High-level languages, compilers, multiprocessors... an elusive mix?

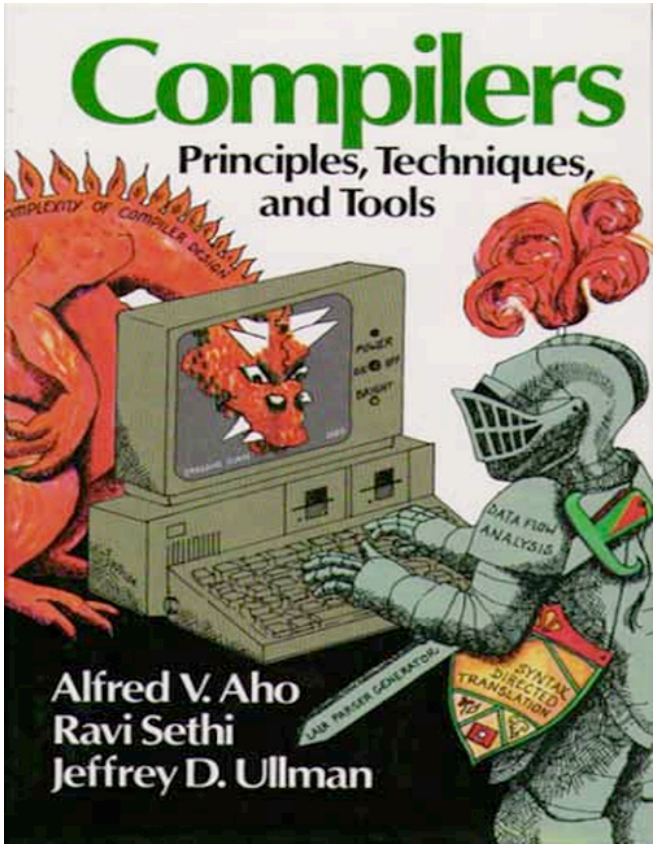
Francesco Zappa Nardelli INRIA Paris

http://www.di.ens.fr/~zappa/projects/weakmemory

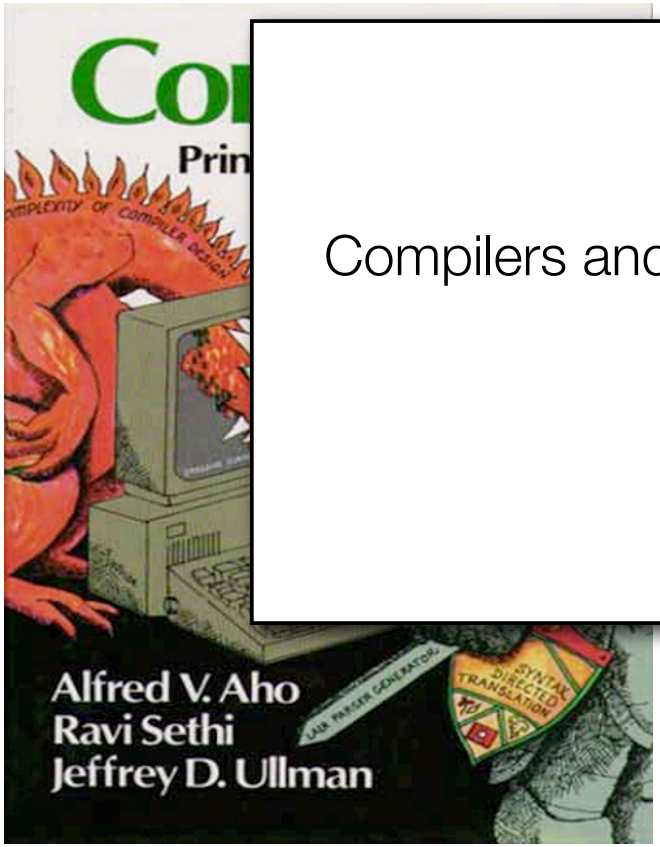
Based on work done by or with

Peter Sewell, Jaroslav Ševčík, Susmit Sarkar, Tom Ridge, Scott Owens, Viktor Vafeiadis, Magnus O. Myreen, Kayvan Memarian, Luc Maranget, Derek Williams, Pankaj Pawan, Thomas Braibant, Mark Batty, Jade Alglave.

Compilers vs. programmers

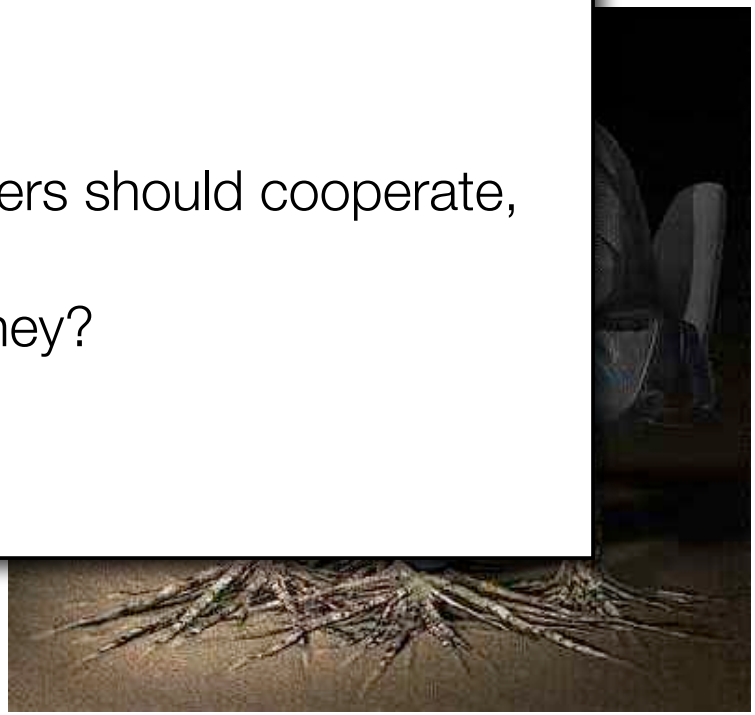


Compilers vs. programmers



Compilers and programmers should cooperate,

don't they?



Constant propagation (an optimising compiler breaks your program)

A simple and innocent looking optimization:

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

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

Constant propagation (an optimising compiler breaks your program)

A simple and innocent looking optimization:

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

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

Consider the two threads below:

`x = y = 0`

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

Intuitively, this program always prints `0`

Constant propagation (an optimising compiler breaks your program)

A simple and innocent looking optimization:

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

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

Consider the two threads below:

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

Sun HotSpot JVM or GCJ: always prints `1`.

Background: lock and unlock

- Suppose that two threads increment a shared memory location:

`x = 0`

`tmp1 = *x;` `*x = tmp1 + 1;`	`tmp2 = *x;` `*x = tmp2 + 1;`

- If both threads read `0`, (even in an ideal world) `x == 1` is possible:

```
tmp1 = *x;  tmp2 = *x;  *x = tmp1 + 1;  *x = tmp2 +1
```

Background: lock and unlock

- **Lock** and **unlock** are primitives that prevent the two threads from interleaving their actions.

`x = 0`

`lock();`
`tmp1 = *x;`
`*x = tmp1 + 1;`
`unlock();`
	`lock();`
	`tmp2 = *x;`
	`*x = tmp2 + 1;`
	`unlock();`

- In this case, the interleaving below is forbidden, and we are guaranteed that `x == 2` at the end of the execution.

FORBIDDEN

`tmp1 = *x;` `tmp2 = *x;` `*x = tmp1 + 1;` `*x = tmp2 +1`

Lazy initialisation (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_;
};
```

Exercise: why is it broken?

Clever programmers use double-check locking

```
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

The instruction

```
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_`).

Thread 2 attempts to reference an object that is not there yet!

The fundamental problem

Problem: You 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 (1)

Consider misaligned 4-byte accesses

`int32_t a = 0`	
`a = 0x44332211`	`if (a == 0x00002211)` `  print "error"`

(Disclaimer

Intel SDM

- *n*-byte
- P6 or I

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

This is called a *out-of-thin air read*:

the program reads a value
that the programmer never wrote.

That pesky hardware (2)

Hardware optimisations can be observed by concurrent code:

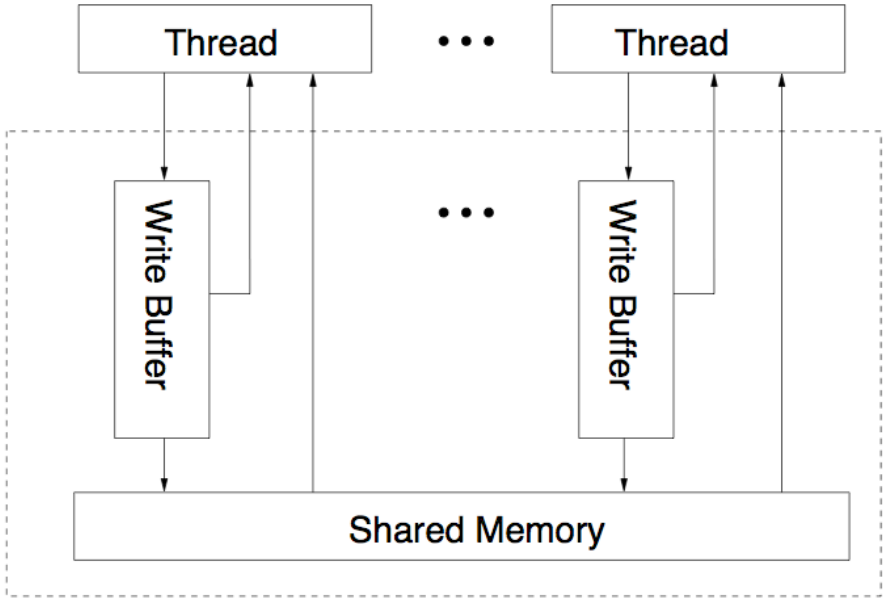
Thread 0	Thread 1
`x = 1`	`y = 1`
`print y`	`print x`

At the end of some executions:

0 0

is printed on the screen,

both on x86 and Power/ARM).



That pesky hardware (2)

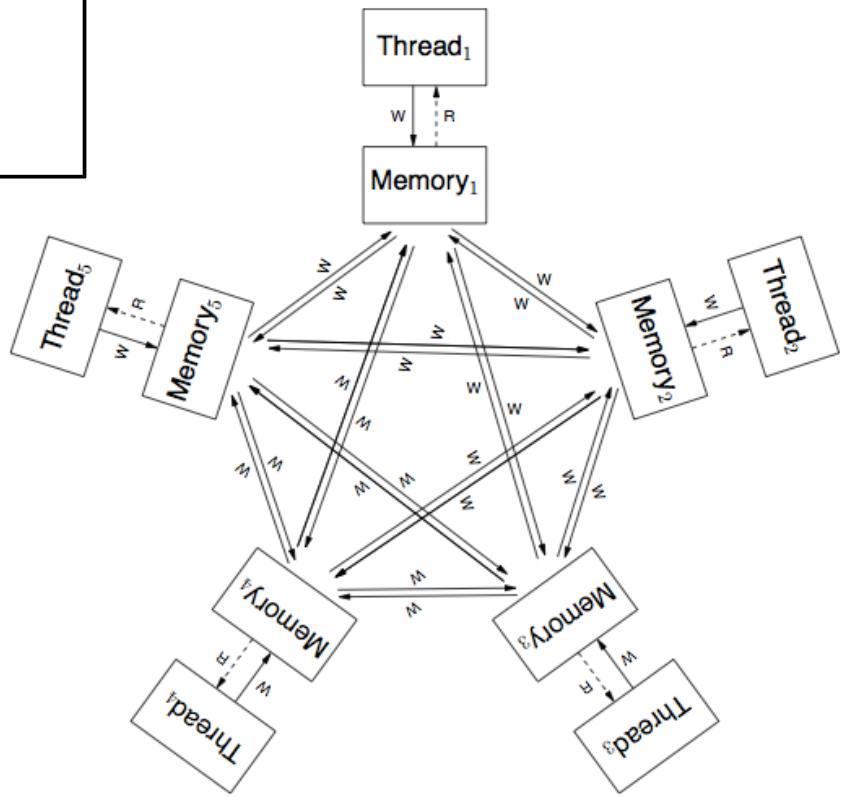
...and differ between architectures...

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

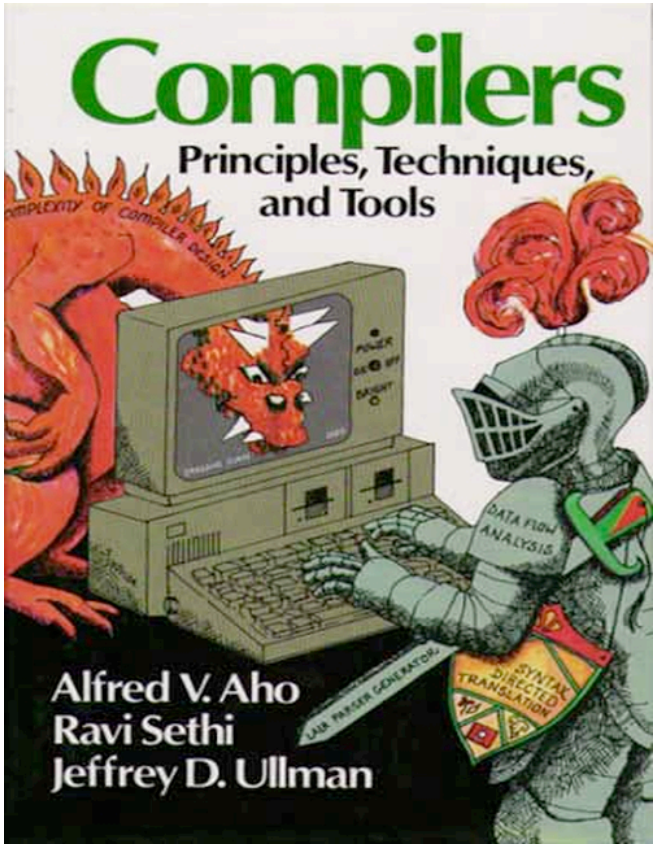
At the end of some executions:

1 0

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



Compilers vs. programmers



Compilers vs. programmers

Tension:

- the programmer wants to understand the code he writes
- the compiler and the hardware want to optimise it.

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?

This lecture

Programming language models

1) defining the semantics of a concurrent programming language

2) data-race freedom

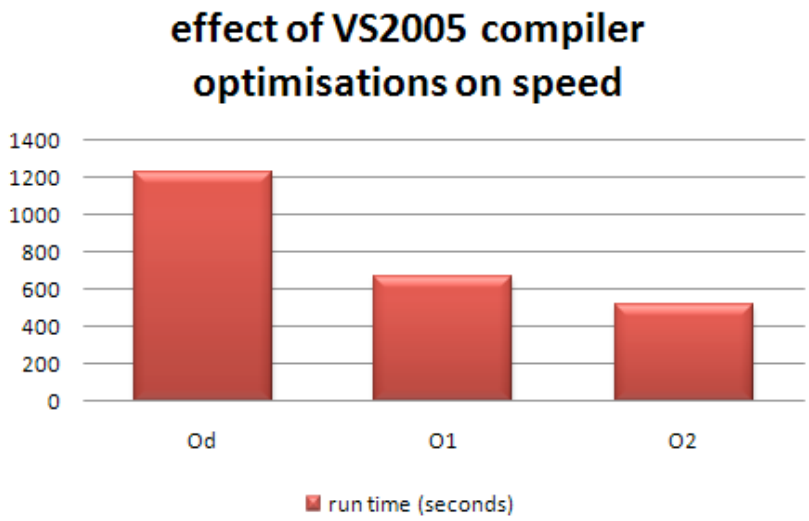
3) soundness of compiler optimisations

Previous lecture: hardware models

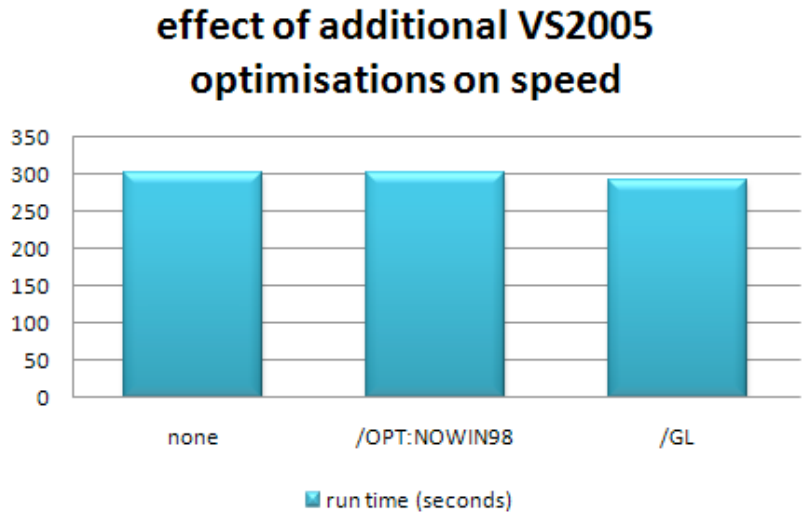
1) why are industrial specs so often flawed?

   focus on x86, with a glimpse of Power/ARM

2) usable models: x86-TSO, PowerARM



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 ;
}
```

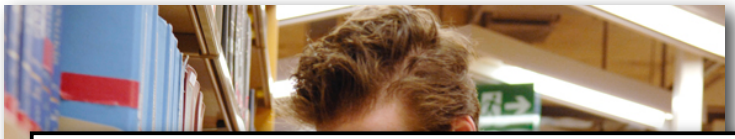
What is an optimisation?

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 ;
}
```

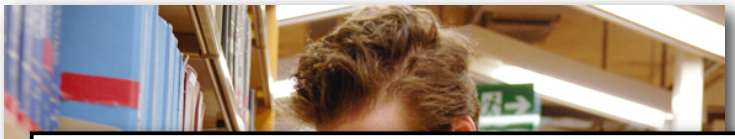
What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

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 ;
}
```

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

What is an optimisation?

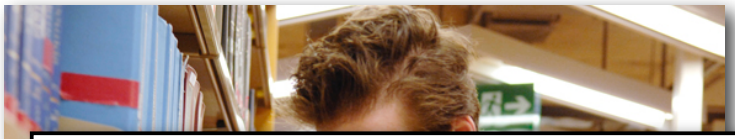
Compiler Writer

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, but rarely:

```
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

Back to our question now:

Which is the semantics of a concurrent program?

Note that the loop body is not executed.

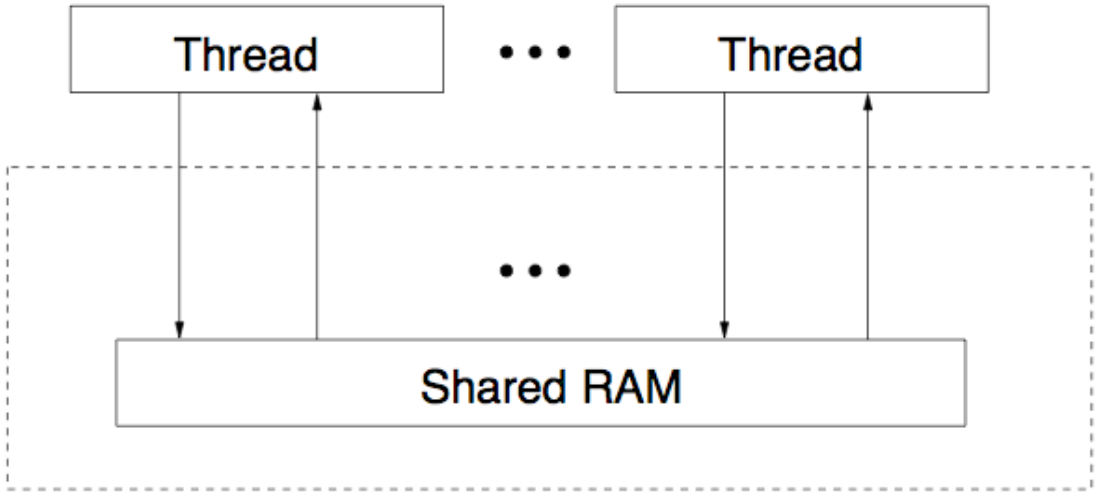
Naive answer: enforce sequential consistency

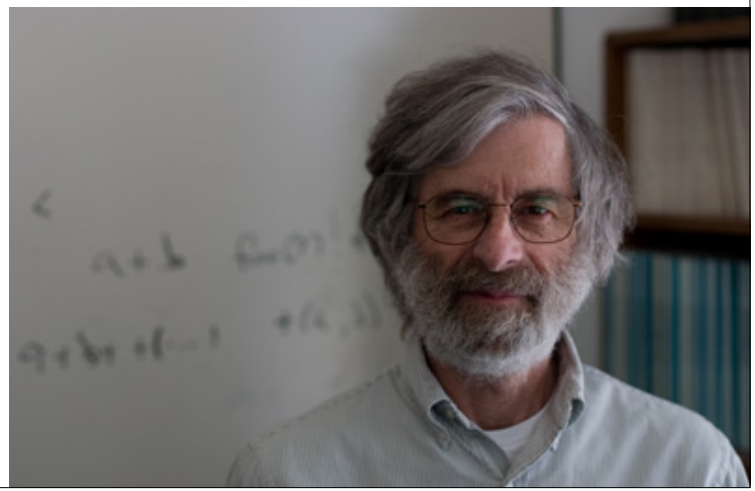
Sequential consistency

Multiprocessors have a ***sequentially consistent*** shared memory when:

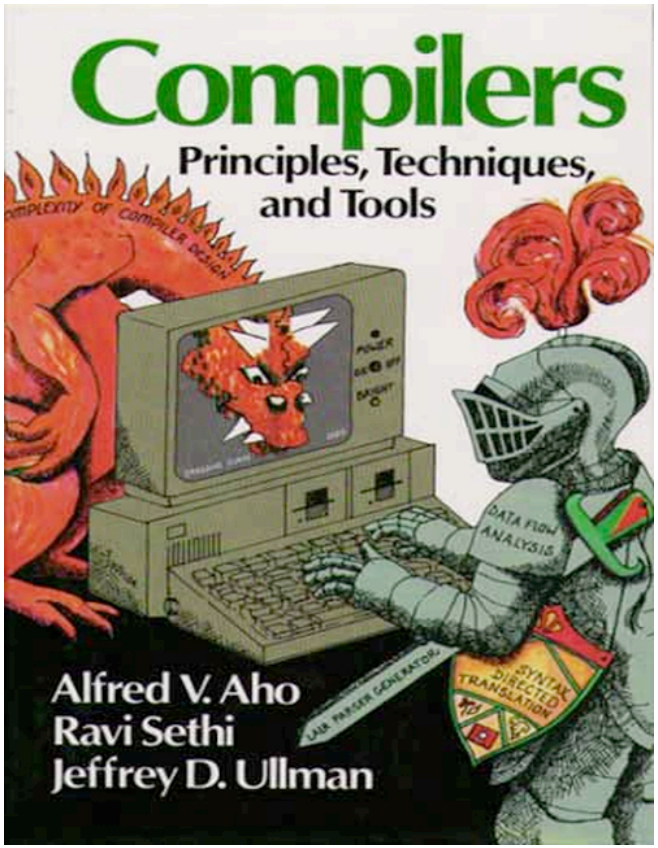
> *...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program...*

Lamport, 1979.

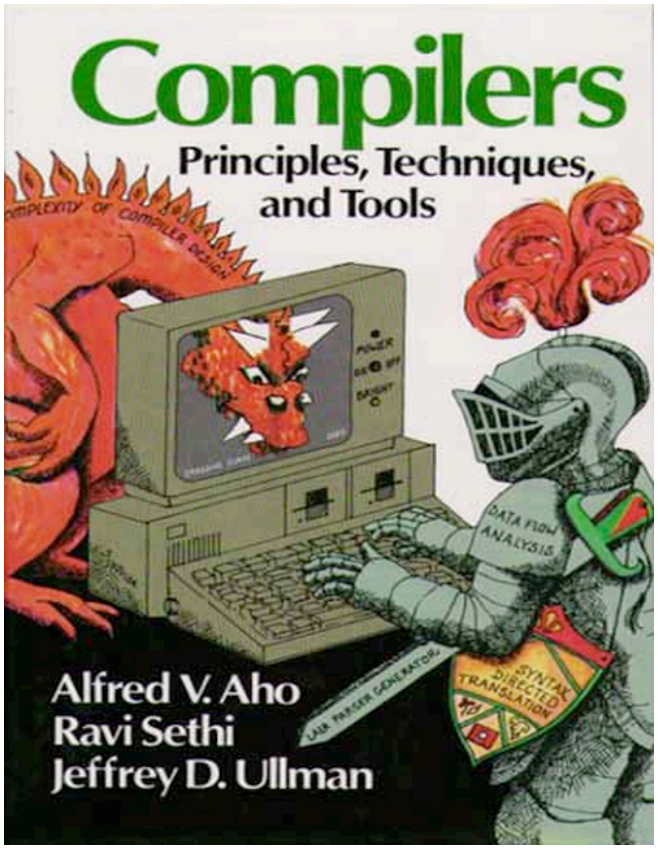




Compilers, programmers & sequential consistency

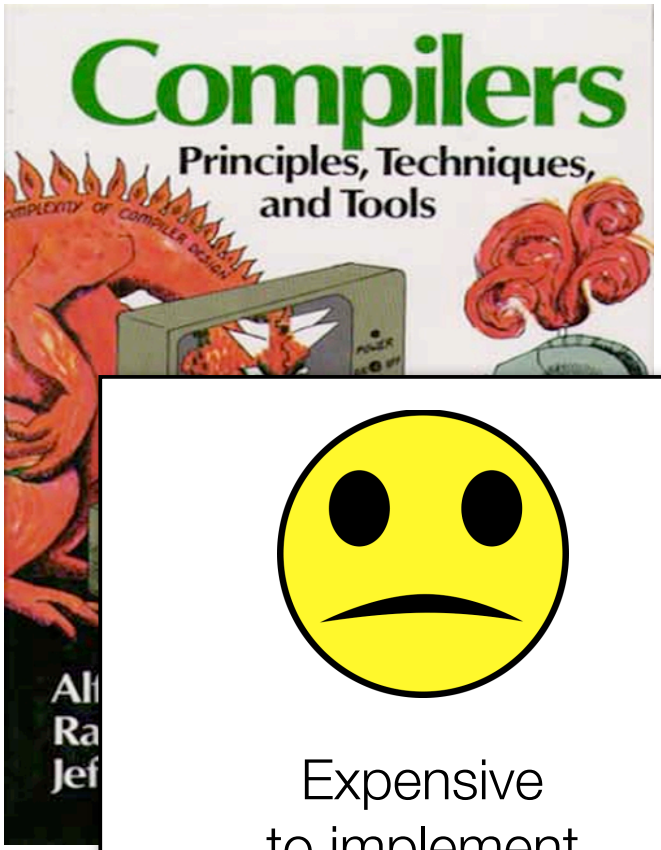


Compilers, programmers & sequential consistency



Simple and intuitive programming model

Compilers, programmers & sequential consistency



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.

Expensive to implement

And this study supposes that the hardware is SC.

SC and hardware

The compiler must insert enough synchronising instructions to prevent hardware reorderings. On x86 we have:

- MFENCE
  flush the local write buffer

- LOCK prefix (e.g. CMPXCHG)
  flush the local write buffer
  globally lock the memory

Initial: [x]=0 ∧ [y]=0	
proc 0	proc 1
MOV [x]←$1 MFENCE MOV EAX←[y]	MOV [y]←$1 MFENCE MOV EBX←[x]
Forbid: EAX=0 ∧ EBX=0	

Initally, [100] = 0

At the end, [100] = 2

proc:0	proc:1
LOCK; INC [100]	LOCK; INC [100]

These consumes hundreds of cycles… ideally should be avoided.

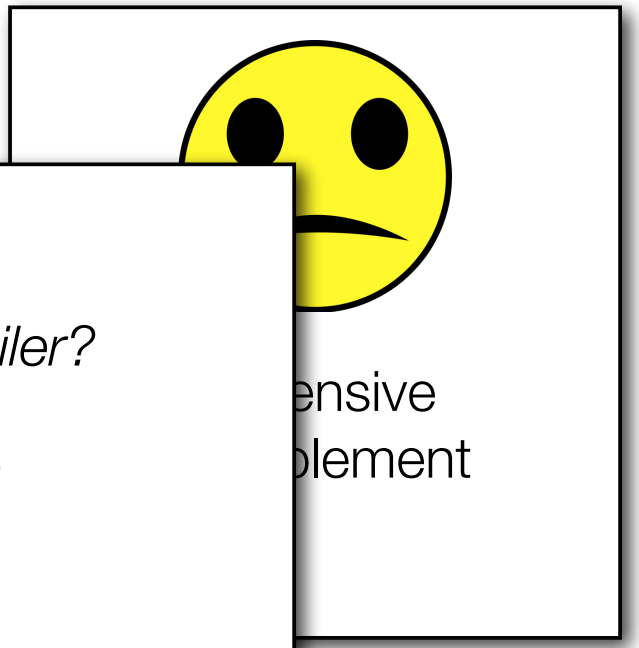
Naively recovering SC on x86 incurs in a ~40% overhead.

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 maximu
program
and SPE



ensive
plement

What is an SC-preserving compiler?

When is a compiler correct?

And this st

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.

Example

$P_1 =$ `*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)?

Example

$P_1 =$ `*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
```

Example

$P_1 =$ `*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:

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

Example

$P_1 =$ `*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:

$W_{t_1}\ x{=}1, R_{t_2}\ x{=}1, R_{t_2}\ x{=}1, P_{t_2}\ 1$
$R_{t_2}\ x{=}0, W_{t_1}\ x{=}1, R_{t_2}\ x{=}1, P_{t_2}\ 2$
$R_{t_2}\ x{=}0, R_{t_2}\ x{=}0, W_{t_1}\ x{=}1, P_{t_2}\ 1$
$R_{t_2}\ x{=}0, R_{t_2}\ x{=}0, P_{t_2}\ 1, W_{t_1}\ x{=}1$

Executions of P2:

$W_{t_1}\ x{=}1, R_{t_2}\ x{=}1, P_{t_2}\ 1$
$R_{t_2}\ x{=}0, W_{t_1}\ x{=}1, P_{t_2}\ 1$
$R_{t_2}\ x{=}0, P_{t_2}\ 1, W_{t_1}\ x{=}1$

Example

$P_1 =$ `*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:

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

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

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

Example

$P_1 =$ `*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:

W_{t_1}

R_{t_2}

R_{t_2}

R_{t_2}

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

Behaviours of P1: $[P_{t_2}\ 1], [P_{t_2}\ 2]$

Behaviours of P2: $[P_{t_2}\ 1]$

General CSE incorrect in SC

```
*x = 1;                 |  if *x=1 then (
*y = 1;                 |    *x = 2;
if *y = 2               |    *y = 2
then print *x           |  )
```

There is only one execution with a printing behaviour:

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

General CSE incorrect in SC

```
*x = 1;
*y = 1;
if *y = 2
then print *x
```

```
if *x=1 then (
  *x = 2;
  *y = 2
)
```

But a compiler would optimise to:

```
*x = 1;
*y = 1;
if *y = 2
then print 1
```

```
if *x=1 then (
  *x = 2;
  *y = 2
)
```

General CSE incorrect in SC

```
*x = 1;          |  if *x=1 then (
*y = 1;          |    *x = 2;
if *y = 2        |    *y = 2
then print 1     |  )
```

The only execution with a printing behaviour in the optimised code is:

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

So the optimisation is not correct.

General CSE incorrect in SC

```
*x = 1;  |  r = *x;
*y = 1;  |  print r;
```

Our first example highlighted that CSE is incorrect in SC.

Here is another example.

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

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

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

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

General CSE incorrect in SC

```
*x = 1;  |  r = *x;
*y = 1;  |  print r;
         |  print *y;
         |  print *x;
```

The observable behaviours are (note that 0 - 1 - 0 is not observable):

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

General CSE incorrect in SC

`*x = 1;`	`r = *x;`
`*y = 1;`	`print r;`
	`print *y;`
	`print *x;`

But a compiler would optimise as:

`*x = 1;`	`r = *x;`
`*y = 1;`	`print r;`
	`print *y;`
	`print r;`

General CSE incorrect in SC

```
*x = 1;  |  r = *x;
*y = 1;  |  print r;
         |  print *y;
         |  print *x;
```

```
*x = 1;  |  r = *x;
*y = 1;  |  print r;
         |  print *y;
         |  print r;
```

Let's compare the behaviours of the two programs:

$[\mathrm{P}_{t_2}\,1, \mathrm{P}_{t_2}\,1, \mathrm{P}_{t_2}\,1]$

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

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

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

$[\mathrm{P}_{t_2}\,0, \mathrm{P}_{t_2}\,0, \mathrm{P}_{t_2}\,0]$

$[\mathrm{P}_{t_2}\,1, \mathrm{P}_{t_2}\,1, \mathrm{P}_{t_2}\,1]$

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

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

$[\mathrm{P}_{t_2}\,0, \mathrm{P}_{t_2}\,0, \mathrm{P}_{t_2}\,0]$

General CSE incorrect in SC

```
*x = 1;  |  r = *x;
*y
```

```
*x = 1;  |  r = *x;
```

The optimised program exhibits a new, unexpected, behaviour.

Let's compare the behaviours of the two programs:

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

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

Reordering incorrect

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

Again, the optimised 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

There are some correct optimisations 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.

(More on this later)

Elimination of adjacent accesses

There are some correct optimisations under SC. For example it is correct to rewrite:

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

Can we define a model that:

1) enables more optimisations than SC, and

2) retains the simplicity of SC?

(More on this later)

Alternative answer: data-race freedom

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

Our exam

	Thread 1
	`x == 1`
	`print *y`

- the prob
  (e.g. sw
  thread 0

viour: 0

programs;

- the prob

...intuition...

Programming languages provide

synchronisation mechanisms

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

de that allows two threads to **access the same data simultaneously in conflicting ways** (e.g. one thread writes the datas read by the other).

The basic solution

Prohibit *data races*

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

Observable behaviour: `0`

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:

- two mem… …emory location …
- a SC exe… …nflicting **operatio**… …t (maybe executed concurrently).

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

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? (focus on 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? (focus on 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? (focus on 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

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 w

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

Compiler/hardware can continue to reorder accesses

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

```
*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();
```

Another example of DRF program

Exercise: is this program DRF?

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

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

Ans

The
par

ot

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, then either my program has a bug OR it has a data-race

Validity of compiler optimisations, summary

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, summary

Transformation	SC	
Memory trace preserving transformations	✓	
Roach-motel reordering	×(✓for locks)	✓
External action reordering	×	✓

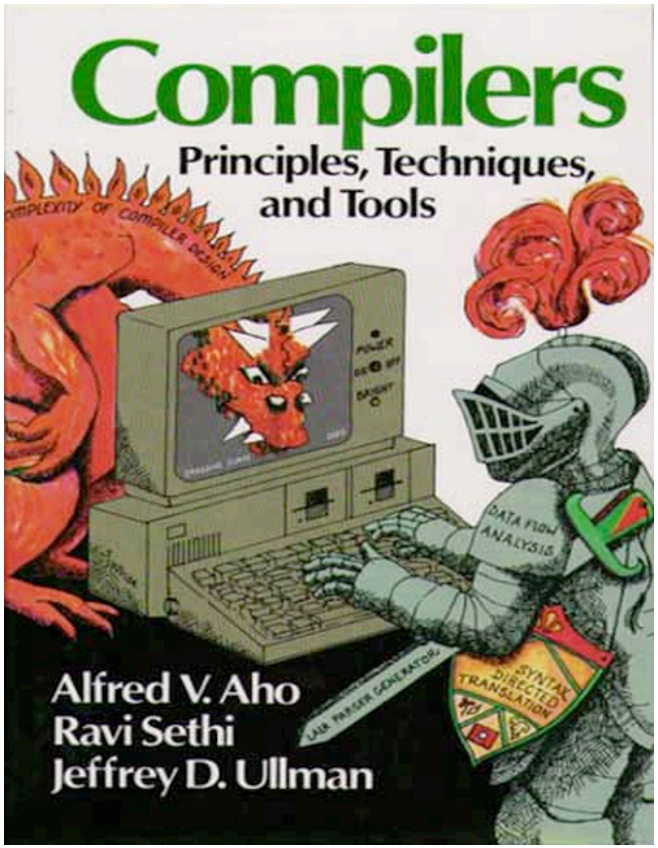
* Optimisations legal only on adjacent statements.

Jaroslav Sevcik

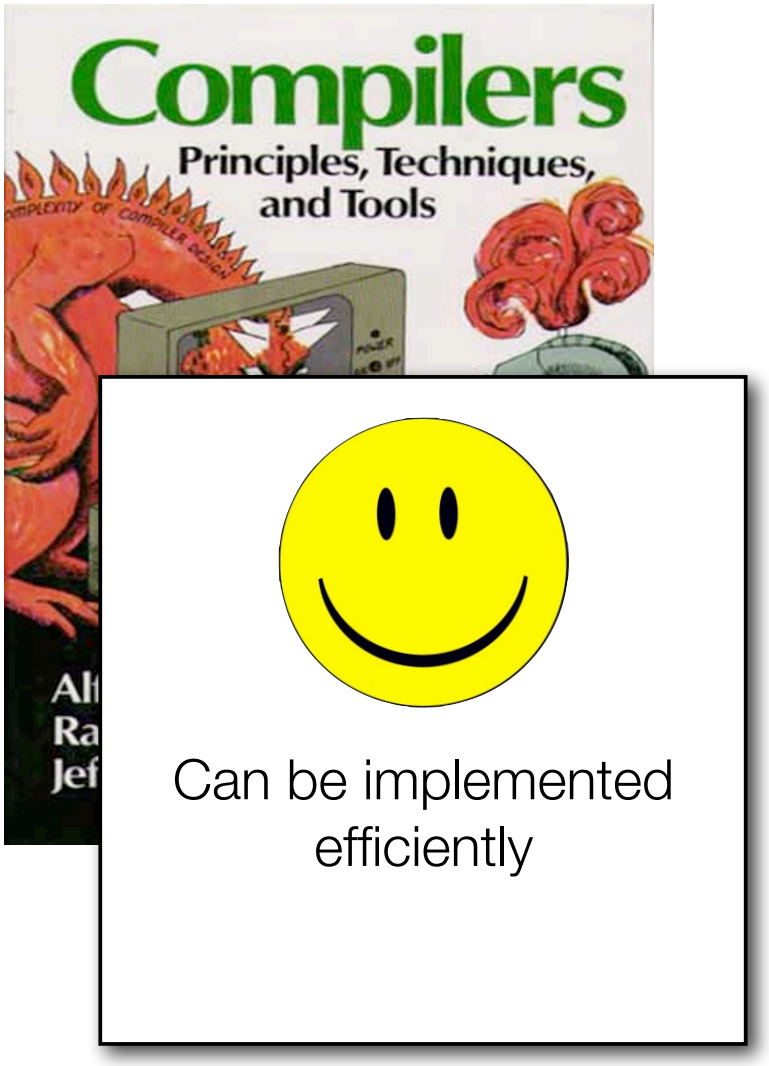
Safe Optimisations for Shared-Memory Concurrent Programs

PLDI 2011

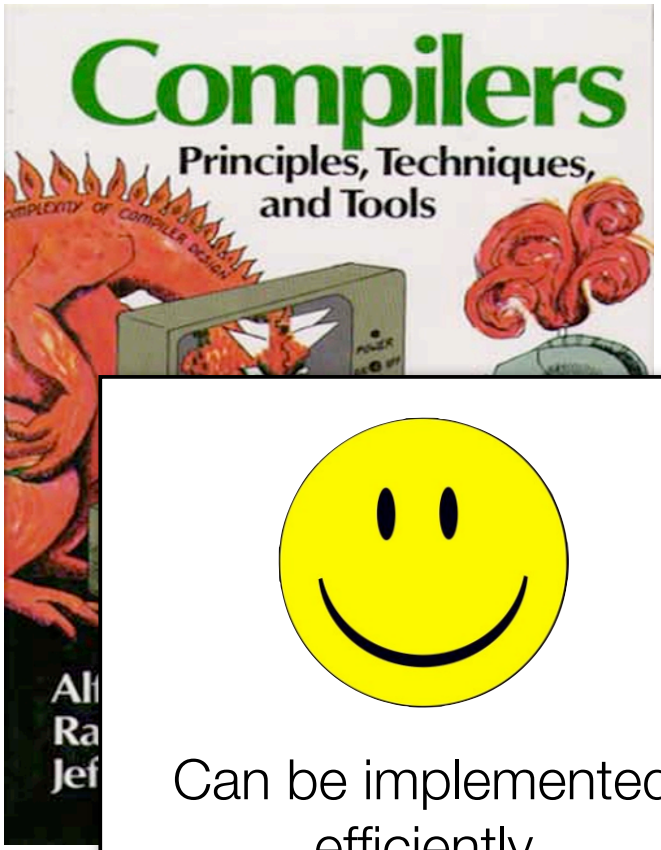
Compilers, programmers & data-race freedom



Compilers, programmers & data-race freedom

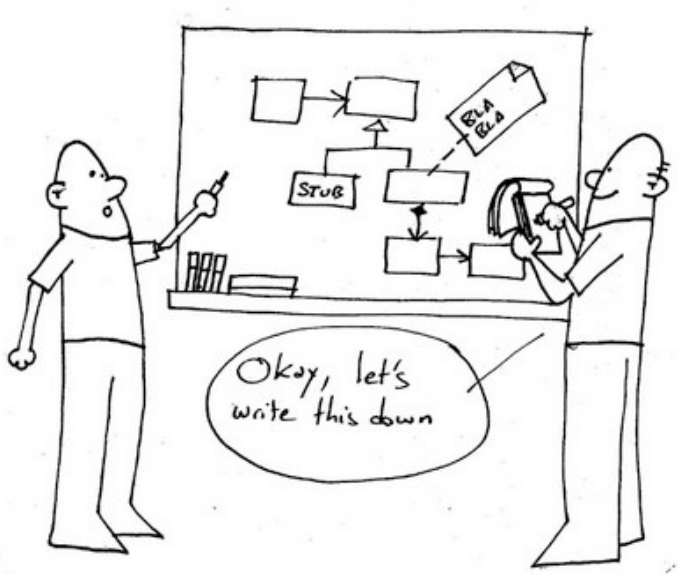


Compilers, programmers & data-race freedom



Can be implemented efficiently

Intuitive programming model (but detecting races is tricky!)



Data-race freedom, formalisation

A toy language: semantics

location, x	shared memory location
register, r	thread-local variable
integer, n	integers
thread_id, t	thread identifier

statement, s ::=		statements
| `r := x`		read from memory
| `x := r`		write to memory
| `r := n`		load constant into register
| `lock`		lock
| `unlock`		unlock
| `print r`		output

`program, p ::= s;…;s` a program is a sequence of statements

system	::=	concurrent system
| t_0:p_0 | … | t_n:p_n		parallel composition of n threads

A toy language: semantics

location, x	shared memory location
register, r	thread-local variable
int	
th	
st	
|	
|	
|	
| `lock`	lock
| `unlock`	unlock
| `print r`	output

We work with a toy language, but this approach scales to the full Java Memory Model or C11/C++11.

`program, p ::= s;…;s` a program is a sequence of statements

system ::= concurrent system

| $t_0:p_0$ | … | $t_n:p_n$	parallel composition of n threads

Traces and tracesets

Definition [trace]: a sequence of memory operations (read, write, thread start, I/O, synchronisation). Thread start is always the first action of thread. All actions in a trace belong to the same thread.

Definition [traceset]: a traceset is a prefix-closed set of traces.

Sample traceset:

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

$$\{[\mathrm{S}(0), \mathrm{R}[\mathtt{x}{=}v], \mathrm{W}[\mathtt{y}{=}v]] \mid v \in V\}$$
$$\cup \{[\mathrm{S}(1), \mathrm{R}[\mathtt{y}{=}v], \mathrm{W}[\mathtt{x}{=}1], \mathrm{X}(v)] \mid v \in V\}$$

Tr

De ad
sta
thr

De

Remarks:

1. Reads can read arbitrary values from memory.
2. Tracesets should not be confused with interleavings.
3. Tracesets do not enforce receptiveness or determinism:

$$\{[\mathrm{S}(0)], [\mathrm{S}(0), \mathrm{R}[\mathtt{x}{=}1]], [\mathrm{S}(0), \mathrm{W}[\mathtt{y}{=}1]]\}$$

is also a valid traceset for the example below.

Sample traceset:

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

$$\{[\mathrm{S}(0), \mathrm{R}[\mathtt{x}{=}v], \mathrm{W}[\mathtt{y}{=}v]] \mid v \in V\}$$
$$\cup \{[\mathrm{S}(1), \mathrm{R}[\mathtt{y}{=}v], \mathrm{W}[\mathtt{x}{=}1], \mathrm{X}(v)] \mid v \in V\}$$

Associate tracesets to toy language programs

$$\langle S,\ \texttt{r := x; s} \rangle \xrightarrow{R[x=v]} \langle S[r=v],\ \texttt{s} \rangle$$

$$\langle S,\ \texttt{x := r; s} \rangle \xrightarrow{W[x=S(r)]} \langle S,\ \texttt{s} \rangle$$

$$\langle S,\ \texttt{r := n; s} \rangle \xrightarrow{\tau} \langle S[r=n],\ \texttt{s} \rangle$$

$$\langle S,\ \texttt{lock; s} \rangle \xrightarrow{L} \langle S,\ \texttt{s} \rangle$$

$$\langle S,\ \texttt{unlock; s} \rangle \xrightarrow{U} \langle S,\ \texttt{s} \rangle$$

$$\langle S,\ \texttt{print r; s} \rangle \xrightarrow{X(S(r))} \langle S,\ \texttt{s} \rangle$$

$$\langle S,\ t_0{:}p_0 \mid \ldots \mid t_n{:}p_n \rangle \xrightarrow{S(i)} \langle S,\ p_i \rangle$$

Tracesets and interleavings

Definition [interleaving]: an interleaving is a sequence of thread-identifier-action pairs.

Example: `y:=1;` ‖ `r2:=v;print r2;`

$$I' = [\langle 0, \mathrm{S}(0)\rangle, \langle 1, \mathrm{S}(1)\rangle, \langle 0, \mathrm{W}[\mathtt{y}{=}1]\rangle, \langle 1, \mathrm{R}[\mathtt{v}{=}0]\rangle, \langle 1, \mathrm{X}(0)\rangle]$$

Given an interleaving *I*, the trace of *tid* in *I* is the sequence of actions of thread *tid* in *I*, e.g.:

trace *1 I'* = [S(1), R[v=0], X(0)].

Conversely, given a traceset, we can compute all the well-formed interleavings (called *executions*)... (next slide)

Tracesets and interleavings

An interleaving *I* is an *execution* of a traceset *T* if:

- for all *tid*, trace *tid I* $\in$ *T* (traces belong to the traceset)

- *tid*s correspond to entry points S(*tid*)

- lock / unlock alternates correctly

- each read sees the most recent write to the same location (read/from).

(The last property enforce the sequentially consistent semantics for memory accesses).

Tracesets and interleavings

An interleaving *I* is an *execution* of a traceset *T* if:

- for
- *tid*
- loc
- eac

(The

Remarks:

1. Interleavings order totally the actions, and do not keep track of which actions happen in parallel.

2. It is however possible to put more structure on interleavings, and recover informations about concurrency.

Happens-before

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

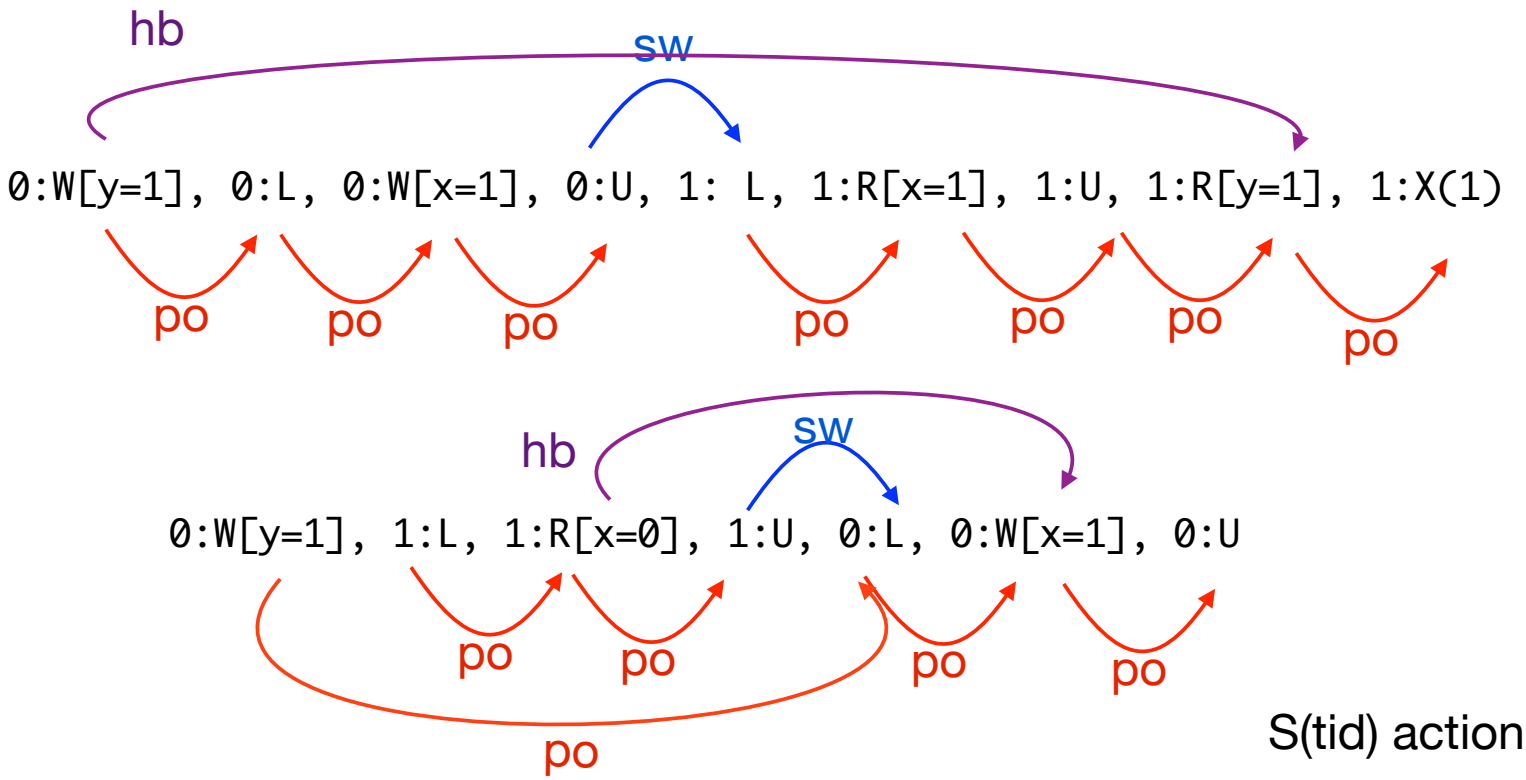
Definition [synchronises with]: in an interleaving *I*, index *i* **synchronises-with** index j, i $<_{sw}$ j, if i < j and A(l_i) = U (unlock), A(l_j) = L (lock).

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

Examples of happens before

Thread 0	Thread 1

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



S(tid) actions omitted.

Data-race freedom

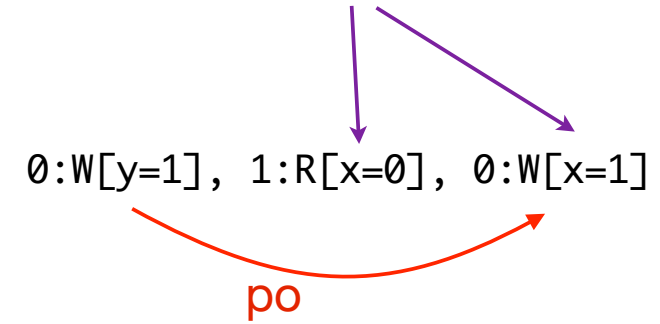
Definition [data-race-freedom]: A traceset is **data-race free** if none of its executions has two adjacent conflicting actions from different threads.

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

A racy program

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

Two conflicting accesses
not related by happens before.



Data-race freedom: equivalence of definitions

Given an execution

$$\alpha \mathbin{+\!\!+} [a] \mathbin{+\!\!+} \beta \mathbin{+\!\!+} [b]$$

of a traceset *T* where [a] and [b] are the first conflicting actions not related by happen-before, we build the interleaving

$$\alpha \mathbin{+\!\!+} \beta' \mathbin{+\!\!+} [a] \mathbin{+\!\!+} [b]$$

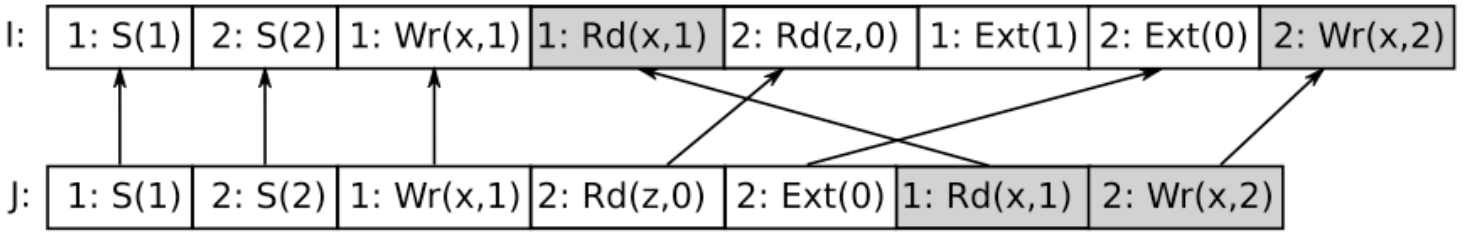
where β' are all the actions from β that strictly happen-before [b].

It remains to show that $\alpha \mathbin{+\!\!+} \beta' \mathbin{+\!\!+} [a] \mathbin{+\!\!+} [b]$ is an execution of *T*.

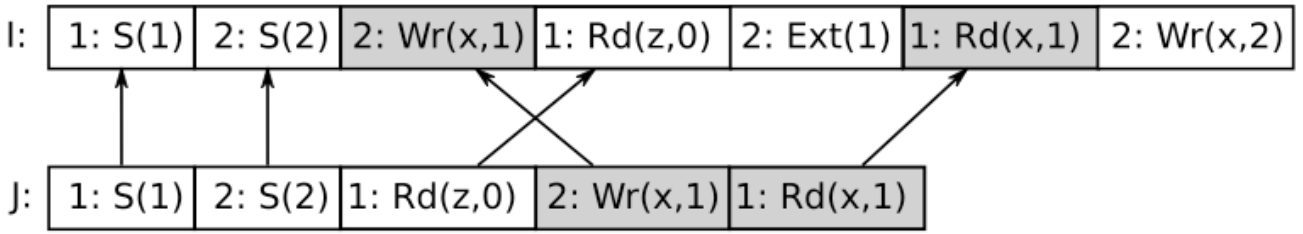
The formal proof is tedious and not easy (see Boyland 2008, Bohem & Adve 2008, Sevcik), here will give the intuitions of the construction on an example.

Data-race freedom: equivalence of definitions

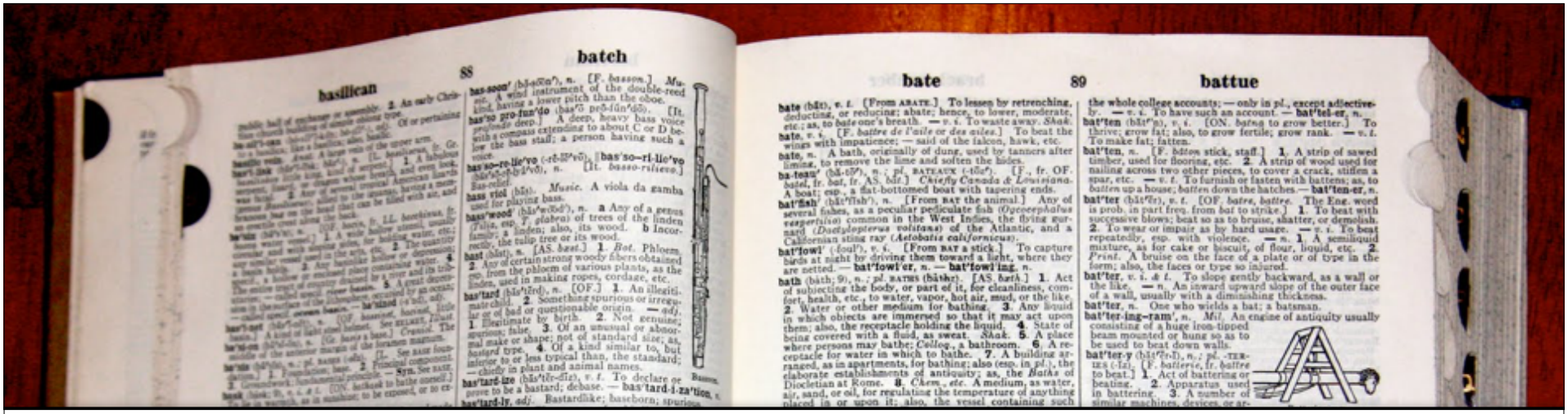
```
Thread 1: x := 1; r1 := x; print r1;
Thread 2: r2 := z; print r2; x := 2;
```



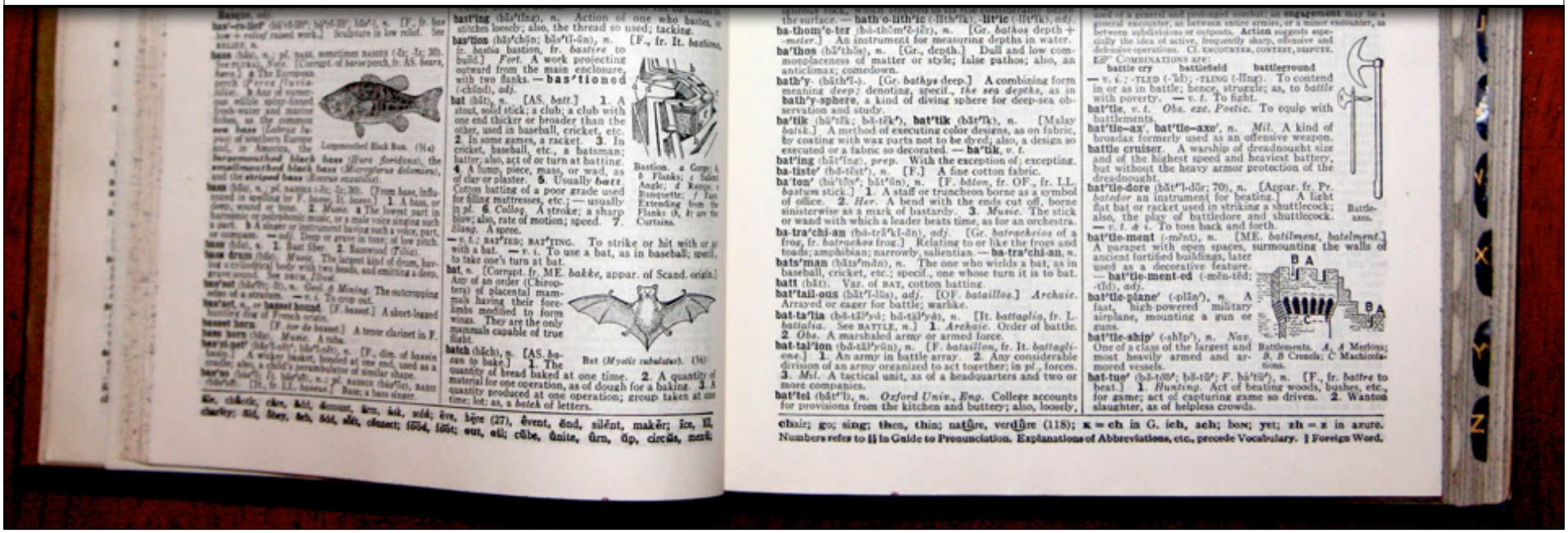
read first



write first



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

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

Data race freedom as a definition

- Core of the C11/C++11 standard.

Hans Boehm & Sarita Adve, PLDI 2008.

- Part of the JSR-133 standard.

Jeremy Manson & Bill Pugh & Sarita Adve, PLDI 2008.

Data race freedom as a definition

- **Core of the C11/C++11 standard.**

Hans Boehm & Sarita Adve, PLDI 2008.

with some escape mechanism called "low level atomics".

Mark Batty & al., POPL 2011.

- **Part of the JSR-133 standard.**

Jeremy Manson & Bill Pugh & Sarita Adve, PLDI 2008.

DRF gives no guarantees for untrusted code: a disaster for Java, which relies on unforgeable pointers for its security guarantees.

JSR-133 is **DRF + some out-of-thin-air guarantees** for all code.

A word on JSR-133

Goal 1: data-race free programs are sequentially consistent;

Goal 2: all programs satisfy some memory safety requirements;

Goal 3: common compiler optimisations are sound.

Out-of-thin-air

Goal 2: all programs satisfy some memory safety requirements.

Programs should never read values that cannot be written by the program:

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

the only possible result should be printing two zeros because no other value appears in or can be created by the program.

Out-of-thin-air

Goal 2: all programs satisfy some memory safety requirements.

Programs should never read values that cannot be written by the program:

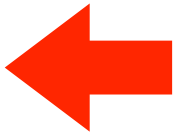
initially `x = y = 0`	
`r1 := x`	`r2 := y`
`y := r1`	`x := r2`
`print r1`	`print r2`

the only possible result should be printing two zeros because no other value appears in or can be created by the program.

Out-of-thin-air

Under DRF, it is correct to speculate on values of writes:

```
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

The transformed program can now print 42. This will be theoretically possible in C++11, but not in Java.

The program above looks benign, why does Java care so much about out-of-thin-air?

Out-of-thin-air

Out-of-thin-air is not so benign for references. Compare:

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

and

initially x = y = null	
r1 := x	r2 := y
y := r1	x := r2
	r2.run()

What should `r2.run()` call?

If we allow out-of-thin-air, then it could do anything!

A word on JSR-133

Goal 1: data-race free programs are sequentially consistent;

Goal 2: all programs satisfy some memory safety requirements;

Goal 3: common compiler optimisations are sound.

The model is intricate, and fails to meet goal 3.

An example: should the source program print 1? can the optimised program print 1?

x = y = 0	
r1 = x y = r1	r2 = y x=(r2==1)?y:1 print r2

HotSpot Optimization

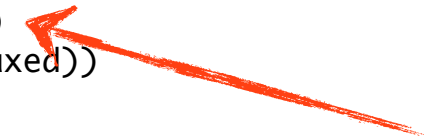
x = y = 0	
r1 = x y = r1	x = 1 r2 = y print r2

Jaroslav Ševčík, David Aspinall, ECOOP 2008

A word on C11/C++11 low-level atomics

```
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

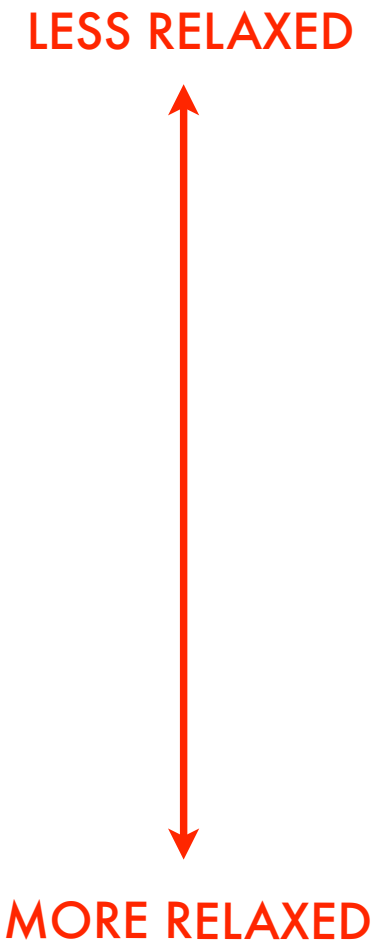


New syntax for memory accesses

Qualifier

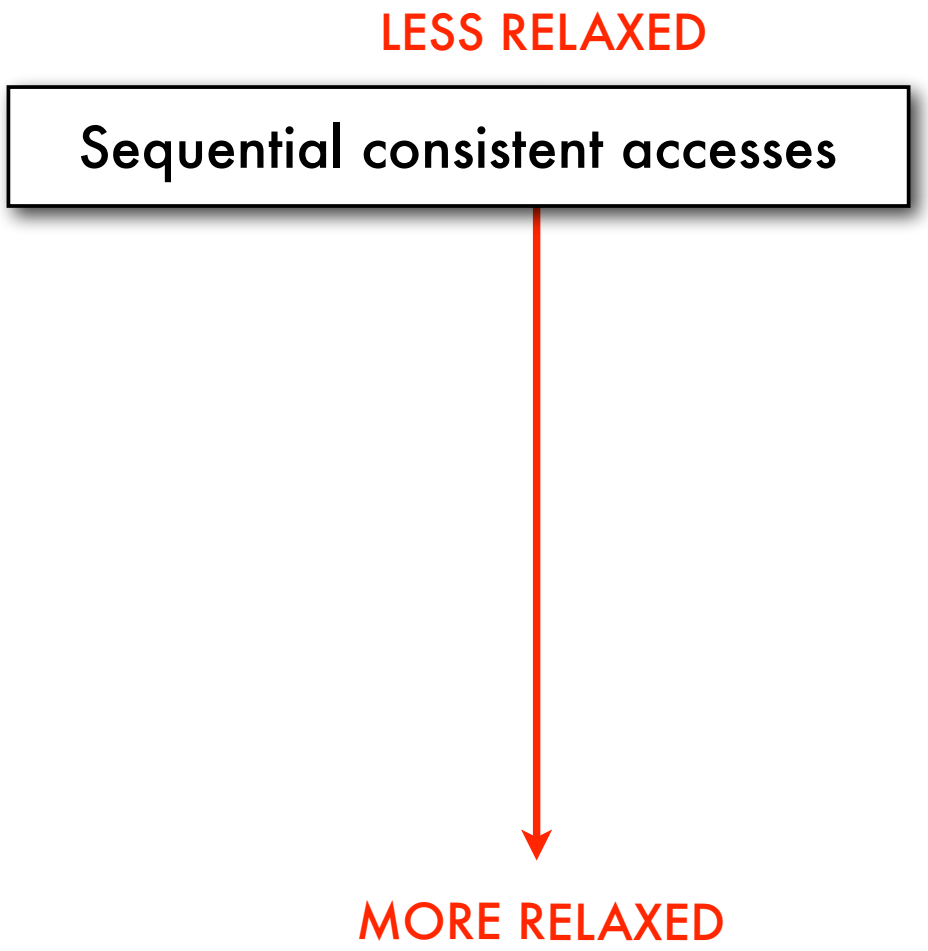
Low level atomics

```
MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED
```

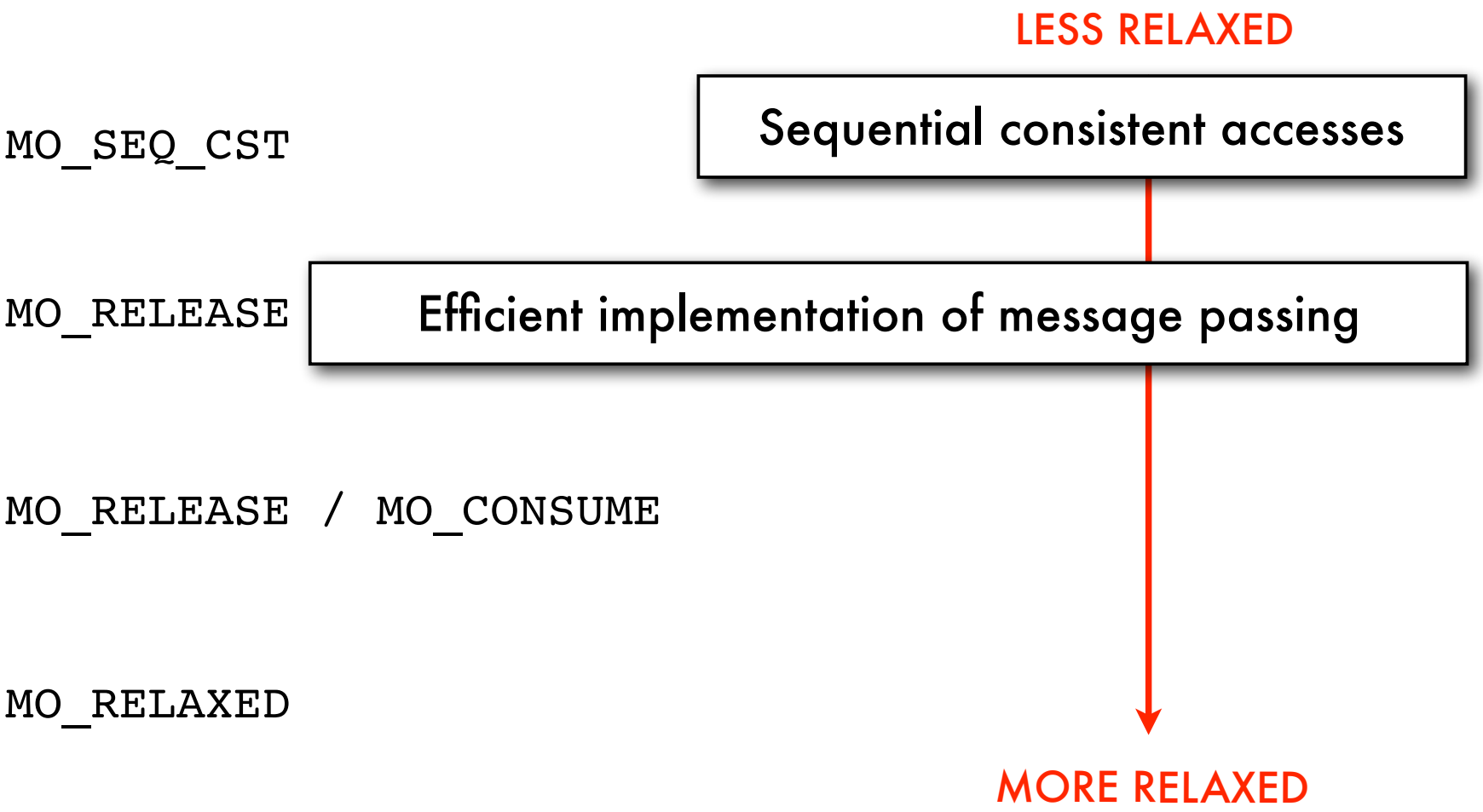


Low level atomics

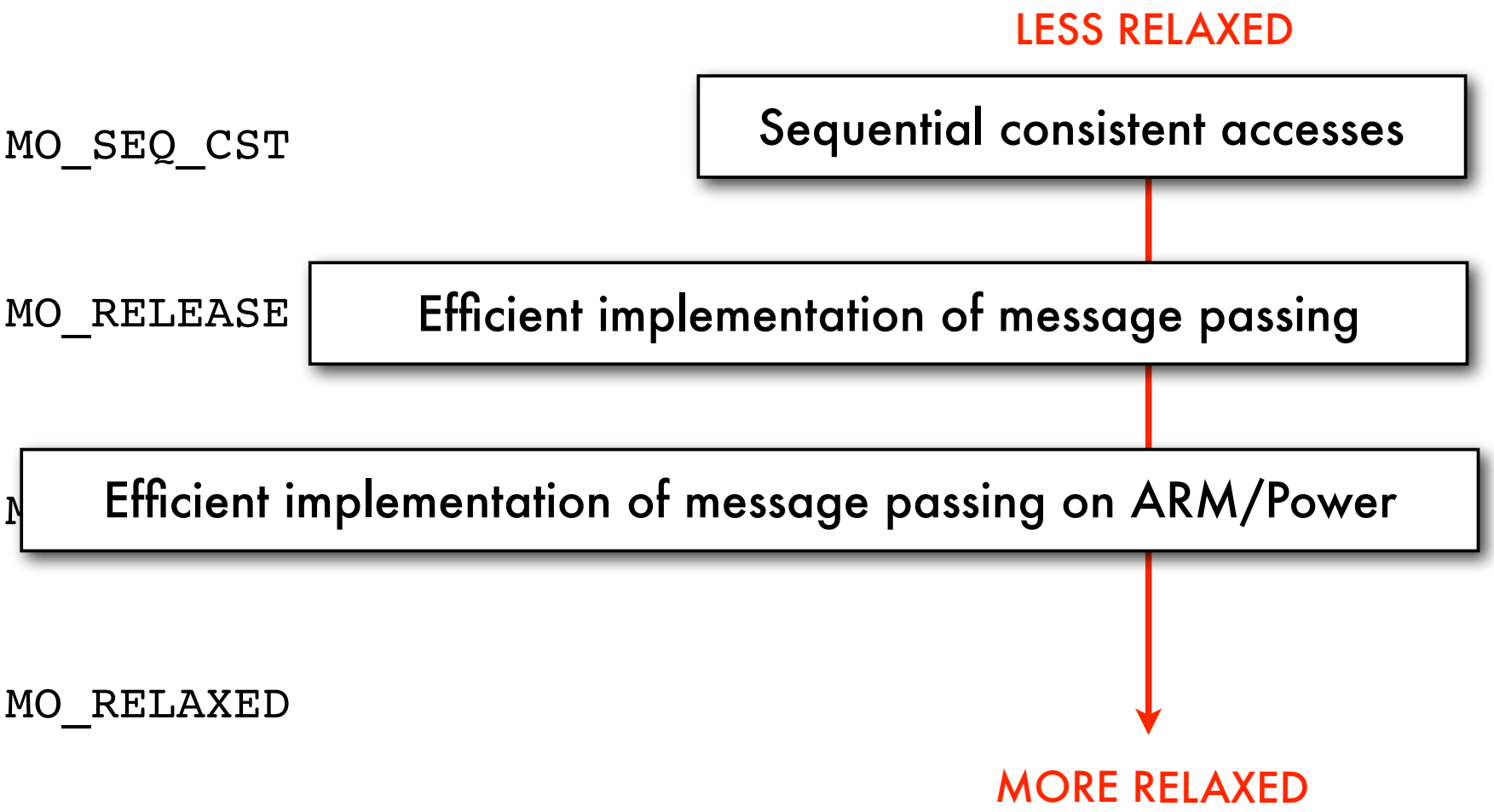
```
MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED
```



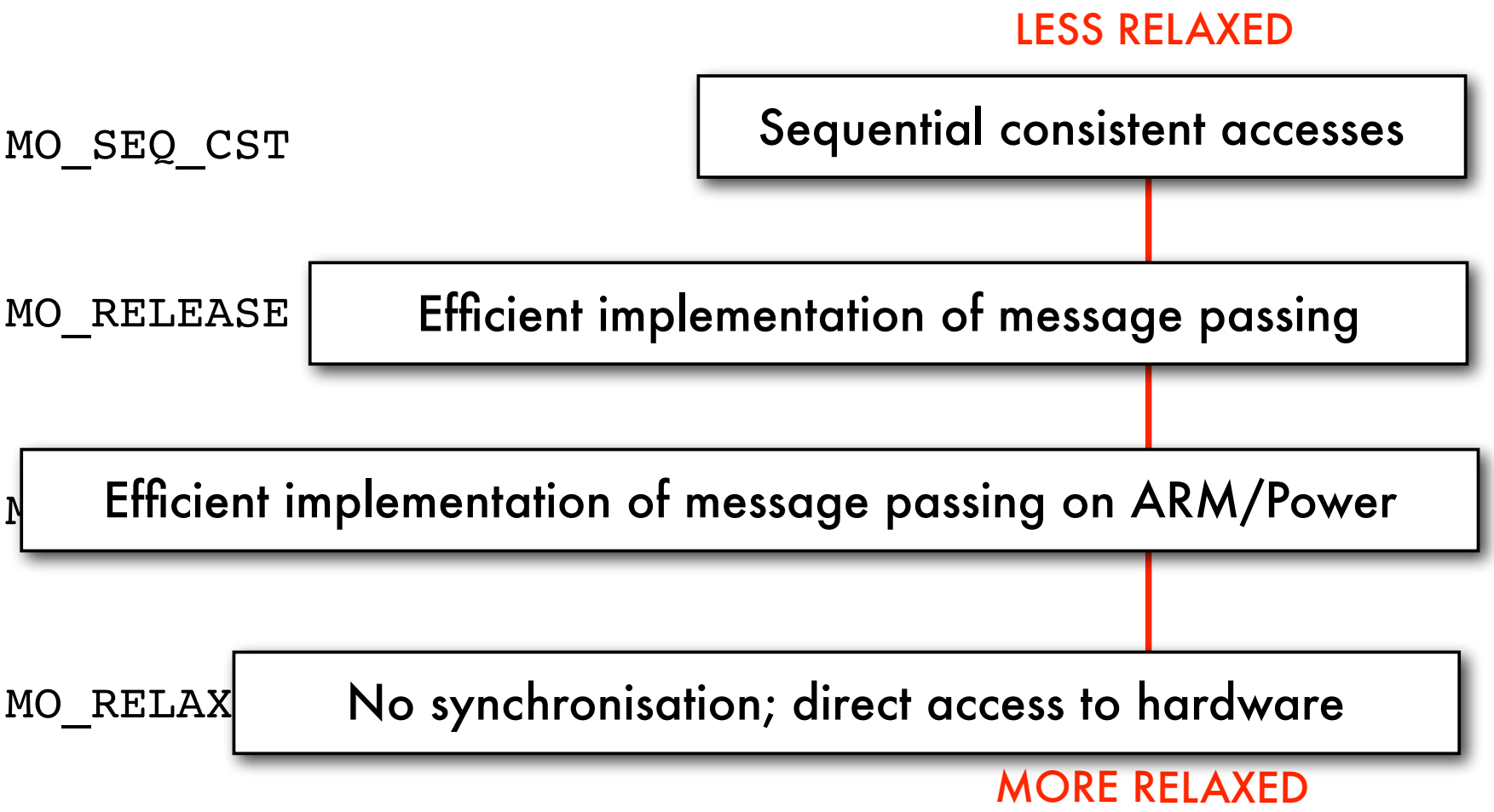
Low level atomics



Low level atomics



Low level atomics

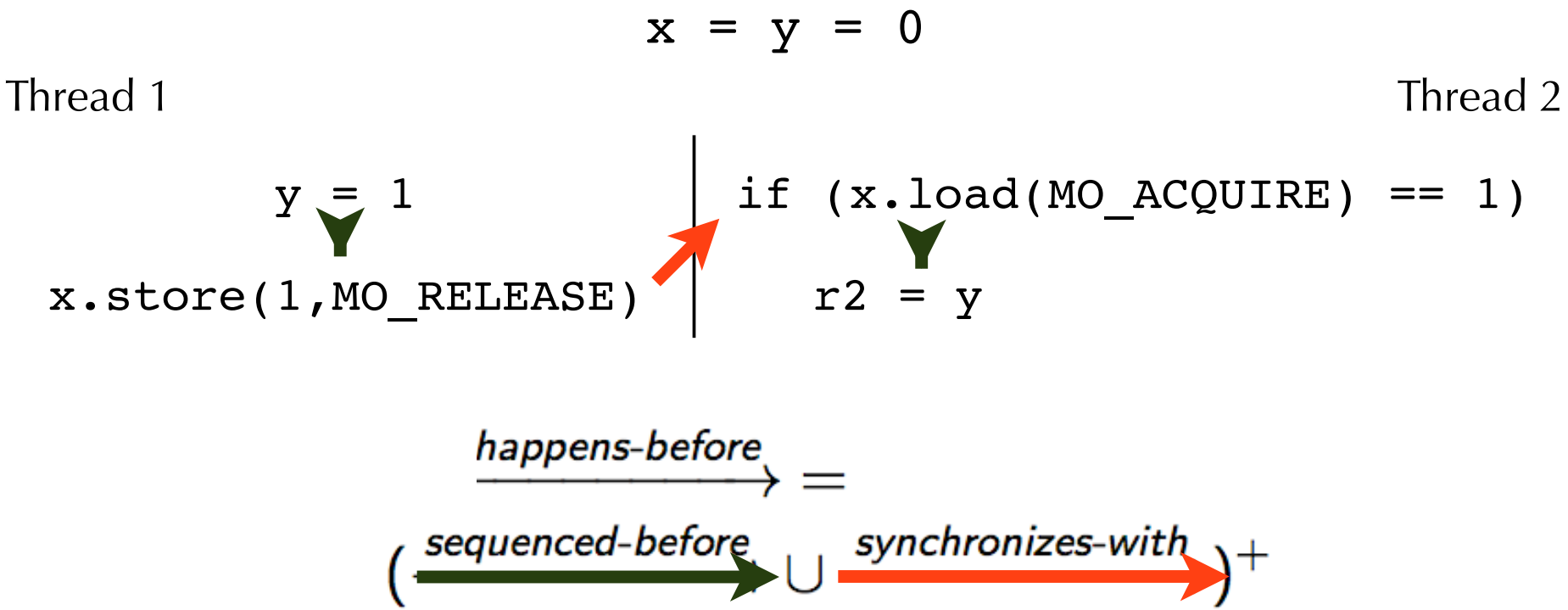


Memory access synchronisation

```
x = y = 0
```

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

Memory access synchronisation



Non-atomic loads must return the *most recent write* in the happens-before order (unique in a DRF program)

Understanding MO_RELAXED

```
x = y = 0
```

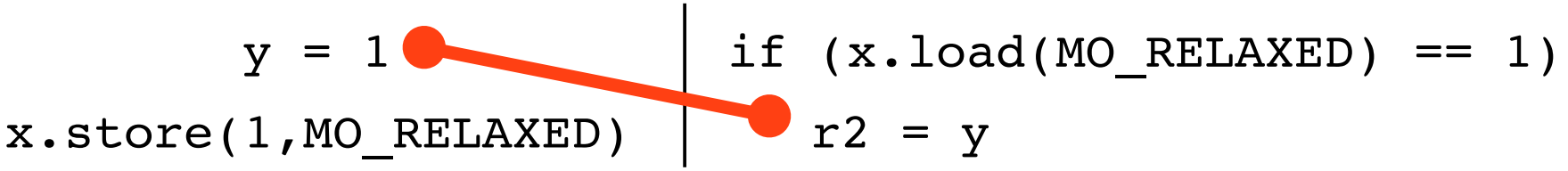
Thread 1 | Thread 2

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

Understanding MO_RELAXED

```
x = y = 0
```

Thread 1 | Thread 2



DATA RACE

Two conflicting accesses not related by happens-before

Understanding MO_RELAXED

```
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)`

WELL DEFINED

but r2 = 0 is possible

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)`

WELL DEFINED

but r2 = 0 is possible

Intuition

the compiler (or hardware) can reorder independent accesses

```
x = y = 0
```

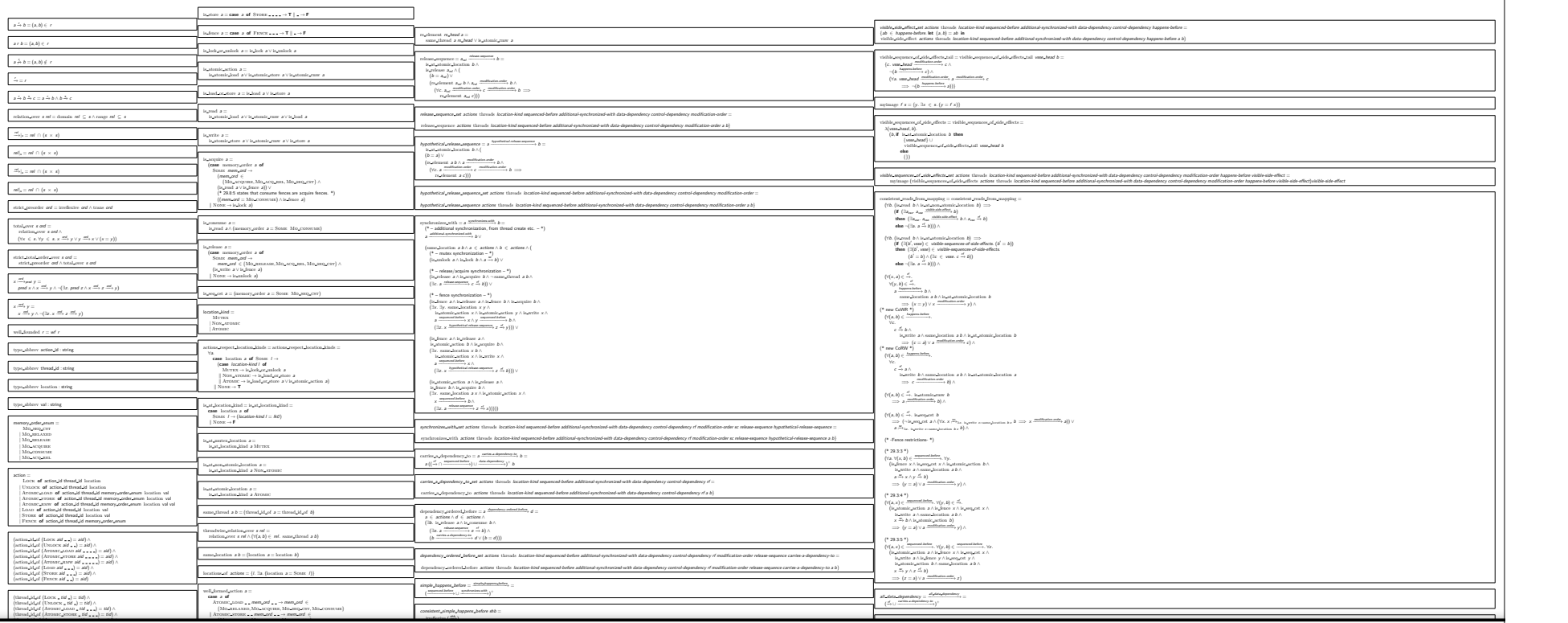
Thread 1	Thread 2
`y.store(1,MO_RELAXED)`	`if (x.load(MO_RELAXED) == 1)`
`x.store(1,MO_RELAXED)`	`  r2 = y.load(MO_RELAXED)`

Allow a RELAXED load to see any store that:

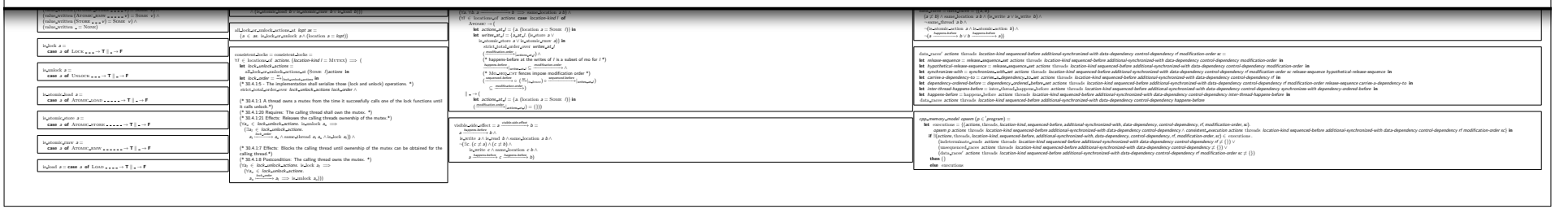
- does not happen-after it
- is not hidden by an intervening store hb-ordered between them

The full model

The full model



We can reason about C concurrency!



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 2 is not affected by Thread 1 and vice-versa

This program is data-race free

This program must print 42

Shared memory

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

This is a *compiler bug*

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

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

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

This program is data-race free

This program must print 42

Shared memory

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

This is a *concurrency compiler bug*

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

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

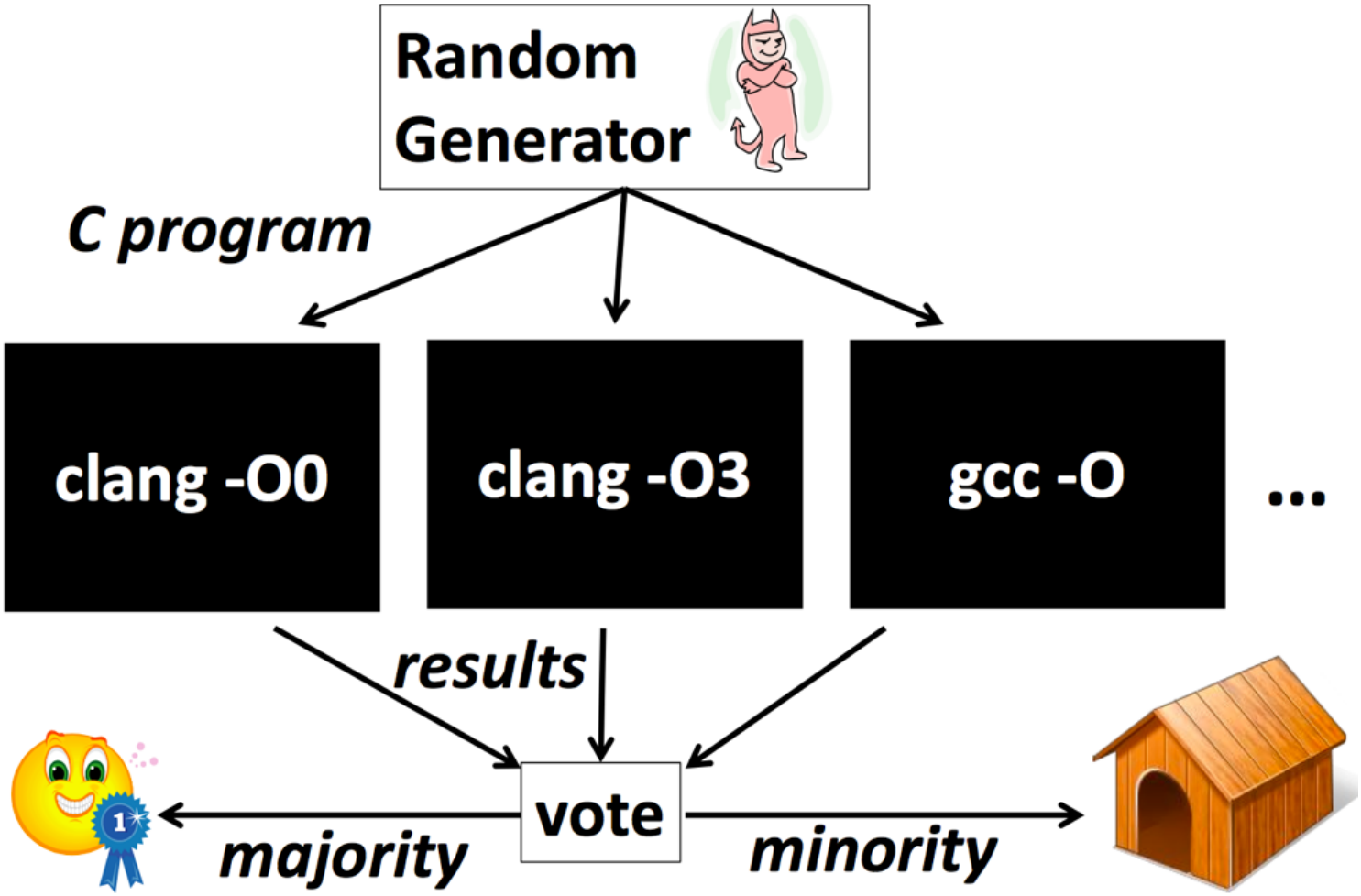
Thread 2 is not affected by Thread 1 and vice-versa

This program is data-race free

This program must print 42

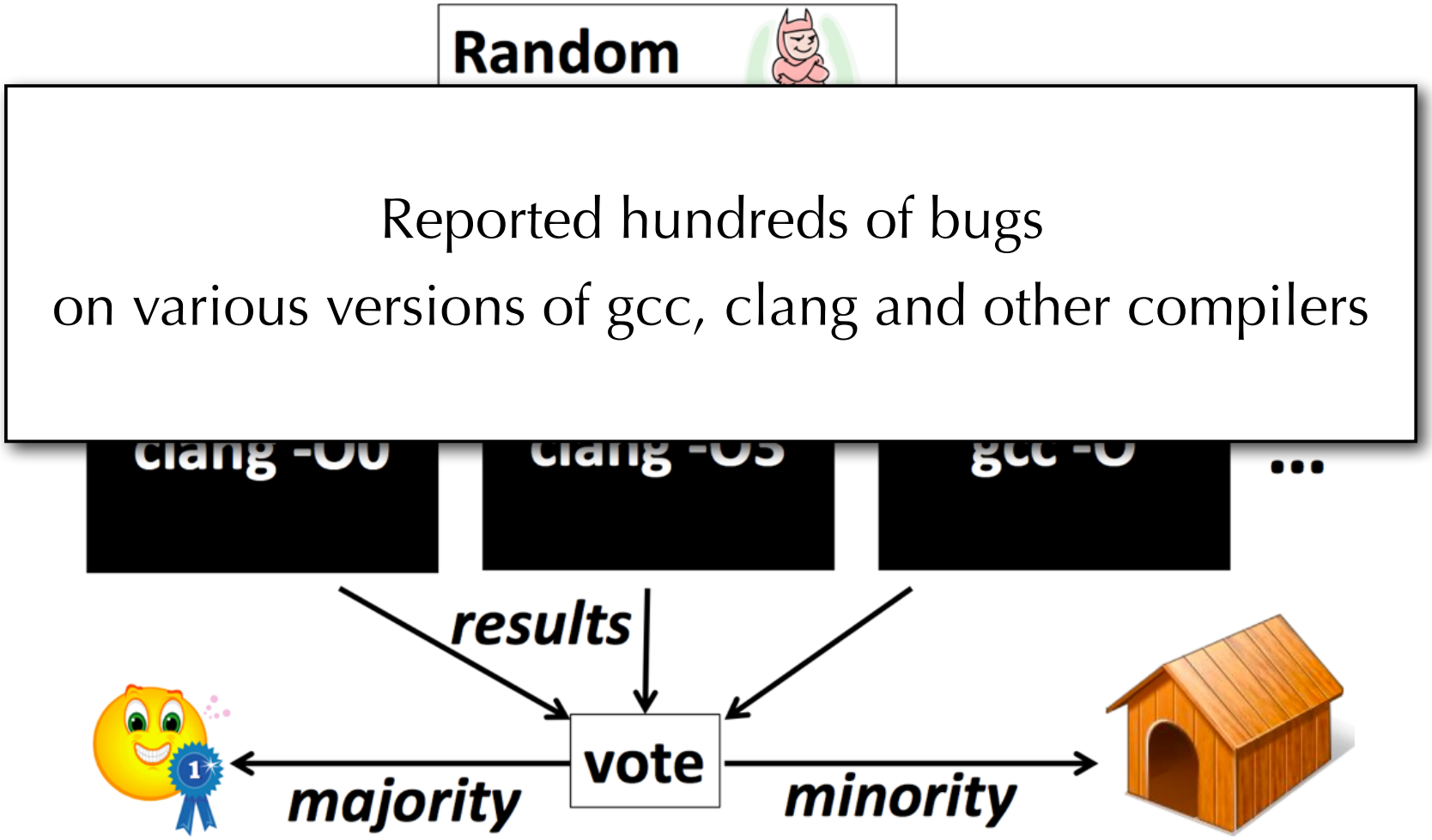
Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



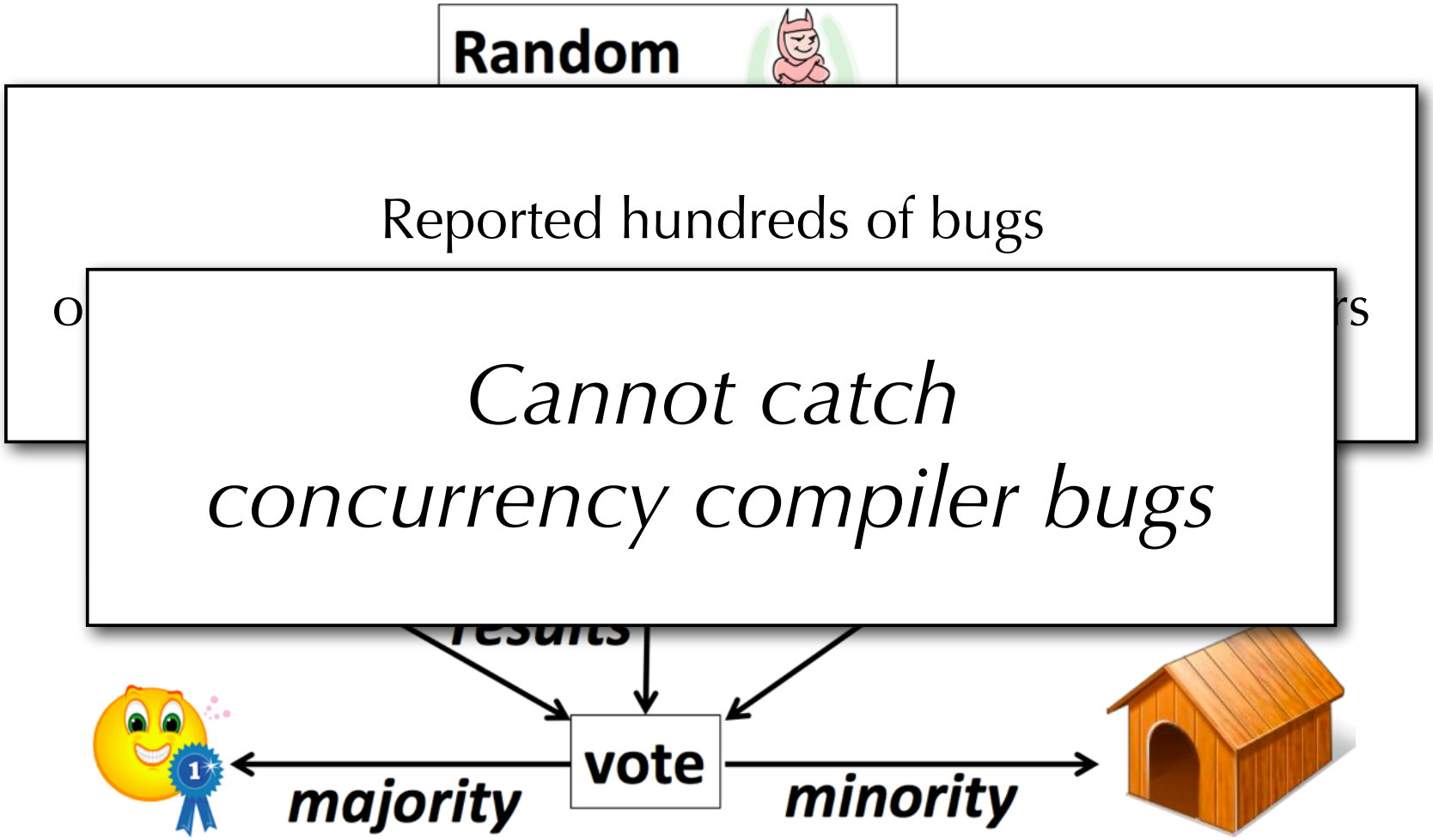
Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



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

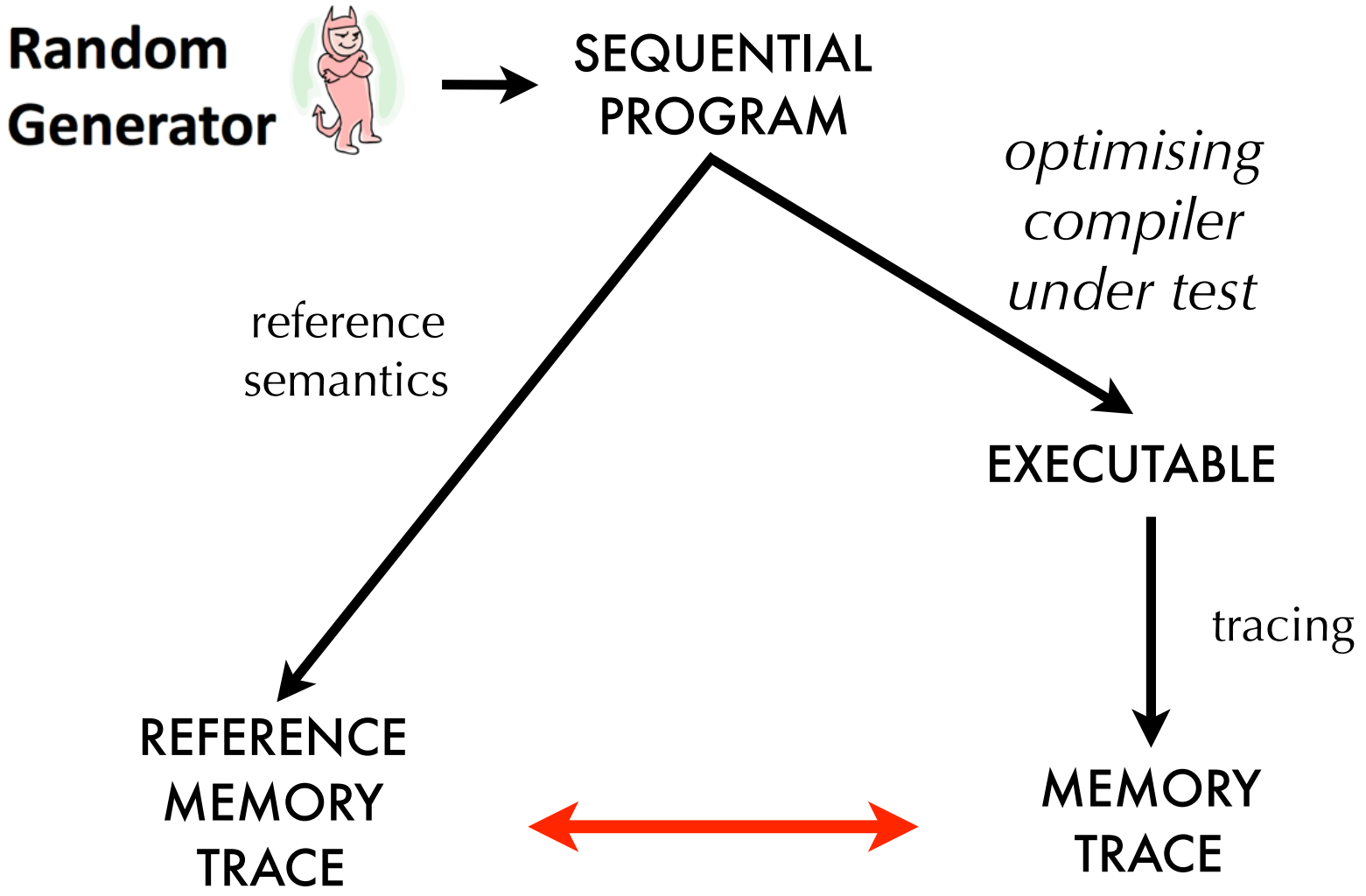
↓

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

Hunt concurrency compiler bugs

=

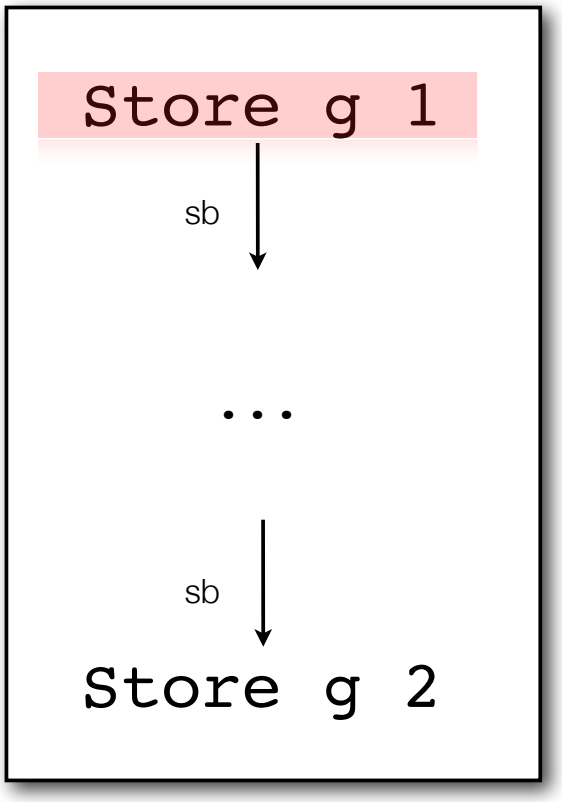
search for transformations of sequential code not sound in an arbitrary non-racy context



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.

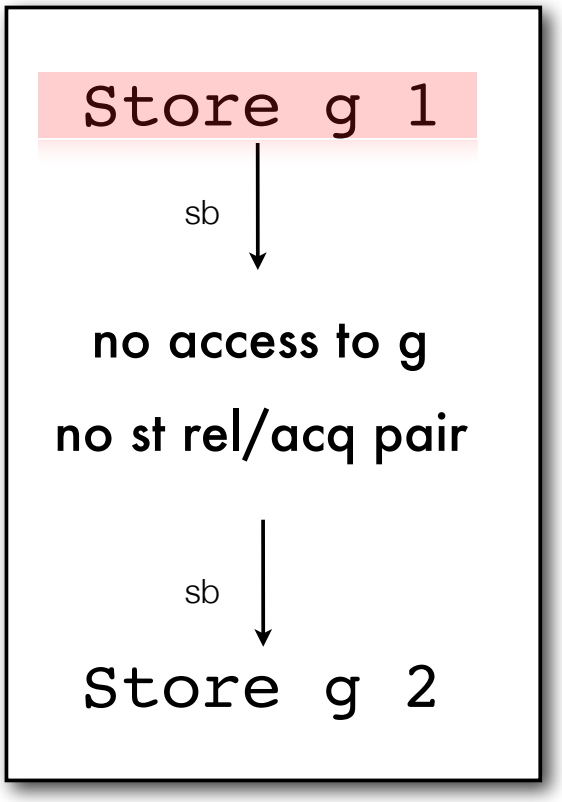
An action is a *release* if it is a possible source of a synchronisation

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

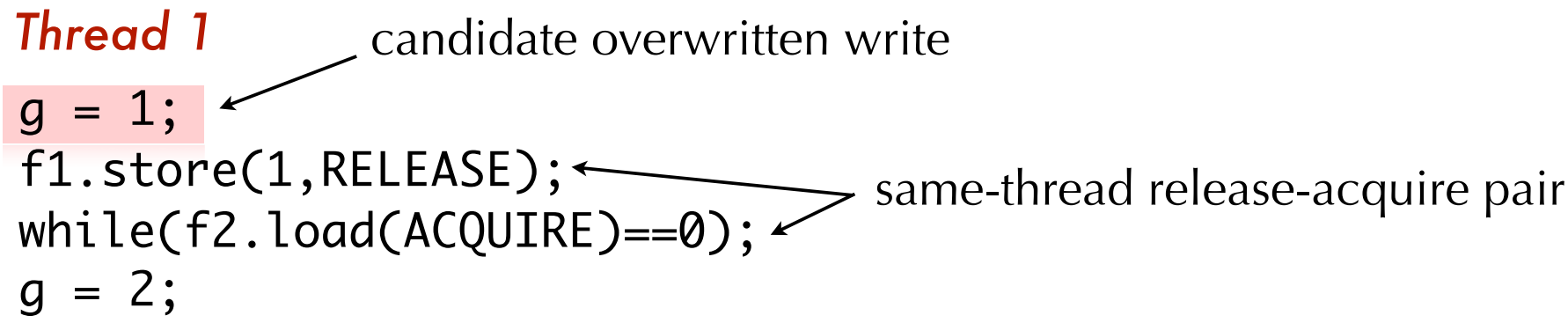

```
g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
```

The soundness condition

Shared memory

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

Thread 1



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

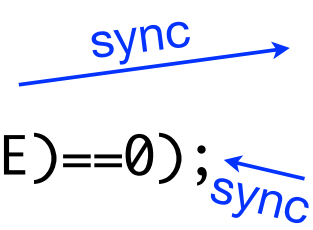
```
g = 0; atomic f1 = f2 = 0;
```

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

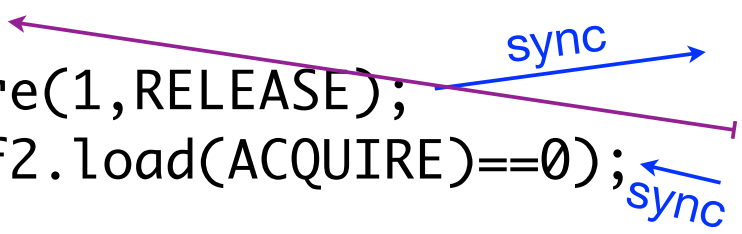
```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

Thread 2

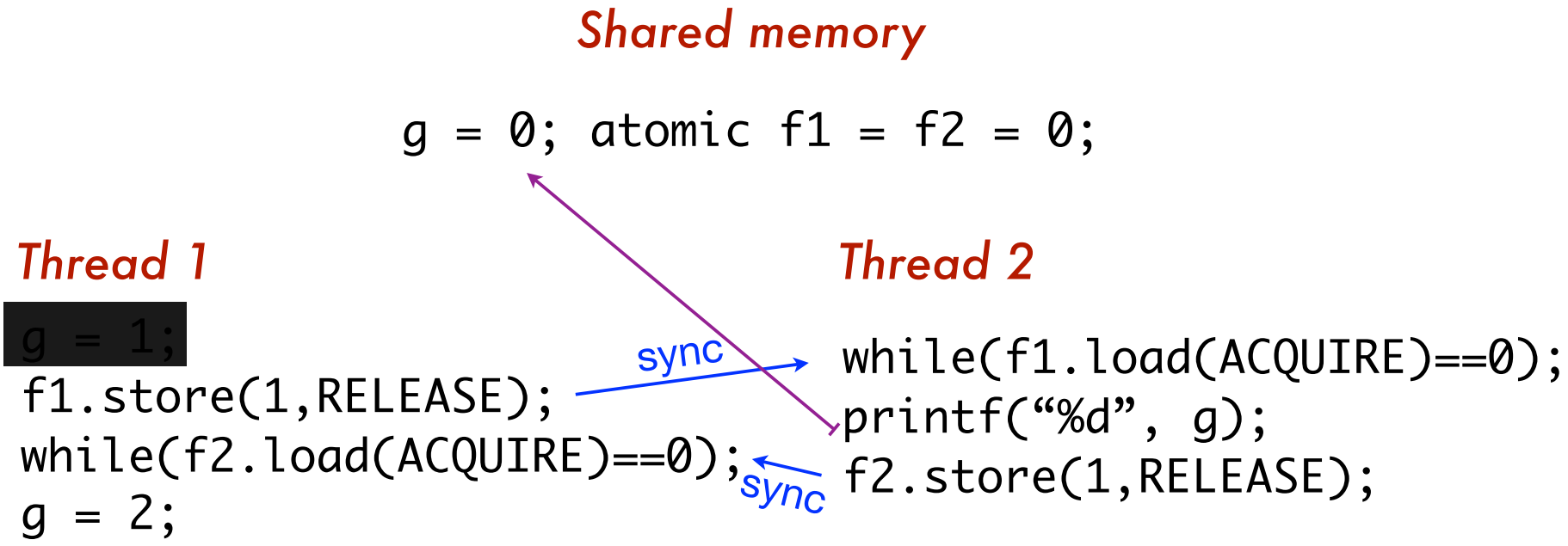
```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```



Thread 2 is non-racy

The program should only print 1

The soundness condition



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;
```

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);
```

The soundness condition

Shared memory

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

Thread 1

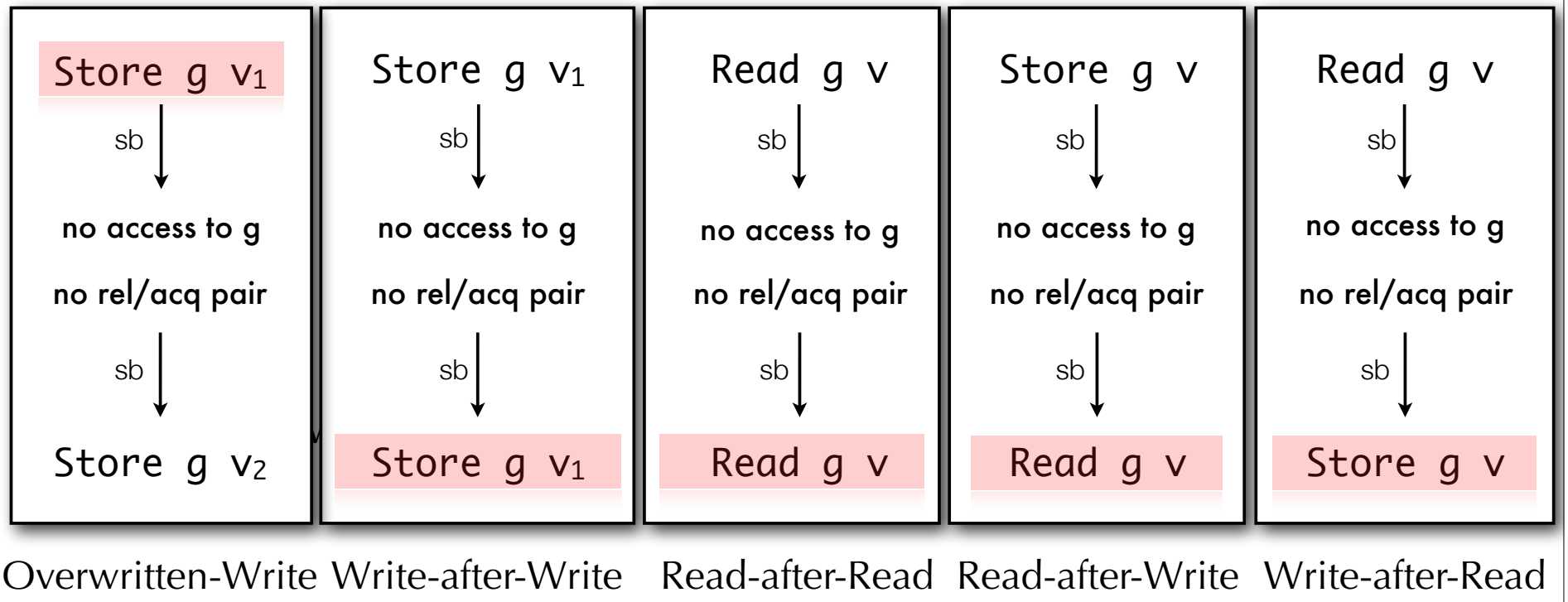
```
g = 1;
f1.store(1,RELEASE);

g = 2;
```

Thread 2

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

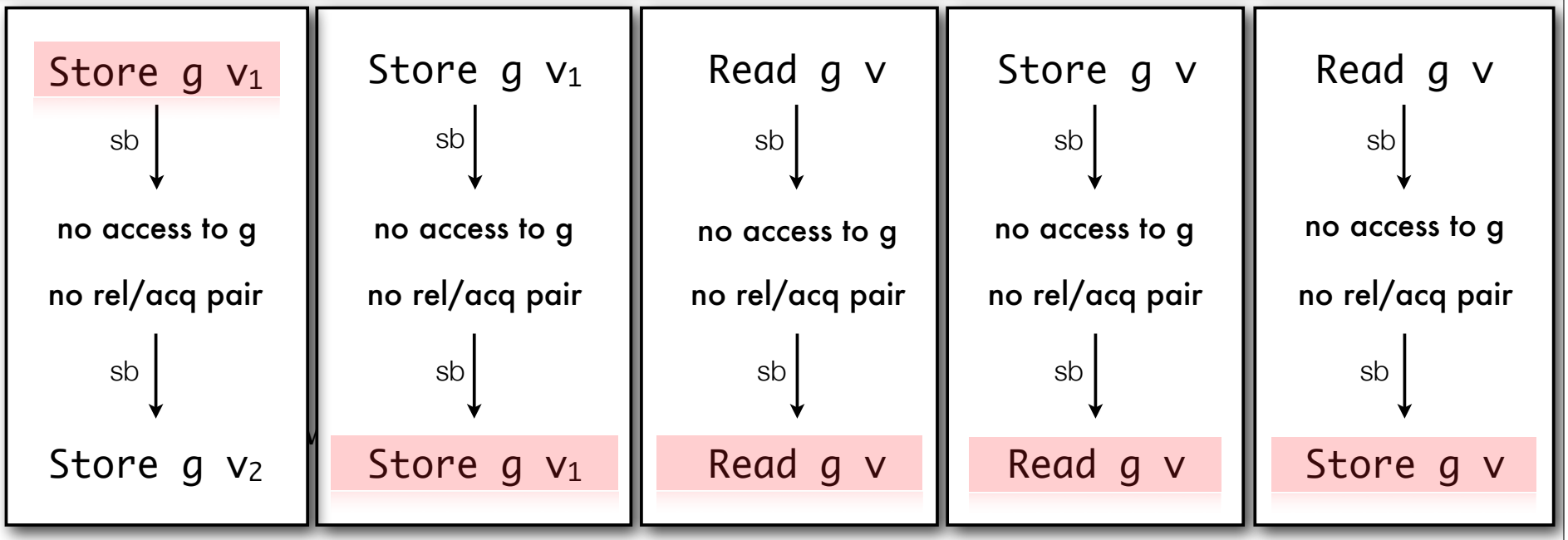
If only a release (or acquire) is present, then
all discriminating contexts *are racy*.
It is sound to optimise the overwritten write.

Eliminations: bestiary



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.



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

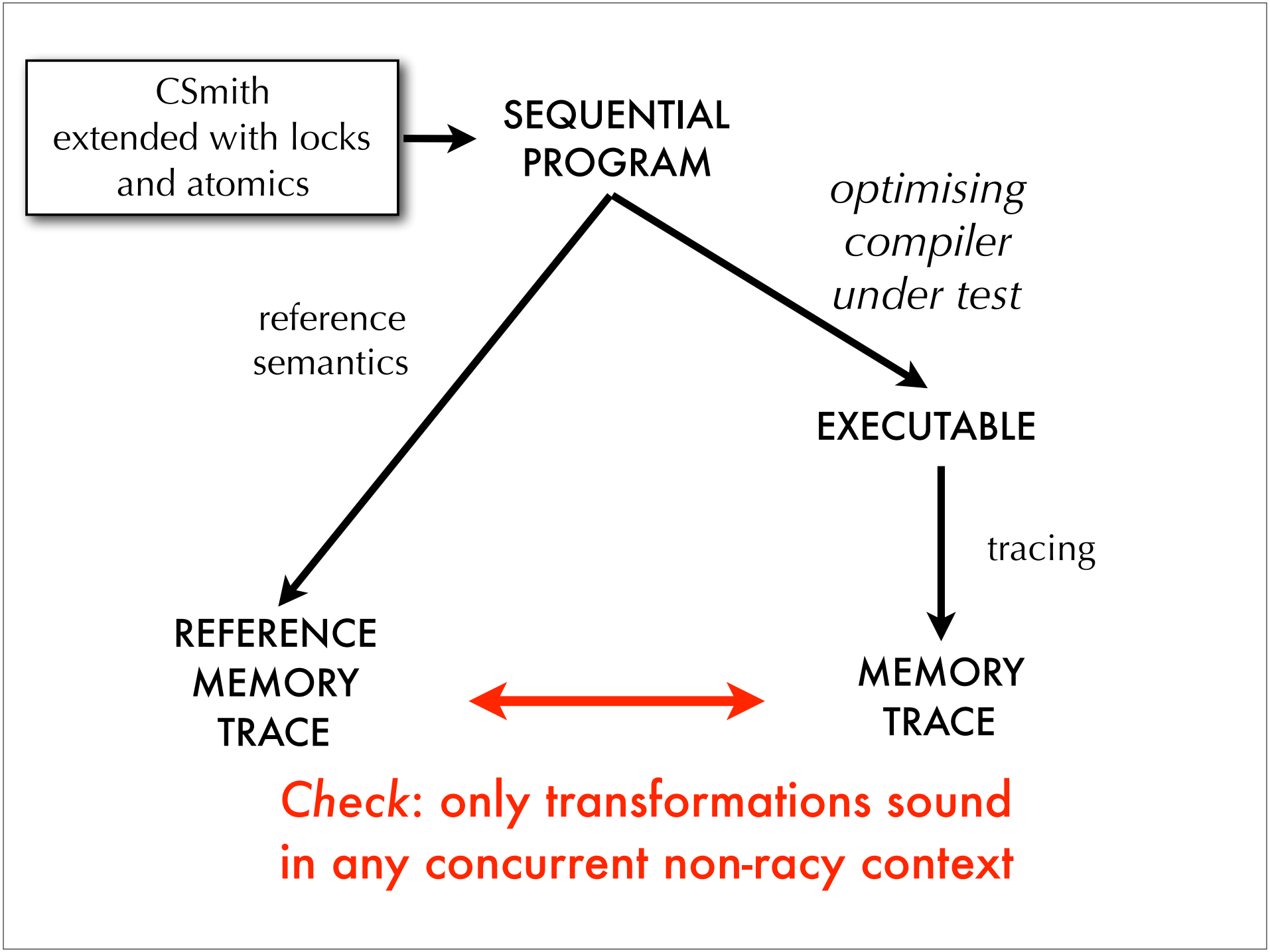
EXECUTABLE

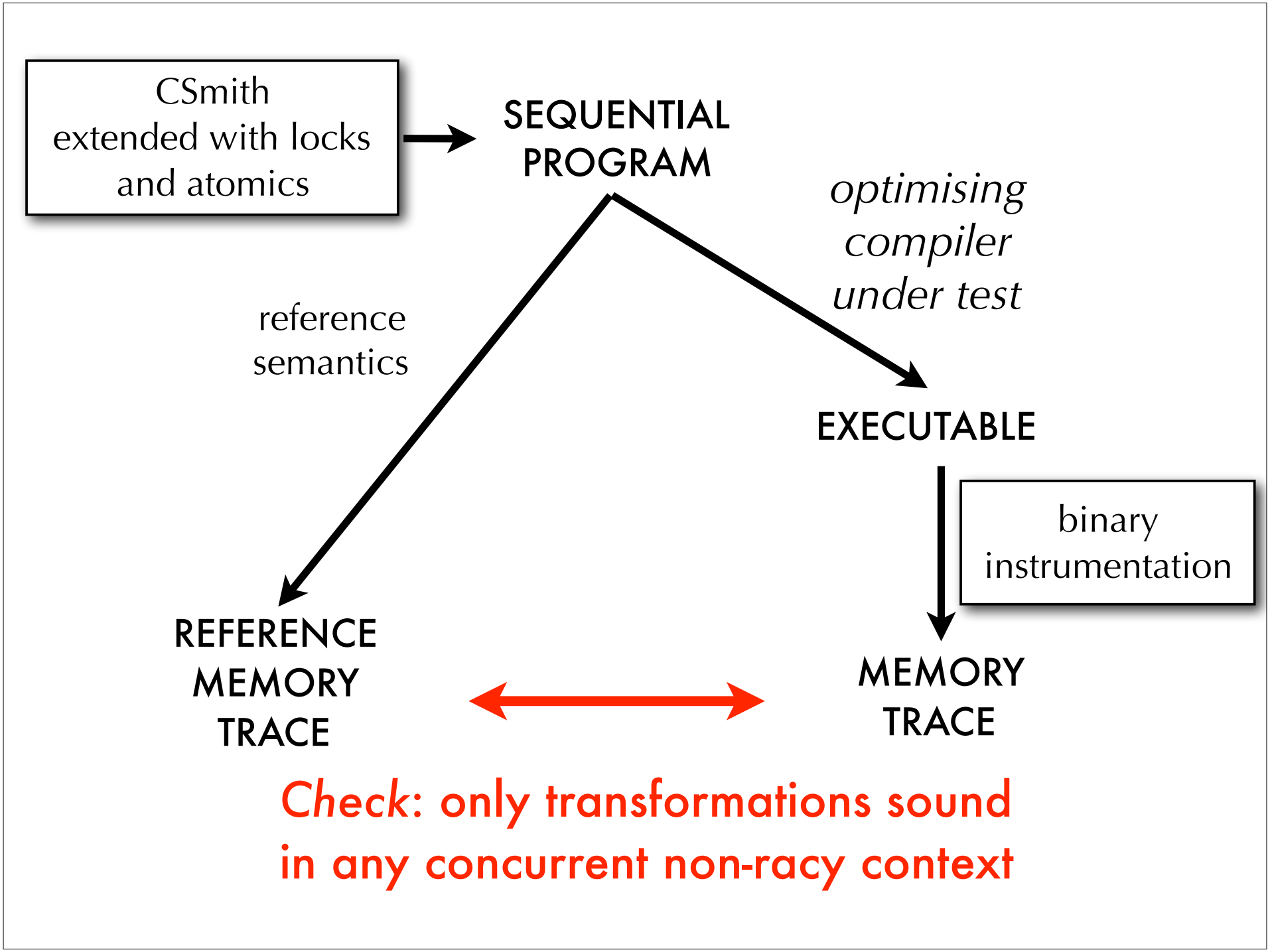
tracing

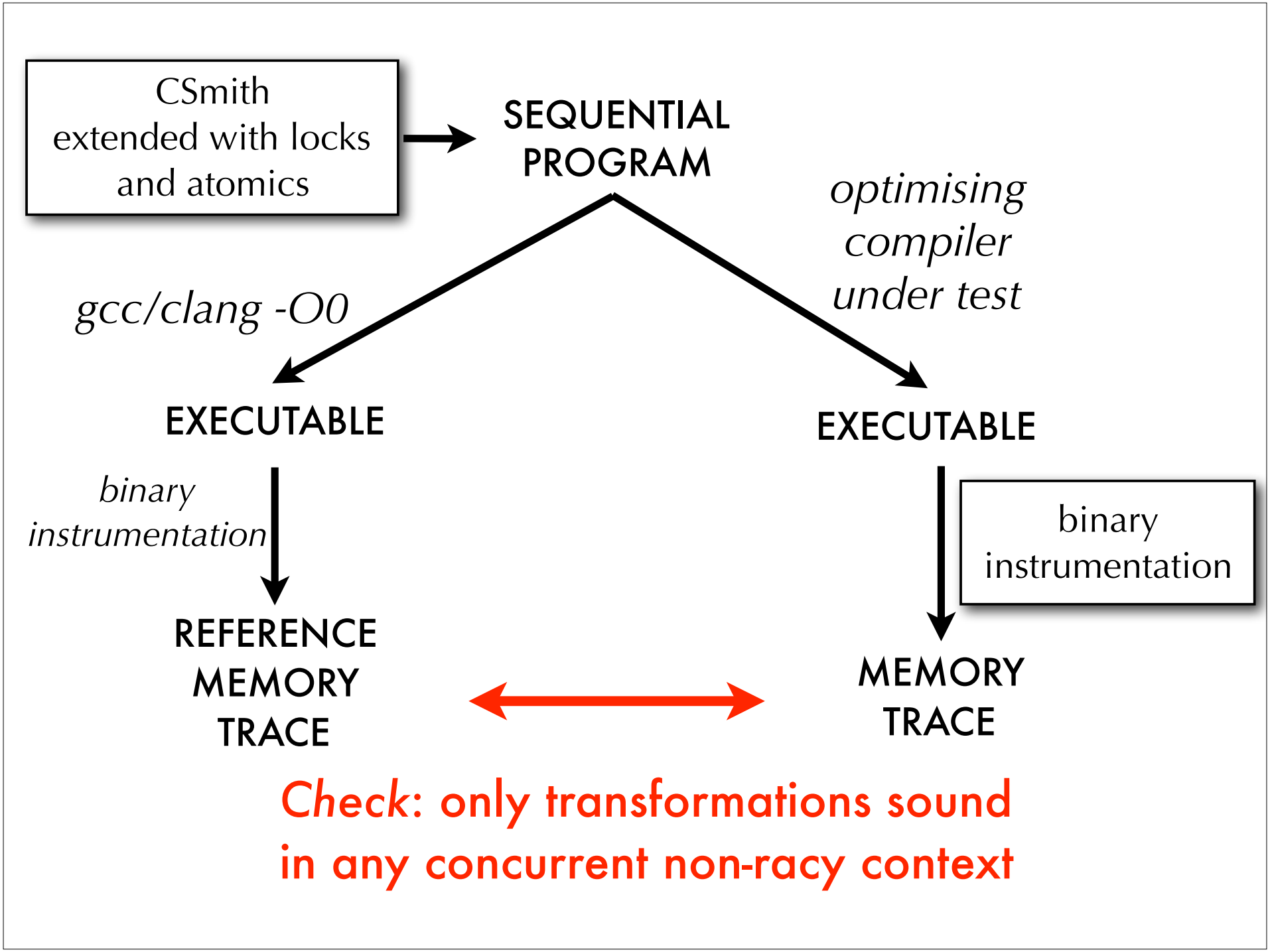
REFERENCE MEMORY TRACE

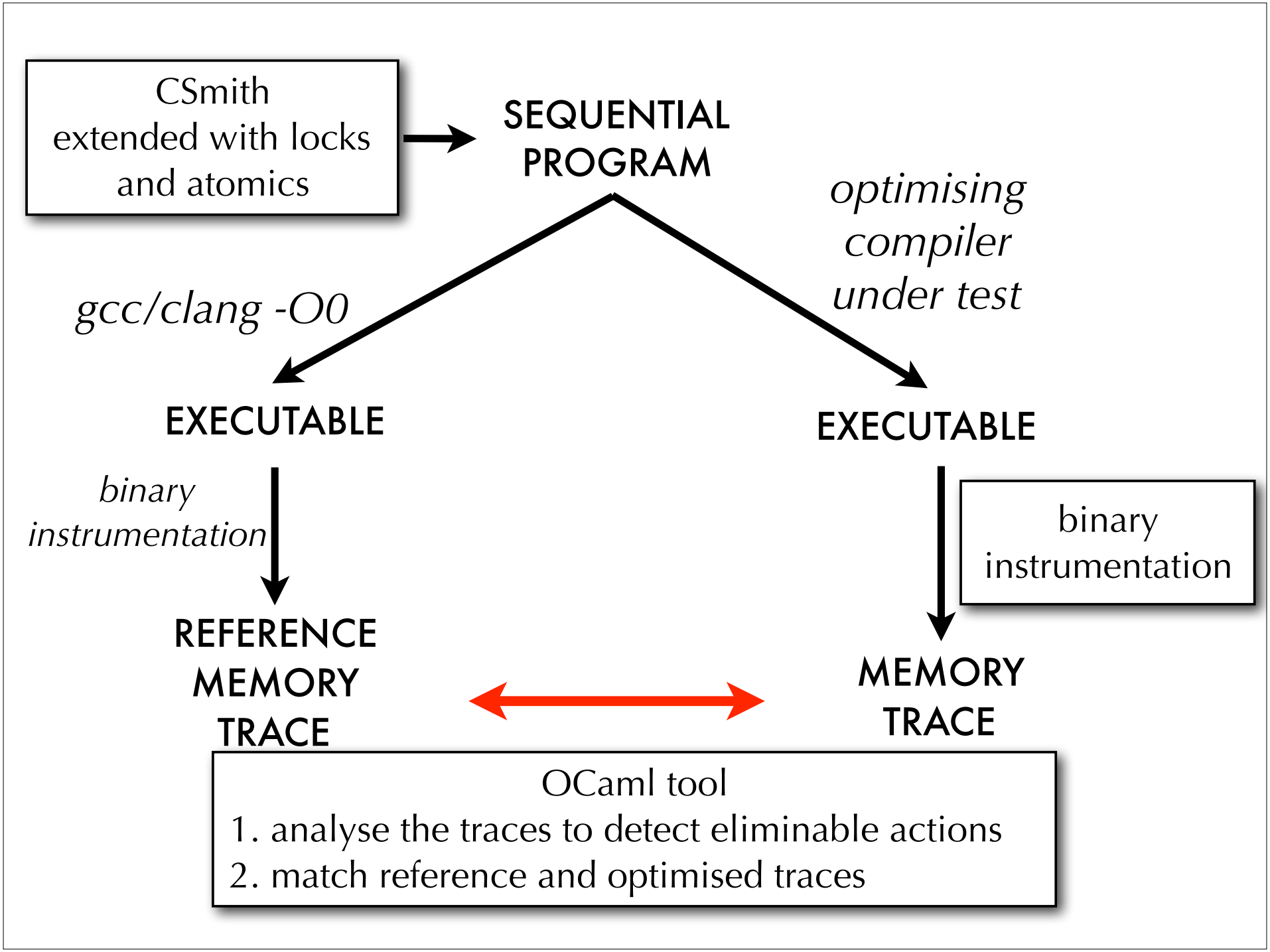
MEMORY TRACE

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
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
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;
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));
}
```

```
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
```

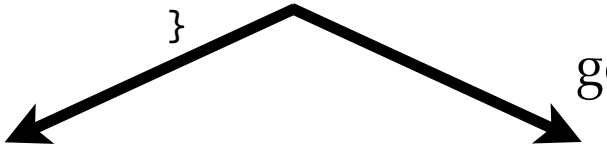
```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

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
```

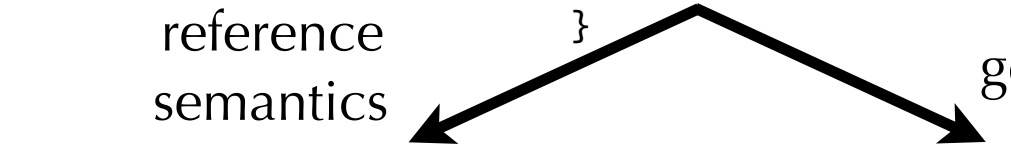
```
Load  g5 1
Store g4 0
Store g6 1
Store g5 2
Load  g4 0
```

```
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
```

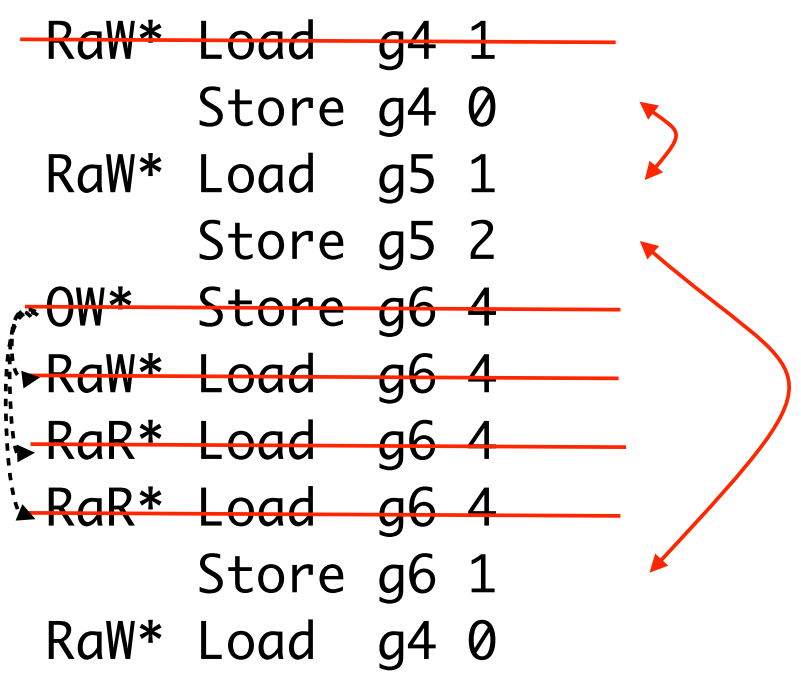
```
void func_1(void){
    int *l8 = &g6;
```

Can match applying
only correct eliminations and reorderings

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

reference semantics

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...

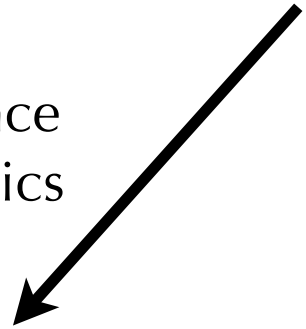
```
int a = 1;
int b = 0;
```

```
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)
    ;
}
```

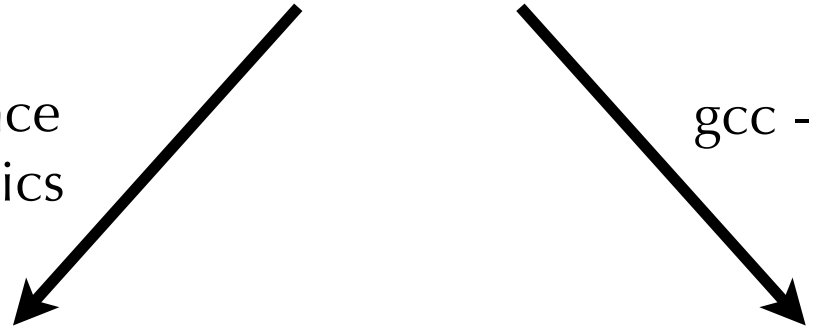
reference semantics



```
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

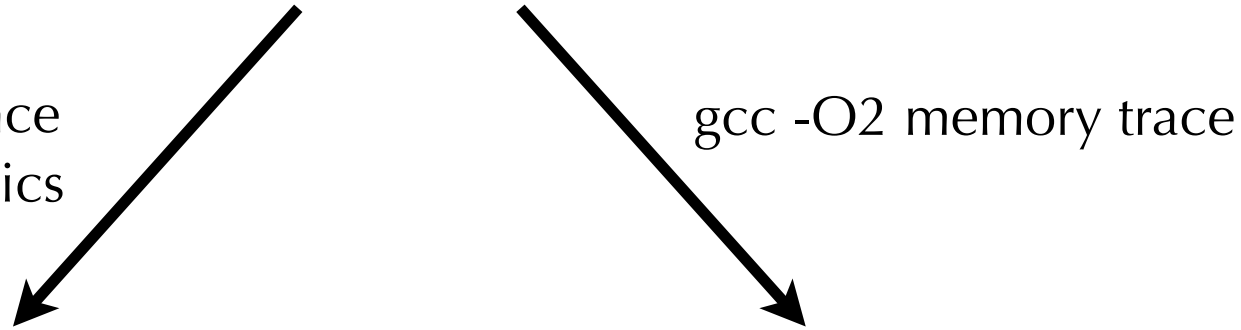
```
Load a 1
```

gcc -O2 memory trace

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

Cannot match some events ⟶ detect compiler bug

```
;
}
```




```
Load a 1
```

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

Applications

2013 - 2015

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

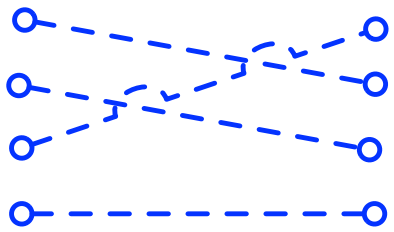
GCC internal invariant: never reorder with an atomic access

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

```
uint16_t 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

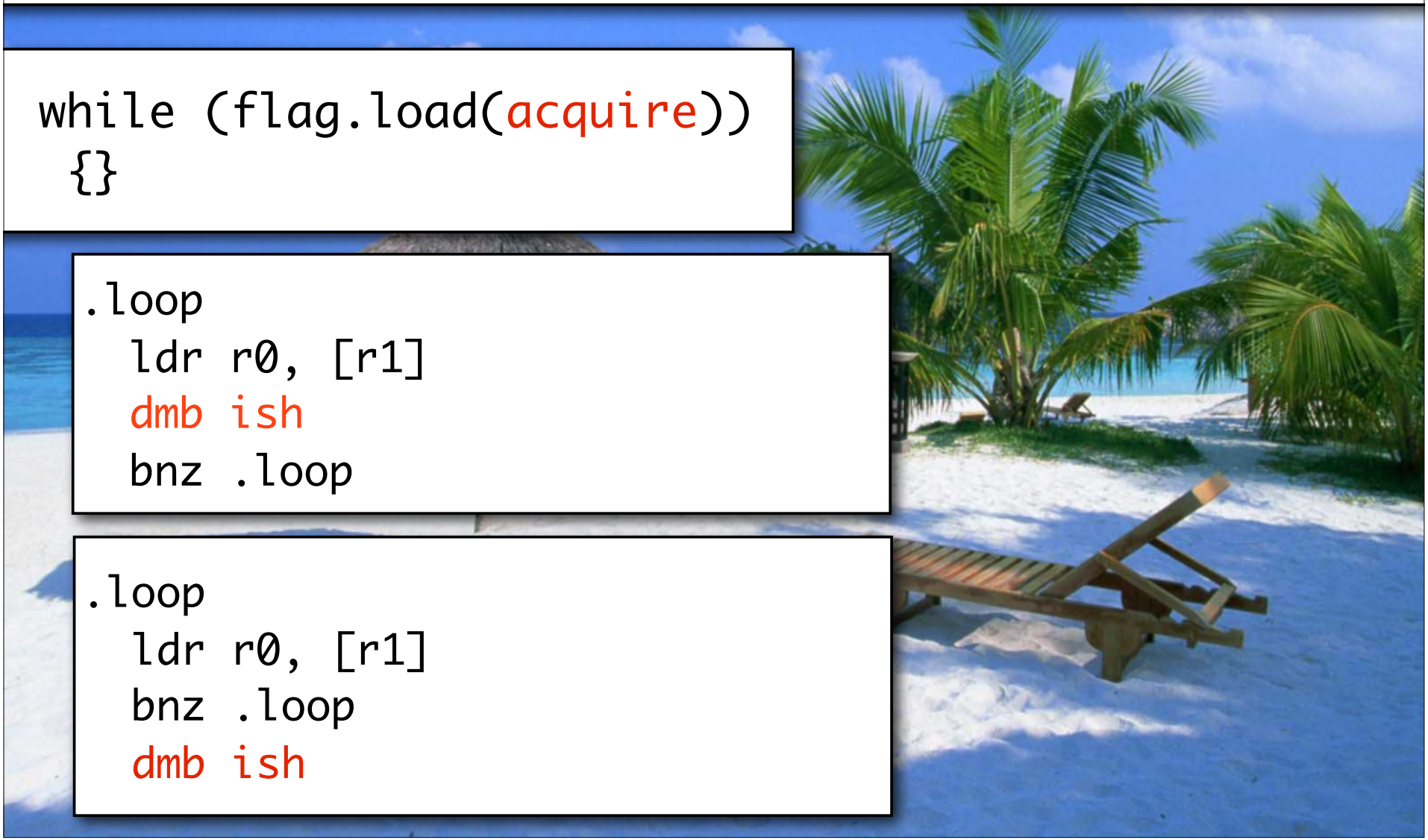
Inform new optimisations

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
```



A word on CompCertTSO

Idea: the programming language memory model *faithfully* mimics the processor model.

Our we might want radically different

programming languages!

(Radically different language = radically different challenges?)

A semantic preserving compiler, CompCertTSO

Intel processors implement the x86-TSO MM

Resources

http://www.cl.cam.ac.uk/~pes20/weakmemory/index.html

Starting point:

J. Sevcik

Safe Optimisations for Shared Memory Concurrent Programs

PLDI 2011

H. Bohem

Threads Cannot Be Implemented as a Library

PLDI 2005

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...

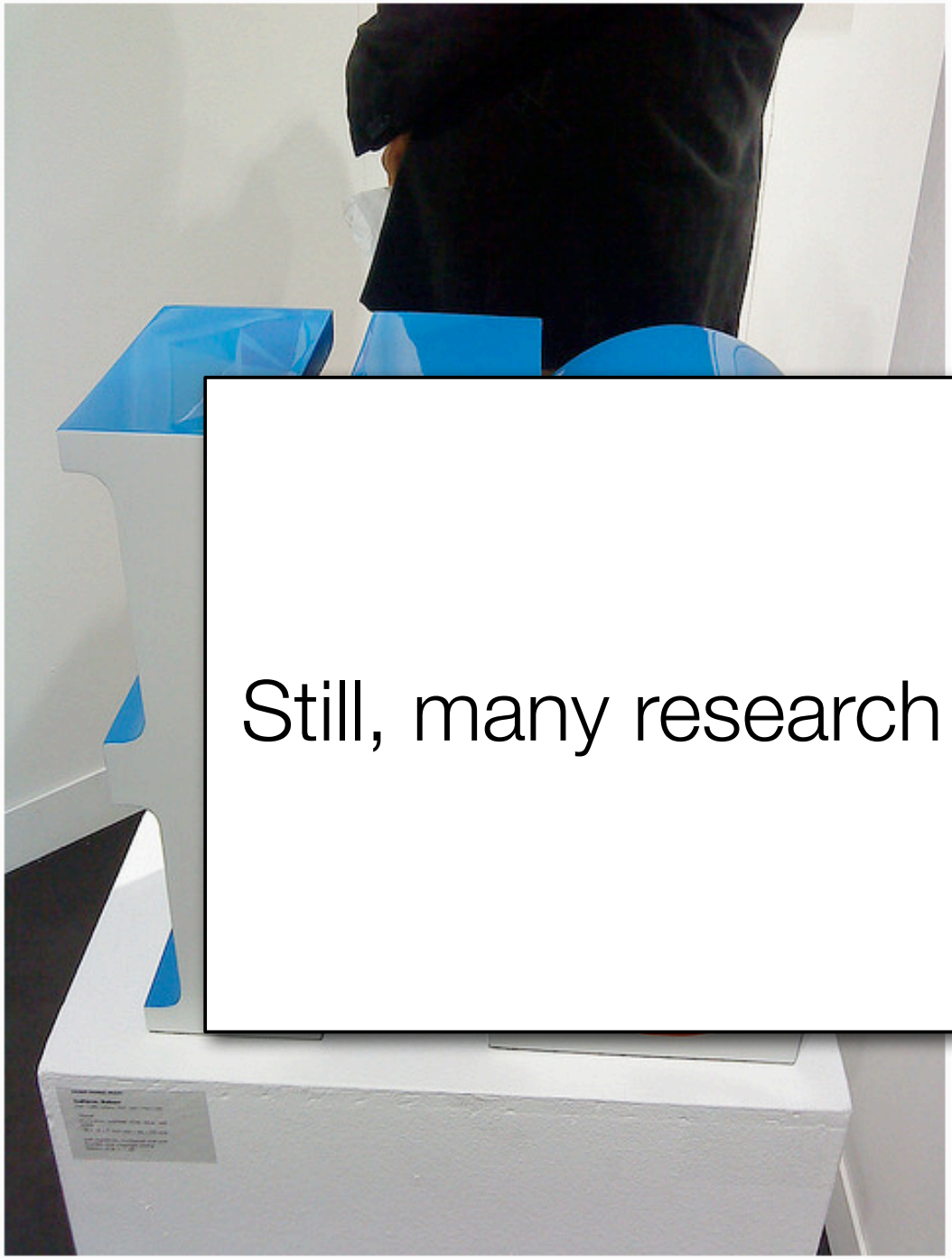
attempt

els.

mmers

se

problems.



The memory models of modern hardware are better understood.

Still, many research opportunities!

attempt

els.

mmers

se

problems.