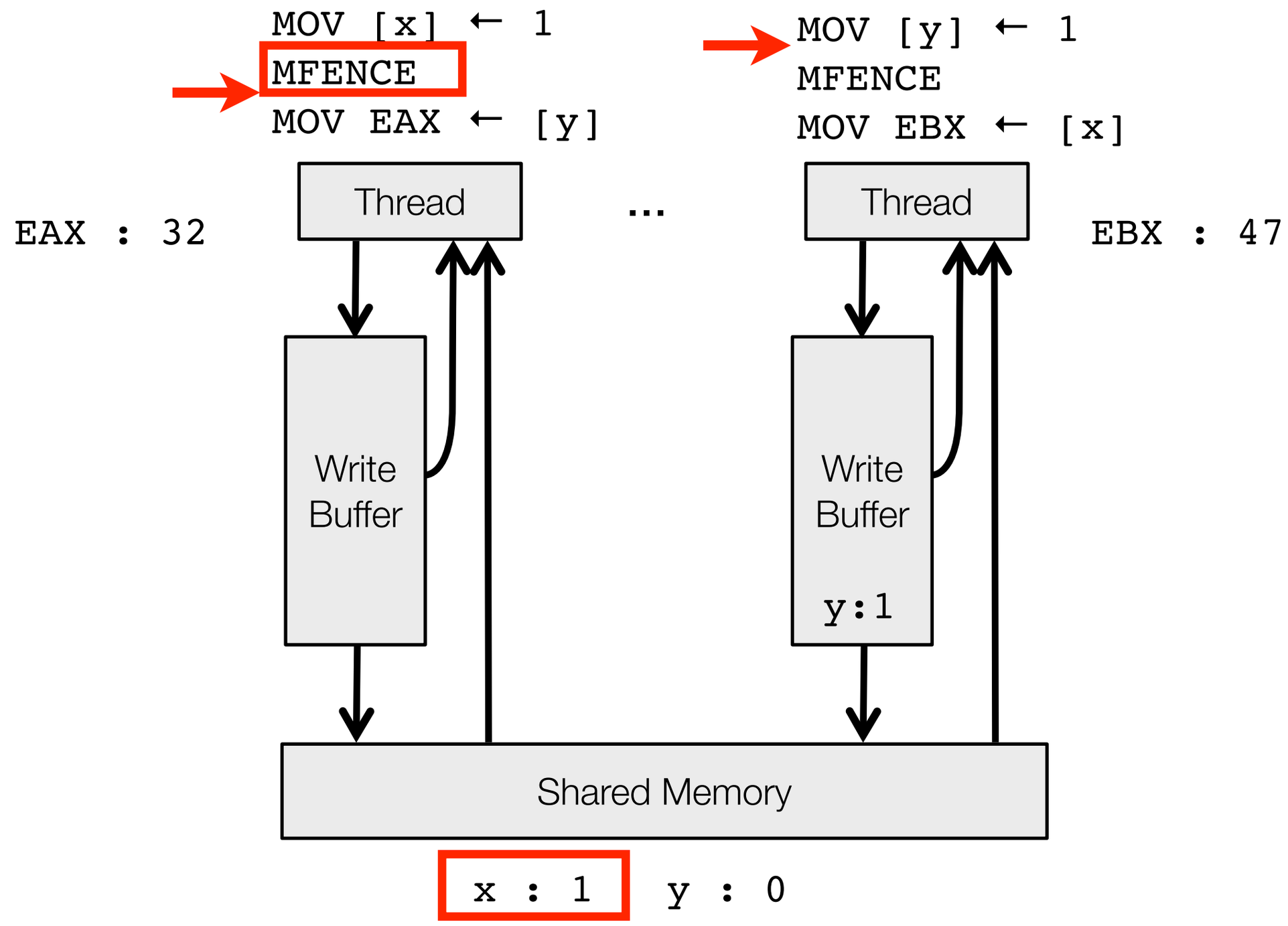


Store buffering + fences



Store buffering + fences

MFENCE blocks until the thread buffer is empty



Who inserts fences?

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

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

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

   Load SEQ_CST: `MFENCE; MOV`

   Store SEQ_CST: `MOV; MFENCE`

Fence instructions

1. *Fences are necessary*

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

   [Attiya et al., POPL 2011]

2. *Fences can be expensive*

Redundant fences (1)

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

```
MFENCE          →    MFENCE
MFENCE               NOP
```

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

Generalisation:

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

FE1

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

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

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.

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

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

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.

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

$$= \emptyset$$

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

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                  ???►   NOP
INSTRUCTION 1                  INSTRUCTION 1
…                              …
INSTRUCTION n                  INSTRUCTION n
MFENCE                         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] MF	MOV [y] ← 1 MFENCE

EAX = EBX = 0
forbidden

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

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)

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

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

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

the analysis for i's pc returned $\bot$ 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 h

– ∀ thread

(i.

t

Stutter conditio

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

(and

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

$\equiv_i t$

Weaktau simulation

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

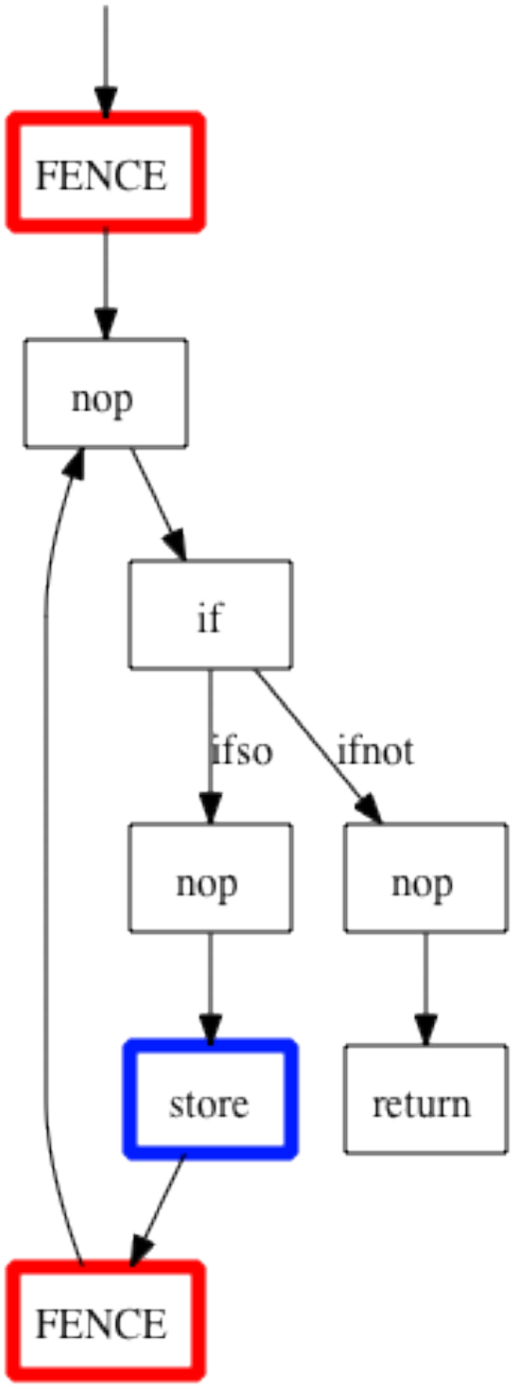
$s \simeq t$ iff

– 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

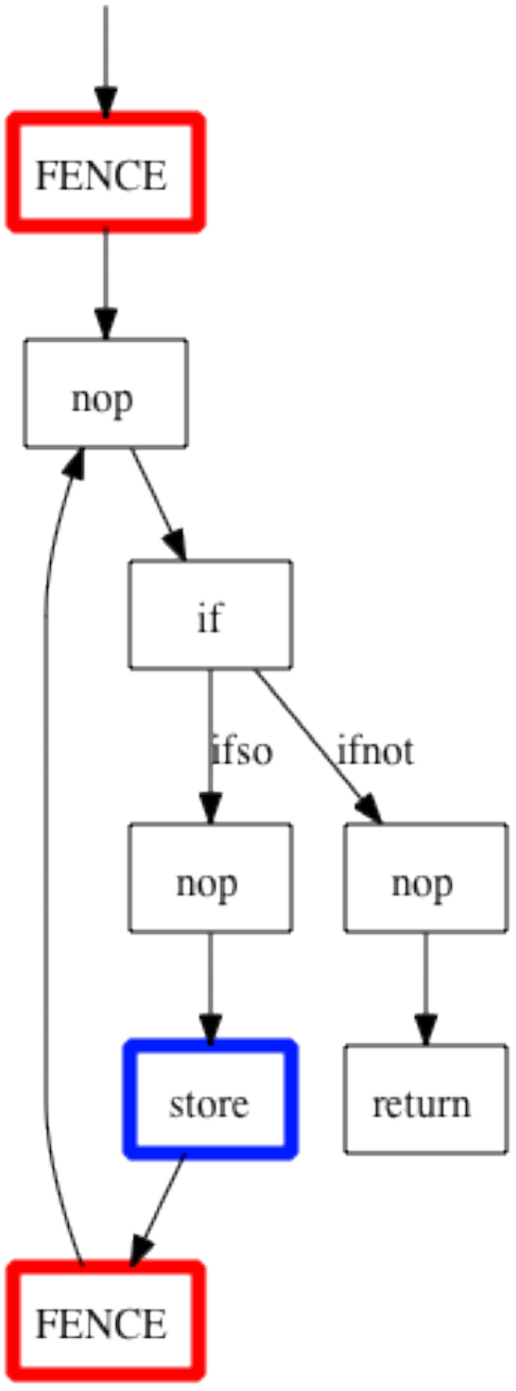


Patterns like that on the left are common.

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

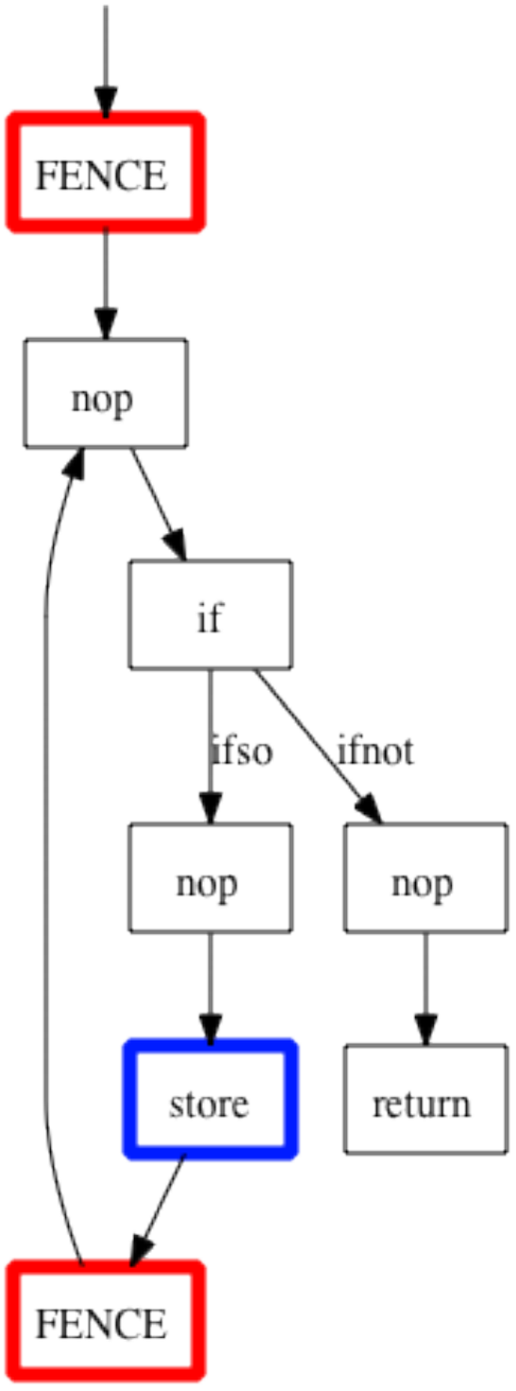


Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

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

A closer look at the RTL



Patterns like that on the left are common.

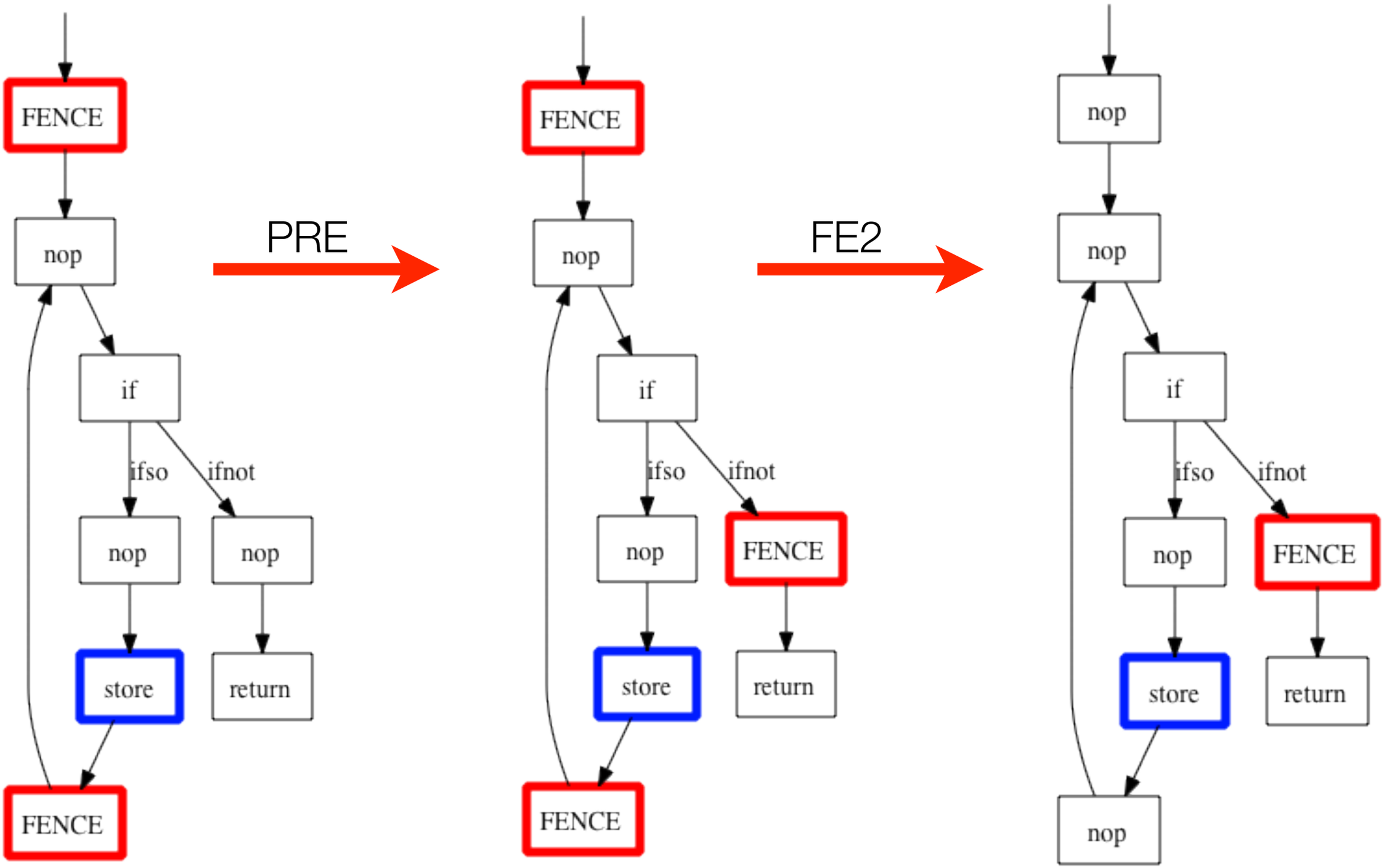
FE1 and FE2 do not optimise these patterns.

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

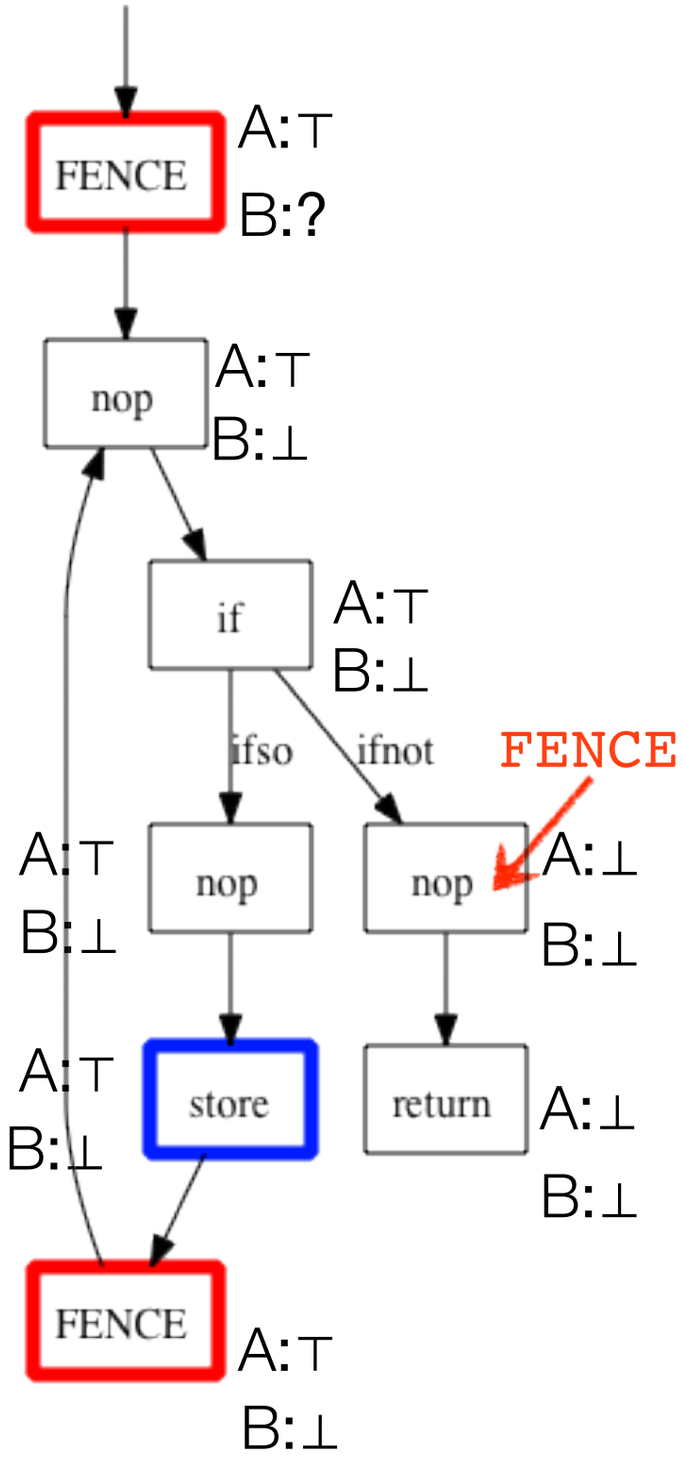
Do you perform PRE?

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

Partial redundancy elimination



Partial redundancy elimination



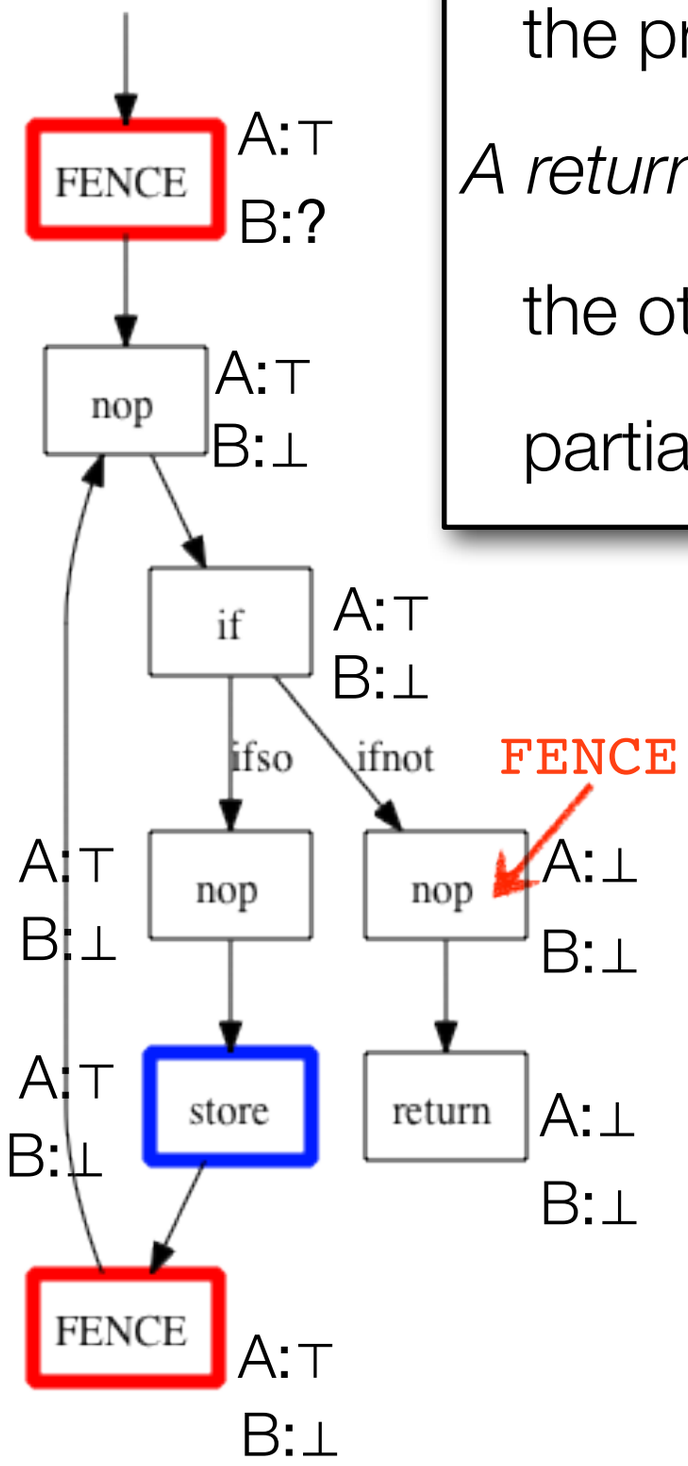
A: a backward analysis returning ⊤ if along *some path* after the current program point there is an atomic instruction with no intervening reads;

B: a forward analysis returning ⊥ if along all paths to the current program point there is a fence with no later reads or atomic instructions.

Replace ***`NOP`*** *with* ***`FENCE`*** *after conditionals if:*

- B returns ⊥
- A returns ⊥
- A returns ⊤ on the other branch

Partial redu



B returns ⊥:
a previous fence will be eliminated if we insert a fence at both branches of conditional nodes.

A returns ⊥:
the previous fence won't be removed by FE2.

A returns ⊤ on the other branch:

the other branch already makes the previous fence partially redundant.

B: a forward analysis returning ⊥ if along all paths to the current program point there is a fence with no later reads or atomic instructions.

Replace ***`NOP`*** *with* ***`FENCE`*** *after conditionals if:*

- B returns ⊥
- A returns ⊥
- A returns ⊤ on the other branch

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

Evaluation should be taken seriously by CS scientists!

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

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

5. Hunting compiler concurrency bugs

Can you guess the output?

```
int g_1 = 1;
int g_2 = 0;
```

Thread 1

```
int func_1(void) {
  int l;
  for (l=0; (l!=4); l++) {
    if (g_1)
      return l;
    for(g_2=0; (g_2>=26); ++g_2)
      ;
  }
}
```

Thread 2

```
void func_2(void){
  g_2 = 42;
  printf("%d", g_2);
}
```

Can you guess the output?

```
int g_1 = 1;
int g_2 = 0;
```

Thread 1

```
int func_1(void) {
  int l;
  for (l=0; (l!=4); l++) {
    if (g_1)
      return l;
    for(g_2=0; (g_2>=26); ++g_2)
      ;
  }
}
```

Thread 2

```
void func_2(void){
  g_2 = 42;
  printf("%d", g_2);
}
```

Thread 1 never updates `g_2`.

This program can only print `42`.

If we compile with gcc 4.7 -O2, sometimes...

```
int g_1 = 1;
int g_2 = 0;
```

Thread 1

```
int func_1(void) {
  int l;
  for (l=0; (l!=4); l++) {
    if (g_1)
      return l;
    for(g_2=0; (g_2>=26); ++g_2)
      ;
  }
}
```

Thread 2

```
void func_2(void){
  g_2 = 42;
  printf("%d", g_2);
}
```

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

```
int func_1(void) {
  int l;
  for (l=0; (l!=4); l++) {
    if (g_1)
      return l;
    for(g_2=0; (g_2>=26); ++g_2)
      ;
  }
}
```

gcc 4.7 -O2

```
func_1:
  movl   g_1(%rip),%edx
  movl   g_2(%rip),%eax
  testl  %edx, %edx
  jne    .L2
  movl   $0, g_2(%rip)
  ret
.L2:
  movl   %eax, g_2(%rip)
  xorl   %eax, %eax
  ret
```

the compiled `func_1` saves and restores `g_2`

```
int g_1 = 1;
int g_2 = 0;
```

```
func_1:
  movl    g_1(%rip),%edx
  movl    g_2(%rip),%eax
  testl   %edx, %edx
  jne     .L2
  movl    $0, g_2(%rip)
  ret
.L2:
  movl    %eax, g_2(%rip)
  xorl    %eax, %eax
  ret
```

```
void func_2(void){
  g_2 = 42;
  printf("%d", g_2);
}
```

- Read g_1.

```
int g_1 = 1;
int g_2 = 0;
```

```
func_1:
  movl    g_1(%rip),%edx
  movl    g_2(%rip),%eax
  testl   %edx, %edx
  jne     .L2
  movl    $0, g_2(%rip)
  ret
.L2:
  movl    %eax, g_2(%rip)
  xorl    %eax, %eax
  ret
```

```
void func_2(void){
  g_2 = 42;
  printf("%d", g_2);
}
```

- Read g_1.
- Read g_2: %eax = 0

```
int g_1 = 1;
int g_2 = 0;
```

```
func_1:
  movl   g_1(%rip),%edx
  movl   g_2(%rip),%eax
  testl  %edx, %edx
  jne    .L2
  movl   $0, g_2(%rip)
  ret
.L2:
  movl   %eax, g_2(%rip)
  xorl   %eax, %eax
  ret
```

```
void func_2(void){
  g_2 = 42;
  printf("%d", g_2);
}
```

- Read g_1.
- Read g_2: %eax = 0
- Write g_2: g_2 = 42

```
int g_1 = 1;
int g_2 = 0;
```

```
func_1:
  movl   g_1(%rip),%edx
  movl   g_2(%rip),%eax
  testl  %edx, %edx
  jne    .L2
  movl   $0, g_2(%rip)
  ret
.L2:
  movl   %eax, g_2(%rip)
  xorl   %eax, %eax
  ret
```

```
void func_2(void){
  g_2 = 42;
  printf("%d", g_2);
}
```

- Read g_1.
- Read g_2: eax = 0
- Write g_2: g_2 = 42
- Copy eax into g_2: g_2 = 0

```
int g_1 = 1;
int g_2 = 0;
```

```
func_1:
  movl   g_1(%rip),%edx
  movl   g_2(%rip),%eax
  testl  %edx, %edx
  jne    .L2
  movl   $0, g_2(%rip)
  ret
.L2:
  movl   %eax, g_2(%rip)
  xorl   %eax, %eax
  ret
```

```
void func_2(void){
  g_2 = 42;
  printf("%d", g_2);
}
```

- Read g_1.
- Read g_2: eax = 0
- Write g_2: g_2 = 42
- Copy eax into g_2: g_2 = 0
- Print 0 (?!)

Is this a compiler bug?

In the C11/C++11 memory model,

data-race free programs must have only sequentially consistent behaviours.

1. Our program was data-race free.
2. Print 0 is not a sequentially consistent behaviour.

Is this a compiler bug?

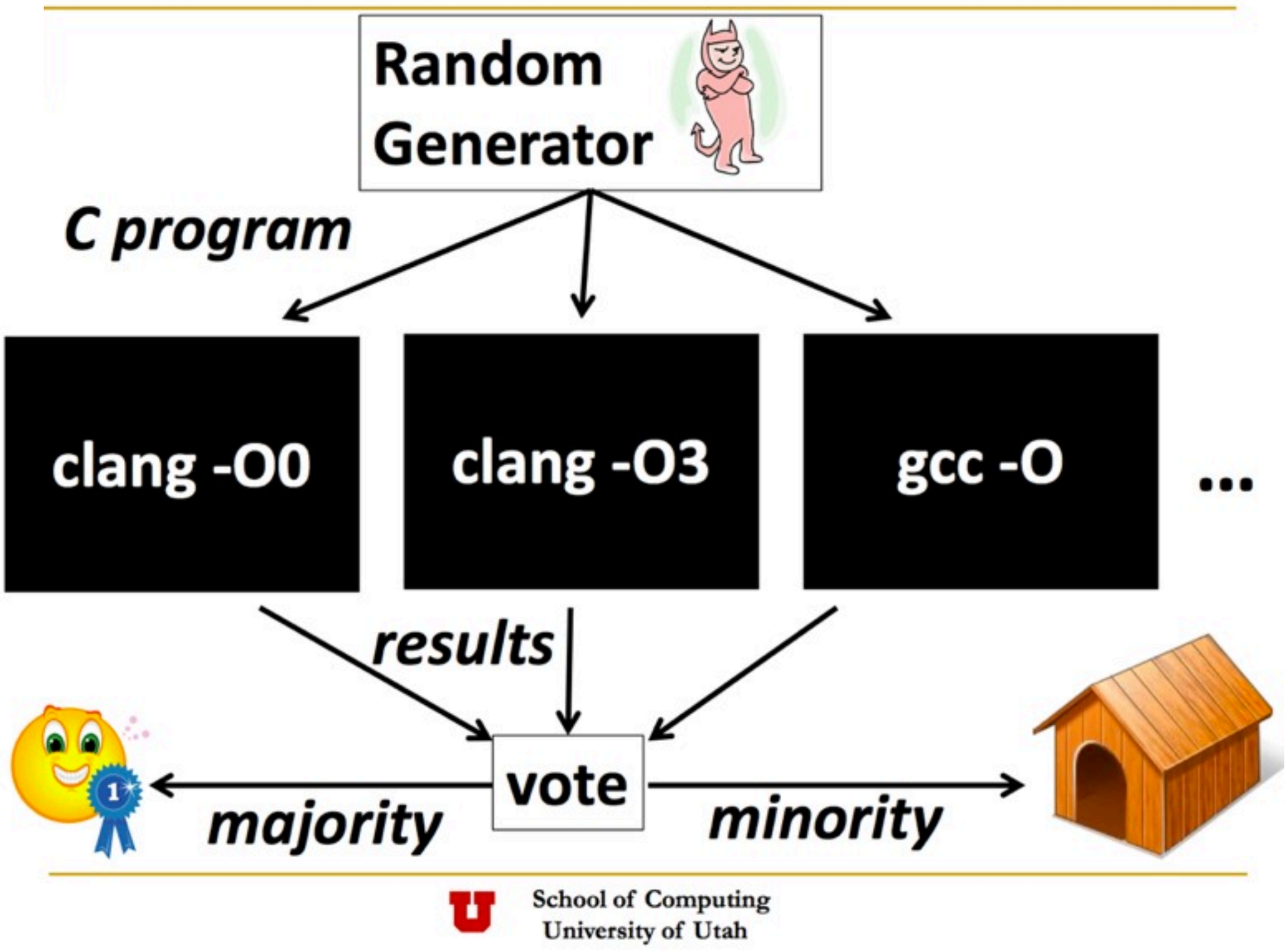
We found a *concurrency compiler bug*.

Can we find more?

1. Our program was data-race free.
2. Print 0 is not a sequentially consistent behaviour.

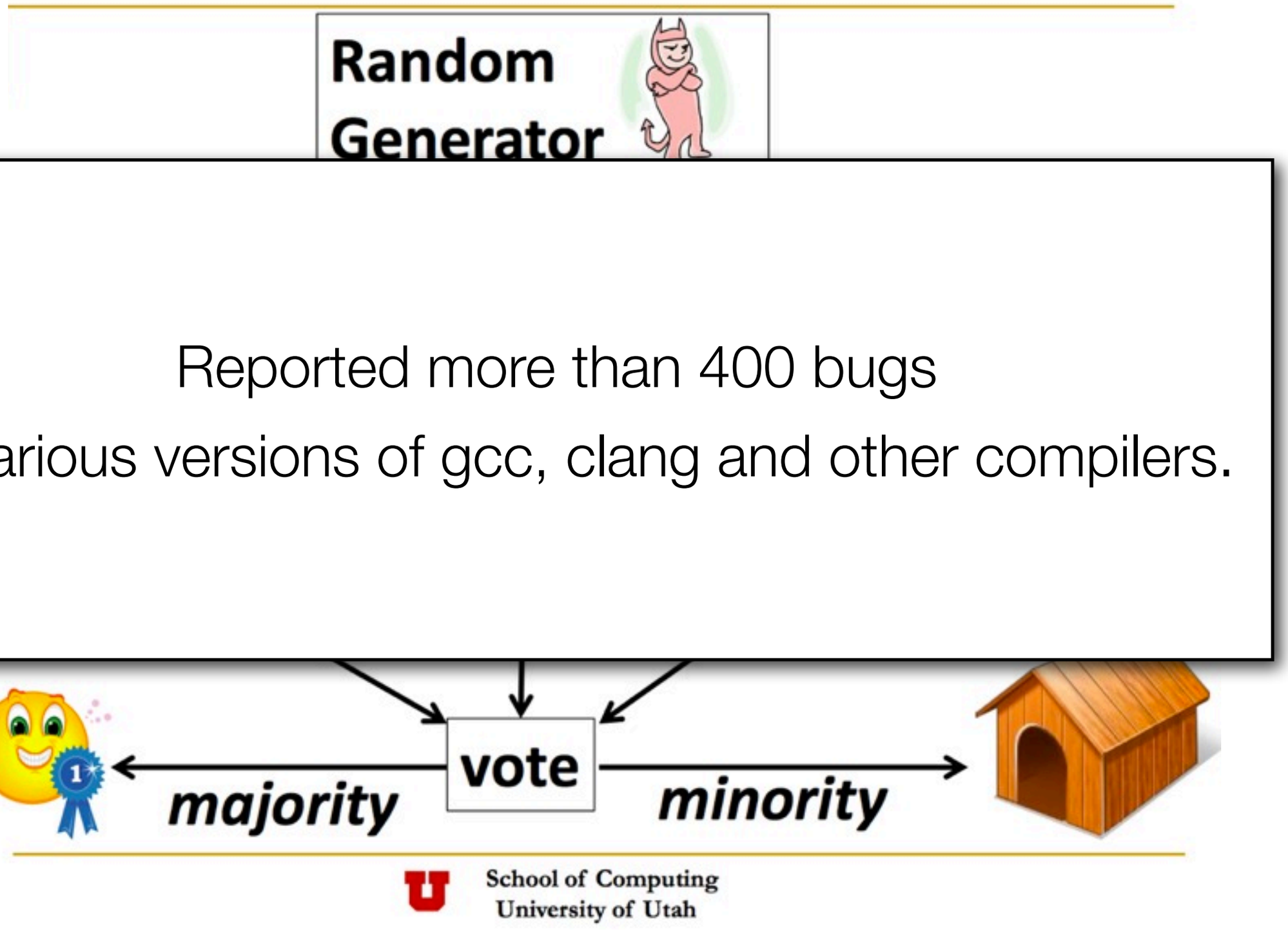
Compiler testing: state of the art

Regehr et al.



Compiler testing: state of the art

Regehr et al.



Generalise to concurrency compiler bugs?

How to deal with non-determinism?

- How to generate random, *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)

Our idea

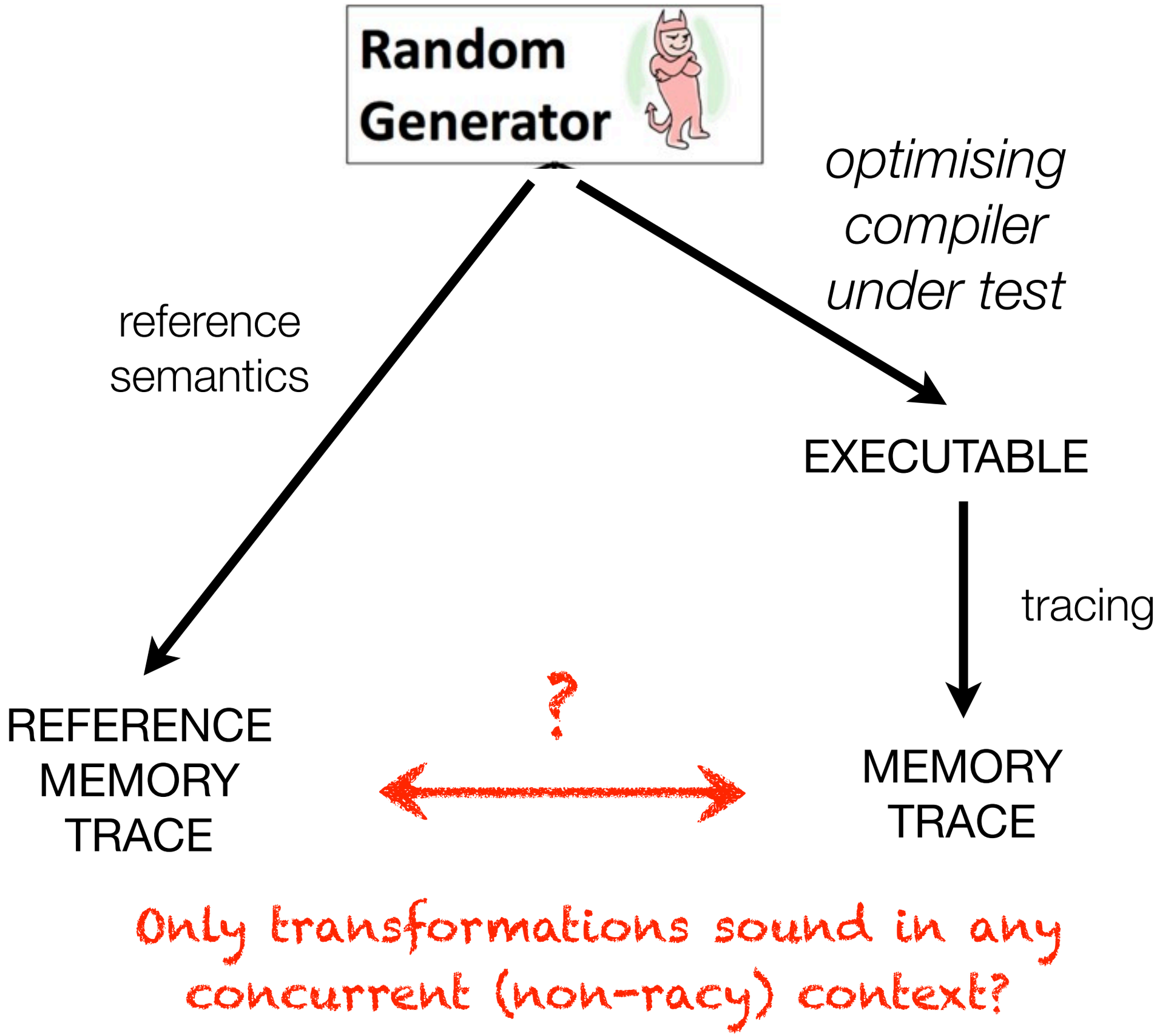
C/C++ functions can be called in arbitrary (non-racy) concurrent contexts

C/C++ compilers can only apply transformations which are 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 context.


Random Generator
optimising compiler under test
reference semantics
EXECUTABLE
tracing
REFERENCE MEMORY TRACE
?
MEMORY TRACE
Only transformations sound in any concurrent (non-racy) context?

```
int g_1 = 1;
int g_2 = 0;
```

```
int func_1(void) {
  int l;
  for (l=0; (l!=4); l++) {
    if (g_1)
      return l;
    for(g_2=0; (g_2>=26); ++g_2)
      ;
  }
}
```

reference semantics

```
Init g_1 1
Init g_2 0
Load g_1 1
```

gcc -O2 trace

```
Init g_1 1
Init g_2 0
Load g_1 1
Load g_2 0
Store g_2 0
```

```
int g_1 = 1;      int func_1(void) {
```

Cannot match some events → compiler bug!

```
    }
}
```

reference semantics

```
Init g_1 1
Init g_2 0
Load g_1 1
```

gcc -O2 trace

```
Init g_1 1
Init g_2 0
Load g_1 1
Load g_2 0
Store g_2 0
```

```
const unsigned int g_3 = 0UL;
long long g_4 = 0x1;
int g_6 = 6L;
volatile unsigned int g_5 = 1UL;
```

```
void func_1(void){
    int *l_8 = &g_6;
    int l_36 = 0x5E9D070FL;
    unsigned int l_107 = 0xAA37C3ACL;
    g_4 &= g_3;
    g_5++;
    int *l_102 = &l_36;
    for (g_6 = 4; g_6 < (-3); g_6 += 1);
    l_102 = &g_6;
    *l_102 = ((*l_8) && (l_107 << 7)*(*l_102));
}
```

reference semantics

```
Load g_4 1
Store g_4 0
Load g_5 1
Store g_5 2
Store g_6 4
Load g_6 4
Load g_6 4
Load g_6 4
Store g_6 1
Load g_4 0
```

gcc -O2 trace

```
Load g_5 1
Store g_4 0
Store g_6 1
Store g_5 2
Load g_4 0
```

```
void func_1(void){
    int *l_8 = &g_6;
}
```

Can match applying only correct eliminations and reorderings

reference semantics

```
Load g_4 1
Store g_4 0
Load g_5 1
Store g_5 2
Store g_6 4
Load g_6 4
Load g_6 4
Load g_6 4
Store g_6 1
Load g_4 0
```

gcc -O2 trace

```
Load g_5 1
Store g_4 0
Store g_6 1
Store g_5 2
Load g_4 0
```

```
void func_1(void){
    int *l_8 = &g_6;
}
```

Can match applying only correct eliminations and reorderings

reference semantics

```
RaW* Load g_4 1
     Store g_4 0
RaW* Load g_5 1
     Store g_5 2
OW*  Store g_6 4
RaW* Load g_6 4
RaR* Load g_6 4
RaR* Load g_6 4
     Store g_6 1
RaW* Load g_4 0
```

gcc -O2 trace

```
Load g_5 1
Store g_4 0
Store g_6 1
Store g_5 2
Load g_4 0
```

```
void func_1(void){
    int *l_8 = &g_6;
}
```

Can match applying only correct eliminations and reorderings

reference semantics

gcc -O2 trace

```
RaW* Load g_4 1
     Store g_4 0
RaW* Load g_5 1
     Store g_5 2
OW*  Store g_6 4
RaW* Load g_6 4
RaR* Load g_6 4
RaR* Load g_6 4
     Store g_6 1
RaW* Load g_4 0
```

```
Load g_5 1
Store g_4 0
Store g_6 1
Store g_5 2
Load g_4 0
```

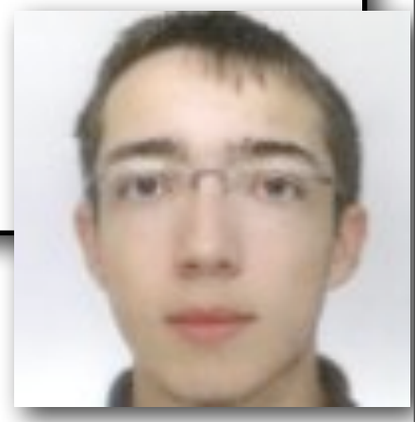
```
void func_1(void){
    int *l_8 = &g_6;
}
```

Can match applying only correct eliminations and reorderings

reference semantics

gcc -O2 trace

Need a theory of sound optimisations in the C11/C++11 memory model



```
RaW* Load g_6 4
RaR* Load g_6 4
RaR* Load g_6 4
     Store g_6 1
RaW* Load g_4 0
```

```
Store g_5 2
Load g_4 0
```

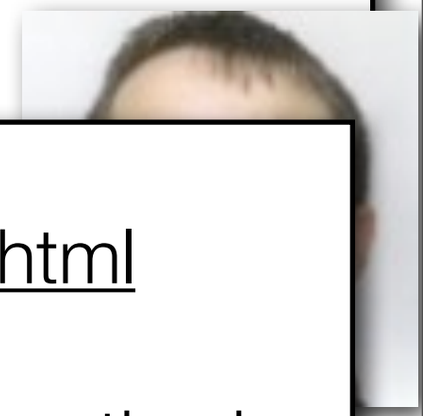
```
void func_1(void){
    int *l_8 = &g_6;
}
```

Can match applying only correct eliminations and reorderings

reference semantics

gcc -O2 trace

Need a theory of sound optimisations
in the C11/C++11 memory model



http://www.di.ens.fr/~zappa/projects/cmmtest/gcc-bugs.html

for a few subtle concurrency compiler bugs catched with this method.

```
Store g_6 1
RaW* Load g_4 0
```

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...

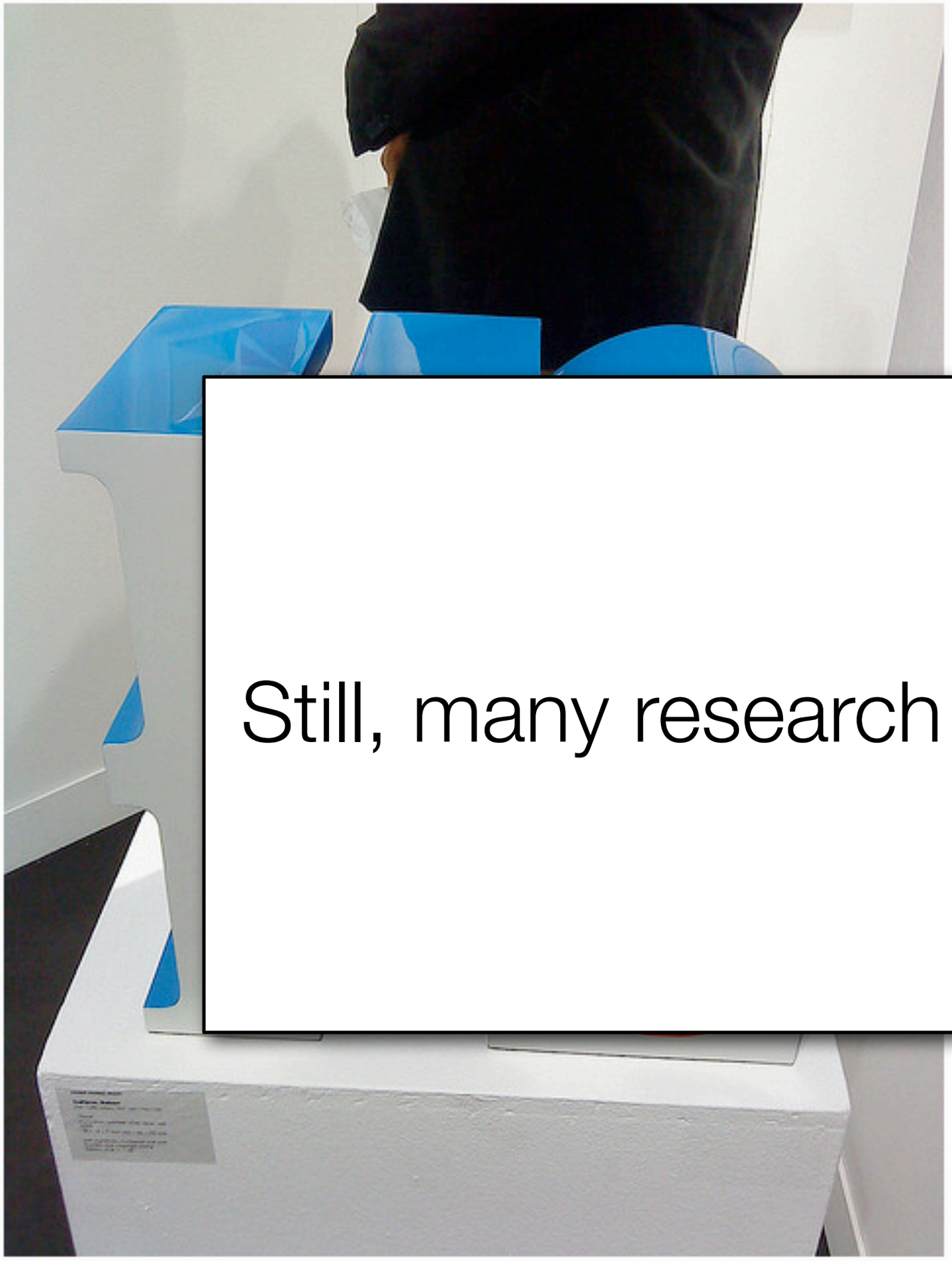
attempt

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.











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) an operational 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.

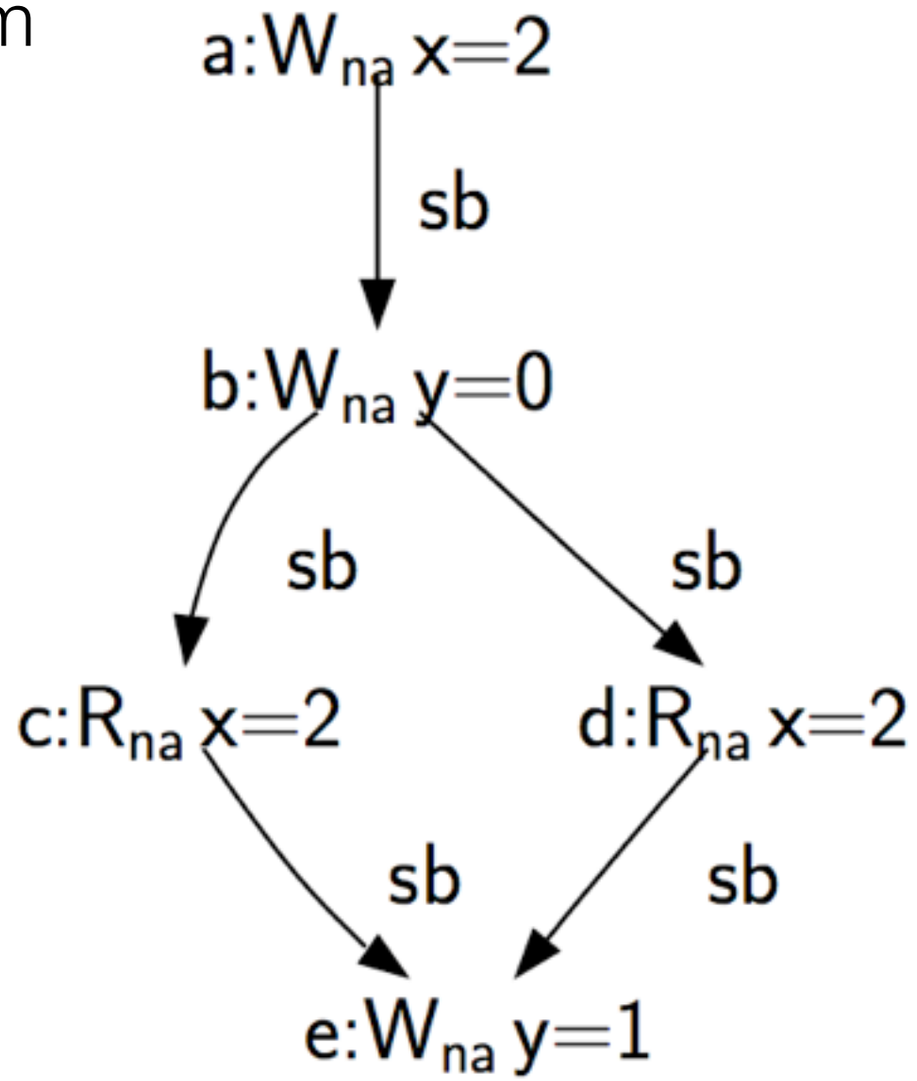
Formally

```
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

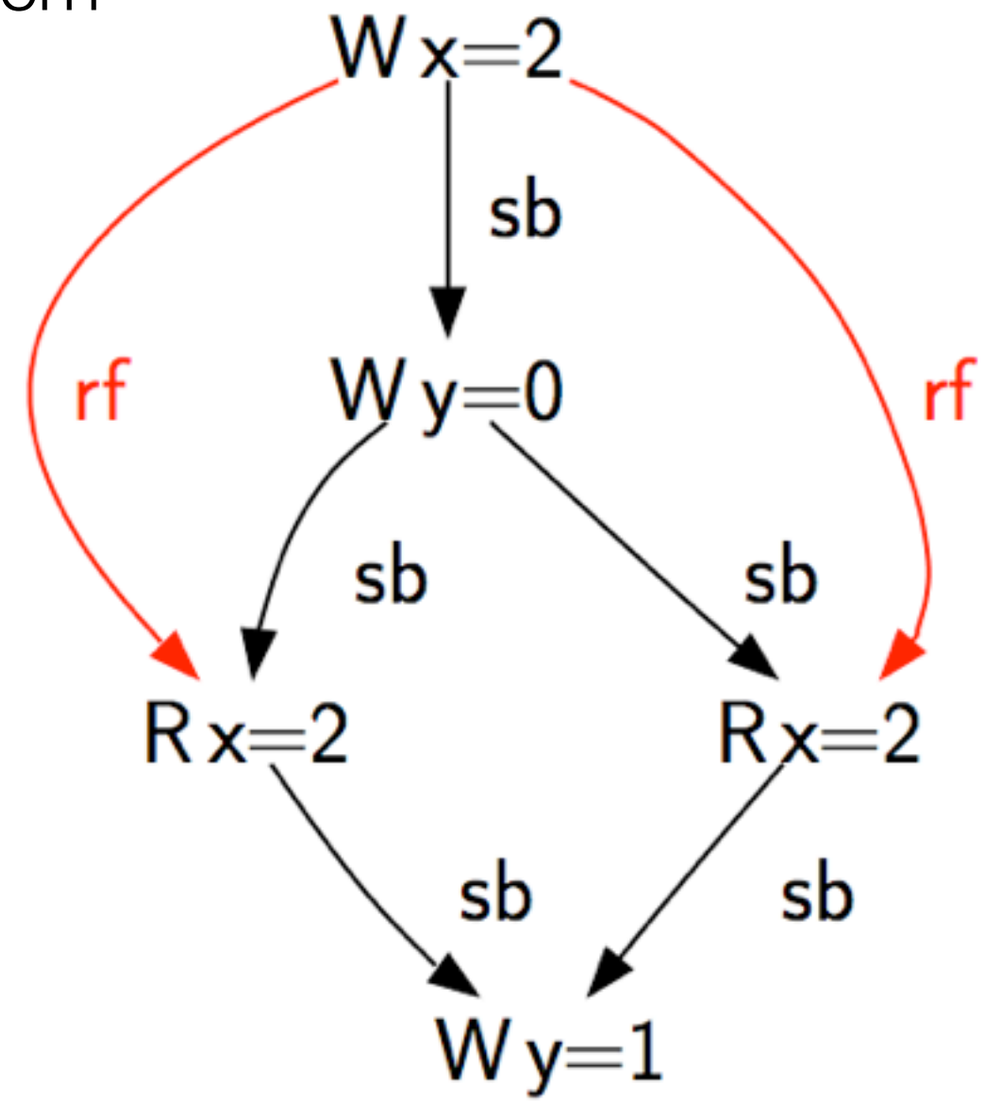
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



A single-threaded example

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

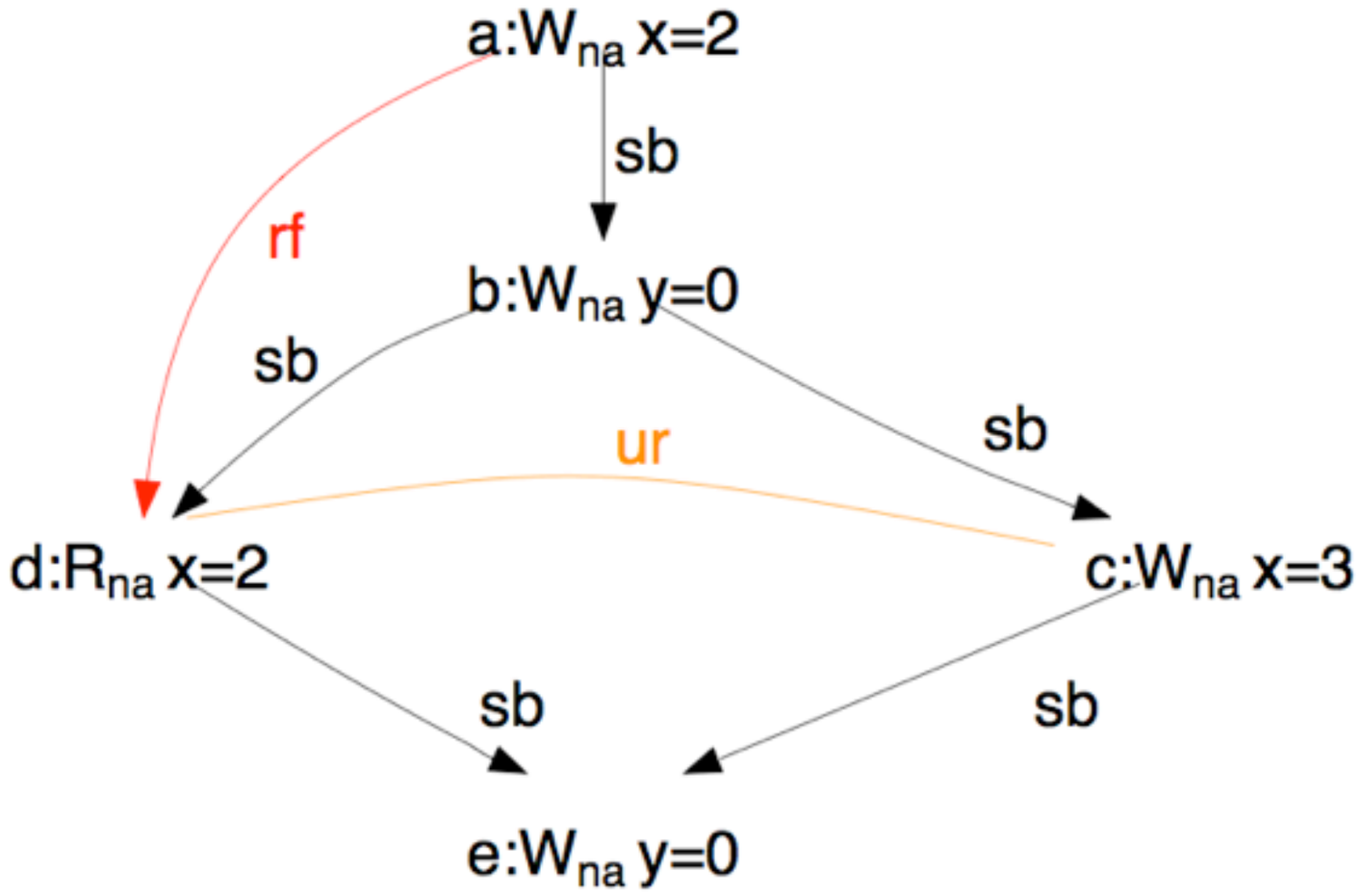
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



A single-threaded ex. with undefined behaviour

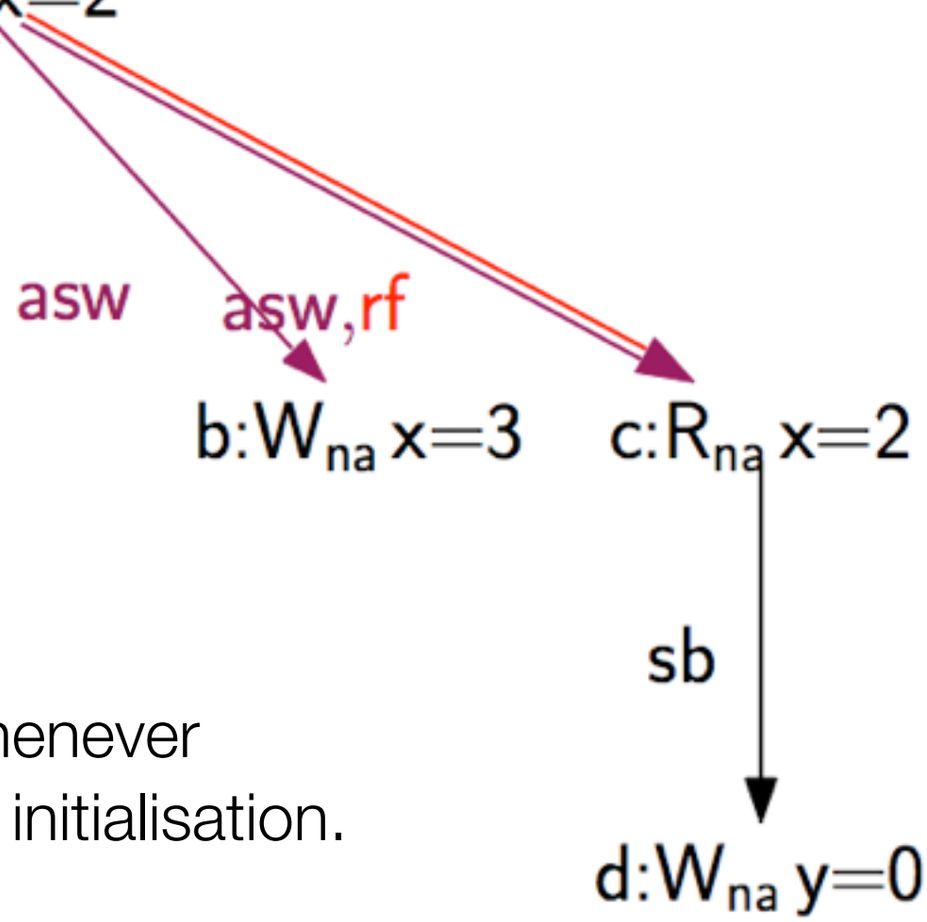
An unsequenced race.

```
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);
```



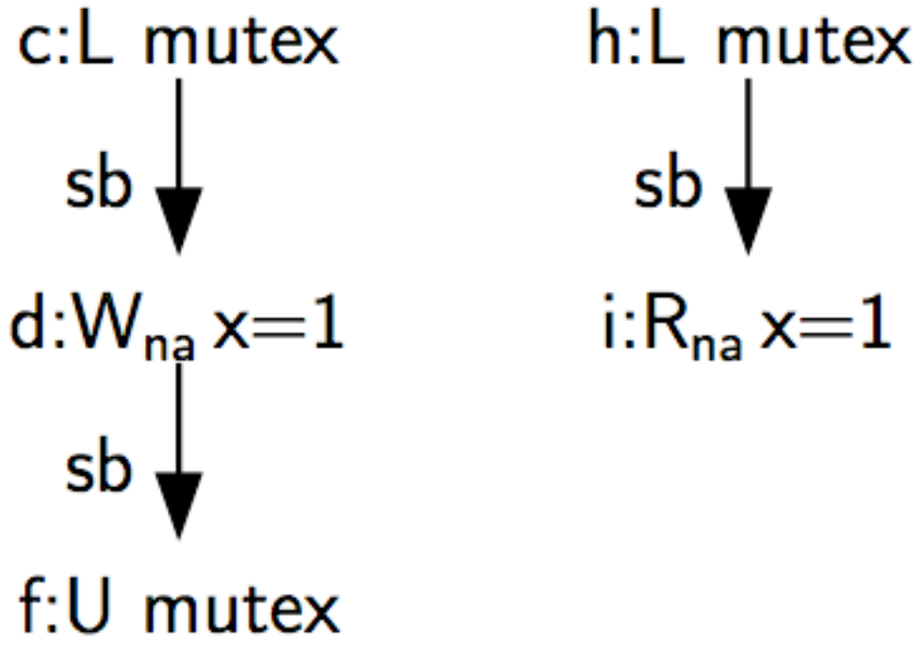
We will omit **`asw`** arrows whenever
we are not interested in the initialisation.

Locks and unlocks

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

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

1. the operational semantics defines the sb arrows

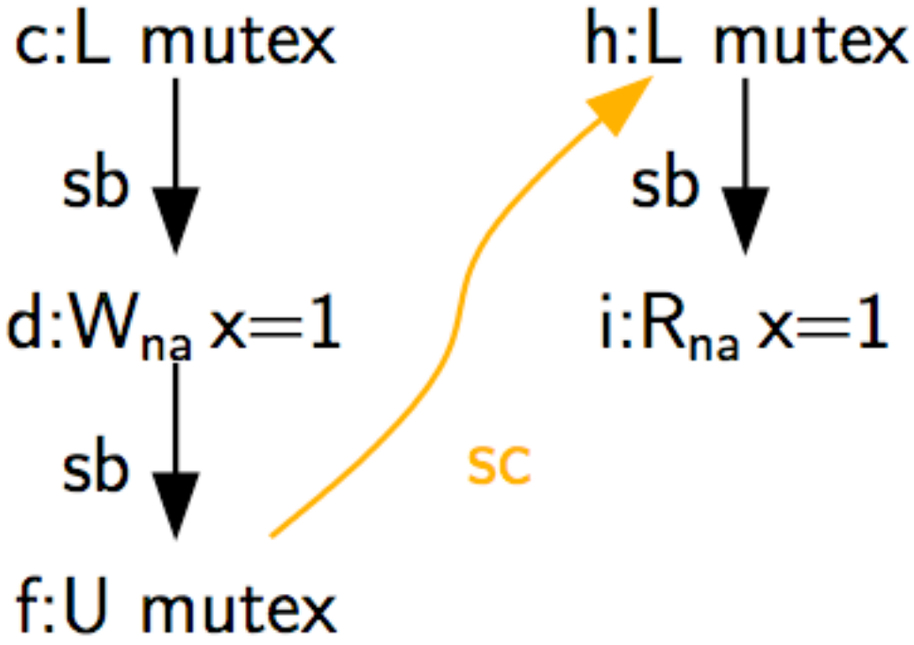


Locks and unlocks

```
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)

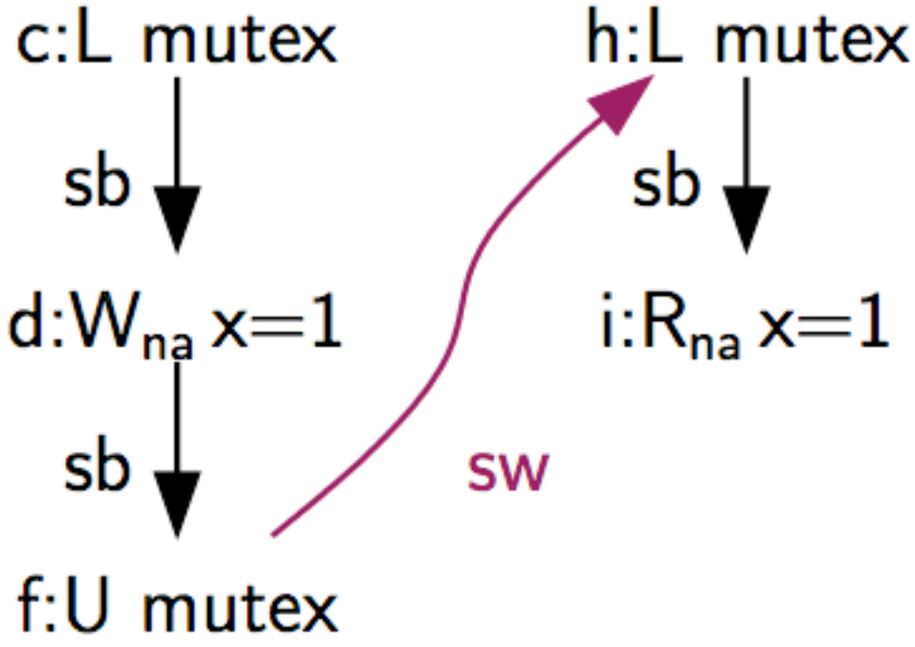


Locks and unlocks

```
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



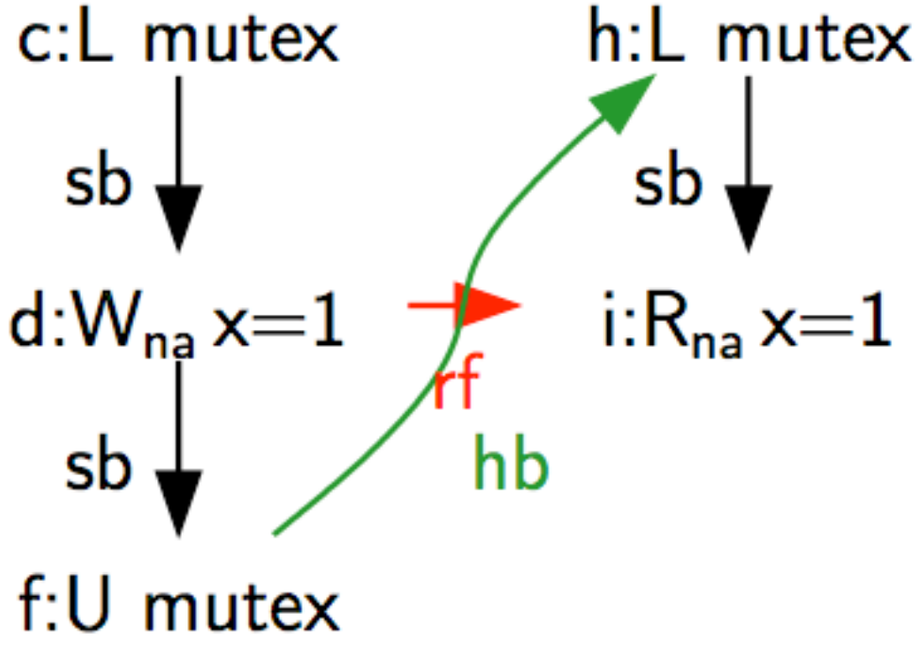
Locks and unlocks

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

```
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

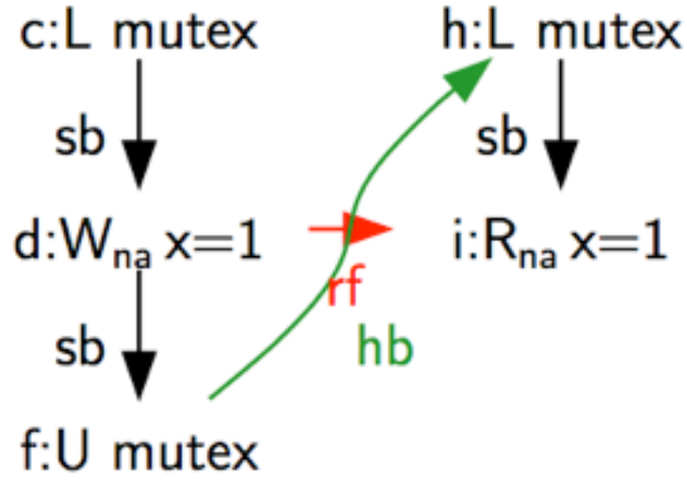
The *happens before* relation is key to the model:

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.

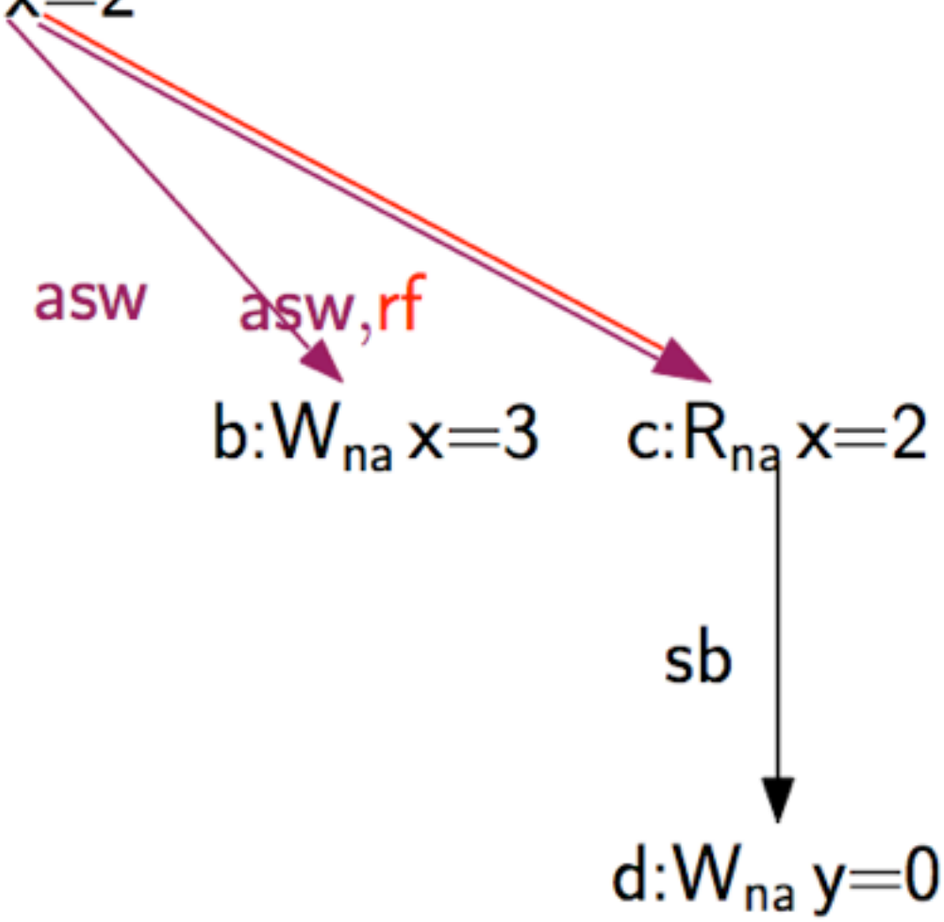
3. data races are defined as an absence of happens before
   between conflicting actions.

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



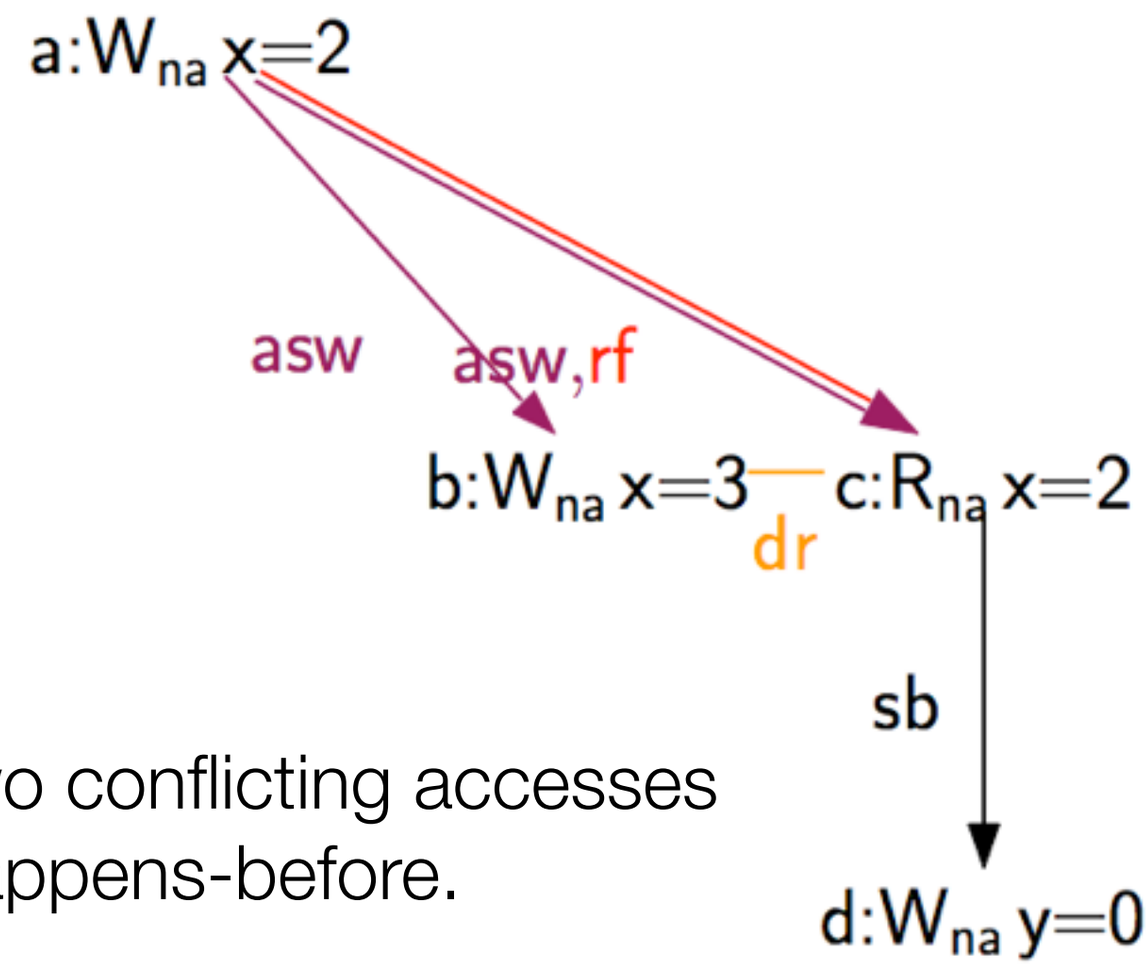
A data race

```
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

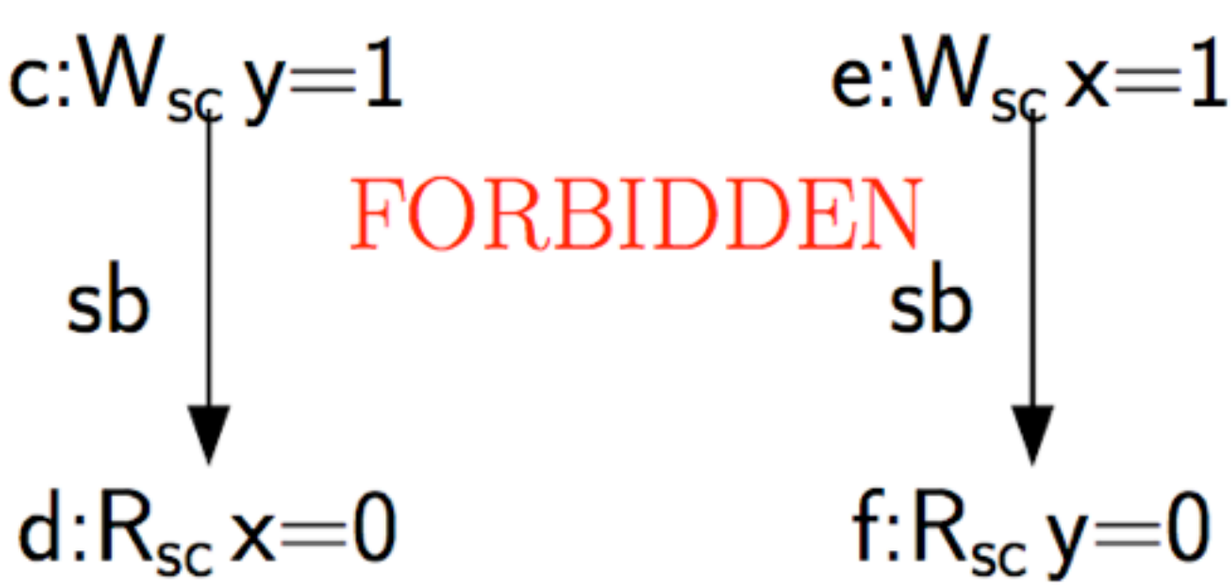
$$
\begin{aligned}
&\text{let } \mathit{data\_races}\ \mathit{actions}\ \mathit{hb} = \\
&\quad \{\ (a,\ b) \mid \forall\ a{\in}\mathit{actions}\ b{\in}\mathit{actions} \mid \\
&\qquad \neg\ (a = b) \wedge \\
&\qquad \text{same\_location}\ a\ b \wedge \\
&\qquad (\text{is\_write}\ a\ \vee\ \text{is\_write}\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

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);
```

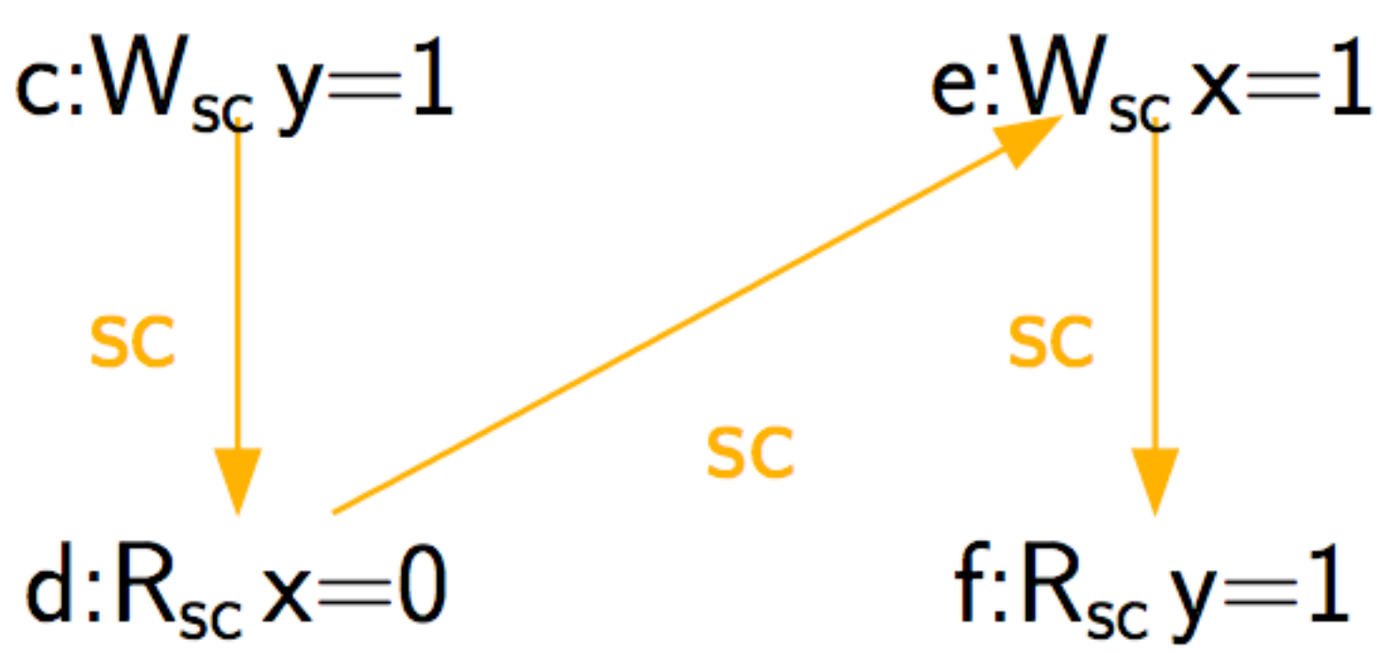


Why is this behaviour forbidden?

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);
```

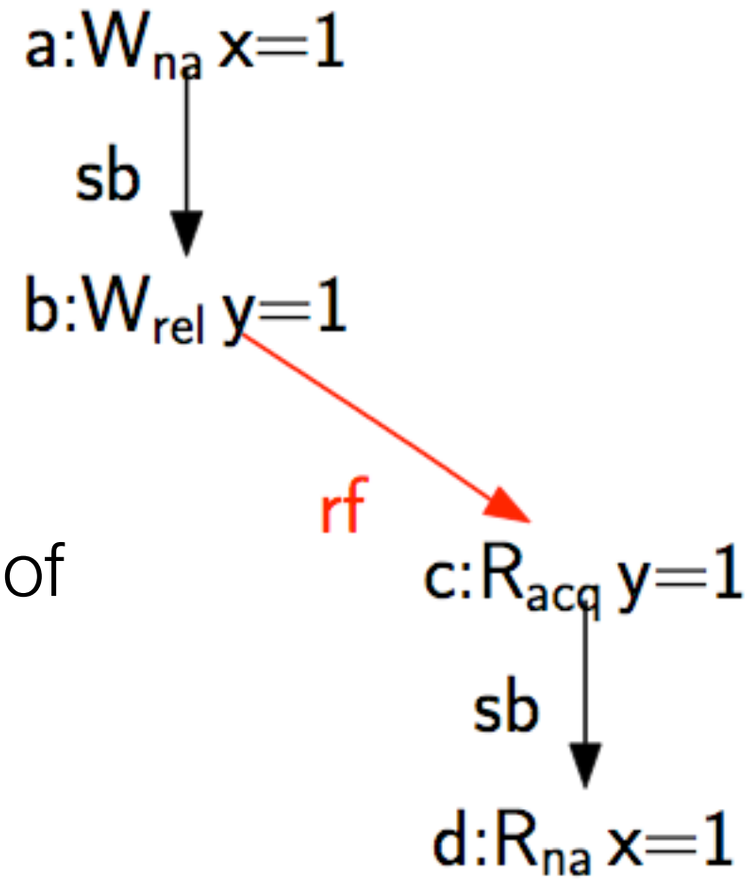


The **`sc`** relation must define a total order over unlocks/locks and **`seq_cst`** accesses… **`sc`** is included in **`hb`**, an **`rf`** must respect **`hb`**.

Expert concurrency: the release-acquire idiom

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

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

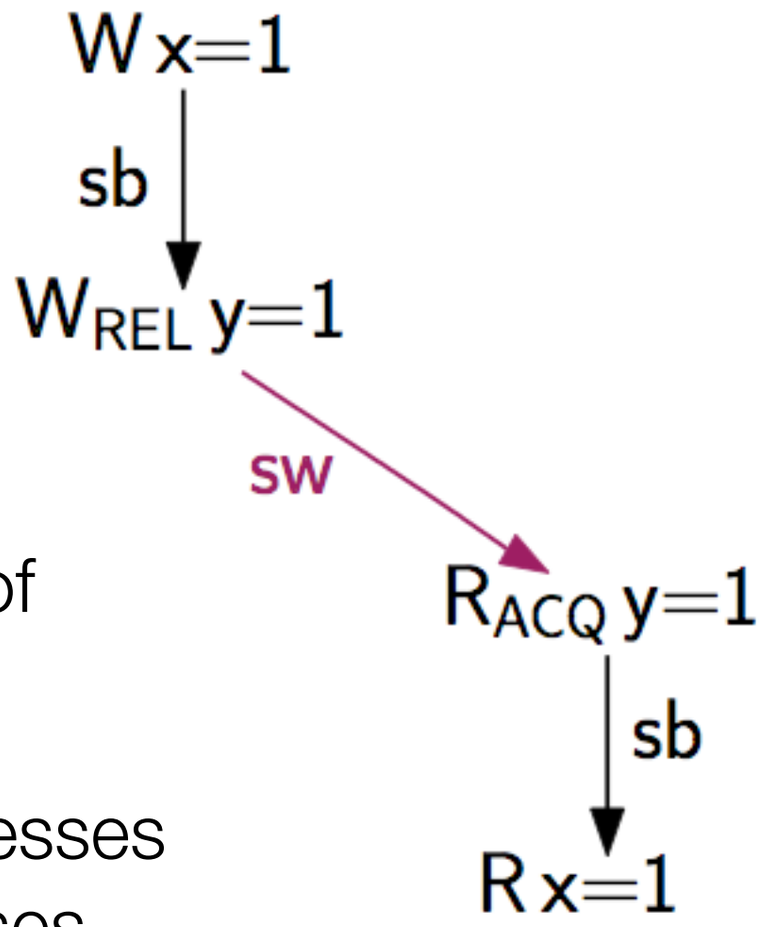


Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

Expert concurrency: the release-acquire idiom

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

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



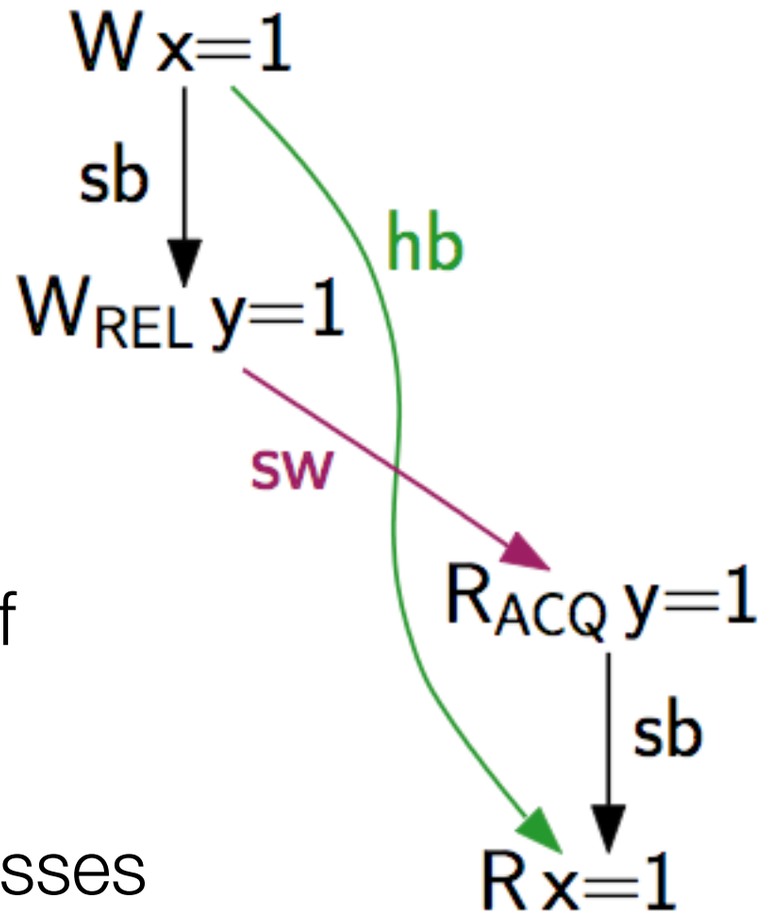
Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

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;
```



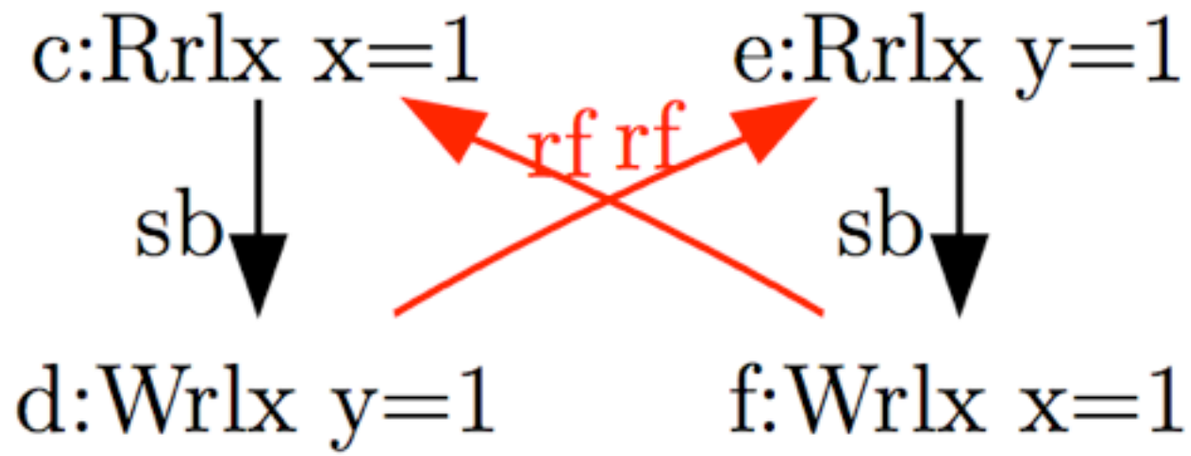
Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

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

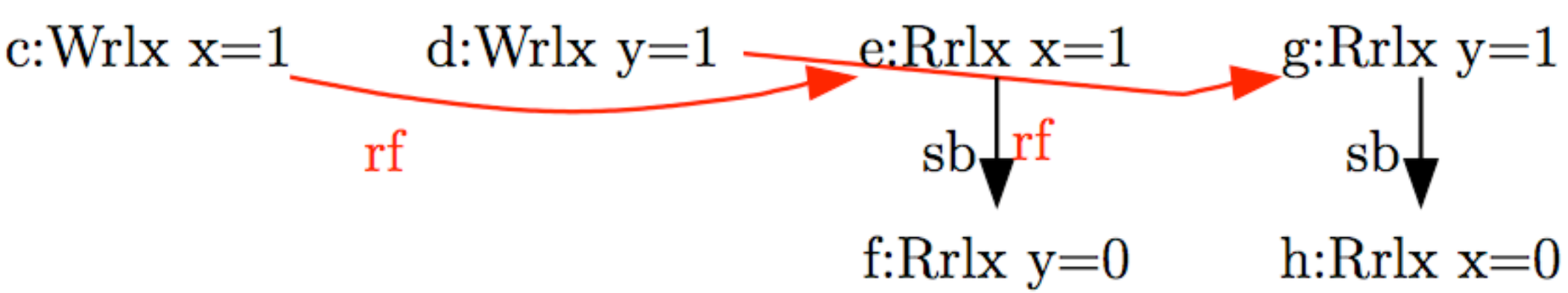
```
x.load(relaxed);          | y.load(relaxed);
y.store(1, relaxed);      | x.store(1, relaxed);
```



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

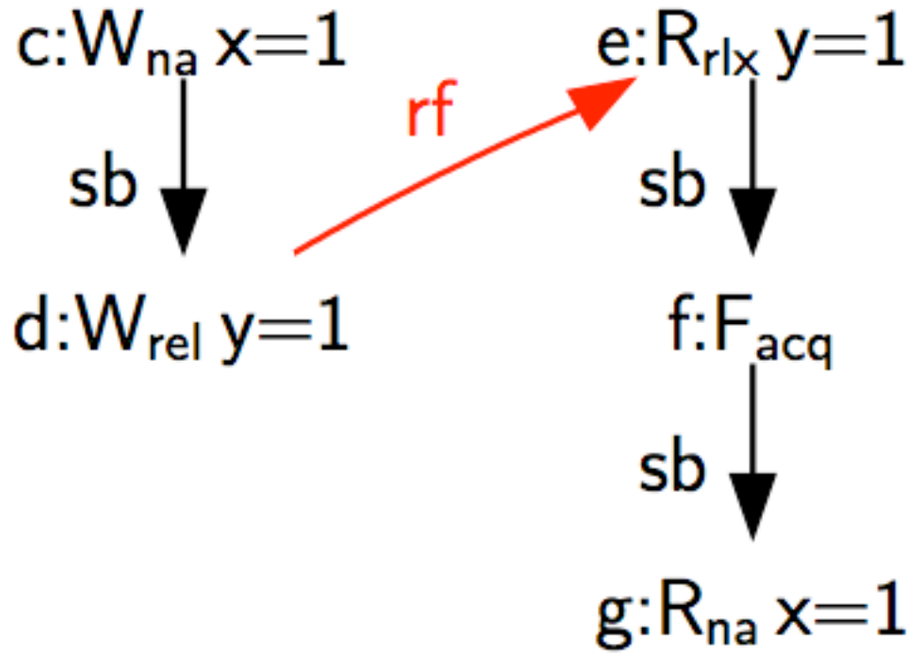
```
// sender                          | // receiver
x = ...                            | while (0 == y.load(relaxed));
y.store(1, release);               | fence(acquire);
                                   | r = x;
```

Expert concurrency: fences avoid excess sync.

```
// sender
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.

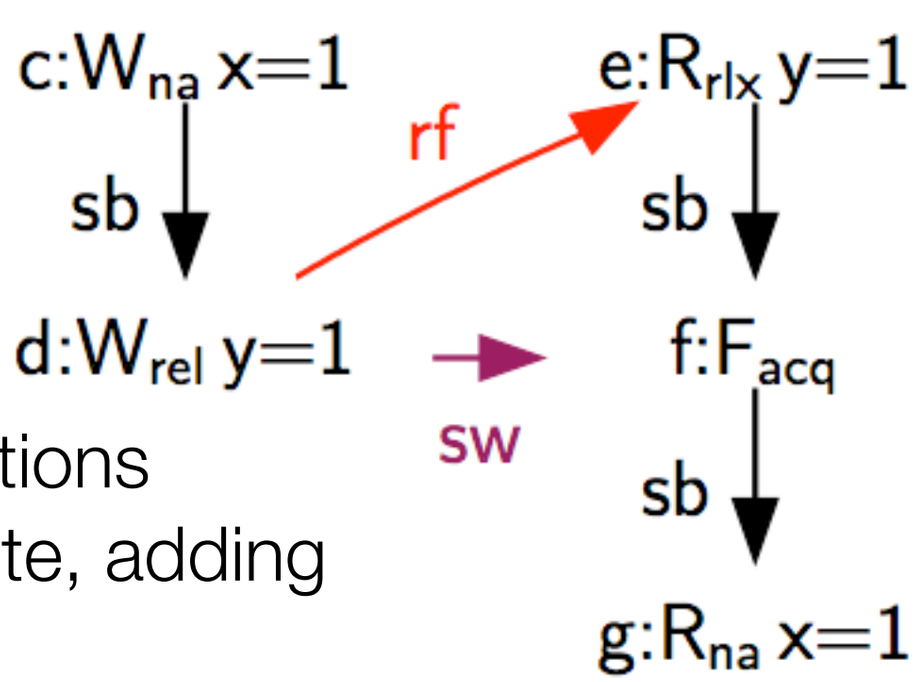


Expert concurrency: fences avoid excess sync.

```
// 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.



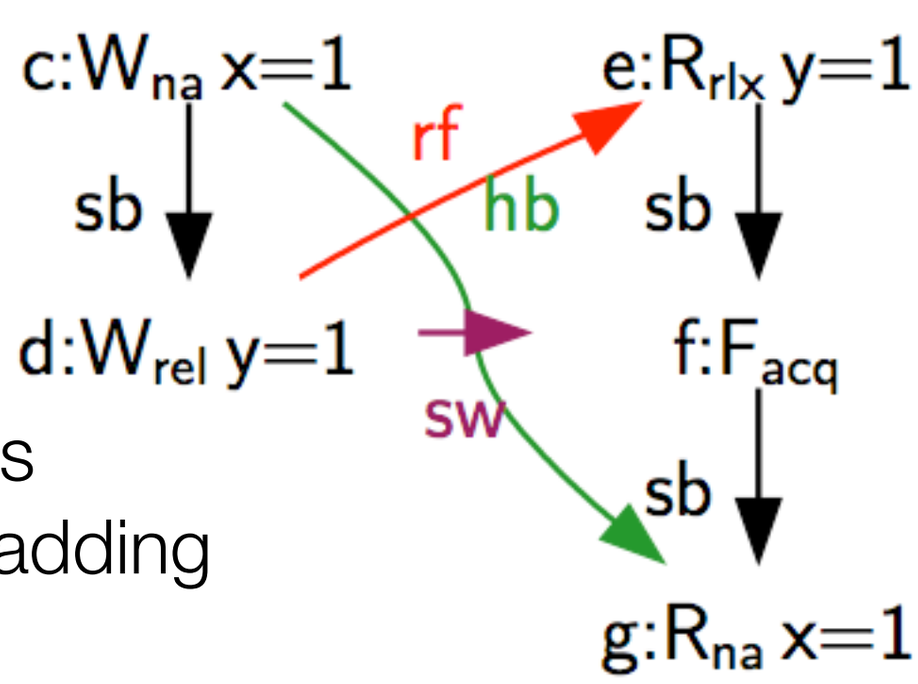
The acquire fence follows the **`sb`**/**`rf`** relations looking for the corresponding release write, adding a **`sw`** arrow.

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.

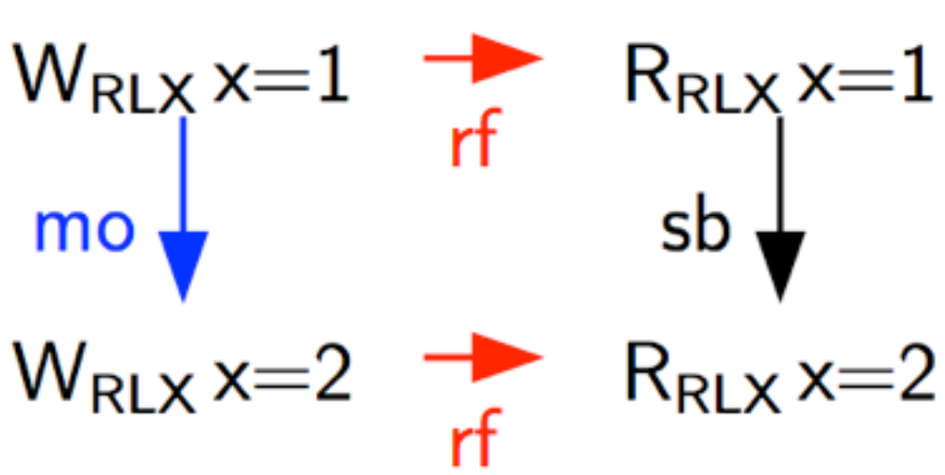


The acquire fence follows the **`sb`**/**`rf`** relations looking for the corresponding release write, adding a **`sw`** arrow.

Happens-before follows as usual...

Modification order

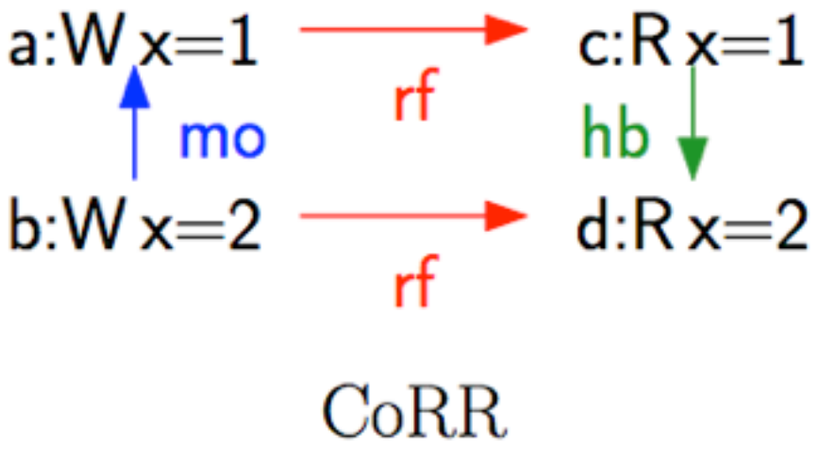
```
   atomic_int x = 0;
x.store(1, relaxed); ||  x.load(relaxed);
x.store(2, relaxed); ||  x.load(relaxed);
```



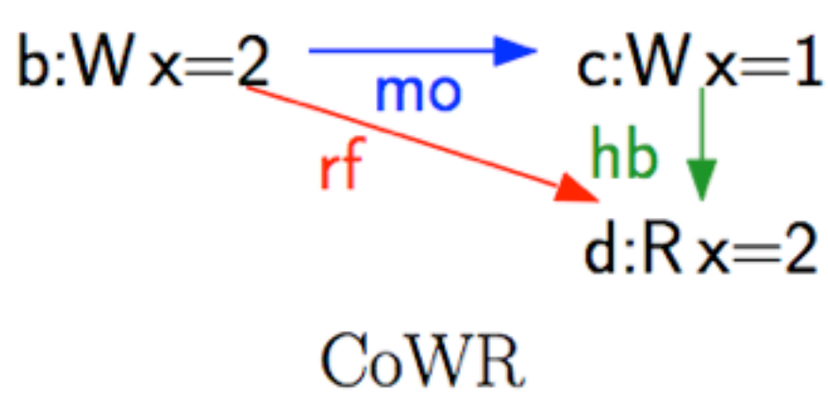
Modification order is a total order over atomic writes of any memory order.

Coherence and atomic reads

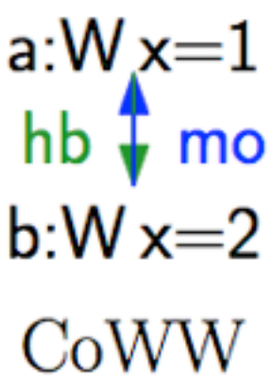
All forbidden:



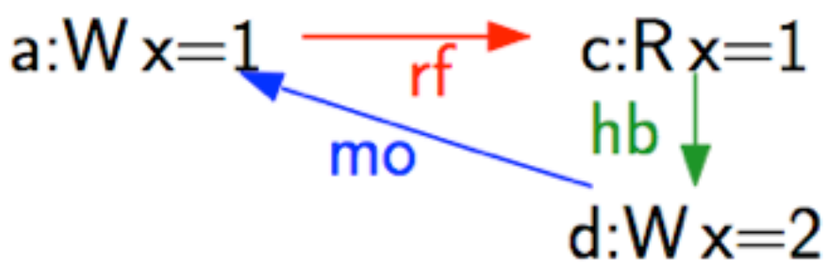
CoRR



CoWR



CoWW



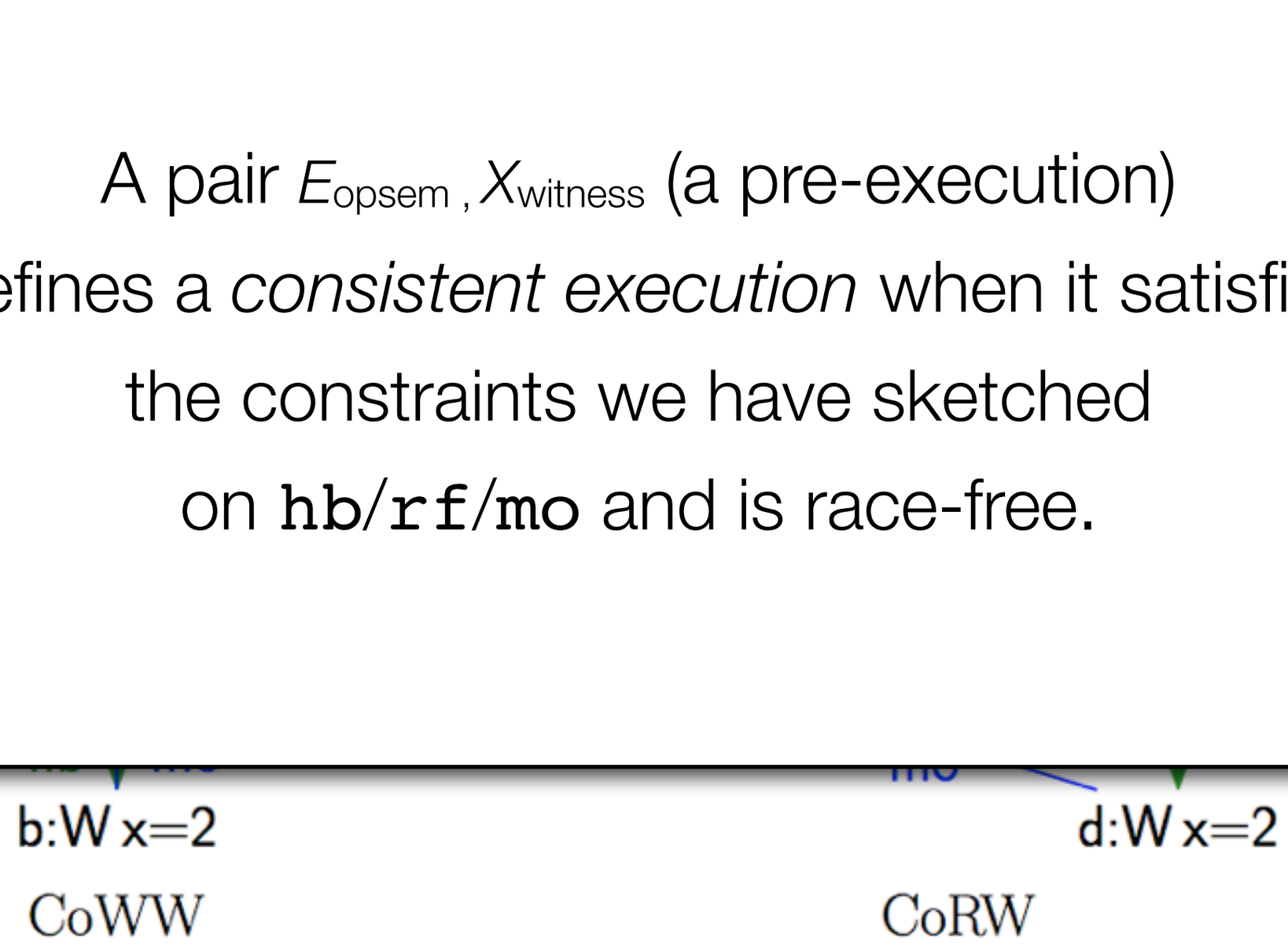
CoRW

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

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

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

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

$\xrightarrow{r}{}^{=} = r$

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

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

$\xrightarrow{rel}{}^{=}_{|s} = rel \cap (s \times s)$

$rel^{=}_{|s} = rel \cap (s \times s)$

$\xrightarrow{rel}{}^{?}_{|s} = rel \cap (s \times s)$

$rel^{?}_{|s} = rel \cap (s \times s)$

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

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

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

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

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

well_founded r = *wf* r

type_abbrev action_id : string

type_abbrev thread_id : string

type_abbrev location : string

type_abbrev val : string

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

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

(action_id_of (Lock *aid* _ _) = *aid*) ∧
(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* = *lkd*)
  | 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 $\subseteq$ *sequenced-before*

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

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

consistent_locks = consistent_locks =
  ∀*l* ∈ locations_of *actions*. (*location-kind l* = Mutex) ⟹ (
  let *lock_unlock_actions* =
    all_lock_or_unlock_actions_at (Some *l*)*actions* **in**
  let *lock_order* = $\xrightarrow{sc}|_{lock\_unlock\_actions}$ **in**
  (* 30.4.1:5 - The implementation shall serialize those (lock and unlock) operations. *)
  strict_total_order_over *lock_unlock_actions lock_order* ∧

  (* 30.4.1:1 A thread owns a mutex from the time it successfully calls one of the lock functions until it calls unlock.*)
  (* 30.4.1:20 Requires: The calling thread shall own the mutex. *)
  (* 30.4.1:21 Effects: Releases the calling threads ownership of the mutex.*)
  (∀*a*$_L$ ∈ *lock_unlock_actions*. is_unlock *a*$_L$ ⟹
    (∃*a*$_l$ *p*. *lock_order*
      a$_l$ $\xrightarrow{lock\_order}$ *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*$_u$ ∈ *lock_unlock_actions*.
      a$_u$ $\xrightarrow{lock\_order}$ *a*$_l$ ⟹ is_unlock *a*$_u$)))

rs_element *rs_head a* =
  same_thread *a rs_head* ∨ is_atomic_rmw *a*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

carries_a_dependency_to = *a* $\xrightarrow{carries\text{-}a\text{-}dependency\text{-}to}$ *b* =
  a $(($\xrightarrow{sequenced\text{-}before}$ ∩ $\xrightarrow{rf}$$)$ ∪ $\xrightarrow{data\text{-}dependency}$$)^+$ *b*

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

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

dependency_ordered_before = *a* $\xrightarrow{dependency\text{-}ordered\text{-}before}$ *d* =
  a ∈ *actions* ∧ *d* ∈ *actions* ∧
  (∃*b*. is_release *a* ∧ is_consume *b* ∧
    (∃*a*$_2$. *a* $\xrightarrow{release\text{-}sequence}$ *a*$_2$ $\xrightarrow{rf}$ *b*) ∧
    (*b* $\xrightarrow{carries\text{-}a\text{-}dependency\text{-}to}$ *d* ∨ (*b* = *d*)))

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

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

simple_happens_before = $\xrightarrow{simple\_happens\_before}$ =
  $($$\xrightarrow{sequenced\text{-}before}$ ∪ $\xrightarrow{synchronize\text{-}with}$$)^+$

consistent_simple_happens_before shb =
  irreflexive ($\xrightarrow{shb}$$^+$)

inter_thread_happens_before = $\xrightarrow{inter\text{-}thread\text{-}happens\text{-}before}$ =
  let *r* = $\xrightarrow{synchronize\text{-}with}$ ∪
    $\xrightarrow{dependency\text{-}ordered\text{-}before}$ ∪
    ($\xrightarrow{synchronize\text{-}with}$ ∘ $\xrightarrow{sequenced\text{-}before}$) **in**
  ($\xrightarrow{r}$ ∪ ($\xrightarrow{sequenced\text{-}before}$ ∘ $\xrightarrow{r}$))$^+$

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

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

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

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

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

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

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

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

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

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

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

consistent_reads_from_mapping = consistent_reads_from_mapping =
  (∀*b*. (is_read *b* ∧ is_at_atomic_location *b*) ⟹
    (**if** (∃*a*$_{sse}$. *a*$_{sse}$ $\xrightarrow{visible\text{-}side\text{-}effect}$ *b*)
    then (∃*a*$_{sse}$. *a*$_{sse}$ $\xrightarrow{visible\text{-}side\text{-}effect}$ *b* ∧ *a*$_{sse}$ $\xrightarrow{rf}$ *b*)
    else ¬(∃*a*. *a* $\xrightarrow{rf}$ *b*))) ∧

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

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

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

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

  (* –Fence restrictions– *)

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

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

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

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

consistent_control_dependency = consistent_control_dependency =
  irreflexive (($\xrightarrow{control\text{-}dependency}$ ∪ $\xrightarrow{all_data_dependency}$$)^+$)

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

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

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

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

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

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

Is C++11 hopelessly complicated?

Programmers cannot be given this model.

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

- Can we compile to x86?

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

- Can we compile to Power?

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

Is C++11 hopelessly complicated?

Simplifications:

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

Derivative models:

- without consume, happens-before is transitive

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

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

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

The current state of the standard

Fixed:

- in some cases, happens-before was cyclic

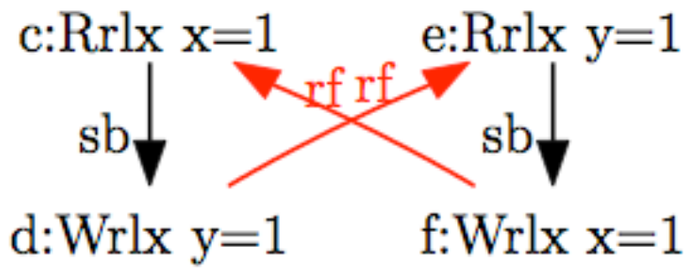
- coherence

- **`seq_cst`** atomics were more broken

Not fixed:

- self satisfying conditional

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



- **`seq_cst`** atomics are still not SC