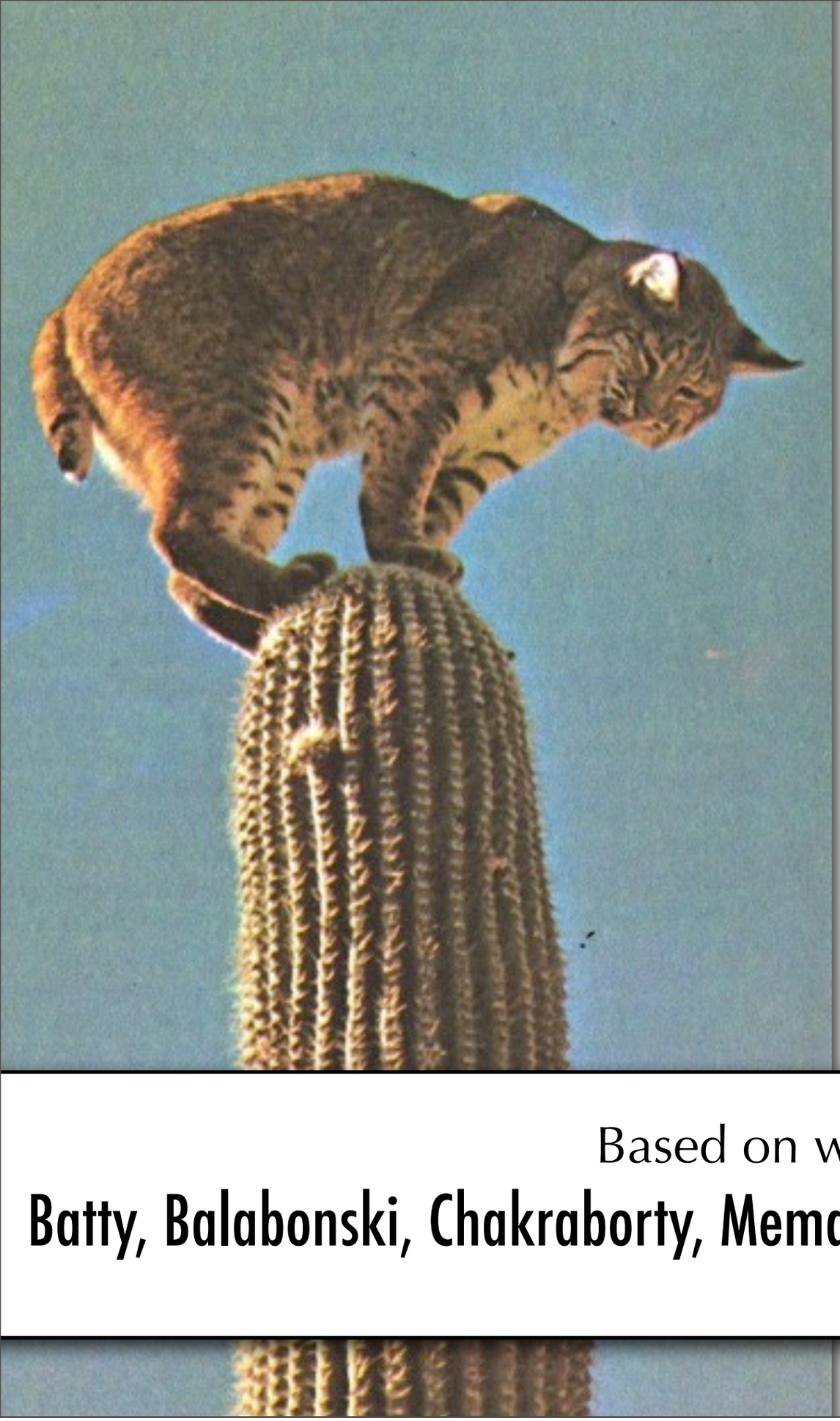


Languages and concurrency

a thorny relationship

Francesco Zappa Nardelli

Inria, France

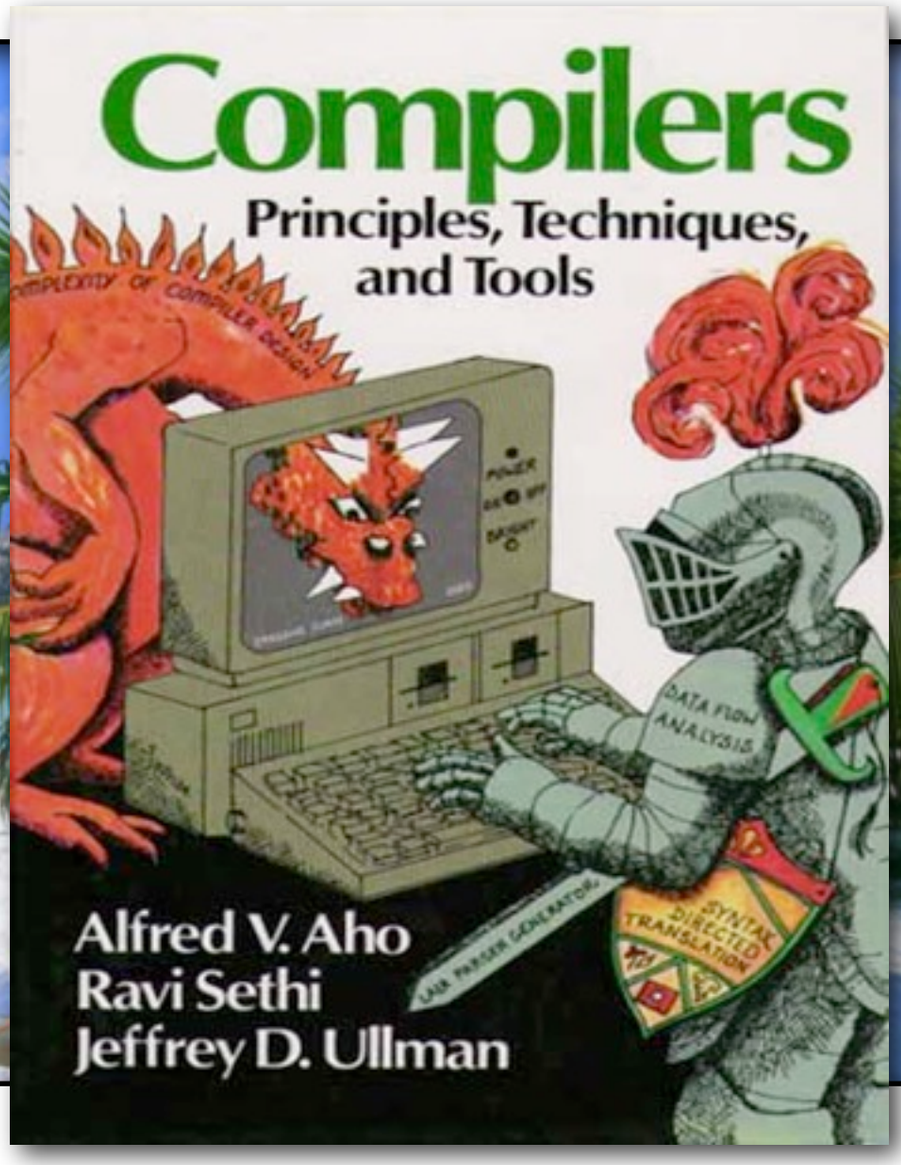
Based on work *done by* or *with*

Batty, Balabonski, Chakraborty, Memariam, Morisset, Owens, Sevcik, Sewell, Vafeiadis

U. Cambridge, U. Kent, MPI-SWS and Inria

Imagine an ideal world

Imagine an ideal world



Programmers and compilers cooperate
to make great software

Constant propagation

A simple, and *innocuous*, optimisation:

Source code

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

Optimised code

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

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

Shared memory concurrency

Shared memory

```
x = y = 0
```

Thread 1

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

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

Thread 2

Shared memory concurrency

Shared memory

```
x = y = 0
```

Thread 1

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

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

Thread 2

Intuitively this program always prints 0

Shared memory concurrency

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

```
x = y = 0
```

Thread 1

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

Shared memory concurrency

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

```
x = y = 0
```

Thread 1

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

(with `print x` struck through and replaced by `print 1`)

Thread 2

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

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

Shared memory concurrency

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

```
x = y = 0
```

An optimising compiler can break your code

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

Lazy initialisation (even an unoptimising compiler breaks your program)

Deferring an object's initialisation util first use: a big win if an object is never used (e.g. device drivers code). Compare:

```
int x = computeInitValue();  // eager initialization
…                            // clients refer to x
```

with:

```
int xValue() {
   static int x = computeInitValue(); // lazy initialization
   return x;
} ...                                 // clients refer to xValue()
```

The singleton pattern

Lazy initialisation is a pattern commonly used. In C++ you would write:

```
class Singleton {
public:
  static Singleton *instance (void) {
    if (instance_ == NULL)
      instance_ = new Singleton;
    return instance_;
  }
…                                  // other methods omitted
private:
  static Singleton *instance_;  // other fields omitted
};


…
Singleton::instance () -> method ();
```

But this code is not thread safe! Why?

Making the singleton pattern thread safe

A simple thread safe version:

```
class Singleton {
public:
  static Singleton *instance (void) {
     Guard<Mutex> guard (lock_); // only one thread at a time
     if (instance_ == NULL)
        instance_ = new Singleton;
     return instance_;
  }
private:
  static Mutex lock_;
  static Singleton *instance_;
};
```

Every call to instance must acquire and release the lock: excessive overhead.

Obvious (broken) optimisation

```
class Singleton {
public:
  static Singleton *instance (void) {
     if (instance_ == NULL) {
        Guard<Mutex> guard (lock_); // lock only if unitialised
        instance_ = new Singleton; }
     return instance_;
  }

private:
  static Mutex lock_;
  static Singleton *instance_;
};
```

Clever programmers use double-check

```
class Singleton {
public:
  static Singleton *instance (void) {
     // First check
     if (instance_ == NULL) {
        // Ensure serialization
        Guard<Mutex> guard (lock_);
        // Double check
        if (instance_ == NULL)
           instance_ = new Singleton;
     }
     return instance_;
  }
private: [..]
};
```

Idea: re-check that the Singleton has not been created after acquiring the lock.

Double-check locking: clever but broken

```
instance_ = new Singleton;
```

does three things:

1) allocate memory

2) construct the object

3) assign to `instance_` the address of the memory

Not necessarily in this order! For example:

```
instance_ =                                  // 3
  operator new(sizeof(Singleton));           // 1
new (instance_) Singleton                    // 2
```

If this code is generated, the order is 1,3,2.

Broken...

```
if (instance_ == NULL) {                          // Line 1
   Guard<Mutex> guard (lock_);
   if (instance_ == NULL) {
     instance_ =
        operator new(sizeof(Singleton));    // Line 2
     new (instance_) Singleton; }}
```

Thread 1:

executes through Line 2 and is suspended; at this point, `instance_` is non-NULL, but no singleton has been constructed.

Thread 2:

executes Line 1, sees `instance_` as non-NULL, returns, and dereferences the pointer returned by `Singleton` (i.e., `instance_`).

Broken...

Problem

```
instance_ =                                  // 3
  operator new(sizeof(Singleton));           // 1
new (instance_) Singleton                    // 2
```

We need a way to specify that step 3 come after steps 1 and 2.

There is no way to specify this in C++

Similar examples can be built for any programming language...

That pesky hardware (1)

Consider misaligned 4-byte accesses:

`int32_t a = 0`

`a = 0x44332211`	`if (a == 0x00002211)` `  print "error"`

(*Disclaimer*: compiler will normally ensure alignment)

Intel SDM x86 atomic accesses:

- *n*-bytes on an *n*-byte boundary (*n* = 1,2,4,16)
- P6 or later: … or if unaligned but within a cache line

Question: what about *multi-word high-level language values*?

That pesky hardware (2)

Hardware optimisations can be observed by concurrent code:

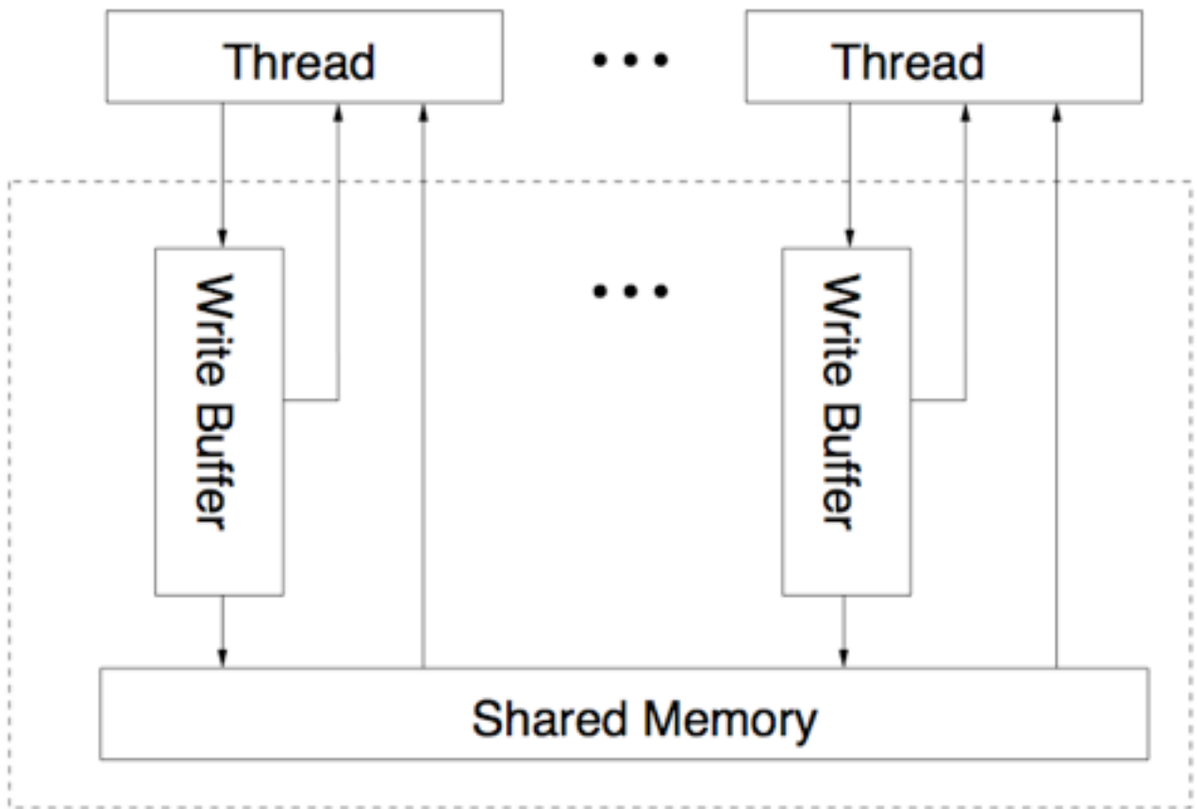
Thread 0	Thread 1
`x = 1`	`y = 1`
`print y`	`print x`

At the end of some executions:

0 0

is printed on the screen,

both on x86 and Power/ARM



That pesky hardware (2)

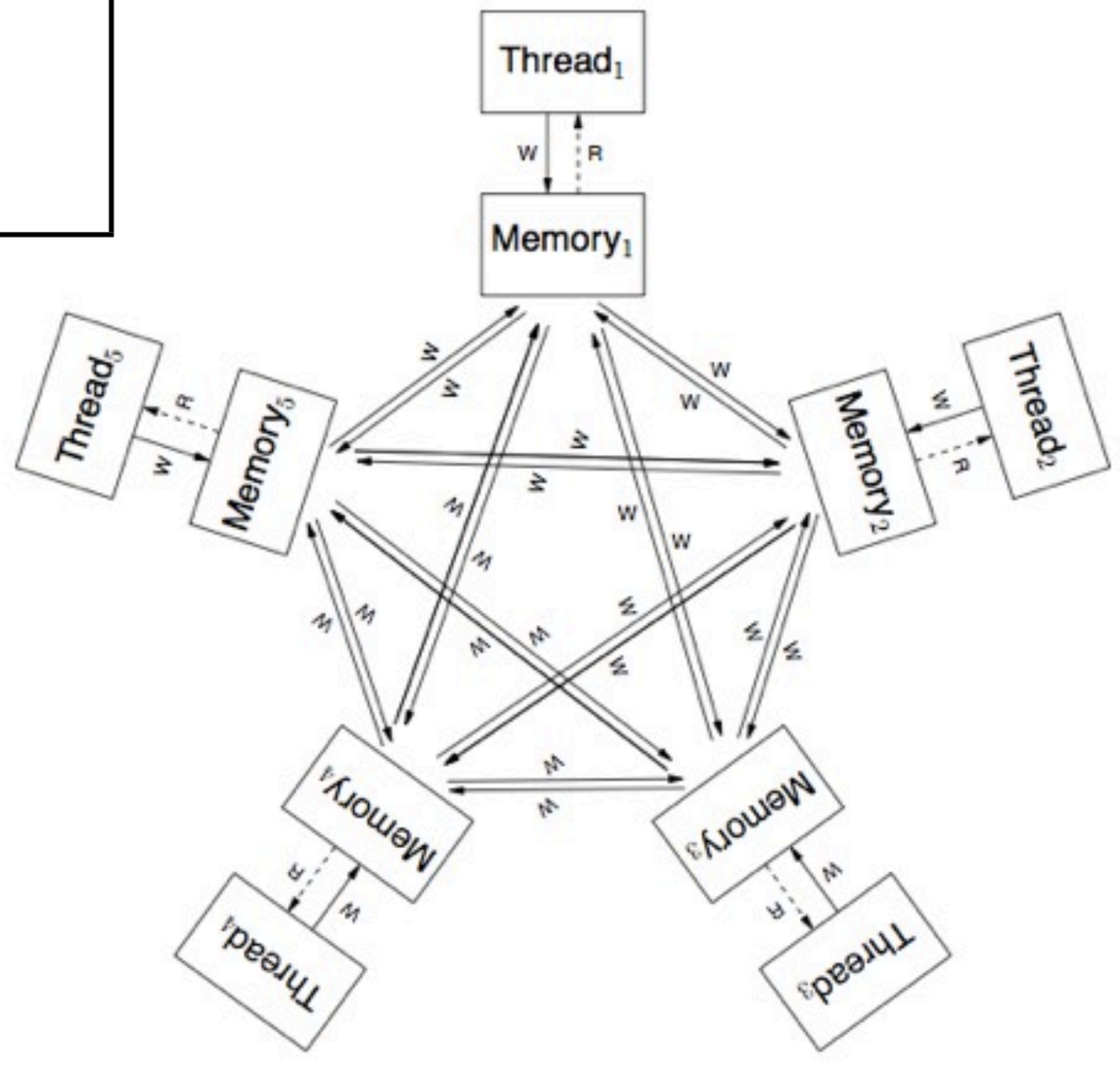
...and differ between architectures...

Thread 0	Thread 1
`x = 1`	`print y`
`y = 1`	`print x`

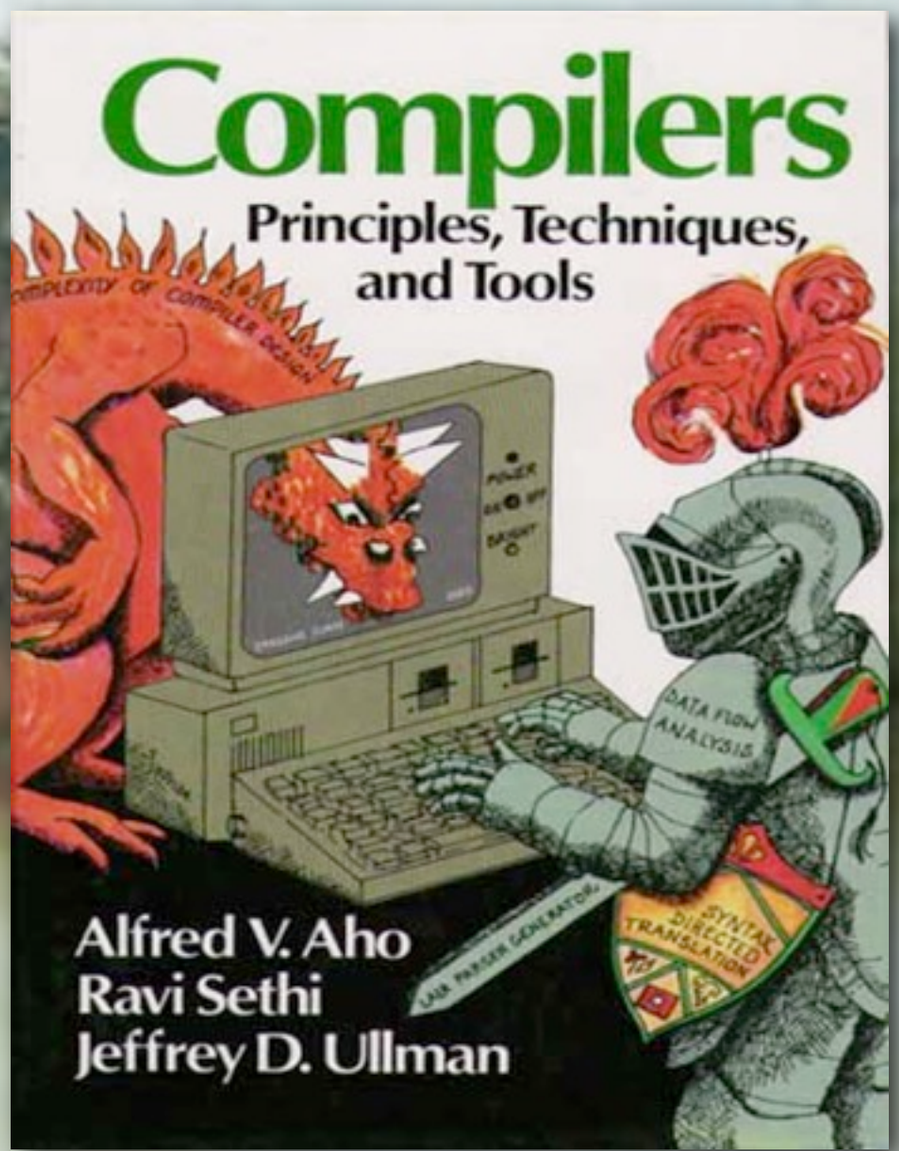
At the end of some executions:

1 0

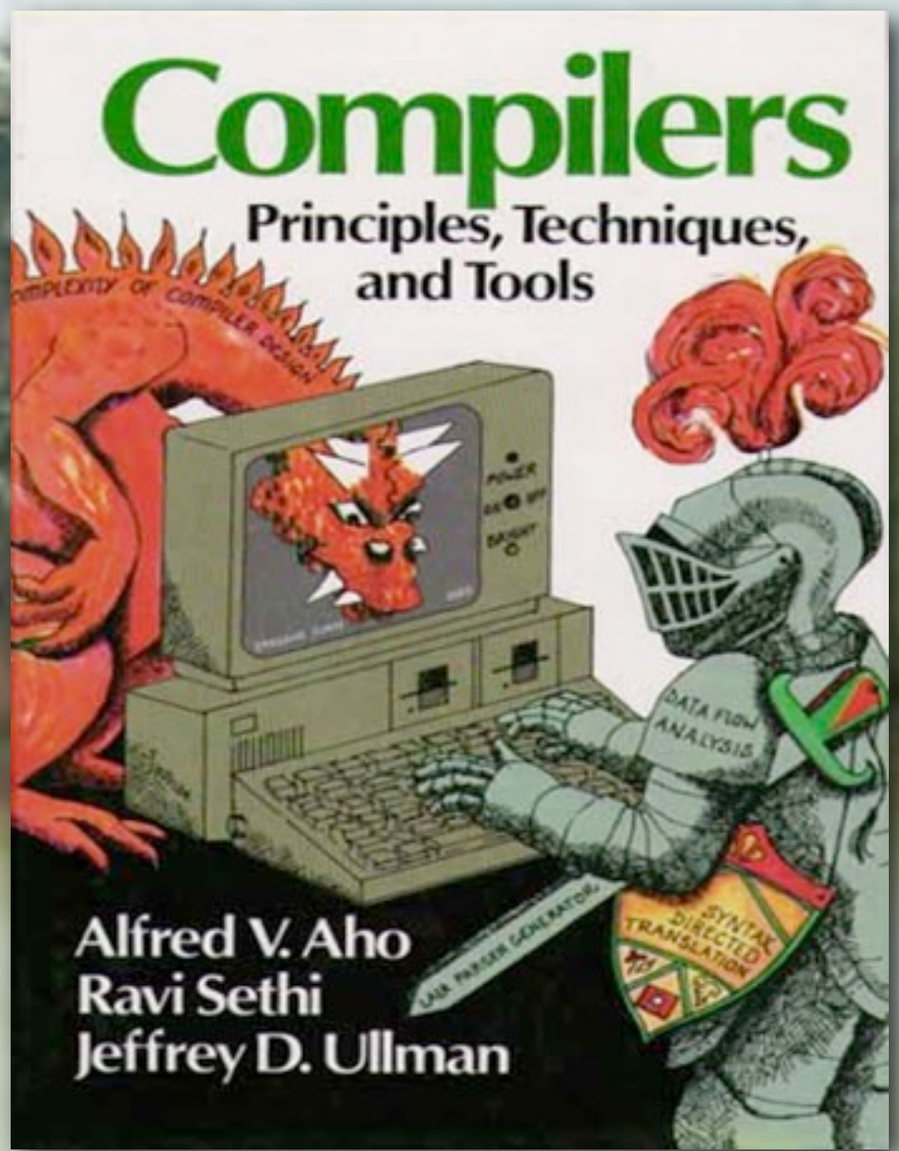
is printed on the screen
on Power/ARM but not on x86.



The fundamental problem

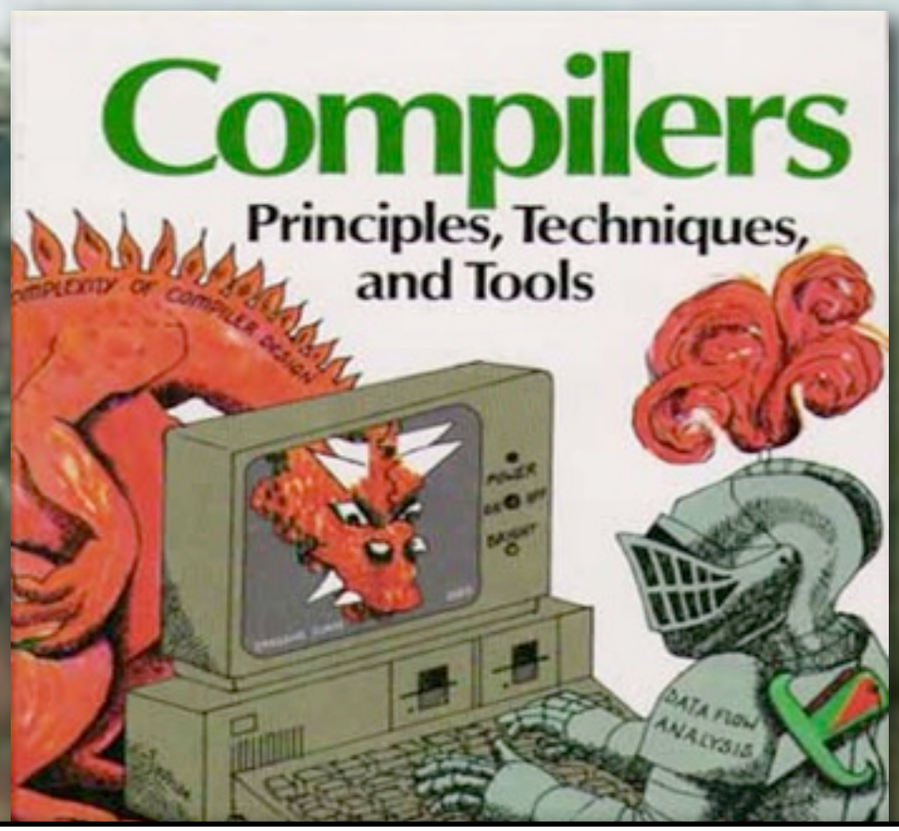


The fundamental problem



The programmer wants to understand the code he writes

The fundamental problem



The programmer wants to understand the code he writes

The compiler
- and the hardware -
try hard to optimise it

The fundamental problem

Which are the valid optimisations that the compiler or the hardware can perform without breaking the expected semantics of a concurrent program?

Which is the semantics of a concurrent program?

The programmer wants to understand the code he writes

The compiler
- and the hardware -
try hard to optimise it

0. Concurrency and optimisations, not so simple

1) The simplest model

2) Data-race freedom (aka. the layman semantics)

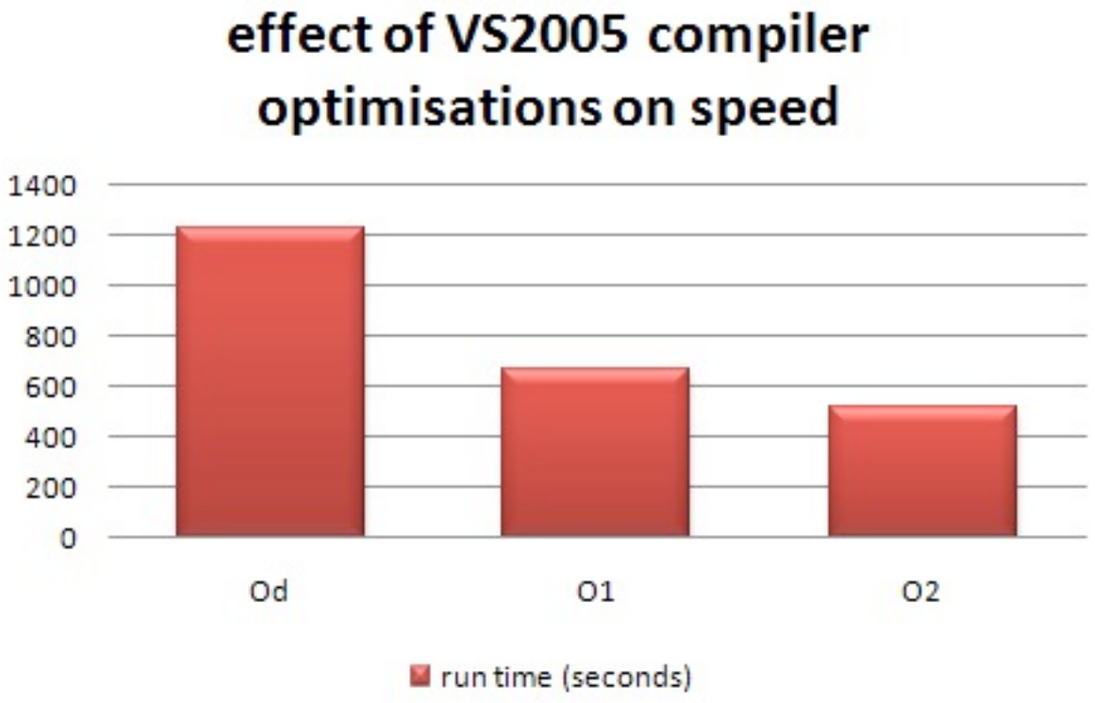
3) How to design a programming language

4) The design of the C11/C++11 languages

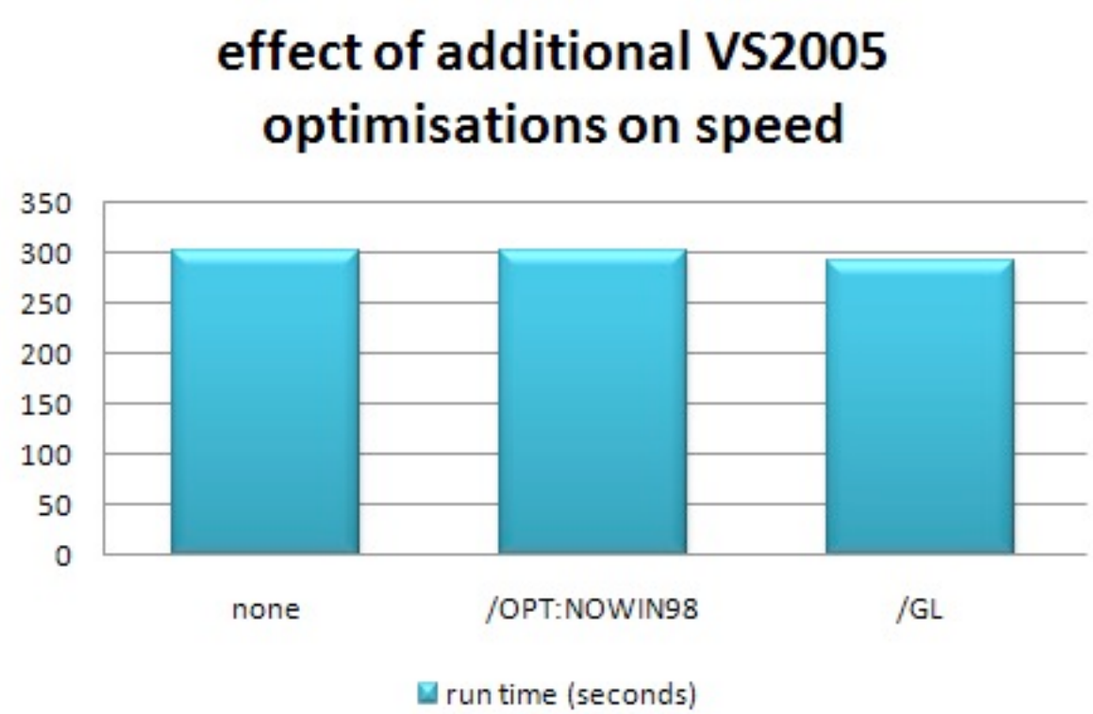
5) Escape lanes are a Pandora's box

6) Exploring alternative models

7) Hope



A brief tour of compiler optimisations



World of optimisations

A typical compiler performs *many optimisations*.

gcc 4.4.1. with -O2 option goes through 147 compilation passes.

computed using -fdump-tree-all and -fdump-rtl-all

Sun Hotspot Server JVM has 18 high-level passes with each pass composed of one or more smaller passes.

`http://www.azulsystems.com/blog/cliff-click/2009-04-14-odds-ends`

World of optimisations

A typical compiler performs many optimisations.

– Common subexpression elimination
    (copy propagation, partial redundancy elimination, value numbering)
– (conditional) constant propagation
– dead code elimination
– loop optimisations
    (loop invariant code motion, loop splitting/peeling, loop unrolling, etc.)
– vectorisation
– peephole optimisations
– tail duplication removal
– building graph representations/graph linearisation
– register allocation
– call inlining
– local memory to registers promotion
– spilling
– instruction scheduling

World of optimisations

However only some optimisations change shared-memory traces:

– *Common subexpression elimination*
(copy propagation, partial redundancy elimination, value numbering)
– (conditional) constant propagation
– dead code elimination
– loop optimisations
(*loop invariant code motion,* loop splitting/peeling, loop unrolling, etc.)
– vectorisation
– *peephole optimisations*
– tail duplication removal
– building graph representations/graph linearisation
– register allocation
– call inlining
– *local memory to registers promotion*
– *spilling*
– instruction scheduling

What is an optimisation?

Compiler Writer

Semanticist

What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

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

What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

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

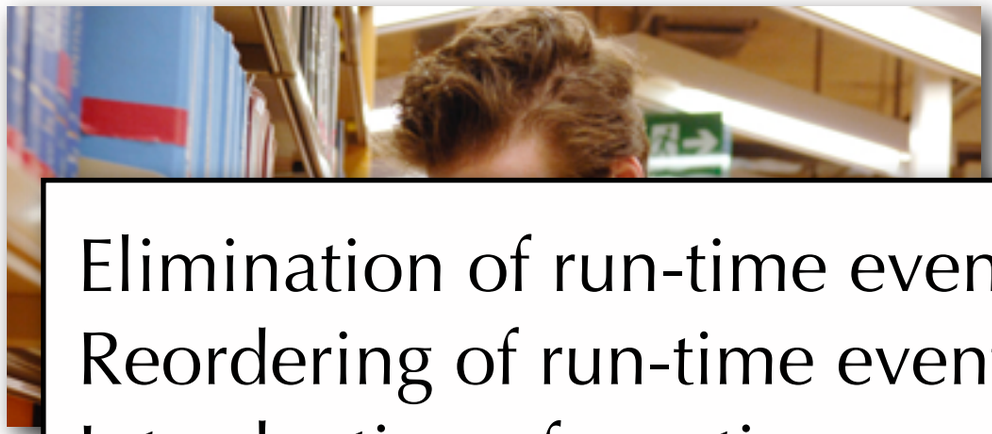
What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist



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

Operations on sets of events

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

What is an optimisation?

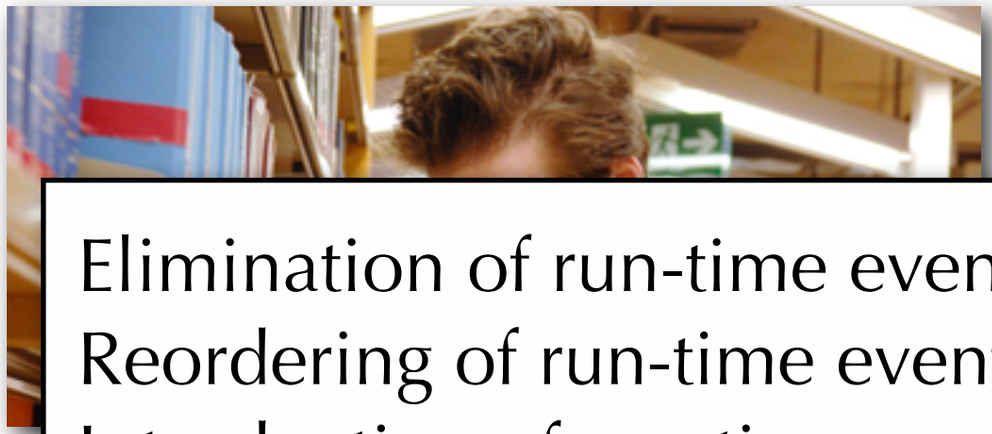
Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

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

Semanticist



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

Operations on sets of events

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

What is an optimisation?

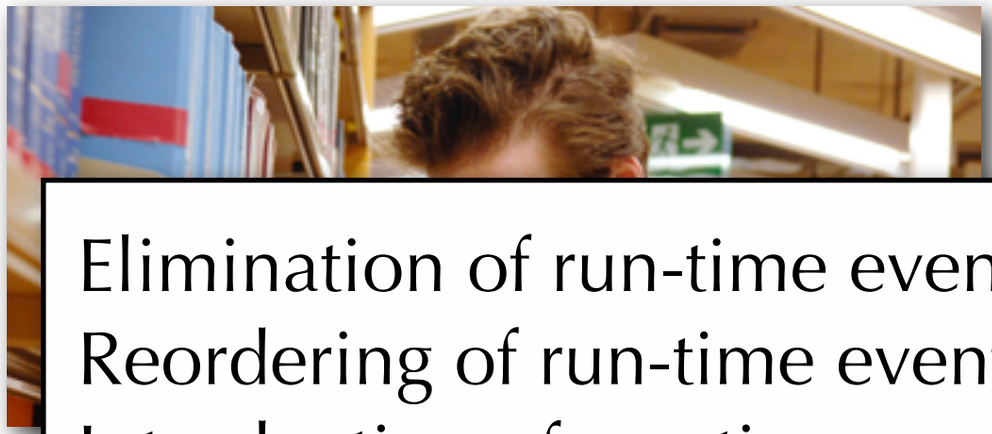
Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

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

Semanticist



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

Operations on sets of events

```
Load y 42
Store z 0

Store x[0] 43
Store z 1

Store x[1] 43
```

Eliminations

This includes common subexpression elimination, dead read elimination, overwritten write elimination, redundant write elimination.

Irrelevant read elimination:

```
r=*x; C → C
```

where `r` is not free in `C`.

Redundant read after read elimination:

```
r1=*x; r2=*x → r1=*x; r2=r1
```

Redundant read after write elimination:

```
*x=r1; r2=*x → *x=r1; r2=r1
```

Reordering

Common subexpression elimination, some loop optimisations, code motion.

Normal memory access reordering:

```
r1=*x; r2=*y → r2=*y; r1=*x
*x=r1; *y=r2 → *y=r2; *x=r1
r1=*x; *y=r2 ⇄ *y=r2; r1=*x
```

Roach motel reordering:

```
memop; lock m → lock m; memop
unlock m; memop → memop; unlock m
```

where `memop` is `*x=r1` or `r1=*x`

Memory access introduction

Can an optimisation introduce memory accesses?

Yes:

```
i = 0;                                  i = 0;
...                          →          …
while (i != 0) {                        tmp = *x;
    j = *x + 1;                         while (i != 0) {
    i = i-1 }                               j = tmp + 1;
                                            i = i-1 }
```

Note that the loop body is not executed.

Memory access introduction

Can an optimisation introduce memory accesses?

Yes:

Back to our question now:

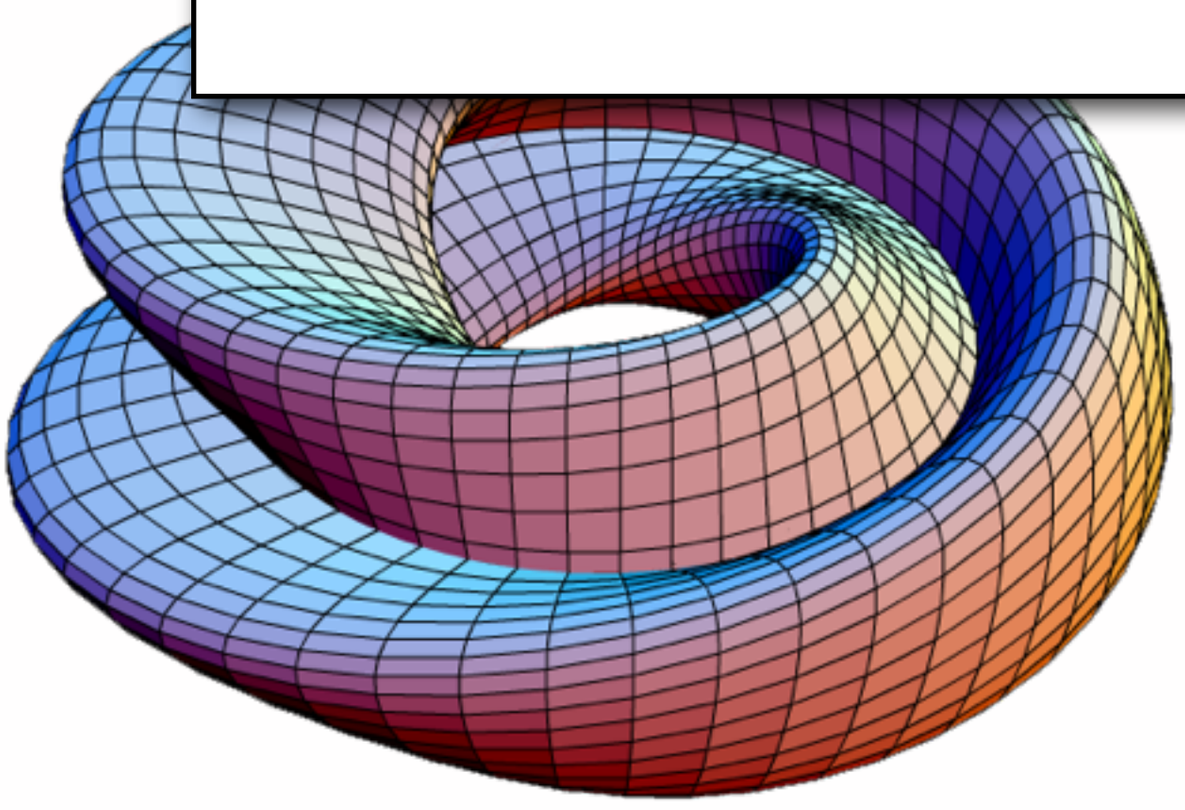
Which is the semantics of a concurrent program?

```
                                    i = i-1 }
```

Note that the loop body is not executed.

The simplest memory model

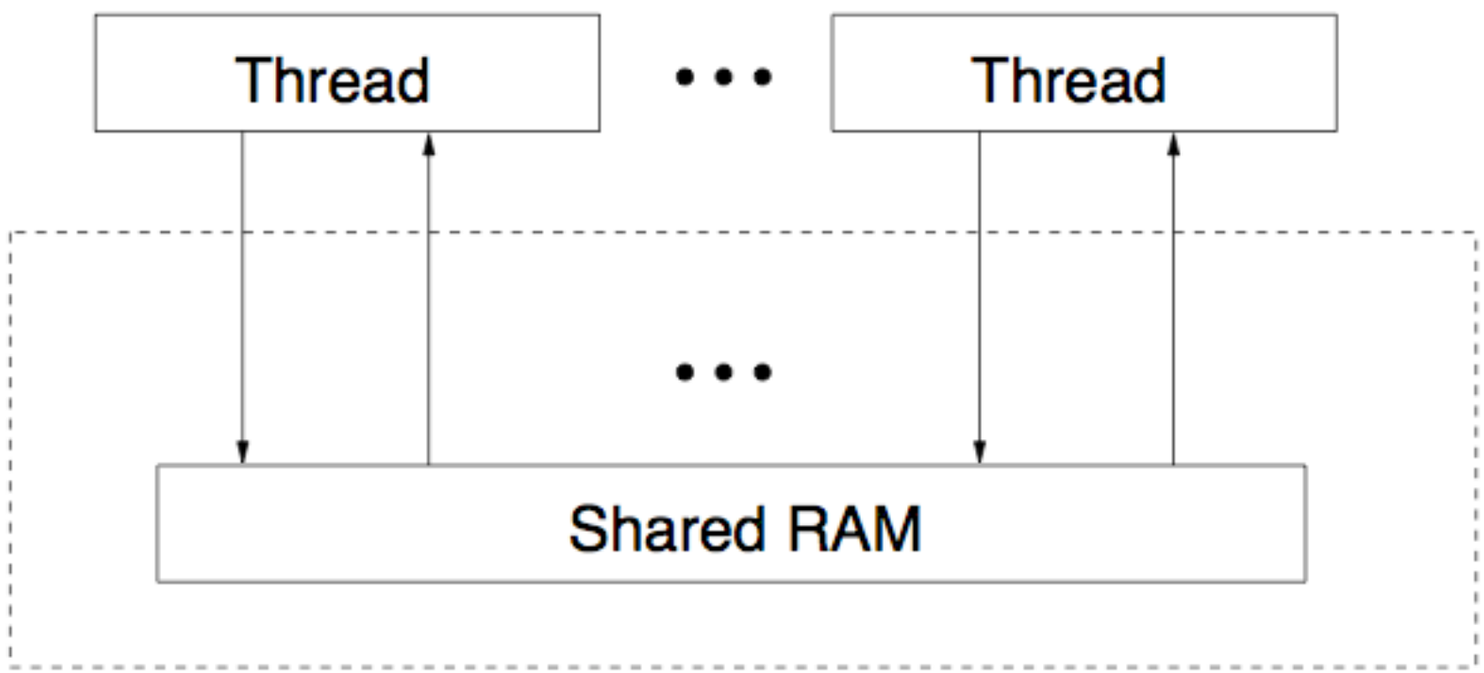
sequential consistency

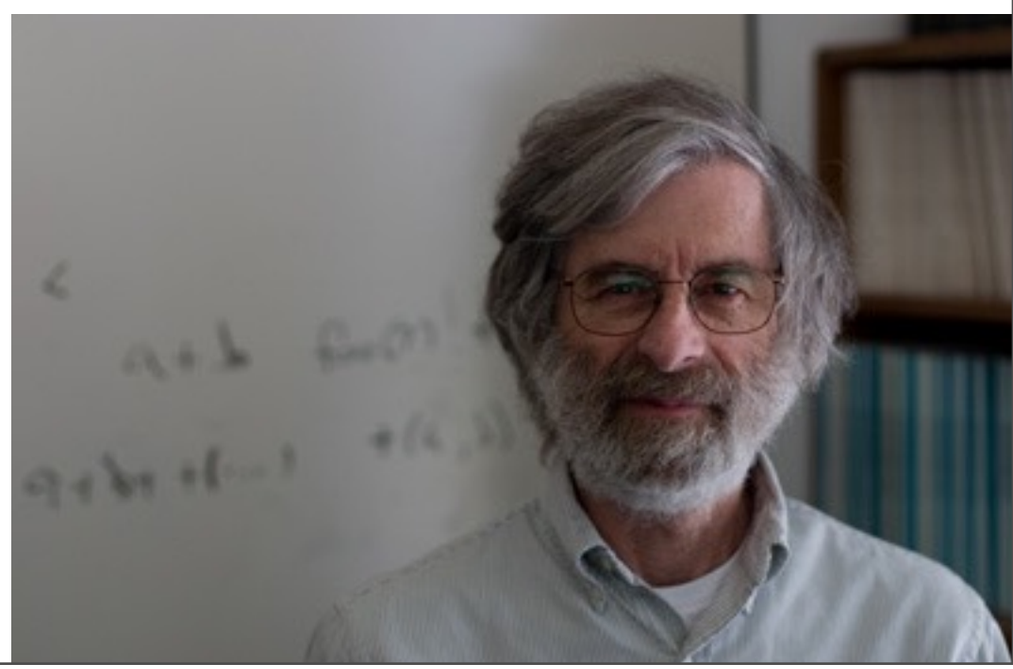


Sequential consistency

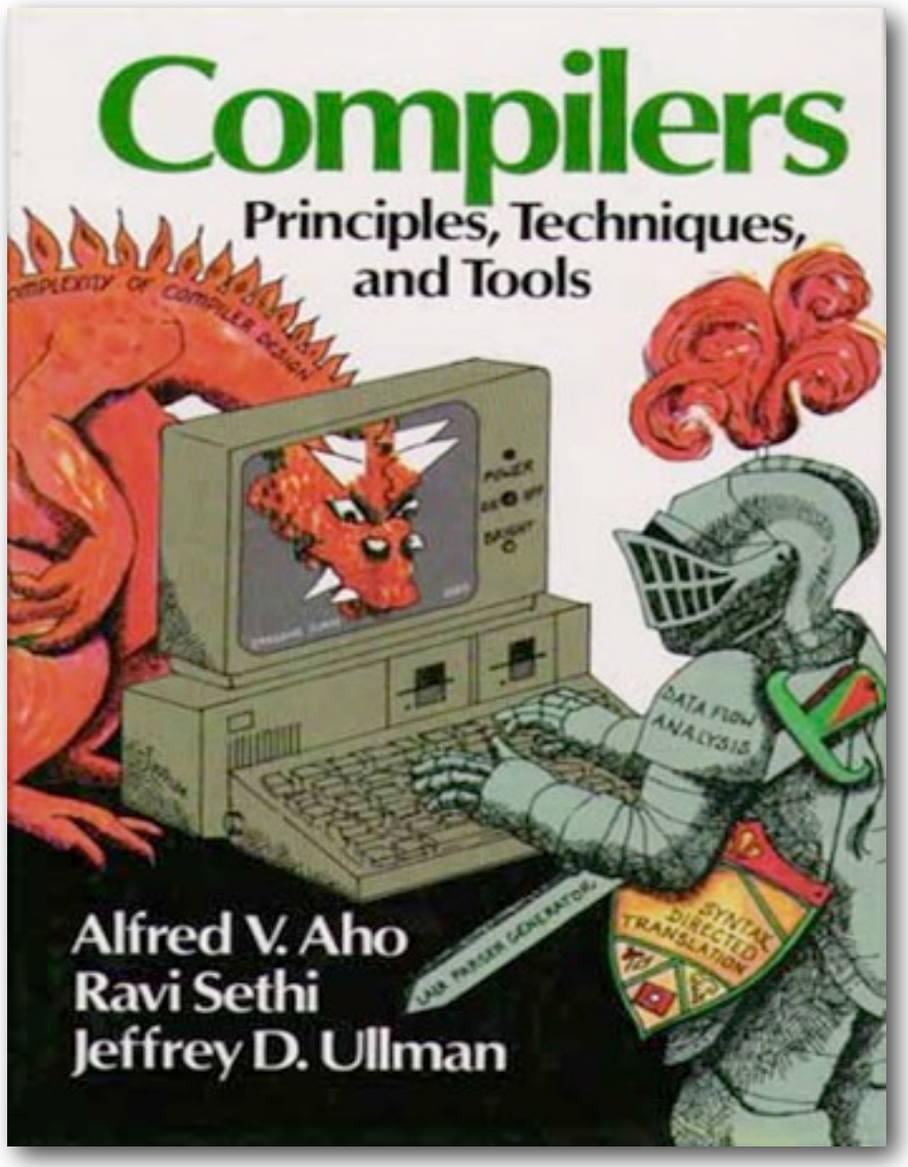
...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program...

Lamport, 1979.

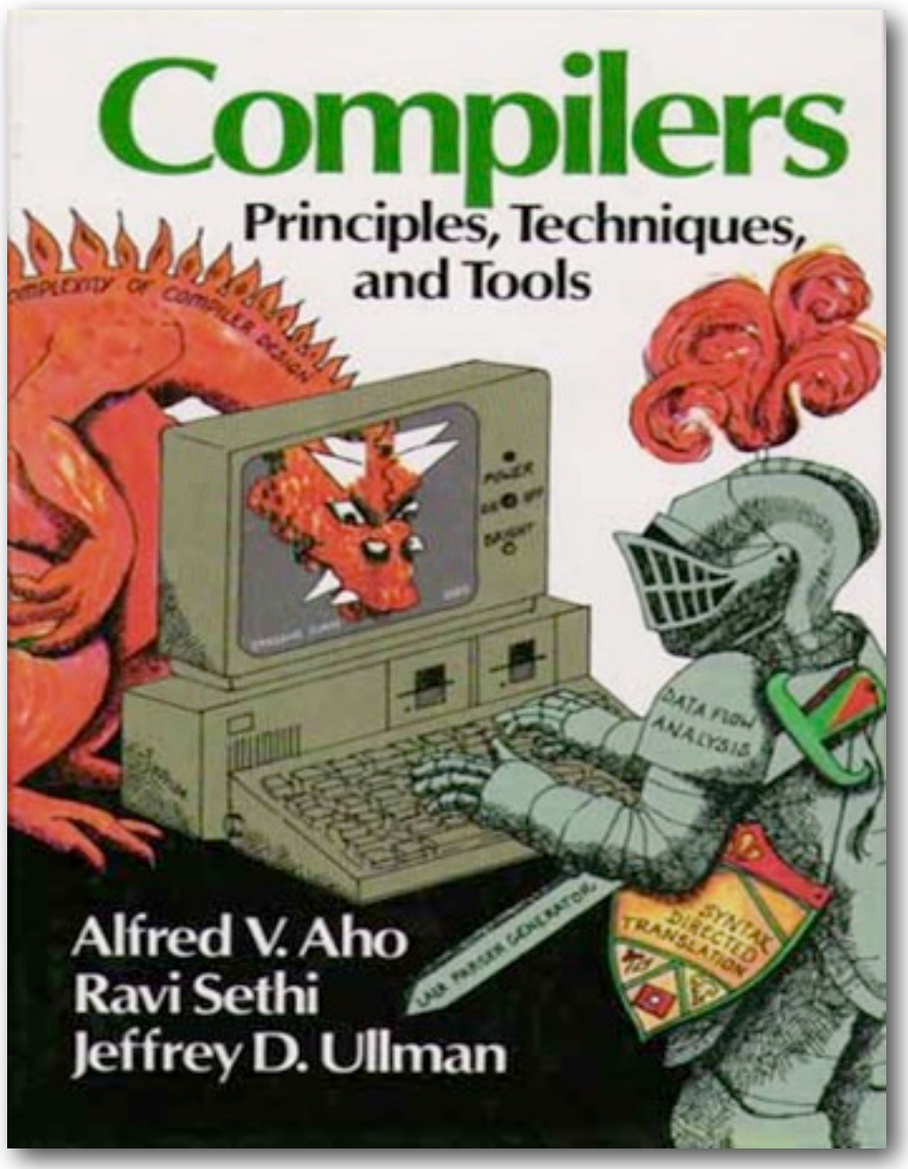




Compilers, programmers & sequential

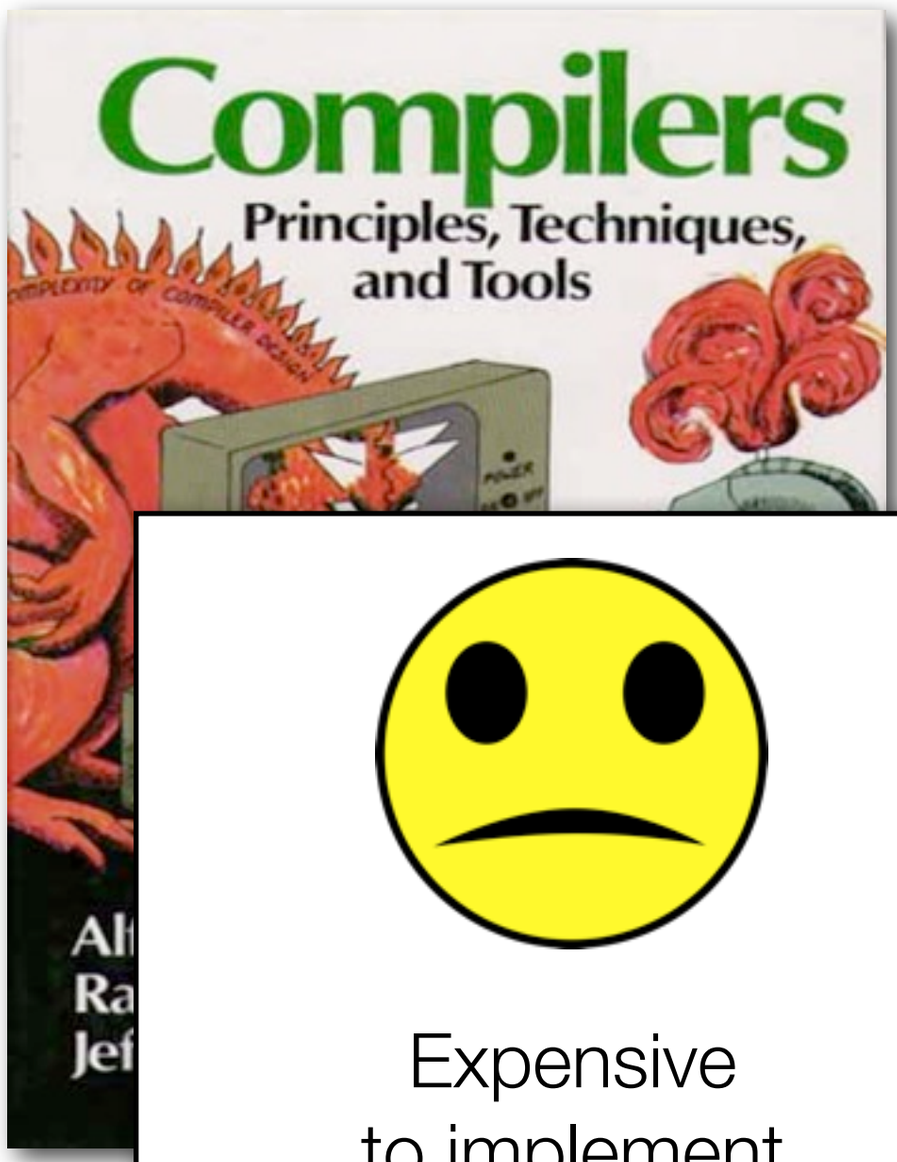


Compilers, programmers & sequential



Simple and intuitive programming model

Compilers, programmers & sequential



Expensive
to implement

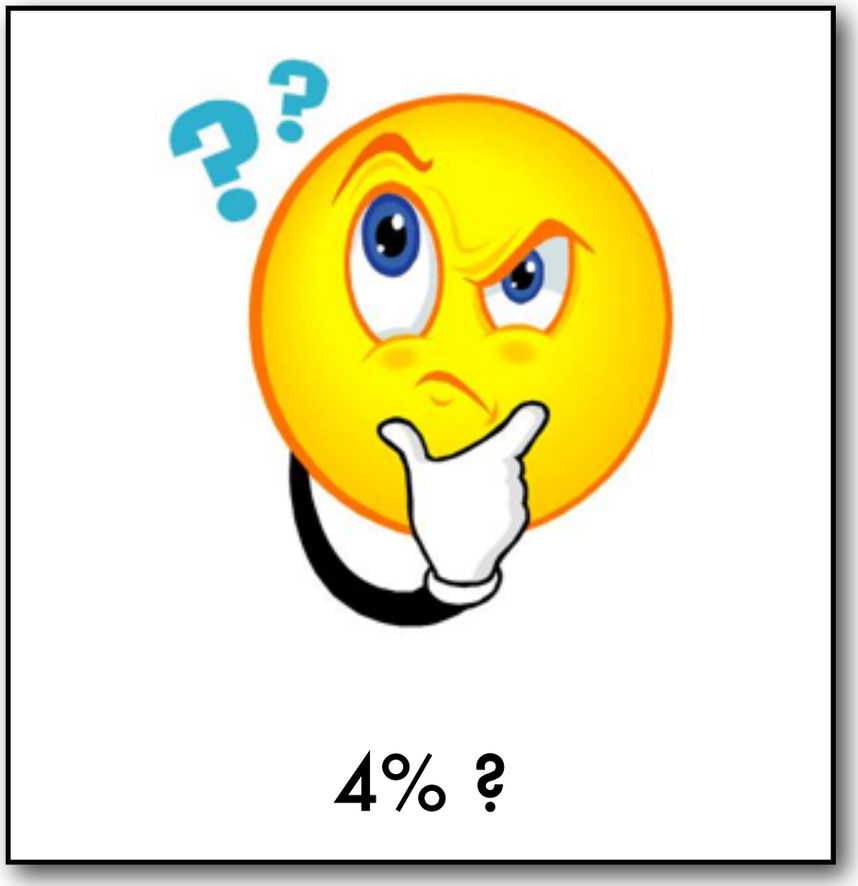
Simple and intuitive
programming model

A Case for an SC-Preserving Compiler

Daniel Marino† Abhayendra Singh* Todd Millstein† Madanlal Musuvathi‡ Satish Narayanasamy*

†University of California, Los Angeles *University of Michigan, Ann Arbor ‡Microsoft Research, Redmond

An SC-preserving compiler, obtained by restricting the optimization phases in LLVM, a state-of-the-art C/C++ compiler, incurs an average slowdown of 3.8% and a maximum slowdown of 34% on a set of 30 programs from the SPLASH-2, PARSEC, and SPEC CINT2006 benchmark suites.



A Case for an SC-Preserving Compiler

Daniel Marino† Abhayendra Singh* Todd Millstein† Madanlal Musuvathi‡ Satish Narayanasamy*

†University of California, Los Angeles *University of Michigan, Ann Arbor ‡Microsoft Research, Redmond

An SC-preserving compiler, obtained by restricting the optimization phases in LLVM, a state-of-the-art C/C++ compiler, incurs an average slowdown of 3.8% and a maximum slowdown of 34% on a set of 30 programs from the SPLASH-2, PARSEC, and SPEC CINT2006 benchmark suites.

EXCESSIVE OVERHEAD

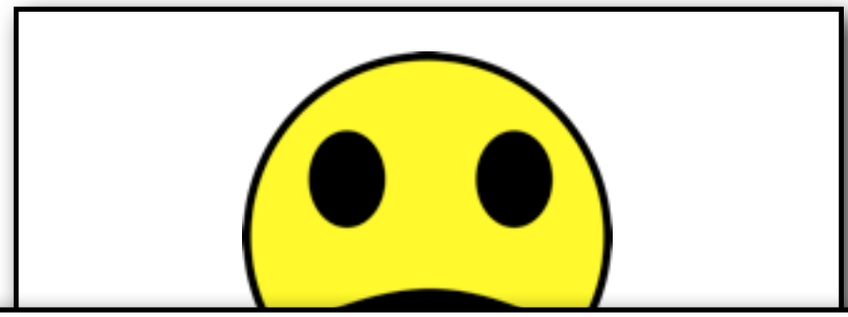
This study assumes that the hardware is SC: these numbers are optimistic lower bounds.

A Case for an SC-Preserving Compiler

Daniel Marino† Abhayendra Singh* Todd Millstein† Madanlal Musuvathi‡ Satish Narayanasamy*

†University of California, Los Angeles *University of Michigan, Ann Arbor ‡Microsoft Research, Redmond

An SC-preserving compiler, obtained by restricting the optimization phases in LLVM, a state-of-the-art C/C++ compiler, incurs an average slowdown of 3.8% and a maximum slowdown of 34% on a set of 30



What is an SC-preserving compiler?

When is a compiler correct?

When is a compiler correct?

A compiler is correct if any behaviour of the compiled program could be exhibited by the original program.

i.e. for any execution of the compiled program, there is an execution of the source program with the *same observable behaviour*.

Intuition: we represent programs as sets of memory action traces, where the trace is a sequence of memory actions of a single thread.

Intuition: the observable behaviour of an execution is the subtrace of external actions.

Exercise

$P_1 =$ `*x = 1`	`r1 = *x; r2 = *x;` `if r1=r2 then print 1 else print 2`
$P_2 =$ `*x = 1`	`r1 = *x; r2 = r1;` `if r1=r2 then print 1 else print 2`

Is the transformation from P1 to P2 correct (in an SC semantics)?

Exercise

$P_1 = *x = 1$

```
r1 = *x; r2 = *x;
if r1=r2 then print 1 else print 2
```

$P_2 = *x = 1$

```
r1 = *x; r2 = r1;
if r1=r2 then print 1 else print 2
```

Exercise

$P_1 = *x = 1$	`r1 = *x; r2 = *x;` `if r1=r2 then print 1 else print 2`
$P_2 = *x = 1$	`r1 = *x; r2 = r1;` `if r1=r2 then print 1 else print 2`

Executions of P1:

$\mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 1$

$\mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 2$

$\mathsf{R}_{t_2}\, x{=}0, \mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{P}_{t_2}\, 1$

$\mathsf{R}_{t_2}\, x{=}0, \mathsf{R}_{t_2}\, x{=}0, \mathsf{P}_{t_2}\, 1, \mathsf{W}_{t_1}\, x{=}1$

Exercise

$P_1 =$ `*x = 1`

```
r1 = *x; r2 = *x;
if r1=r2 then print 1 else print 2
```

$P_2 =$ `*x = 1`

```
r1 = *x; r2 = r1;
if r1=r2 then print 1 else print 2
```

Executions of P1:

$\mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 2$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{R}_{t_2}\, x{=}0, \mathsf{P}_{t_2}\, 1, \mathsf{W}_{t_1}\, x{=}1$

Executions of P2:

$\mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{P}_{t_2}\, 1, \mathsf{W}_{t_1}\, x{=}1$

Exercise

$P_1 =$ `*x = 1`

```
r1 = *x; r2 = *x;
if r1=r2 then print 1 else print 2
```

$P_2 =$ `*x = 1`

```
r1 = *x; r2 = r1;
if r1=r2 then print 1 else print 2
```

Executions of P1:

$\mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 2$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{R}_{t_2}\, x{=}0, \mathsf{P}_{t_2}\, 1, \mathsf{W}_{t_1}\, x{=}1$

Behaviours of P1: $[\mathsf{P}_{t_2}\, 1], [\mathsf{P}_{t_2}\, 2]$

Executions of P2:

$\mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{P}_{t_2}\, 1, \mathsf{W}_{t_1}\, x{=}1$

Behaviours of P2: $[\mathsf{P}_{t_2}\, 1]$

Exercise

$P_1 =$ `*x = 1`

```
r1 = *x; r2 = *x;
if r1=r2 then print 1 else print 2
```

$P_2 =$ `*x = 1`

```
r1 = *x; r2 = r1;
if r1=r2 then print 1 else print 2
```

Executions of P1:

Executions of P2:

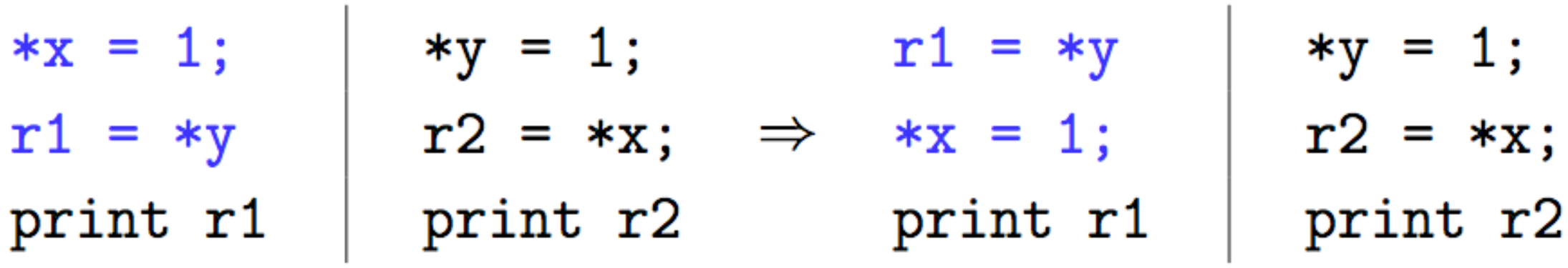
It is correct to rewrite P1 into P2, but not the opposite!

t_2 , t_2 , t_2 , t_1

Behaviours of P1: $[\mathrm{P}_{t_2}\ 1], [\mathrm{P}_{t_2}\ 2]$

Behaviours of P2: $[\mathrm{P}_{t_2}\ 1]$

Another exercise



Another exercise

```
*x = 1;          |  *y = 1;          r1 = *y          |  *y = 1;
r1 = *y          |  r2 = *x;    ⇒    *x = 1;          |  r2 = *x;
print r1         |  print r2         print r1         |  print r2
```

$[\mathsf{P}_{t_1}\, 0, \mathsf{P}_{t_2}\, 1]$
$[\mathsf{P}_{t_1}\, 1, \mathsf{P}_{t_2}\, 0]$
$[\mathsf{P}_{t_1}\, 1, \mathsf{P}_{t_2}\, 1]$

$[\mathsf{P}_{t_1}\, 0, \mathsf{P}_{t_2}\, 1]$
$[\mathsf{P}_{t_1}\, 1, \mathsf{P}_{t_2}\, 0]$
$[\mathsf{P}_{t_1}\, 1, \mathsf{P}_{t_2}\, 1]$
$[\mathsf{P}_{t_1}\, 0, \mathsf{P}_{t_2}\, 0]$

Another exercise

`*x = 1;`	`*y = 1;`	`r1 = *y`	`*y = 1;`

Again, the transformed program exhibits a new behaviour

$$[\mathsf{P}_{t_1}\,0, \mathsf{P}_{t_2}\,1]$$
$$[\mathsf{P}_{t_1}\,1, \mathsf{P}_{t_2}\,0]$$
$$[\mathsf{P}_{t_1}\,1, \mathsf{P}_{t_2}\,1]$$

$$[\mathsf{P}_{t_1}\,0, \mathsf{P}_{t_2}\,1]$$
$$[\mathsf{P}_{t_1}\,1, \mathsf{P}_{t_2}\,0]$$
$$[\mathsf{P}_{t_1}\,1, \mathsf{P}_{t_2}\,1]$$
$$[\mathsf{P}_{t_1}\,0, \mathsf{P}_{t_2}\,0]$$

Elimination of *adjacent* accesses

Some optimisations are correct under SC.

For example it is correct to rewrite:

```
r1 = *x; r2 = *x     →     r1 = *x; r2 = r1
```

The basic idea: whenever we perform the read r1 = *x in the optimised program, we perfom *both* reads in the source program.

Elimination of *adjacent* accesses

Some optimisations are correct under SC.

Not really satisfying... Can we define a model that:

1) enables more optimisations than SC, and
2) retains the simplicity of SC?

The basic idea: whenever we perform the read r1 = *x in the optimised program, we perfom *both* reads in the source program.

The layman solution

forbid data-races

Data-race freedom

Our examples again:

Thread 0	Thread 1
`*y = 1`	`if *x == 1`
`*x = 1`	`then print *y`

Observable behaviour: `0`

- the problematic transformations (e.g. swapping the two writes in thread 0) *do not change the meaning of single-threaded programs*

- the problematic transformations are *detectable* only by code that allows *two threads to access the same data simultaneously in conflicting ways* (e.g. one thread writes the datas read by the other).

Data-race freedom

Thread 0	Thread 1
*x = 1	if *x == 1

...intuition...

Programming languages provide
synchronisation mechanisms

if these are used (and implemented) correctly,
we might avoid the issues above...

conflicting ways (e.g. one thread writes the datas read by the other).

The basic solution

Thread 0	Thread 1
`*y = 1`	`if *x == 1`
`*x = 1`	`then print *y`

Observable behaviour: `0`

Prohibit *data races*

Defined as follows:

- two memory operations **conflict** if they access the same memory location and at least one is a store operation;
- a **SC execution** (interleaving) **contains a data race** if two conflicting operations corresponding to different threads are adjacent (maybe executed concurrently).

Example: a data race in the example above:

$$\mathsf{W}_{t_1}\, y{=}1, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, y{=}1, \mathsf{P}_{t_2}\, 1$$

The basic solution

Thread 0	Thread 1
`*y = 1`	`if *x == 1`
`*x = 1`	`then print *y`

Observable behaviour: `0`

Prohibit *data races*

Defined as follows:

The definition of data race quantifies *only* over the sequential consistent executions

executed concurrently).

Example: a data race in the example above:

$$\mathsf{W}_{t_1}\, y{=}1, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, y{=}1, \mathsf{P}_{t_2}\, 1$$

How do we avoid data races? (high-level languages)

- **Locks**

  No lock(l) can appear in the interleaving unless prior lock(l) and unlock(l) calls from other threads balance.

- **Atomic variables**

  Allow concurrent access "exempt" from data races (called volatile in Java).

Example:

Thread 0	Thread 1

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

How do we avoid data races? (high-level languages)

Thread 0	Thread 1

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

This program is data-race free:

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

*y = 1; lock(); tmp = *x; unlock(); lock(); *x = 1; unlock(); if tmp=1

*y = 1; lock(); tmp = *x; unlock(); if tmp=1; lock(); *x = 1; unlock();

lock();tmp = *x;unlock(); *y = 1; lock(); *x = 1; unlock(); if tmp=1

lock(); tmp = *x; unlock(); if tmp=1; *y = 1; lock();*x = 1;unlock();

lock();tmp = *x;unlock(); *y = 1; if tmp=1; lock(); *x = 1; unlock();
```

How do we avoid data races? (high-level languages)

- `lock()`, `unlock()` are opaque for the compiler: viewed as potentially modifying any location, memory operations cannot be moved past them
- `lock()`, `unlock()` contain *"sufficient fences"* to prevent hardware reordering across them and global orderering

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

*y = 1; lock(); tmp = *x; unlock(); lock(); *x = 1; unlock(); if tmp=1

*y = 1; lock(); tmp = *x; unlock(); if tmp=1; lock(); *x = 1; unlock();

lock();tmp = *x;unlock(); *y = 1; lock(); *x = 1; unlock(); if tmp=1

lock(); tmp = *x; unlock(); if tmp=1; *y = 1; lock();*x = 1;unlock();

lock();tmp = *x;unlock(); *y = 1; if tmp=1; lock(); *x = 1; unlock();
```

guages)

Compiler/hardware can continue to reorder accesses

Intuition:
compiler/hardware do not know about threads
but only racing threads can tell the difference!

be

moved past them

- `lock()`, `unlock()` contain *"sufficient fences"* to prevent hardware reordering across them and global orderering

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

*y = 1; lock(); tmp = *x; unlock(); lock(); *x = 1; unlock(); if tmp=1

*y = 1; lock(); tmp = *x; unlock(); if tmp=1; lock(); *x = 1; unlock();

lock();tmp = *x;unlock(); *y = 1; lock(); *x = 1; unlock(); if tmp=1

lock(); tmp = *x; unlock(); if tmp=1; *y = 1; lock();*x = 1;unlock();

lock();tmp = *x;unlock(); *y = 1; if tmp=1; lock(); *x = 1; unlock();
```

Validity of compiler optimisations,

Transformation	SC	DRF
Memory trace preserving transformations	✓	✓
Redundant read after read elimination	✓*	✓
Redundant read after write elimination	✓*	✓
Irrelevant read elimination	✓	✓
Redundant write before write elimination	✓*	✓
Redundant write after read elimination	✓*	✓
Irrelevant read introduction	✓	×
Normal memory accesses reordering	×	✓
Roach-motel reordering	×(✓for locks)	✓
External action reordering	×	✓

* Optimisations legal only on adjacent statements.

Validity of compiler optimisations,

Transformation	SC
Memory trace preserving transformations	✓

Jaroslav Sevcik

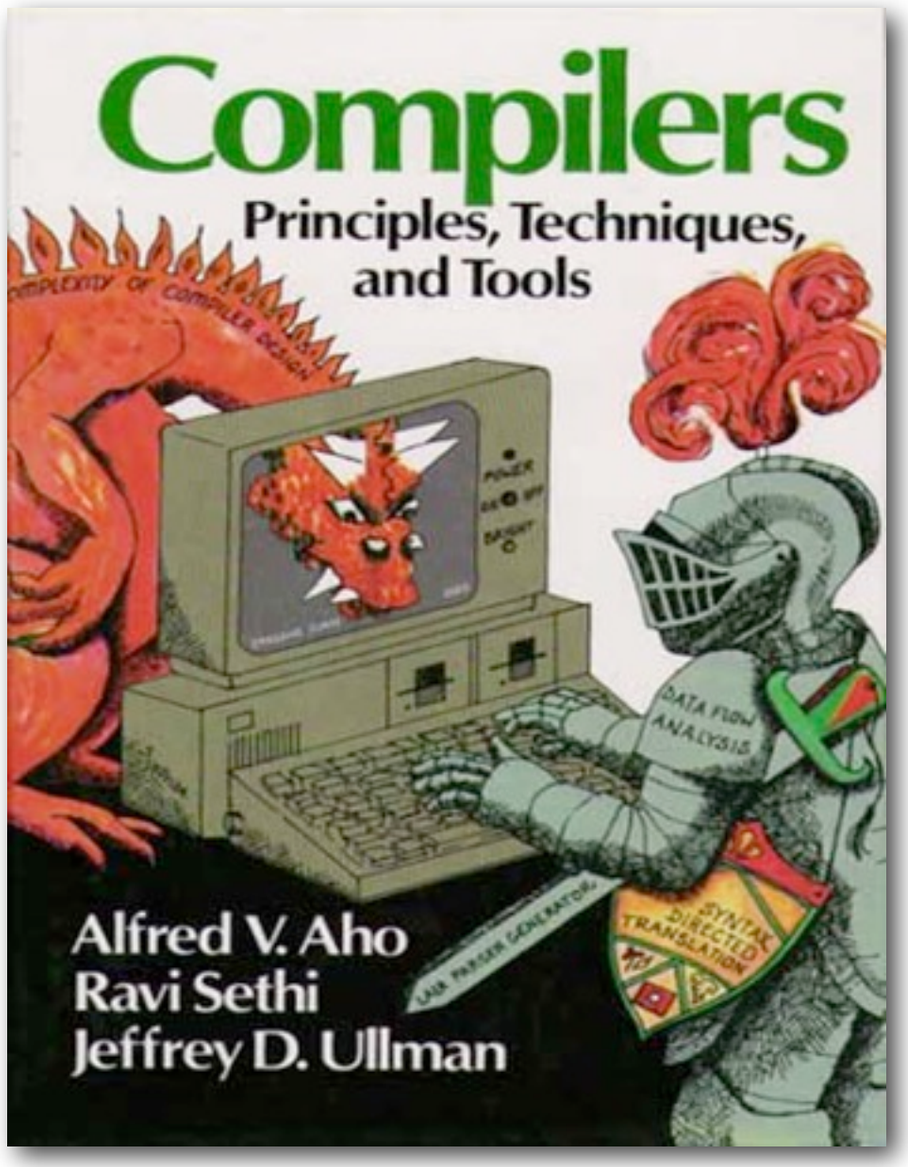
Safe Optimisations for Shared-Memory Concurrent Programs

PLDI 2011

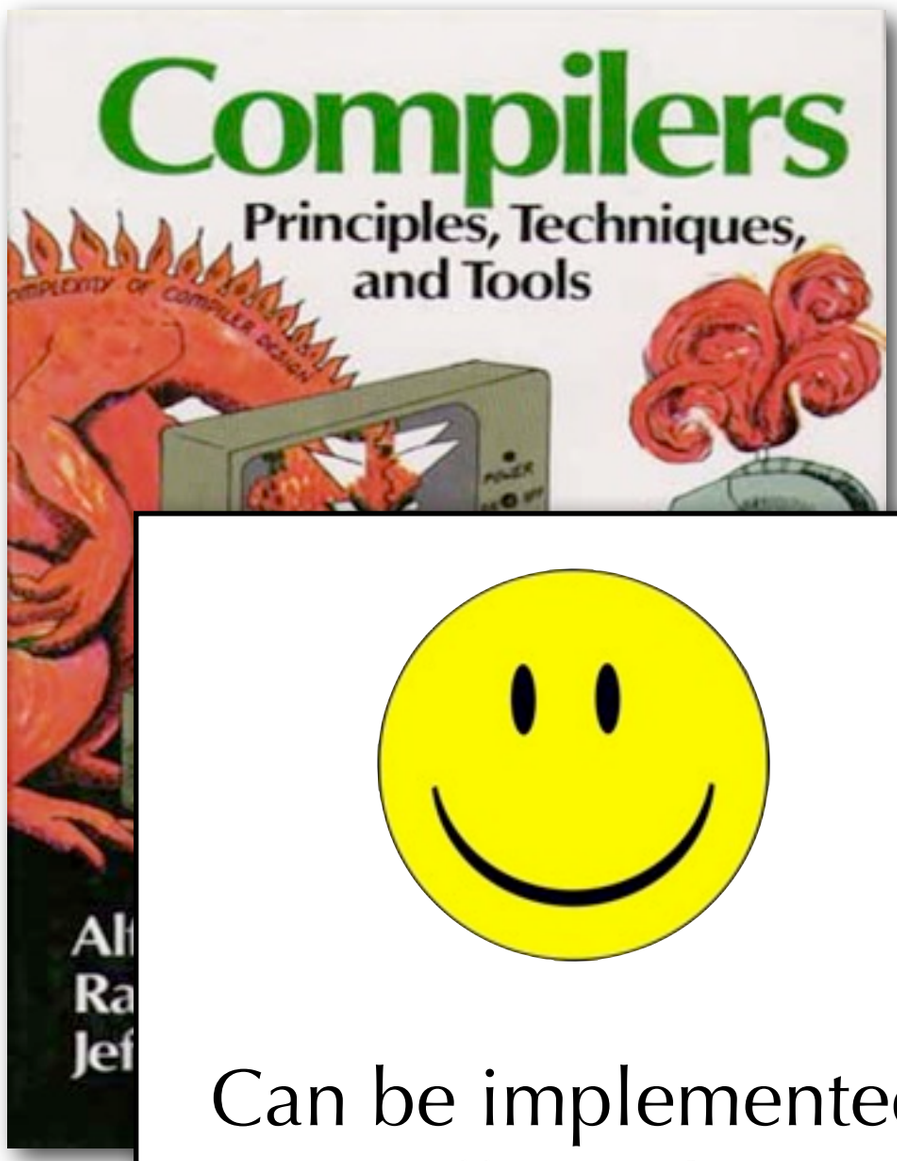
Roach-motel reordering	×(✓for locks)	✓
External action reordering	×	✓

* Optimisations legal only on adjacent statements.

Compilers, programmers & data-race

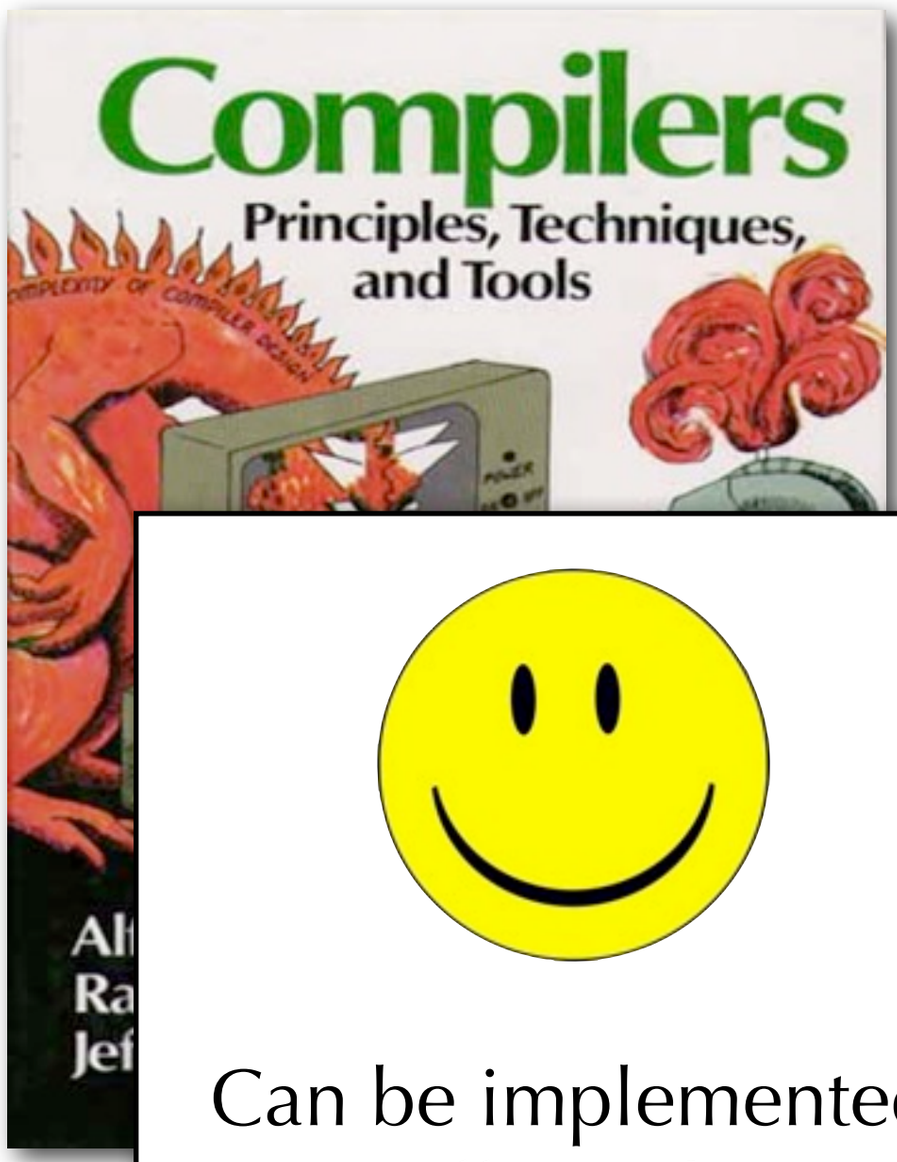


Compilers, programmers & data-race



Can be implemented efficiently

Compilers, programmers & data-race



Can be implemented efficiently

Intuitive programming model (but detecting races is tricky!)

Another example of DRF program

Exercise: is this program DRF?

Thread 0	Thread 1
`if *x == 1`	`if *y == 1`
`then *y = 1`	`then *x = 1`

Another example of DRF program

Exercise: is this program DRF?

Thread 0	Thread 1
`if *x == 1`	`if *y == 1`
`then *y = 1`	`then *x = 1`

Answer: yes!

The writes cannot be executed in any SC execution, so they cannot participate in a data race.

Another example of DRF program

Exercise: is this program DRF?

Data-race freedom is not the ultimate panacea

- the absence of data-races is hard to verify / test (undecidable)
- imagine *debugging*...

  my program ended with a *wrong* result:
  my program has a bug OR it has a data-race

  my program ended with a *correct* result:
  my program is correct OR it has a data-race

Defining programming language memory models

Option 1

Don't.

No concurrency.

Implemented by highly-successful programming languages (**OCaml**)

Poor match for current trends

Option 2

Don't.

No shared memory

A good match for some problems (see Erlang, MPI, …)

Option 3

Don't.

But language ensures data-race freedom

Possible:

- syntactically ensuring data accesses protected by associated locks
- fancy effect type systems (**don't miss Pottier's lecture on Friday**)

Not suitable for general purpose programming.

Option 4

Don't.

Leave it (sort of) up to the hardware

Example:

MLton, a high performance ML-to-x86 compiler with concurrency extensions

Accesses to ML refs exhibit the underlying x86-TSO behaviour (atomicity is guaranteed though)

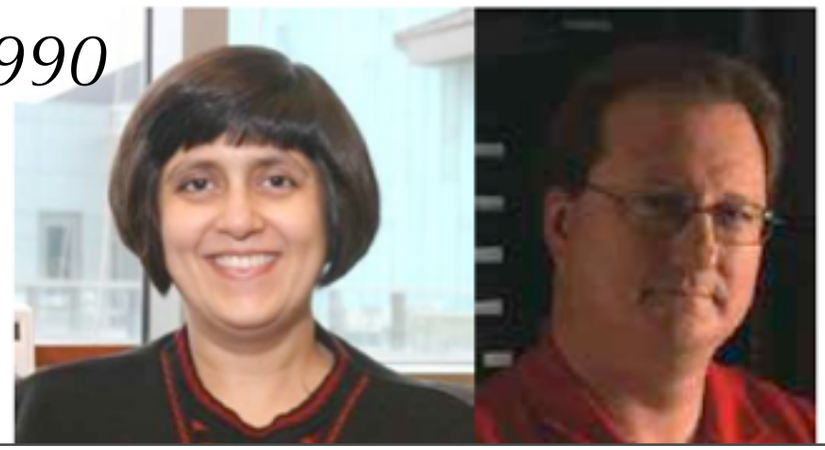
Option 5

Do.

Use data race freedom as a definition

1. Programs that race-free have only sequentially consistent behaviours
2. Programs that have a race in some execution can behave in any way

Sarita Adve & Mark Hill, 1990



Option 5

Do.

Use data race freedom as a definition

Pro:

- simple
- strong guarantees for most code
- allows lots of freedom for compiler and hardware optimisations

Cons:

- undecidable premise
- can't write racy programs (escape mechanisms?)

Ada 83

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

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

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

Data-races are errors

Posix Threads Specification

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

Data-races are errors

C++ 2011 / C1x

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

Data-races are errors

C++ 2011 / C1x

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

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

Data-races are errors

Escape lanes for expert programmers

Low-level atomics in C11/C++11

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

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

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

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

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

Atomic variable declaration

New syntax
for memory accesses

Qualifier

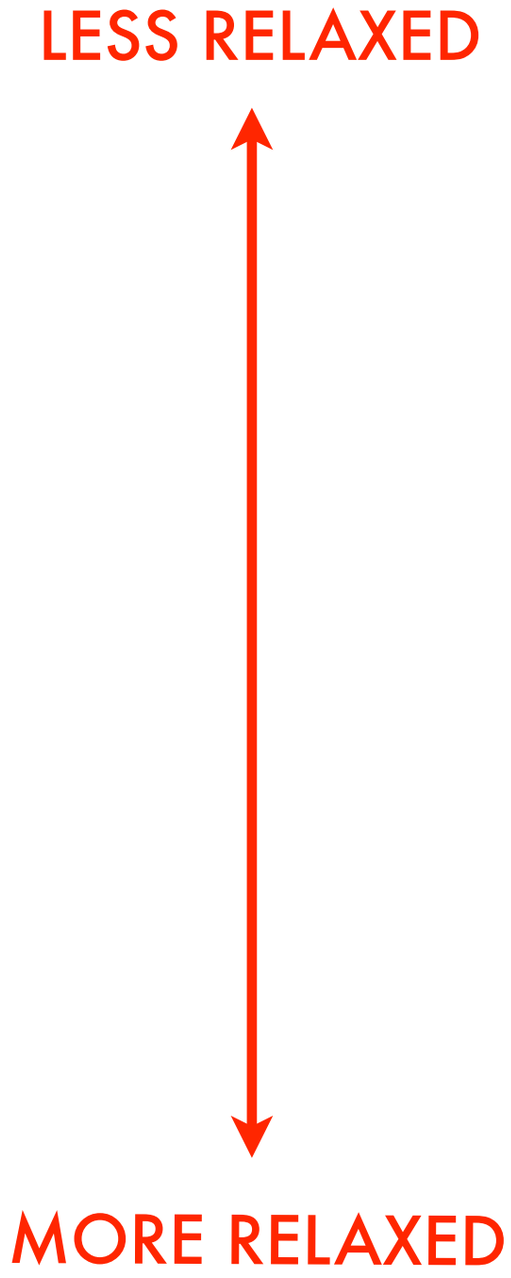
The qualifiers

MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED



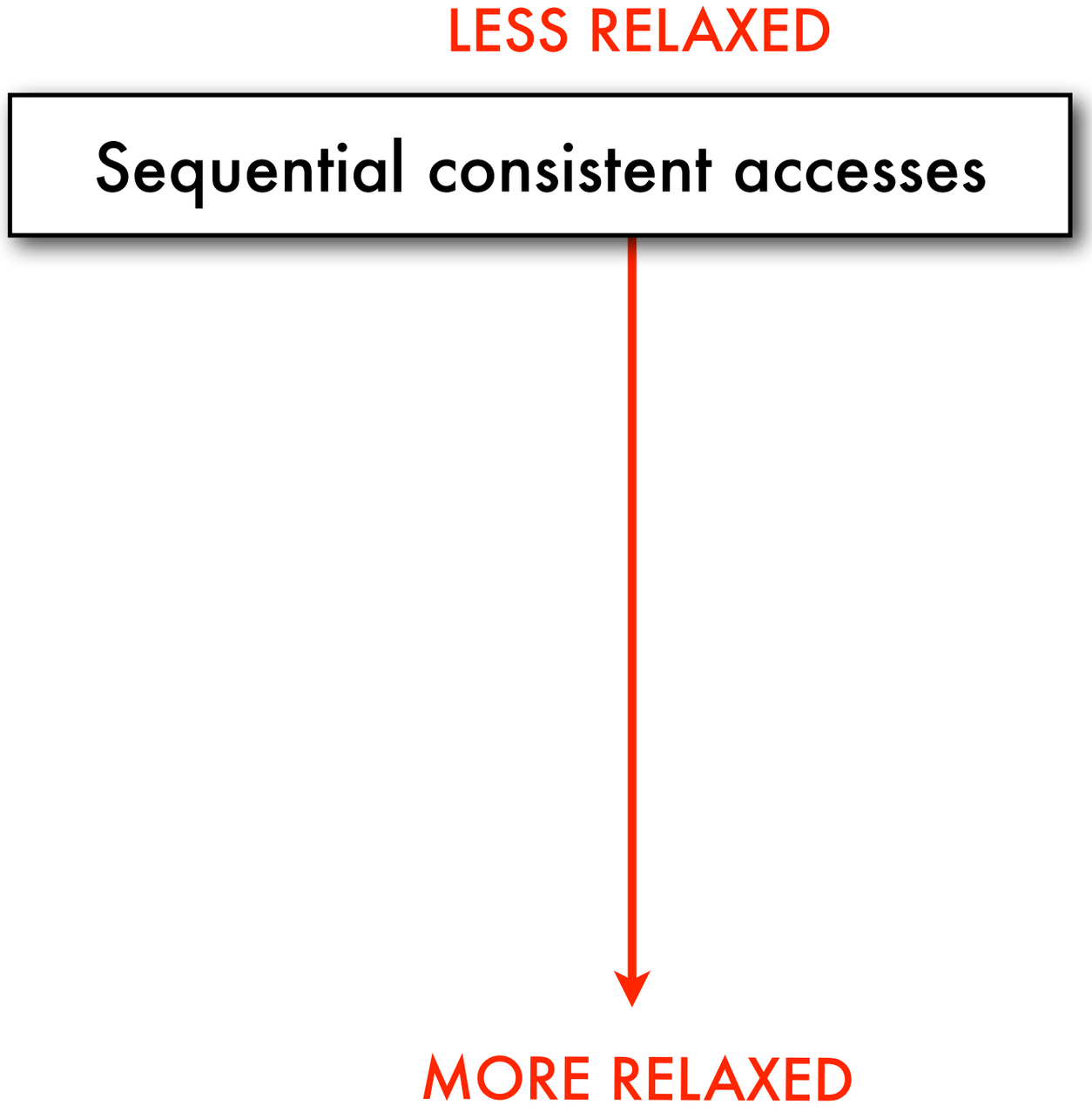
The qualifiers

`MO_SEQ_CST`

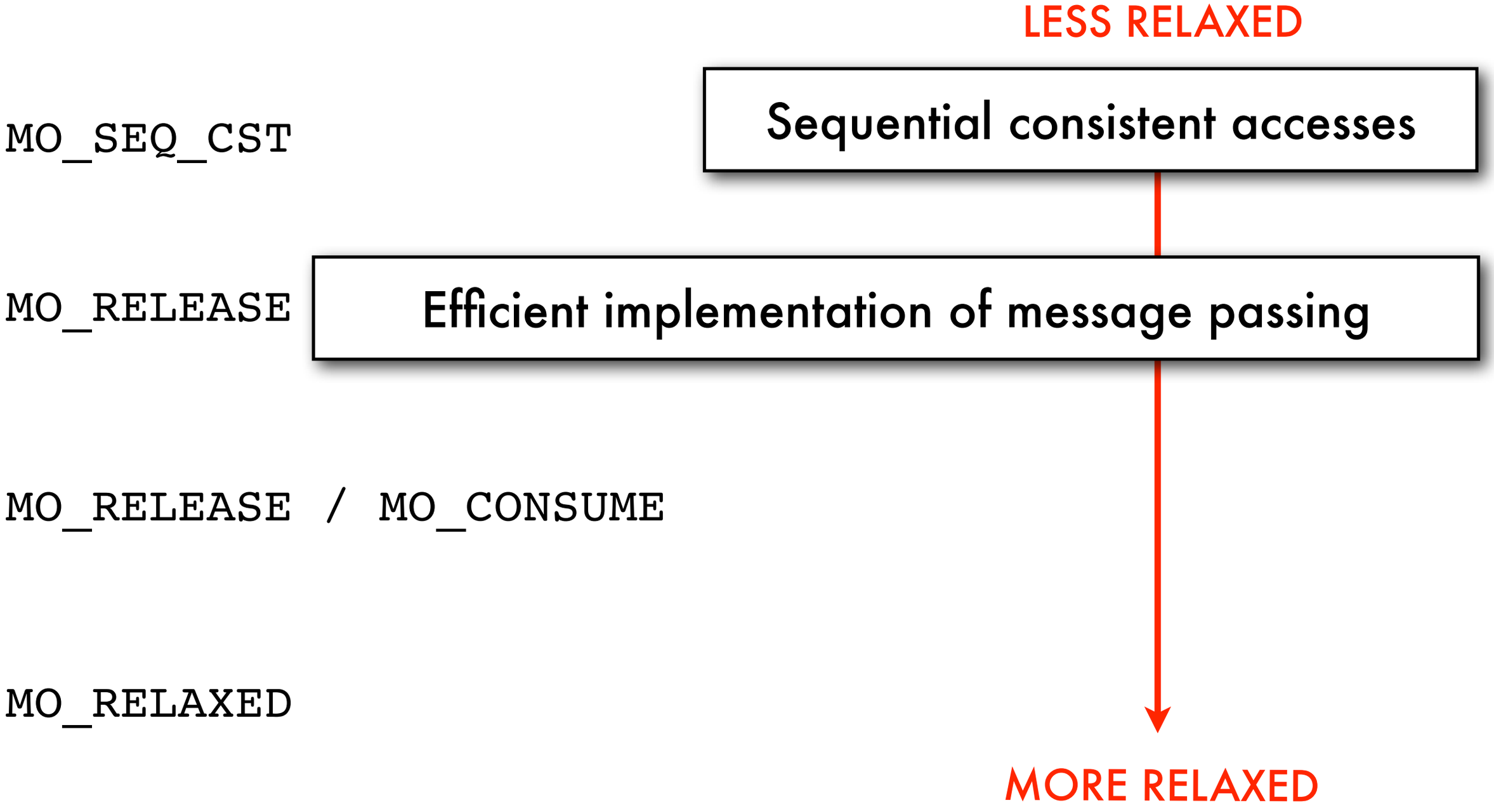
`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

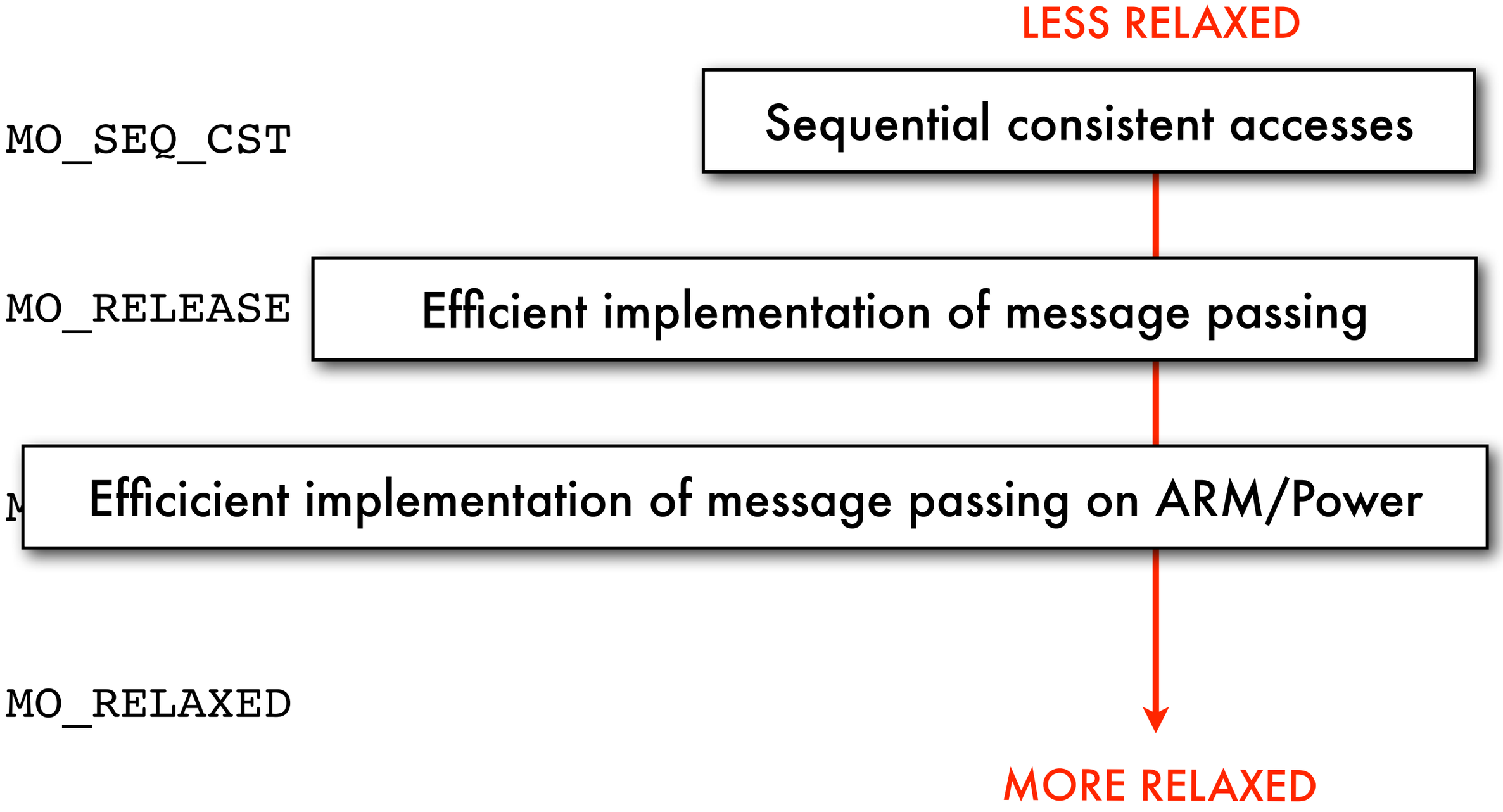
`MO_RELAXED`



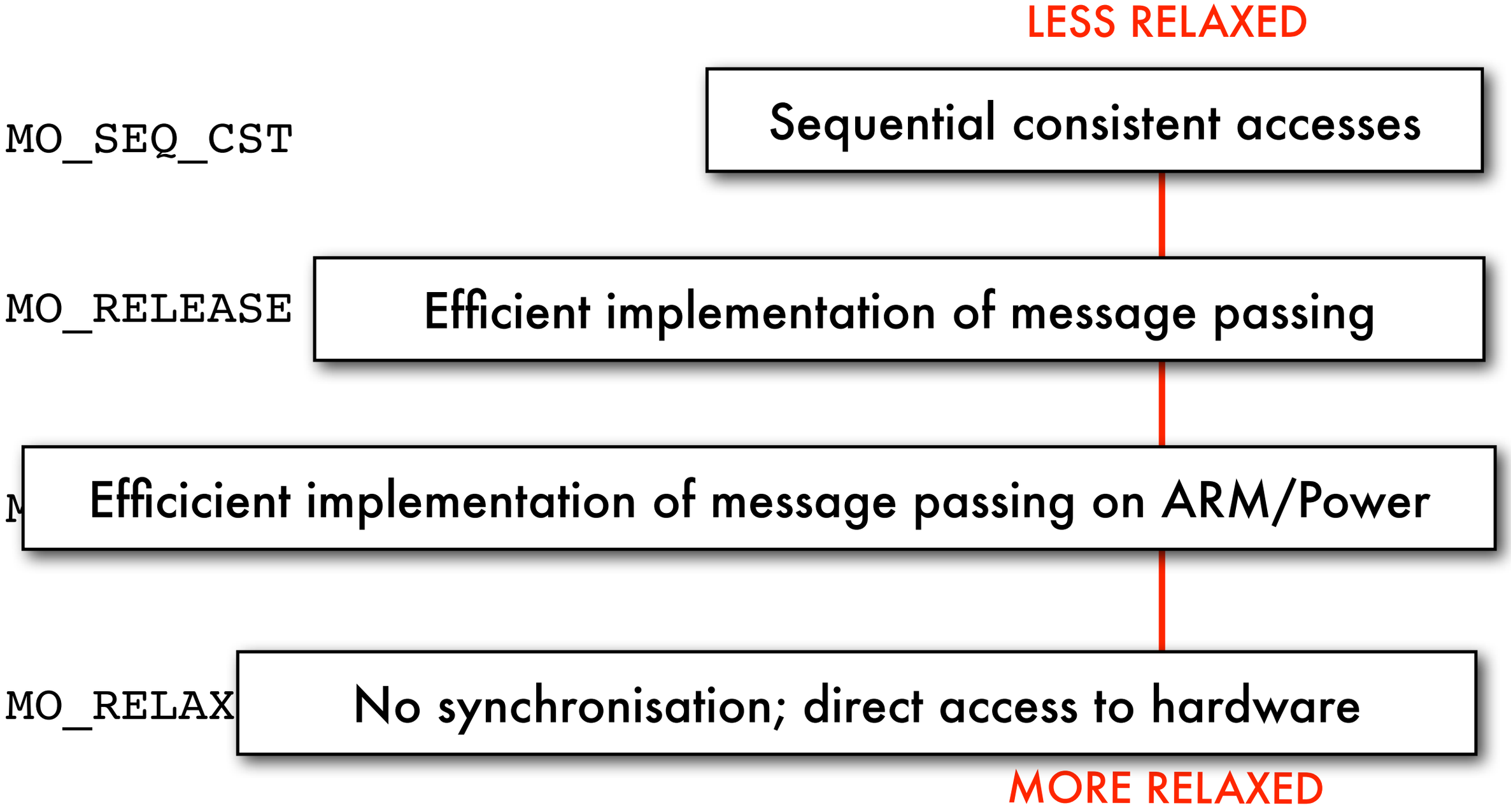
The qualifiers



The qualifiers



The qualifiers



MO_SEQ_CST

The compiler must ensure that `MO_SEQ_CST` accesses have sequentially consistent semantics.

Thread 0	Thread 1
`x.store(1,MO_SEQ_CST)`	`y.store(1,MO_SEQ_CST)`
`r1 = y.load(MO_SEQ_CST)`	`r2 = x.load(MO_SEQ_CST)`

The program above cannot end with `r1 = r2 = 0`.

Sample compilation on x86:

store: `MOV; MFENCE`
load: `MOV`

Sample compilation on Power:

store: `HWSYNC; ST`
load: `HWSYNC; LD; CMP; BC; ISYNC`

MO_RELAXED

MO_RELAXED accesses can be reordered by compiler/hardware

Thread 0	Thread 1
`x.store(1,MO_RELAXED)`	`y.store(1,MO_RELAXED)`
`r1 = y.load(MO_RELAXED)`	`r2 = x.load(MO_RELAXED)`

The program above **can** end with `r1 = r2 = 0`.

Sample compilation on x86:

store: **`MOV`**
load: **`MOV`**

Sample compilation on Power:

store: **`ST`**
load: **`LD`**

MO_RELEASE / MO_ACQUIRE

Supports a fast implementation of the message passing idiom:

Thread 0	Thread 1
`x.store(1,MO_RELAXED)`	`r1 = y.load(MO_ACQUIRE)`
`y.store(1,MO_RELEASE)`	`r2 = x.load(MO_RELAXED)`

The program above cannot end with `r1 = 1` and `r2 = 0`.

Accesses to the data structure can be reordered/optimised (`MO_RELAXED`).

Sample compilation on x86:

store: `MOV`
load: `MOV`

Sample compilation on Power:

store: `LWSYNC; ST`
load: `LD; CMP; BC; ISYNC`

MO_RELEASE / MO_CONSUME

Supports a fast implementation of the message passing idiom on Power:

Thread 0	Thread 1
`x.store(1,MO_RELAXED)`	`r1 = y.load(x,MO_CONSUME)`
`y.store(&x,MO_RELEASE)`	`r2 = (*r1).load(MO_RELAXED)`

The program above cannot end with `r1 = 1` and `r2 = 0`.

The two loads have an address dependency, Power won't reorder them.

Sample compilation on x86:

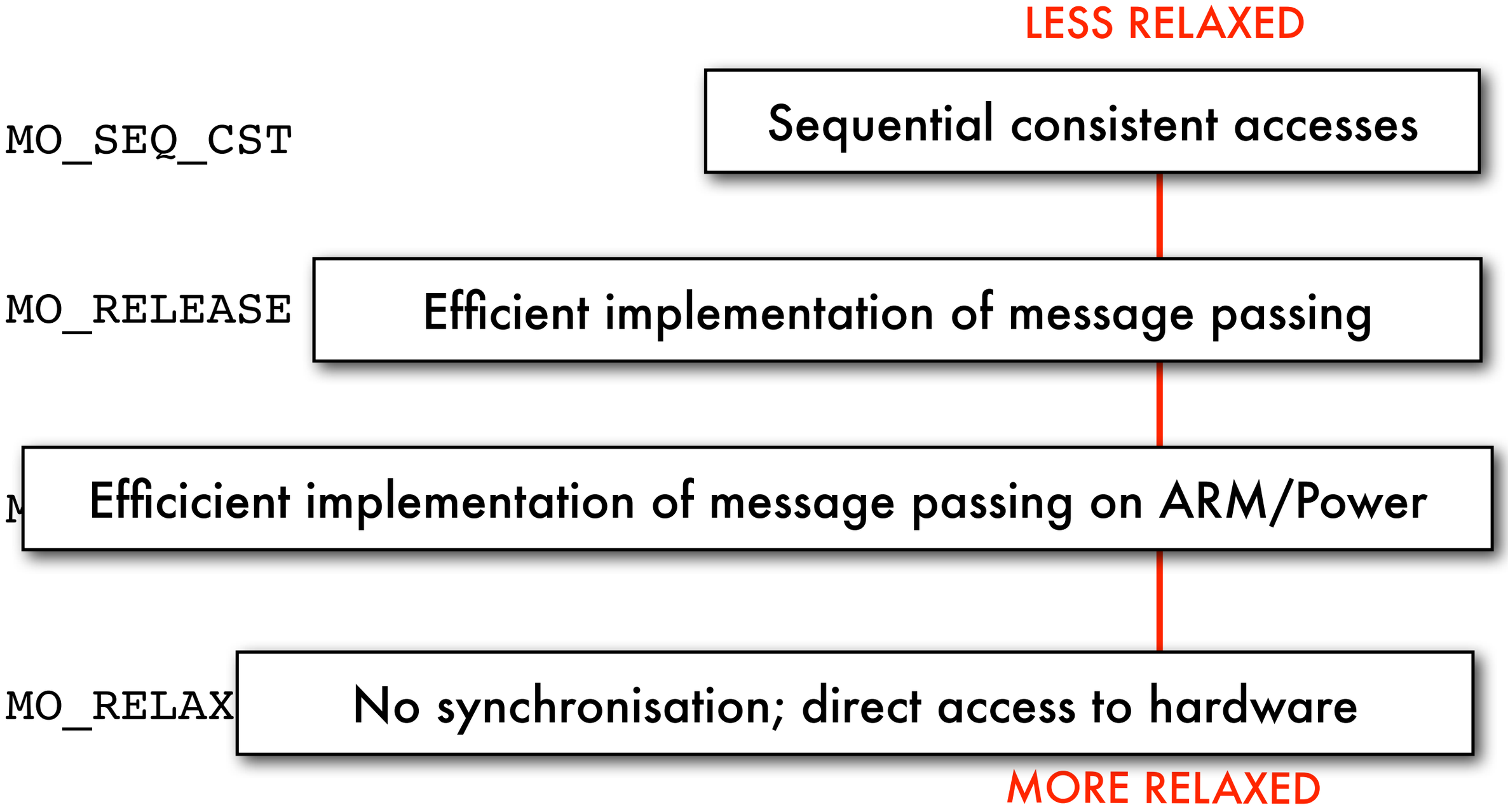
store: **`MOV`**
load: **`MOV`**

Sample compilation on Power:

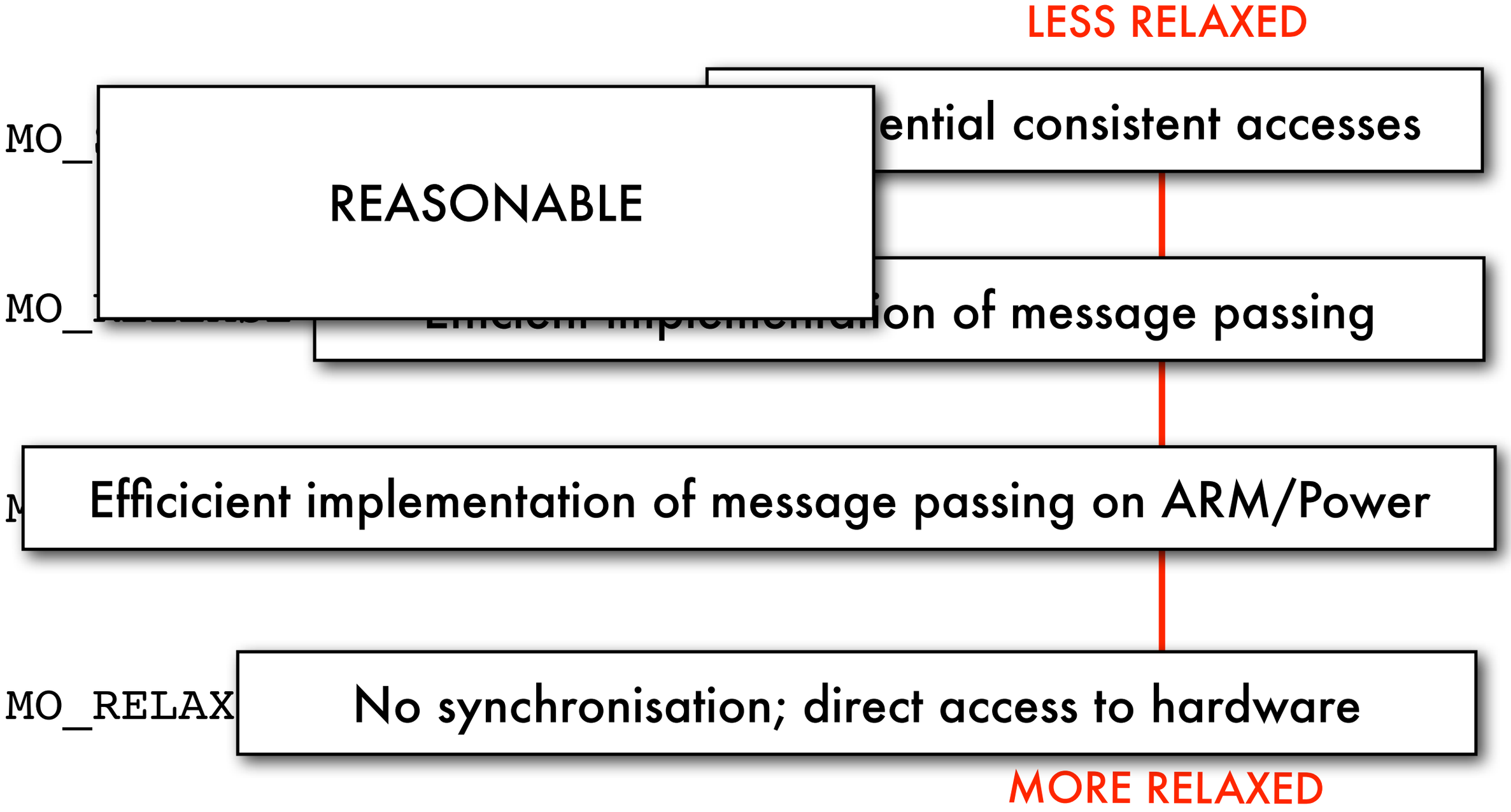
store: **`LWSYNC; ST`**
load: **`LD`**

Escape lanes pitfalls

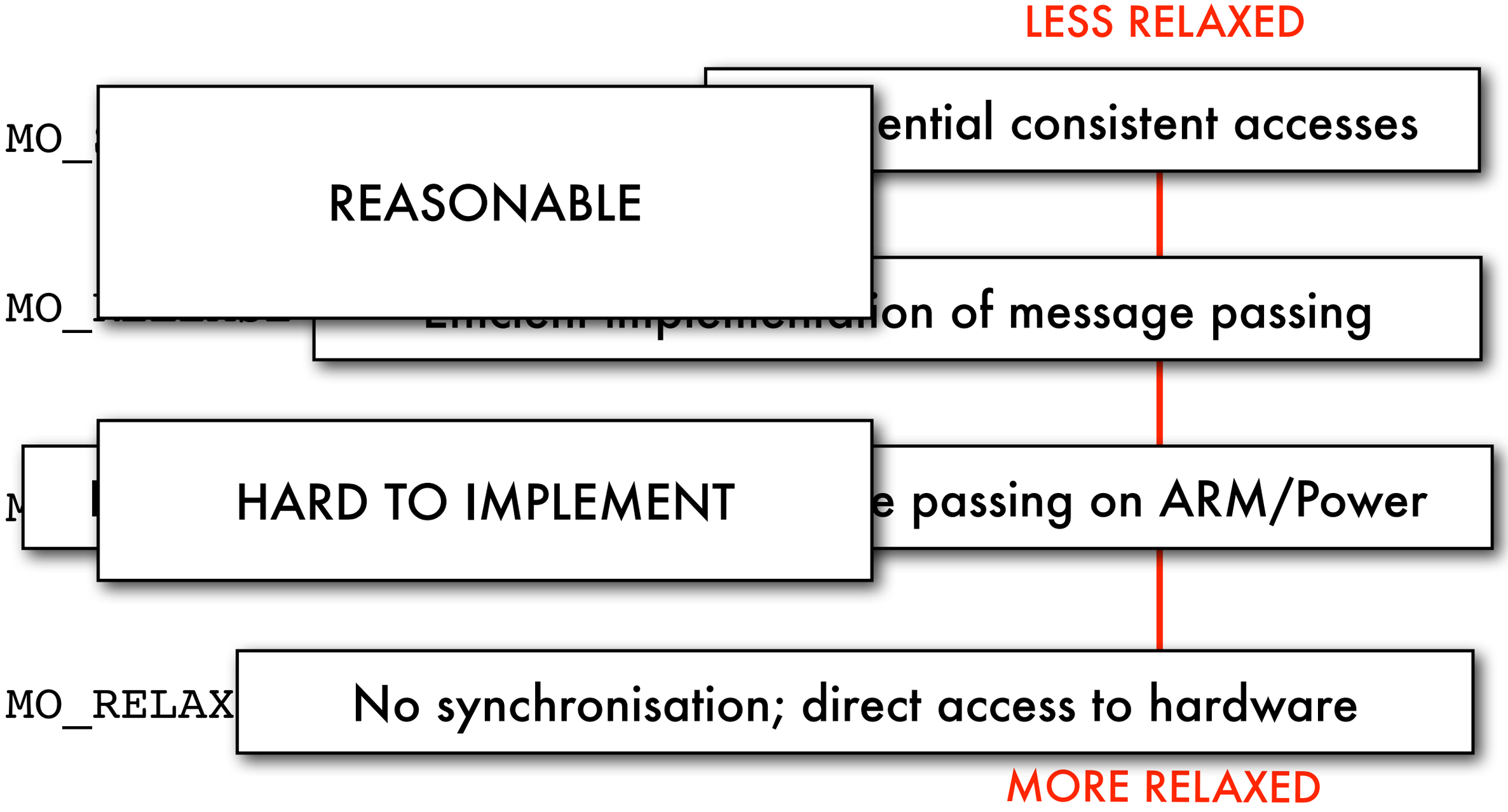
The qualifiers



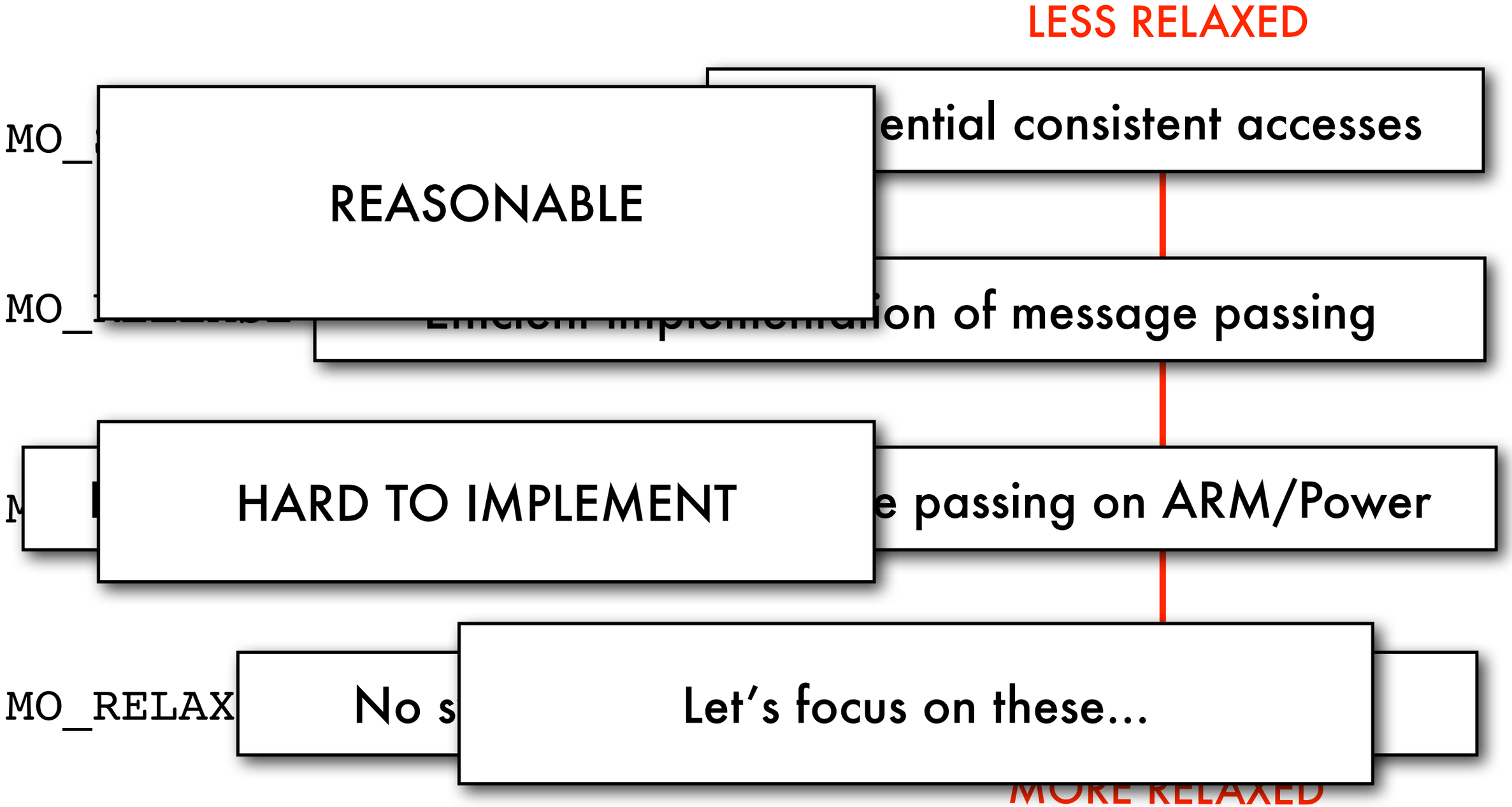
The qualifiers



The qualifiers



The qualifiers



Memory access synchronisation

```
x = y = 0
```

Thread 1

```
y = 1

x.store(1,MO_RELEASE)
```

Thread 2

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

   r2 = y
```

Memory access synchronisation

```
x = y = 0
```

Thread 1

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

Thread 2

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

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

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

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1 | Thread 2

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

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1

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

Thread 2

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

DATA RACE

Two conflicting accesses not related by happens-before

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1

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

Thread 2

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

WELL DEFINED

but r2 = 0 is possible

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1

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

Thread 2

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

Allow a RELAXED load to see any store that:

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

Intuition
the compiler (or hardware) can reorder independent accesses

```
x = y = 0
```

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

Allow a RELAXED load to see any store that:

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

Shorthand

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

Out of thin-air reads

Out-of-thin-air

```
x = y = 0
```

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

Out-of-thin-air

`x = y = 0`

Thread 1

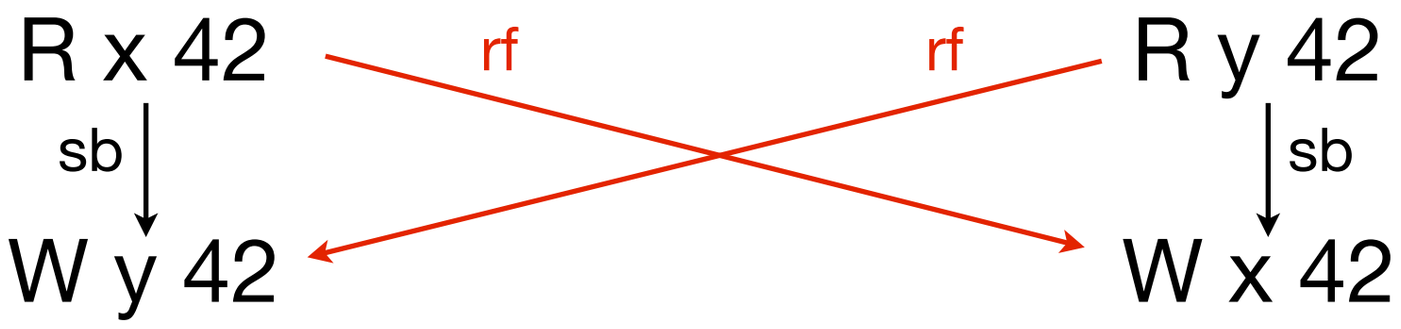
```
r1 = x
y = r1
```

Thread 2

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

r1 = r2 = 42

is a valid execution.



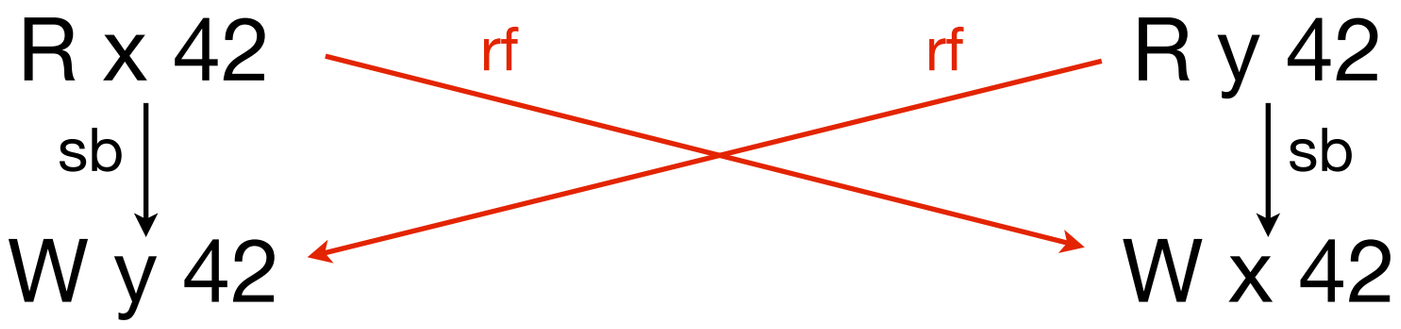
Intuition
the compiler (or hardware) can reorder independent accesses

`x = y = 0`

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

r1 = r2 = 42

is a valid execution.



Out-of-thin-air reads

Thread 1

```
x = y = 0
```

Thread 2

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

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

Out-of-thin-air reads

Thread 1

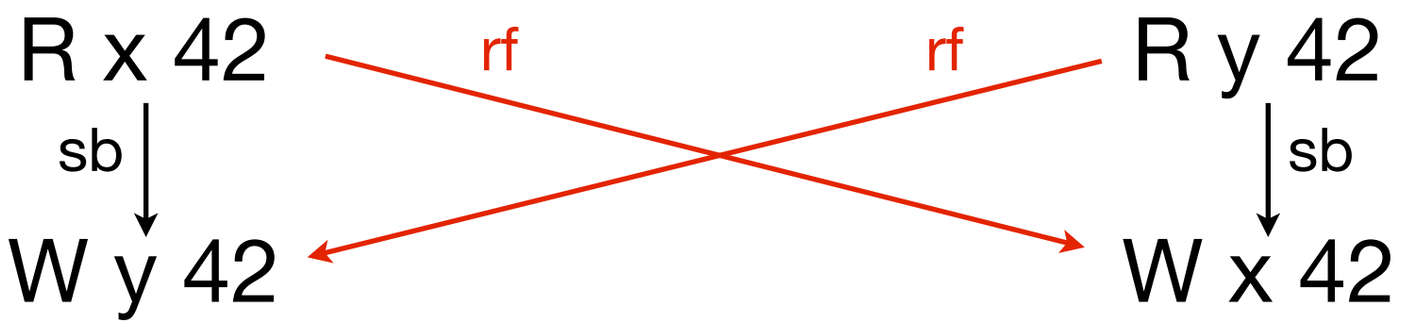
```
x = y = 0
```

Thread 2

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

r1 = r2 = 42

is also an allowed execution



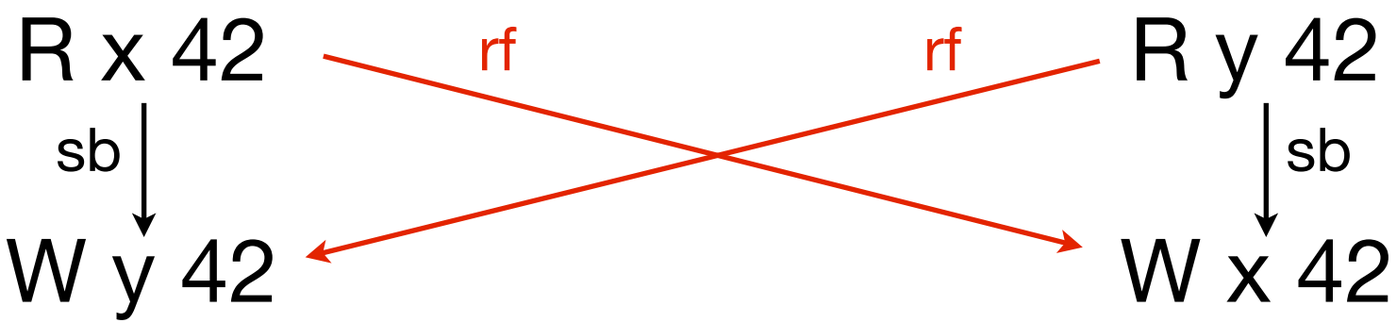
the value 42 appears *out-of-thin-air*

`x = y = 0`

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

r1 = r2 = 42

is also an allowed execution



Speculation can justify out-of-thin-air reads

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

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

initially x = y = 0

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

Speculation can justify out-of-thin-air reads

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

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

initially x = y = 0

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

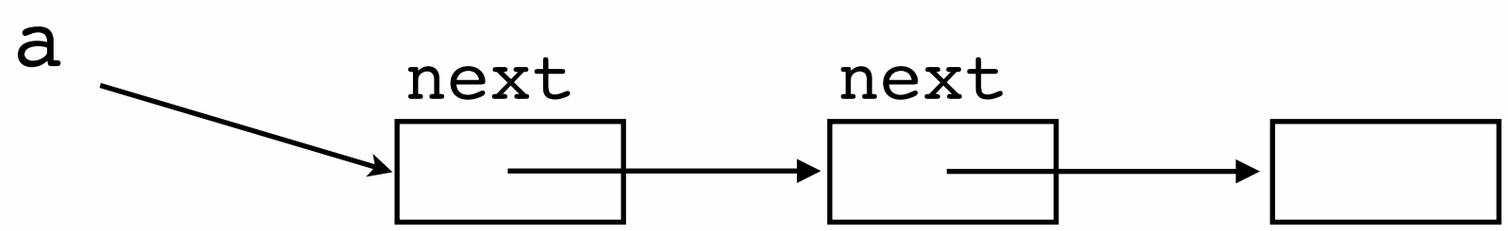
It does not happen in practice... even if it might!

Consequences of out-of-thin-air reads

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

Thread 1

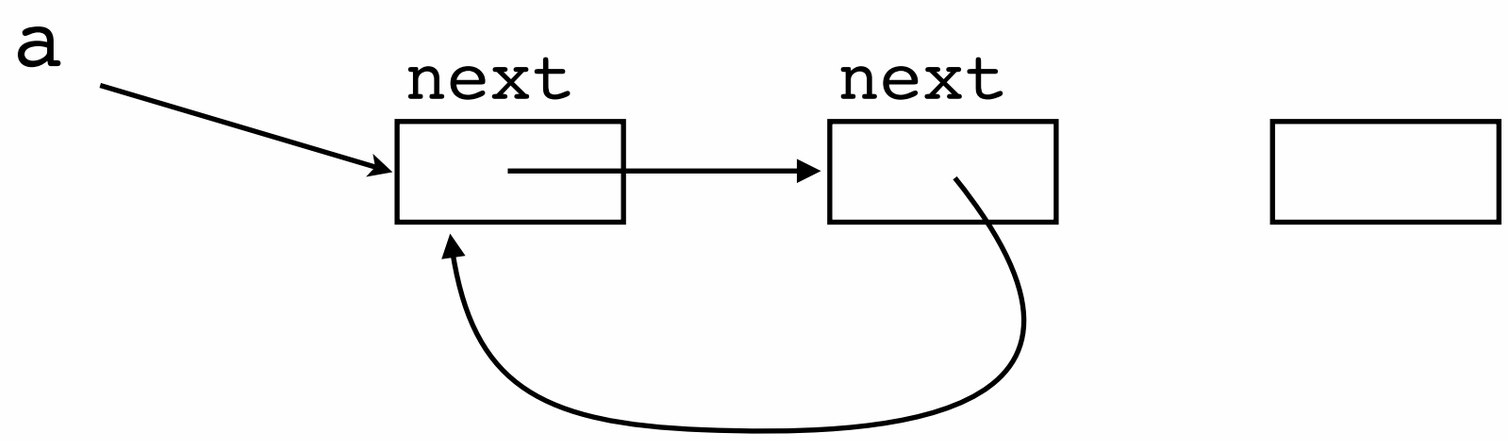
```
r1 = a->next
r1->next = a
```



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

Thread 1

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



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

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

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

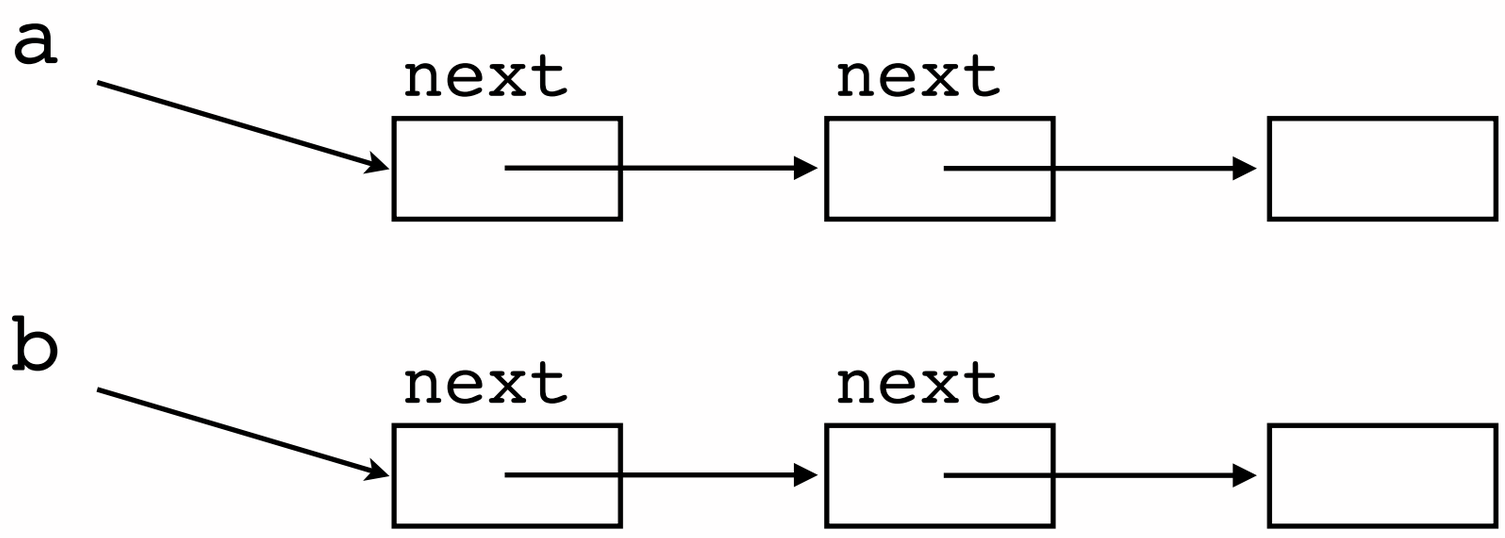
Thread 1	Thread 2
`r1 = a->next`	`r2 = b->next`
`r1->next = a`	`r2->next = b`

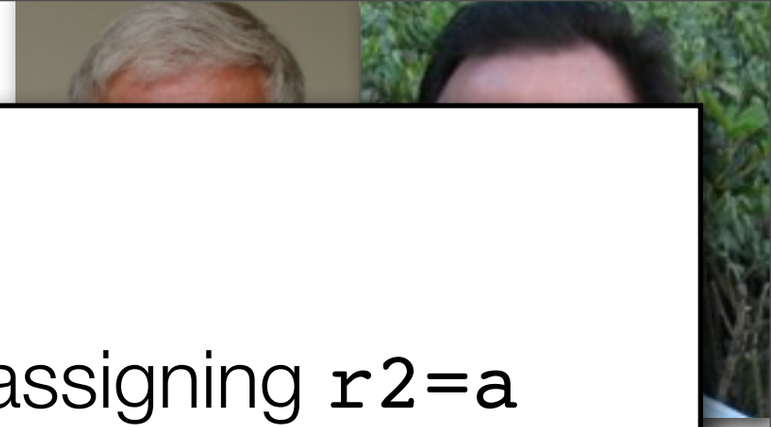
If **a** and **b** initially reference disjoint data-structures

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

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

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





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

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

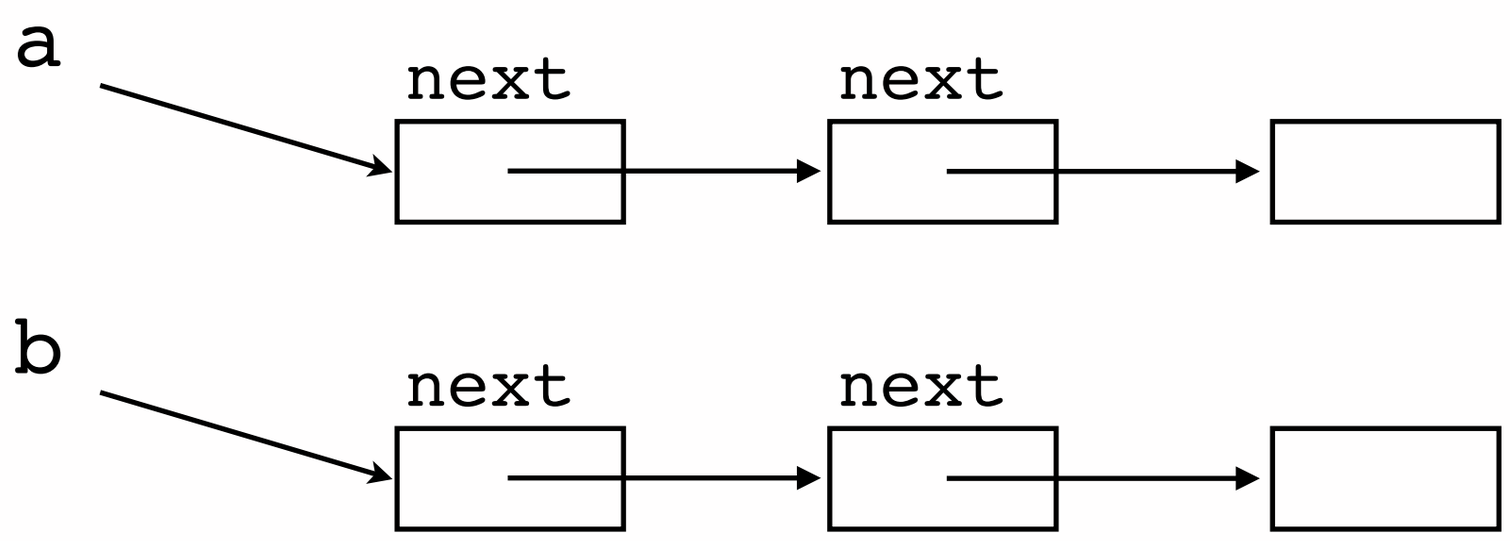
(and symmetrically to justify `r1=b`)

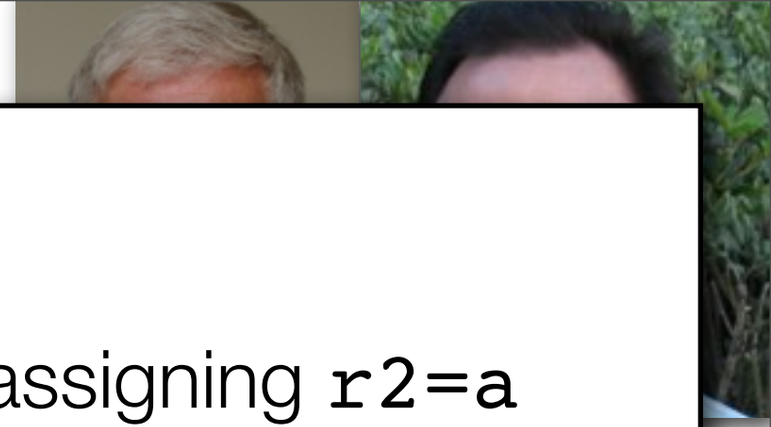
Thread 1

```
r1 = a->next

r1->next = a
```

Thread 2

```
r2 = b->next

r2->next = b
```





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

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

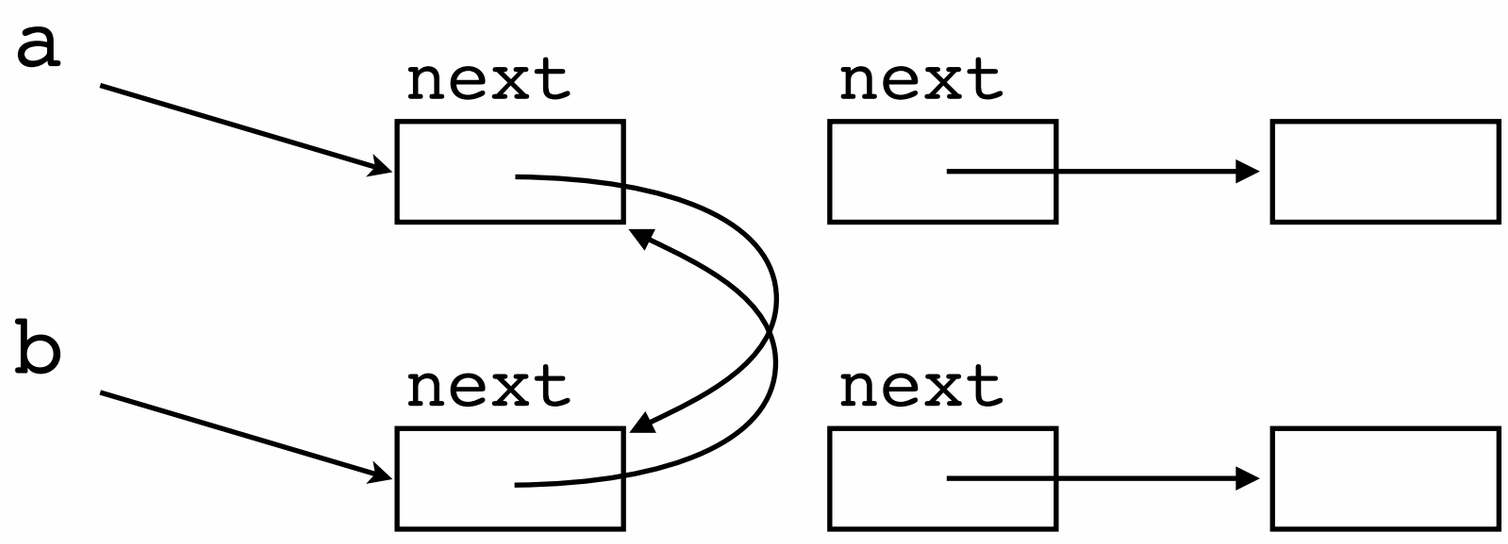
(and symmetrically to justify `r1=b`)

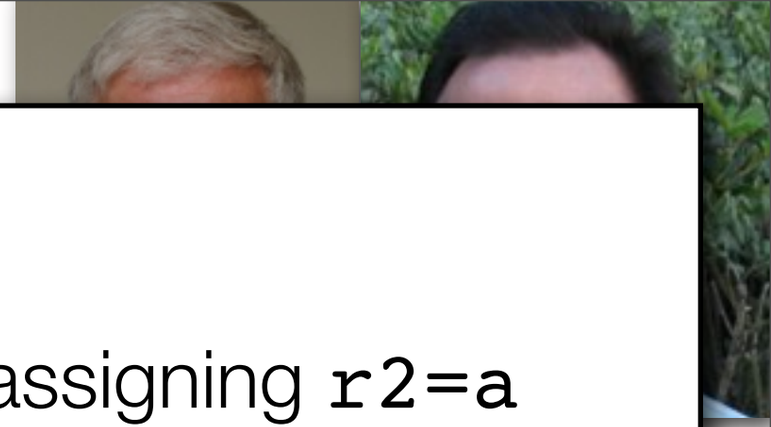
Thread 1

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

Thread 2

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



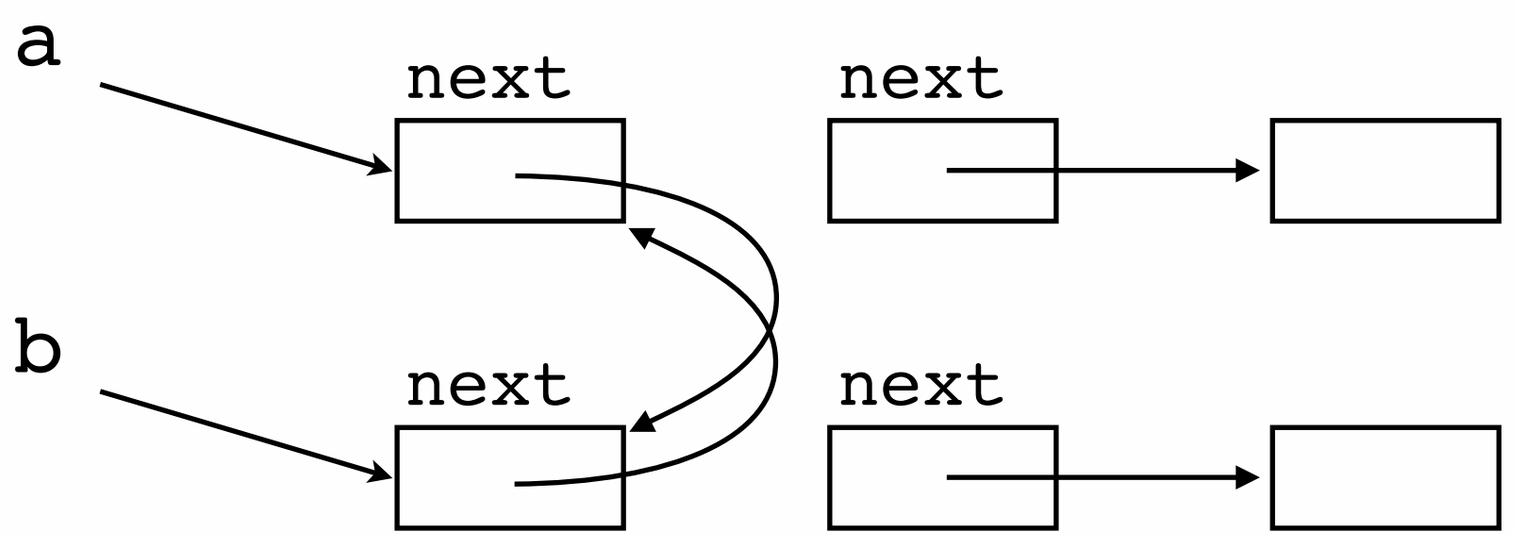


If the compiler speculates `r1=b` and `r2=a`, then

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

(and symmetrically to justify `r1=b`)

Break our basic intuitions about memory and sharing!



Common compiler optimisations are unsound in C11

x = y = a = 0

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

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

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

Remark 1

This code is not racy!

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

```
x = y = a = 0

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

Remark 2

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

is the only possible final state

```
                    x = y = a = 0

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

Consider sequentialisation:

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

(ought to be correct)

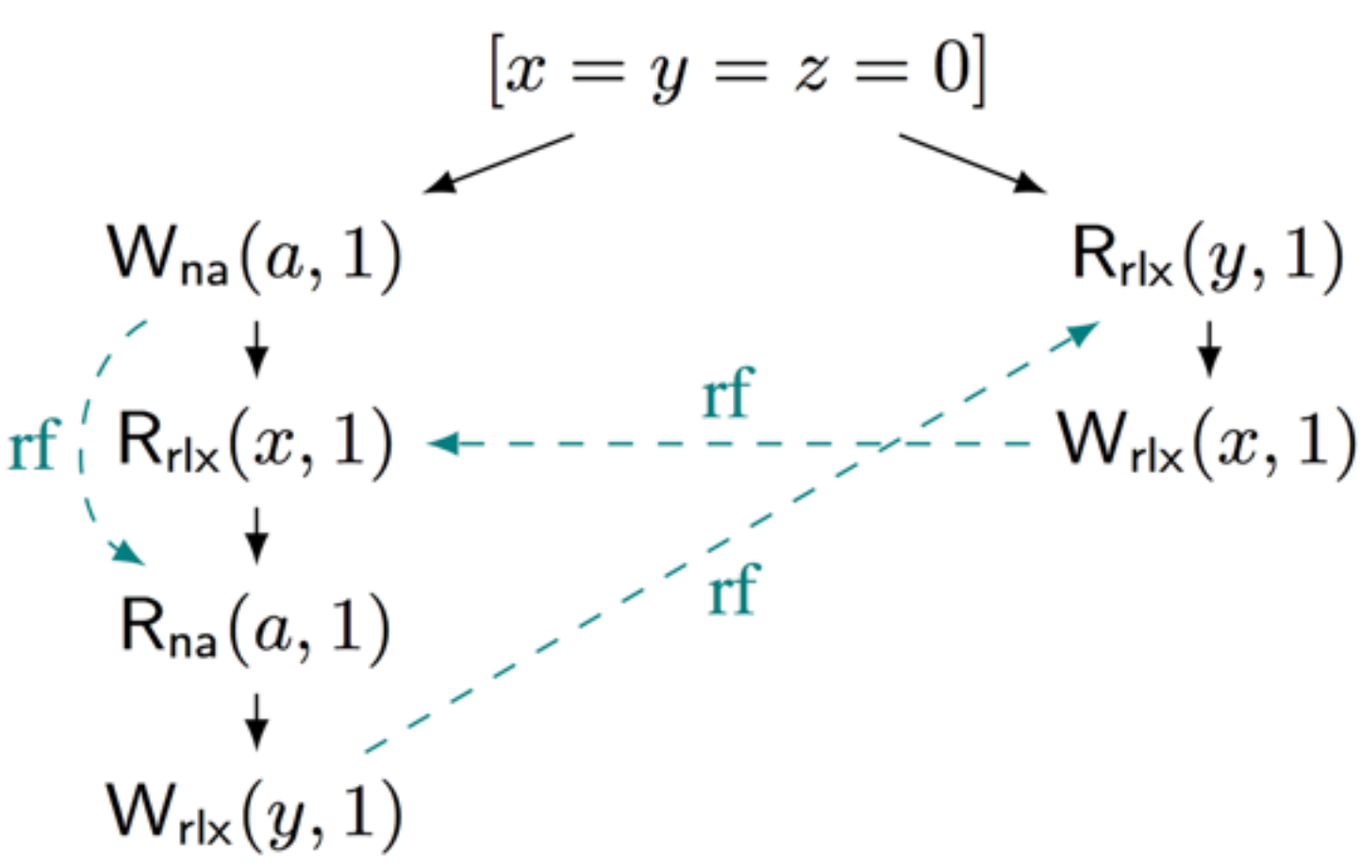
```
x = y = a = 0
```

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

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

```
                    x = y = a = 0

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

```
                    x = y = a = 0

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



a = 1

x = y = 42

is also possible

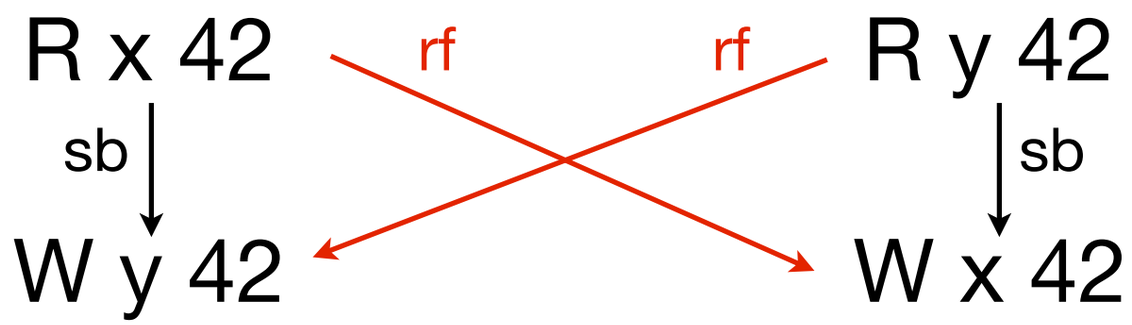
```
                    x = y = a = 0

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

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

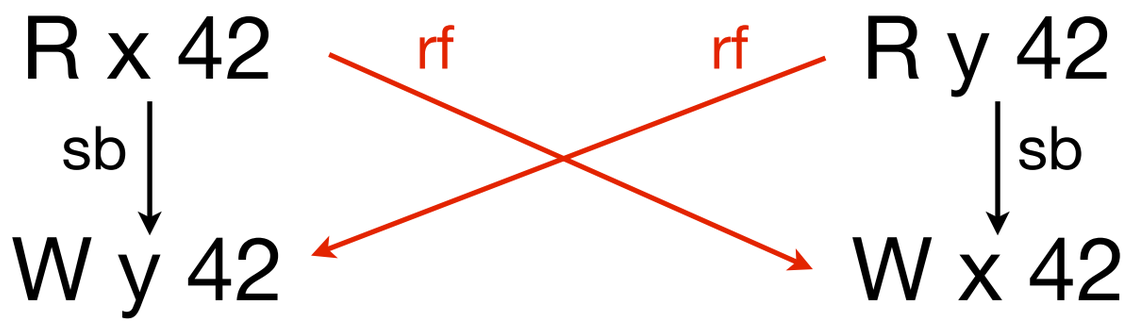
including *expression linearisation* and *roach-motel reorderings*

Are there any solutions?

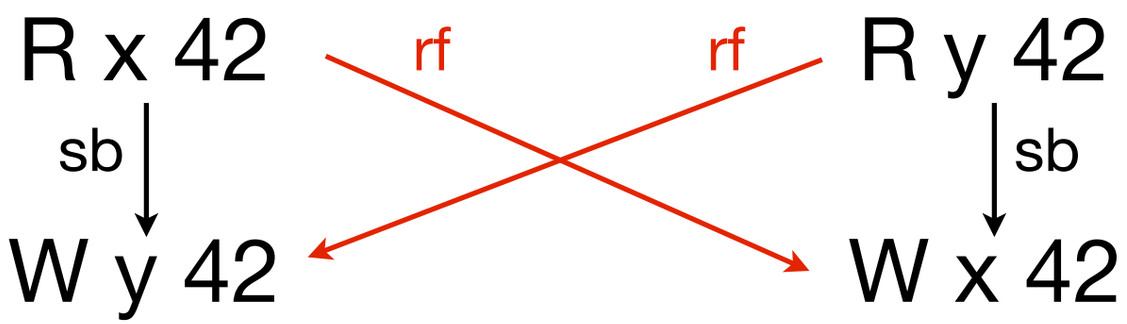
Thread 0	Thread 1
`r1 = x`	`r2 = y`
`y = r1`	`x = 42`



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

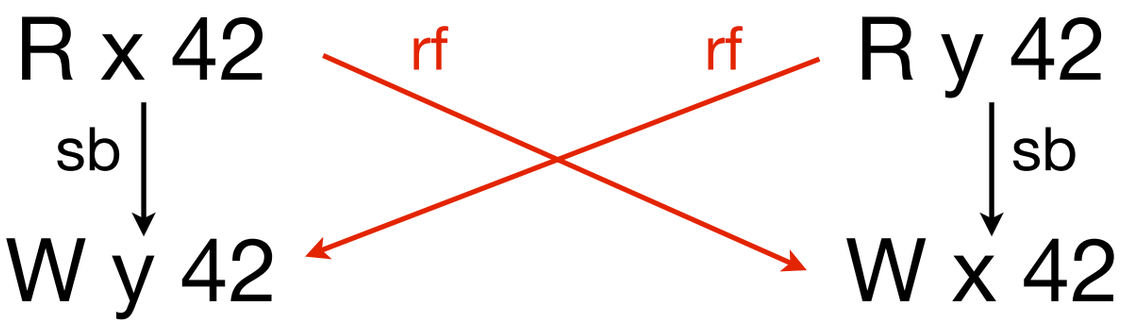


Thread 0	Thread 1
`r1 = x`	`r2 = y`
`y = r1`	`x = 42`

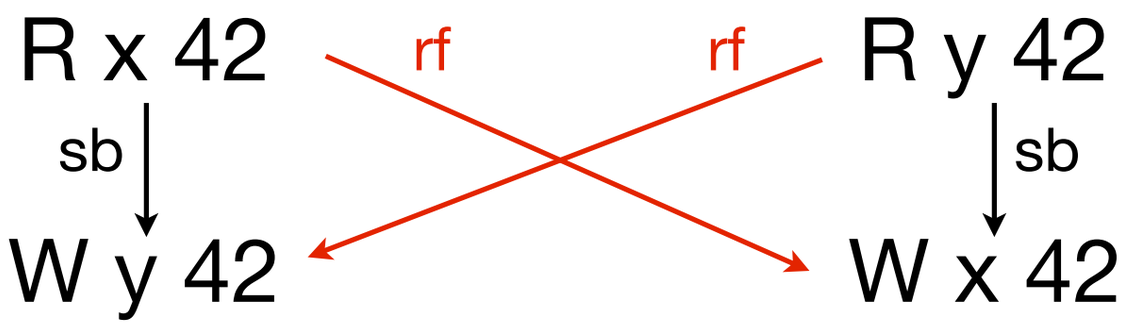


r1 = r2 = 42. Can you spot the difference?

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

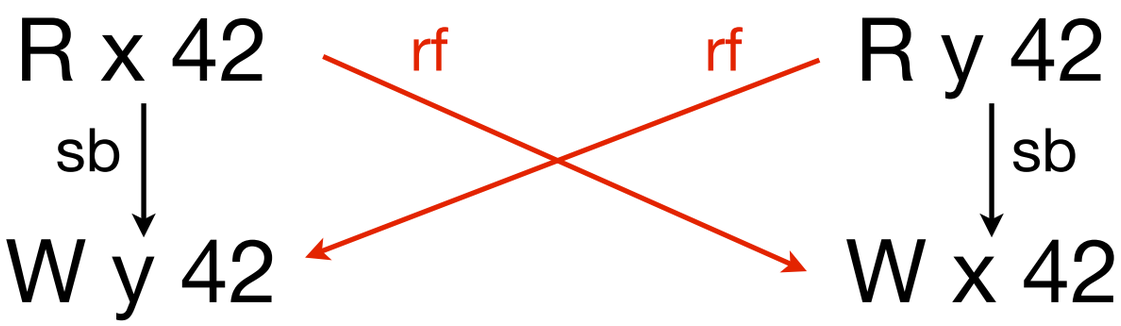


Thread 0	Thread 1
`r1 = x`	`r2 = y`
`y = r1`	`x = 42`



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

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

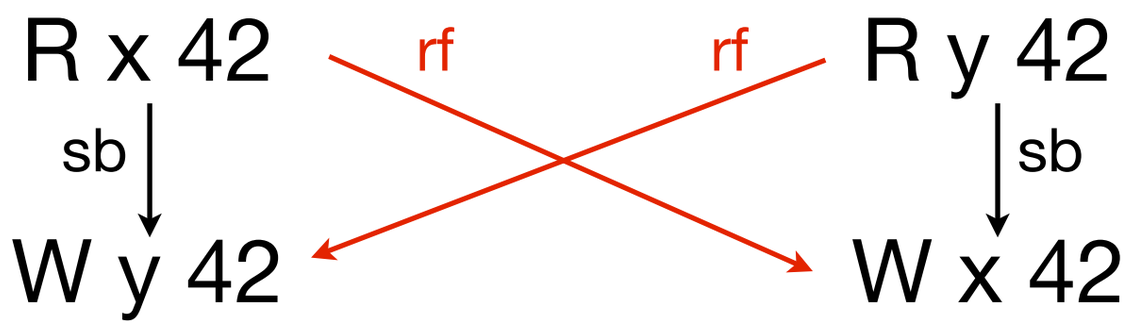


Solution 1.

Prohibit executions with dependency cycles

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

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



Compiler writers
do not want to track all dependencies

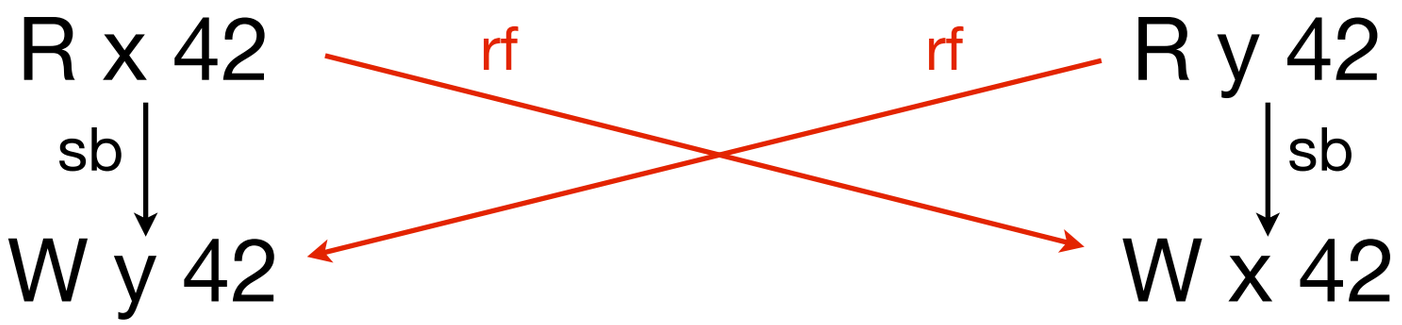
Compiler writers
do not want to track all dependencies

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

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

Solution 2. Brute force

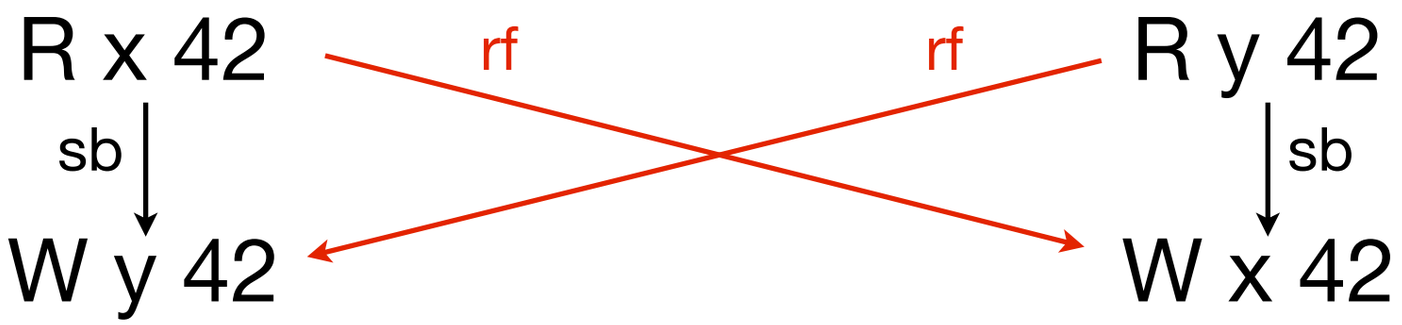
Disallow cycles altogether



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

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

Disallow cycles altogether



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

Solution 3. less brute force

Allow cycles but make this racy

by allowing a to read 1

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

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

Allow cycles but make this racy

by allowing a to read 1

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

State of the art

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

Current proposal for C++14

Currently, there is no really satisfactory proposal for the semantics of a general-purpose shared-memory concurrent programming language.

Currently, there is no really satisfactory proposal for the semantics of a general-purpose shared-memory concurrent programming language.

Remarkable and disturbing.

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.

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`

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

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`

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

C11 states that this program *must* print `42`

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

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

gcc 4.7 -O2

The outer loop can be (and is) optimised away

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

The compiled code saves and restores `b`

Correct result in a sequential setting

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

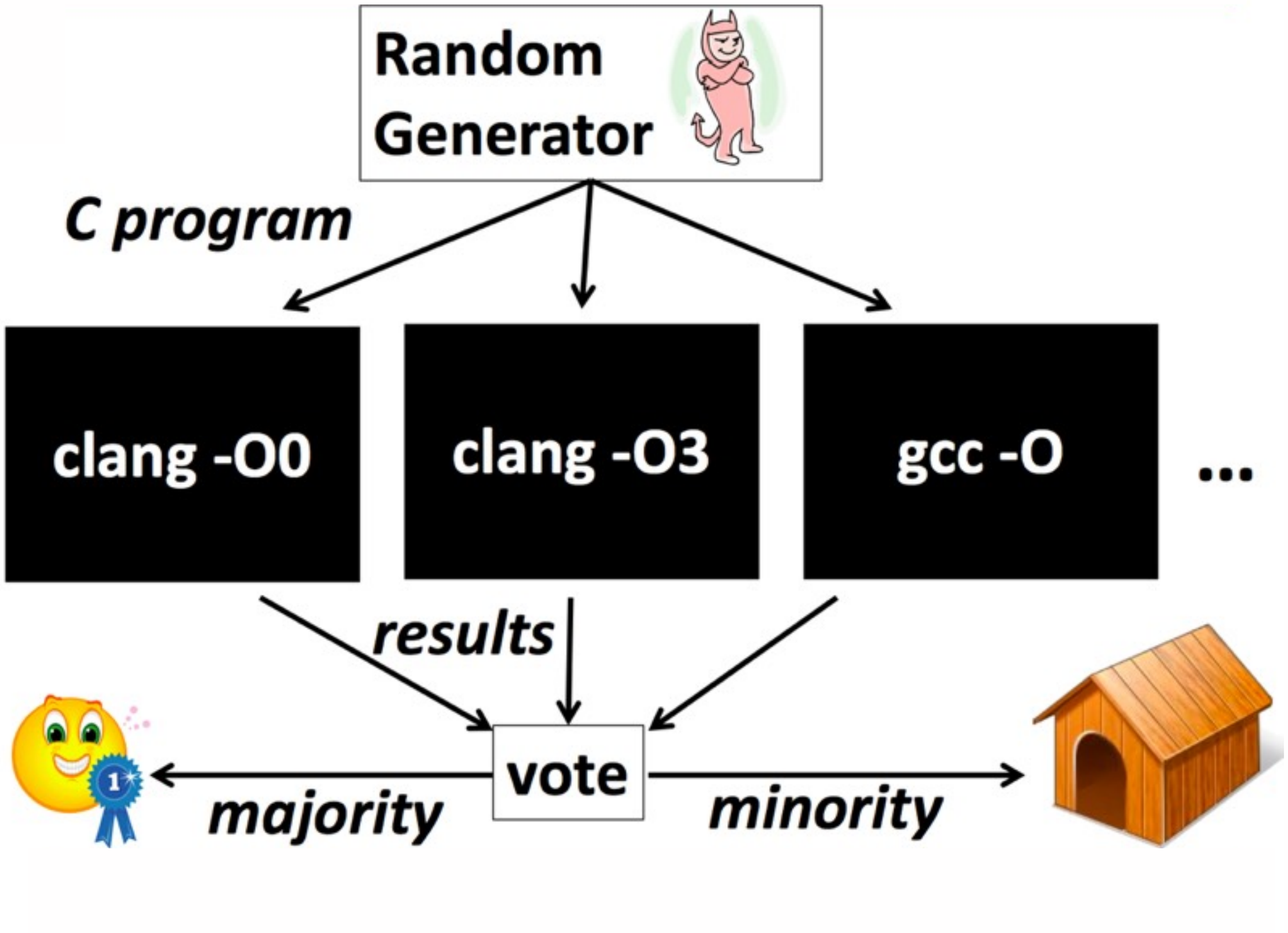
```
b = 42;
printf("%d\n", b);
```

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

The horror, the horror... a subtle compiler bug!

Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

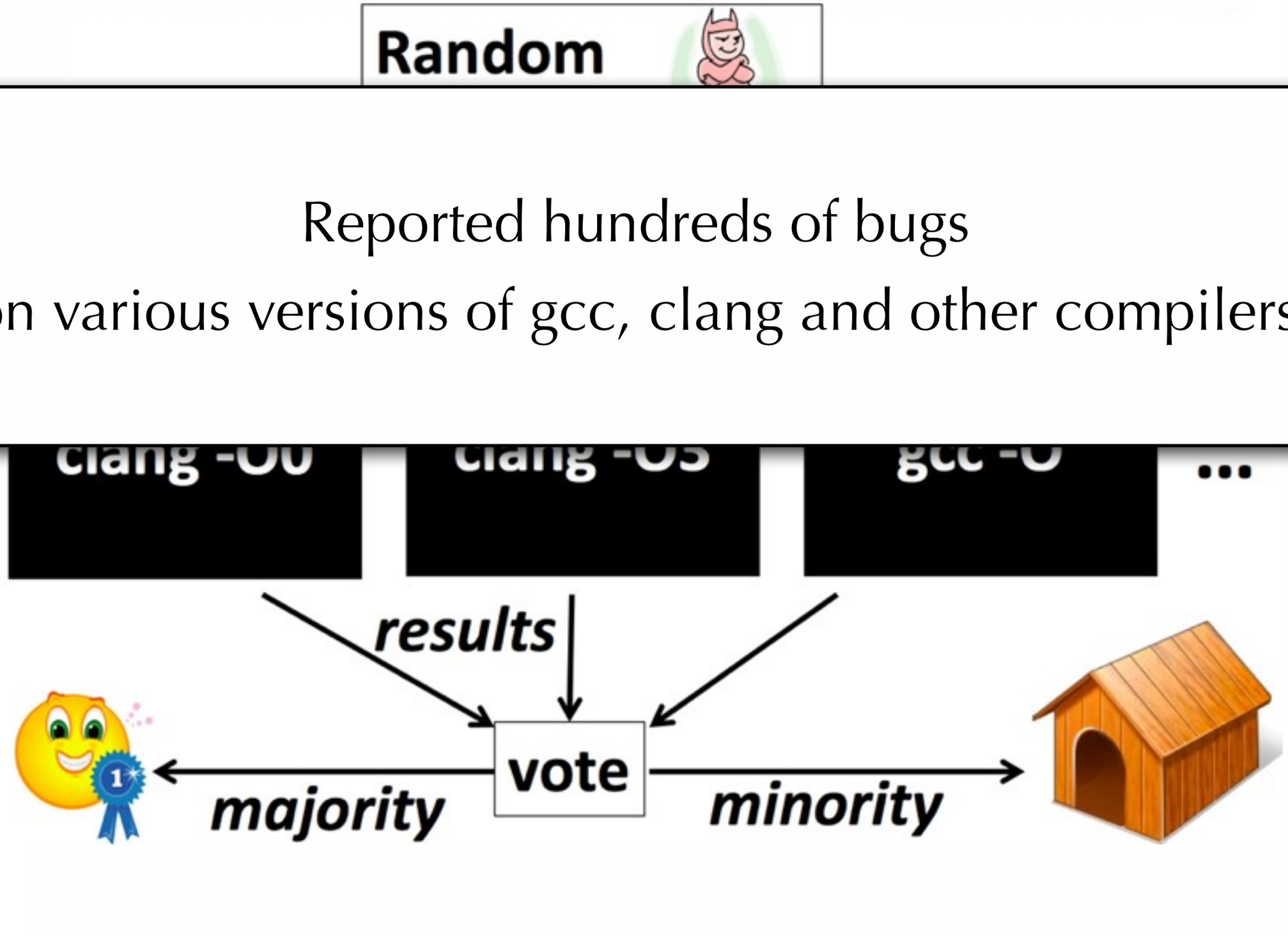


Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

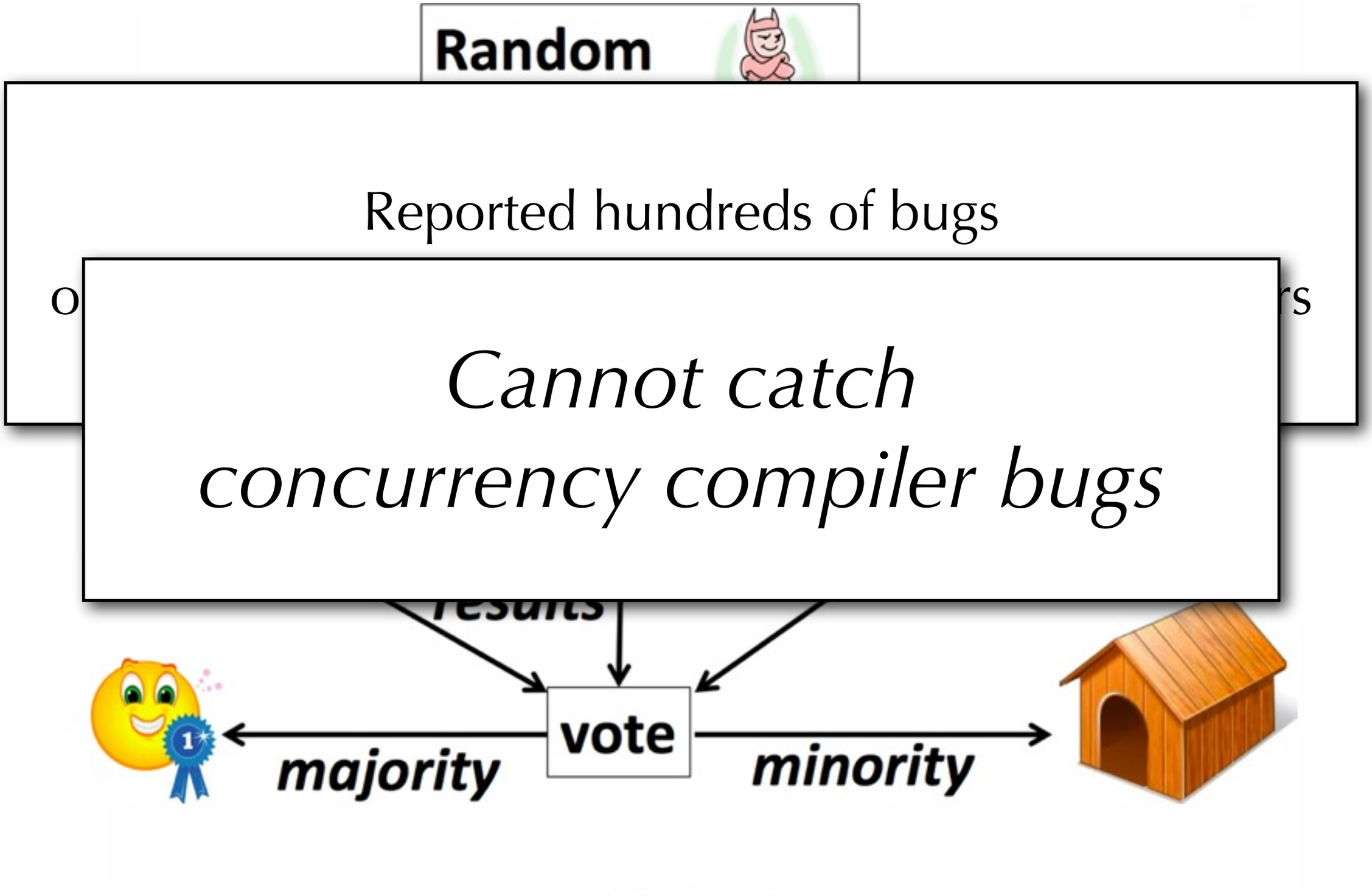
Reported hundreds of bugs

on various versions of gcc, clang and other compilers



Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



Hunting *concurrency* compiler bugs?

How to deal with non-determinism?

How to generate non-racy interesting programs?

How to capture all the behaviours of concurrent programs?

A compiler can optimise away behaviours:

how to test for correctness?

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

Idea

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

↓

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

Hunt concurrency compiler bugs

=

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

Random Generator

SEQUENTIAL PROGRAM

optimising compiler under test

reference semantics

EXECUTABLE

tracing

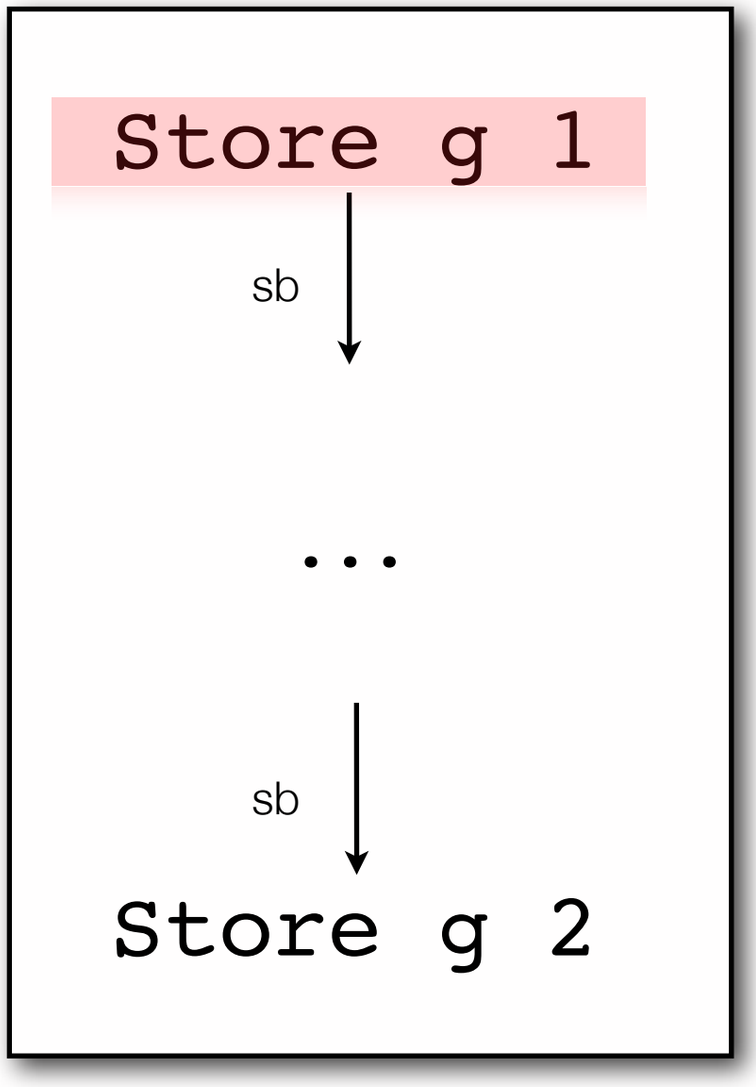
REFERENCE MEMORY TRACE

MEMORY TRACE

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

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

Elimination of *overwritten writes*



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

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

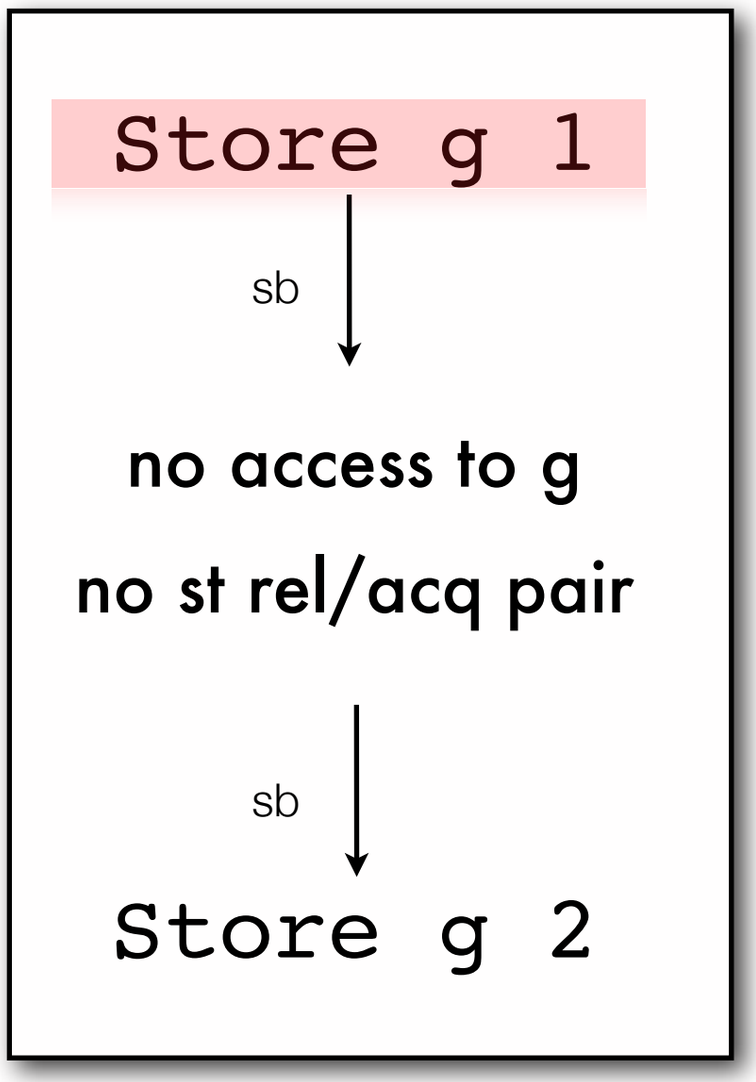
An action is a *release* if it is a possible source of a synchronisation

unlock mutex, release or seq_cst atomic write

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

lock mutex, acquire or seq_cst atomic read

Elimination of *overwritten writes*



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

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

The soundness condition

Shared memory

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

Thread 1

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

The soundness condition

Shared memory

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

Thread 1

candidate overwritten write

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

The soundness condition

Shared memory

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

Thread 1

candidate overwritten write

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

same-thread release-acquire pair

The soundness condition

Shared memory

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

Thread 1

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

Thread 2

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

The soundness condition

Shared memory

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

Thread 1

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

Thread 2

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

Thread 2 is non-racy

The soundness condition

Shared memory

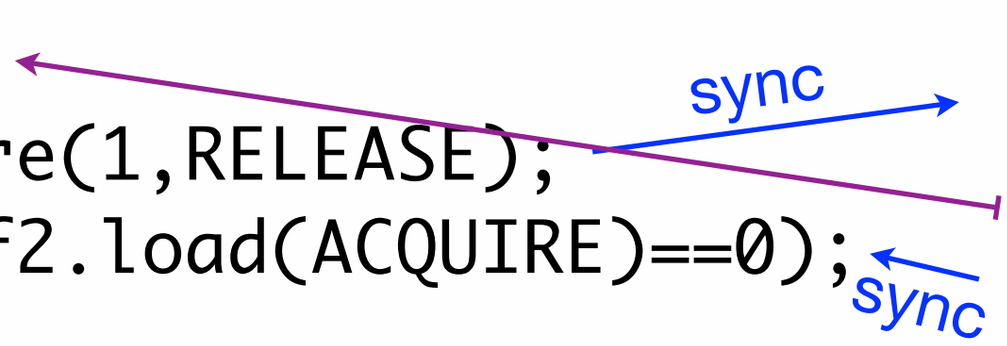
```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

Thread 2

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



Thread 2 is non-racy

The program should only print 1

The soundness condition

Shared memory

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

Thread 1

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

Thread 2

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

sync

sync

Thread 2 is non-racy

The program should only print 1

If we perform overwritten write elimination it prints 0

The soundness condition

Shared memory

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

Thread 1

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

sync →

Thread 2

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

The soundness condition

Shared memory

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

Thread 1

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

g = 2;
```

sync →

Thread 2

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

The soundness condition

Shared memory

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

Thread 1

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

g = 2;
```

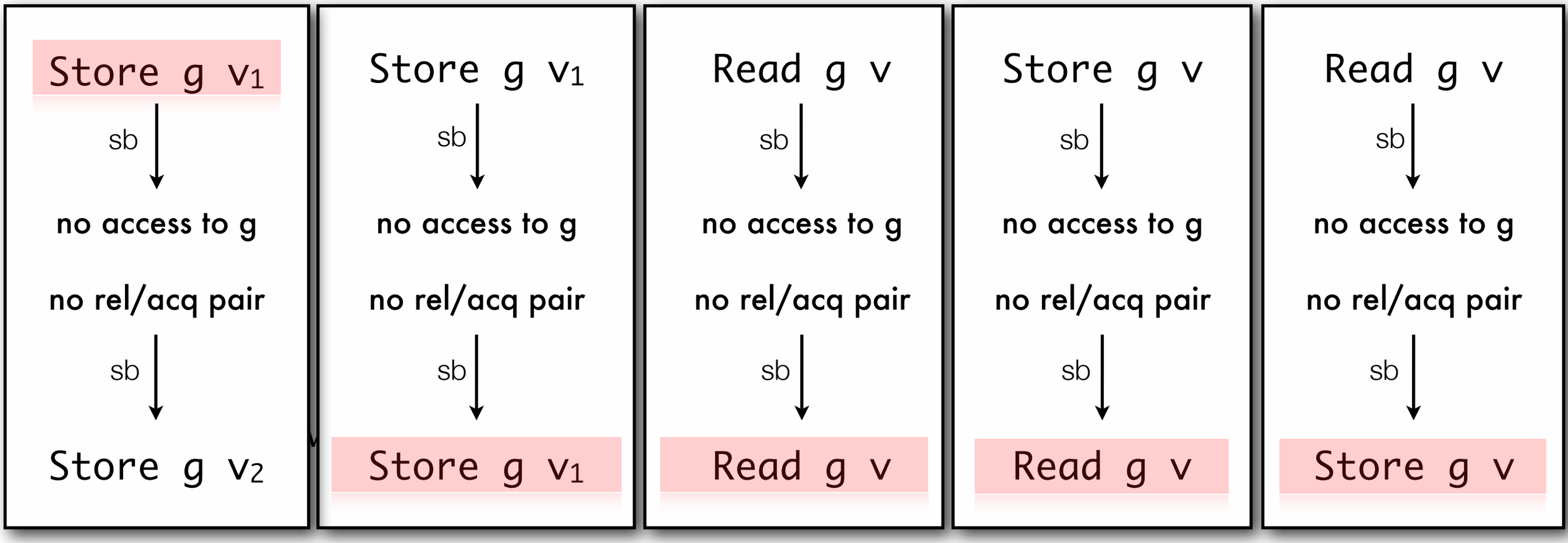
Thread 2

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

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

It is sound to optimise the overwritten write.

Eliminations: bestiary

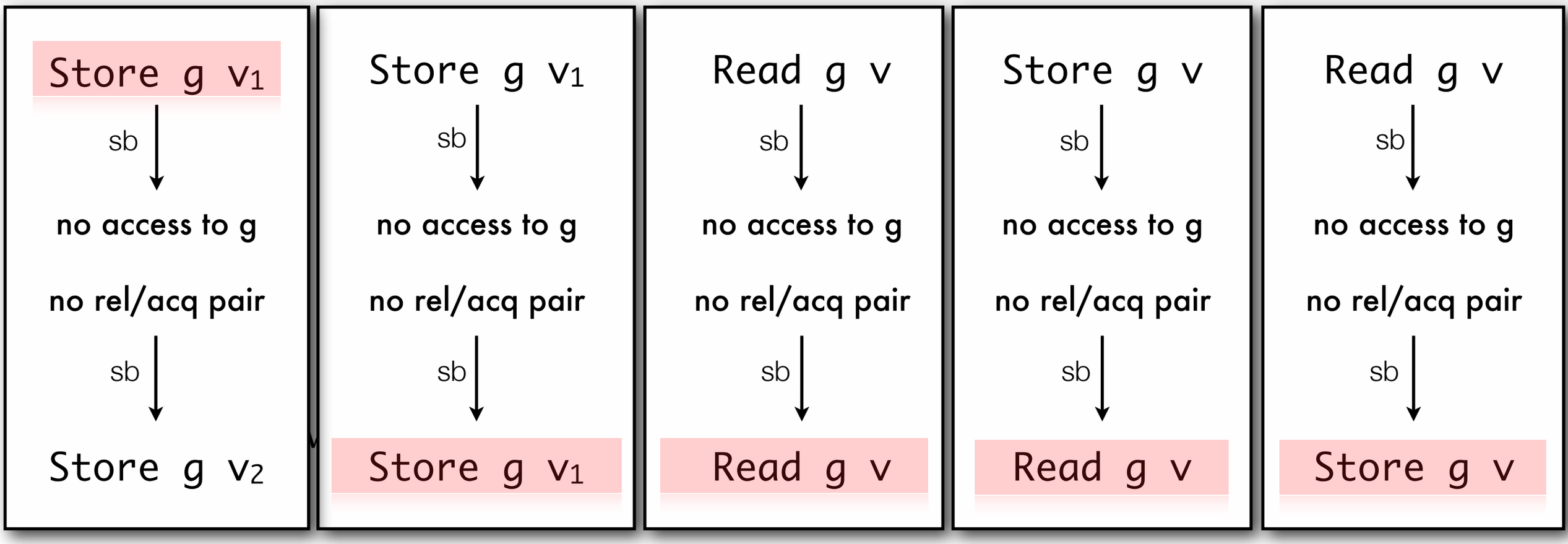


Overwritten-Write Write-after-Write Read-after-Read Read-after-Write Write-after-Read

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

Also correctness statements for

reorderings, **merging**, and **introductions** of events.



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

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

From theory to the Cmmtest tool

Random Generator

SEQUENTIAL PROGRAM

optimising compiler under test

reference semantics

EXECUTABLE

tracing

REFERENCE MEMORY TRACE

MEMORY TRACE

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

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

optimising
compiler
under test

reference
semantics

EXECUTABLE

tracing

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

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

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

optimising
compiler
under test

reference
semantics

EXECUTABLE

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

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

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

gcc/clang -O0

optimising
compiler
under test

EXECUTABLE

EXECUTABLE

binary
instrumentation

REFERENCE
MEMORY
TRACE

binary
instrumentation

MEMORY
TRACE

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

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

gcc/clang -O0

optimising
compiler
under test

EXECUTABLE

EXECUTABLE

binary
instrumentation

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

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

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

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

Start with a randomly generated well-defined program

```
const unsigned int g3 = 0UL;  void func_1(void){
long long g4 = 0x1;                 int *l8 = &g6;
int g6 = 6L;                        int l36 = 0x5E9D070FL;
volatile unsigned int g5 = 1UL;     unsigned int l107 = 0xAA37C3ACL;
                                    g4 &= g3;
                                    g5++;
                                    int *l102 = &l36;
                                    for (g6 = 4; g6 < (-3); g6 += 1);
                                    l102 = &g6;
                                    *l102 = ((*l8) && (l107 << 7)*(*l102));
                                }
```

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

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

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

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

reference semantics

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

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

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

reference semantics

gcc -O2 memory trace

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

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

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

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

reference semantics

gcc -O2 memory trace

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

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

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

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

reference semantics

gcc -O2 memory trace

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

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

```
Init g3 0
```

```
void func_1(void){
    int *l8 = &g6;
```

Can match applying
only correct eliminations and reorderings

```
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

reference semantics

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

gcc -O2 memory trace

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

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

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

If we focus on the miscompiled initial example...

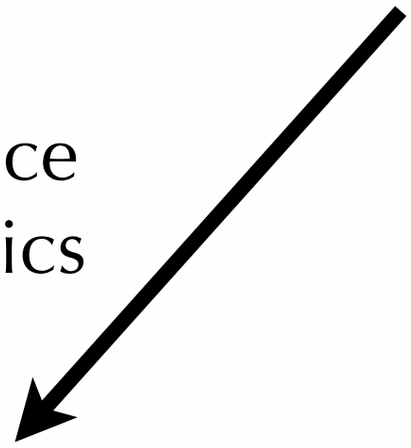
```
int a = 1;
int b = 0;
```

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

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

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

reference semantics



```
Load a 1
```

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

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

reference semantics

```
Load a 1
```

gcc -O2 memory trace

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

Cannot match some events ⟶ detect compiler bug

```
;
}
```

reference semantics

```
Load a 1
```

gcc -O2 memory trace

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

Applications

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

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

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

Remark: these bugs break the Posix thread model too.

All promptly fixed.

2. Checking compiler invariants

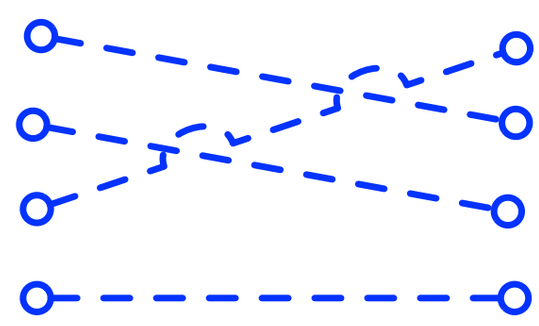
GCC internal invariant: never reorder with an atomic access

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

...not a bug, but fixed anyway

```
atomic_uint a;
int32_t g1, g2;

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

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



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

3. Detecting unexpected behaviours

```
uint16_t g                          uint16_t g

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

Correct or not?

3. Detecting unexpected behaviours

```
uint16_t g                          uint16_t g
for (; g==0; g--);      ⟶          g=0;
```

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

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

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

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

3. Detecting unexpected behaviours

```
uint16_t g                          uint16_t g
for (; g==0; g--);      ⟶          g=0;
```

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

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

False positives in Thread Sanitizer

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

Proving the correctness of mappings for atomics

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

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

Inform new optimisations

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

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

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

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

Inform new optimisations

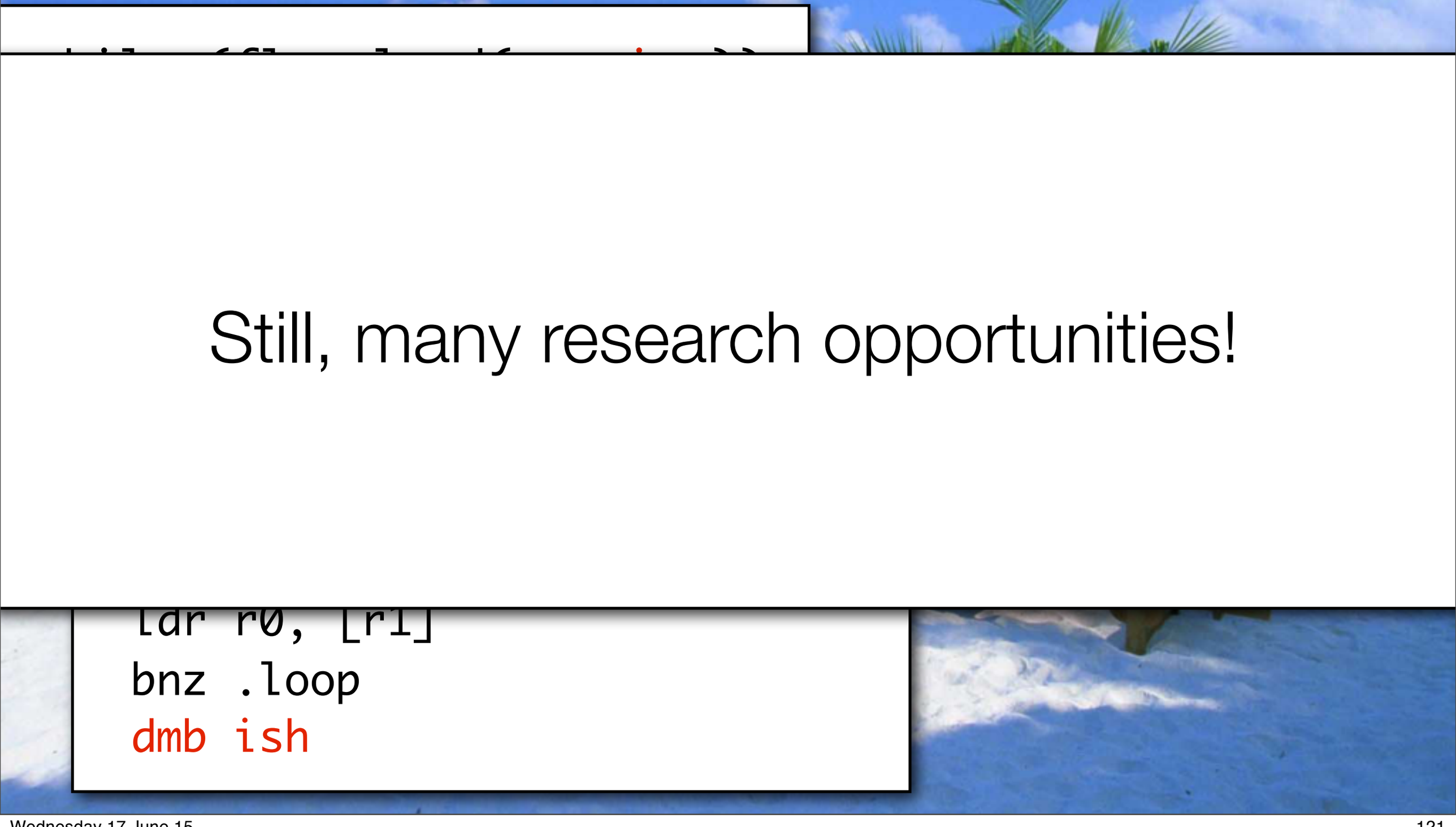
e.g. the work by Robin Morisset on the Arm LLVM backend

Still, many open problems...

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

Inform new optimisations

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



Still, many research opportunities!

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

Thank you! Questions?