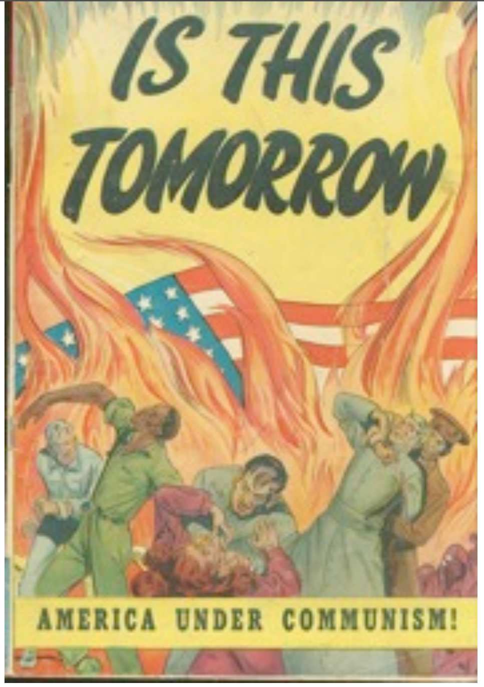
Part 1. Shared memory: an elusive abstraction

Francesco Zappa Nardelli INRIA Paris-Rocquencourt

http://moscova.inria.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.



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

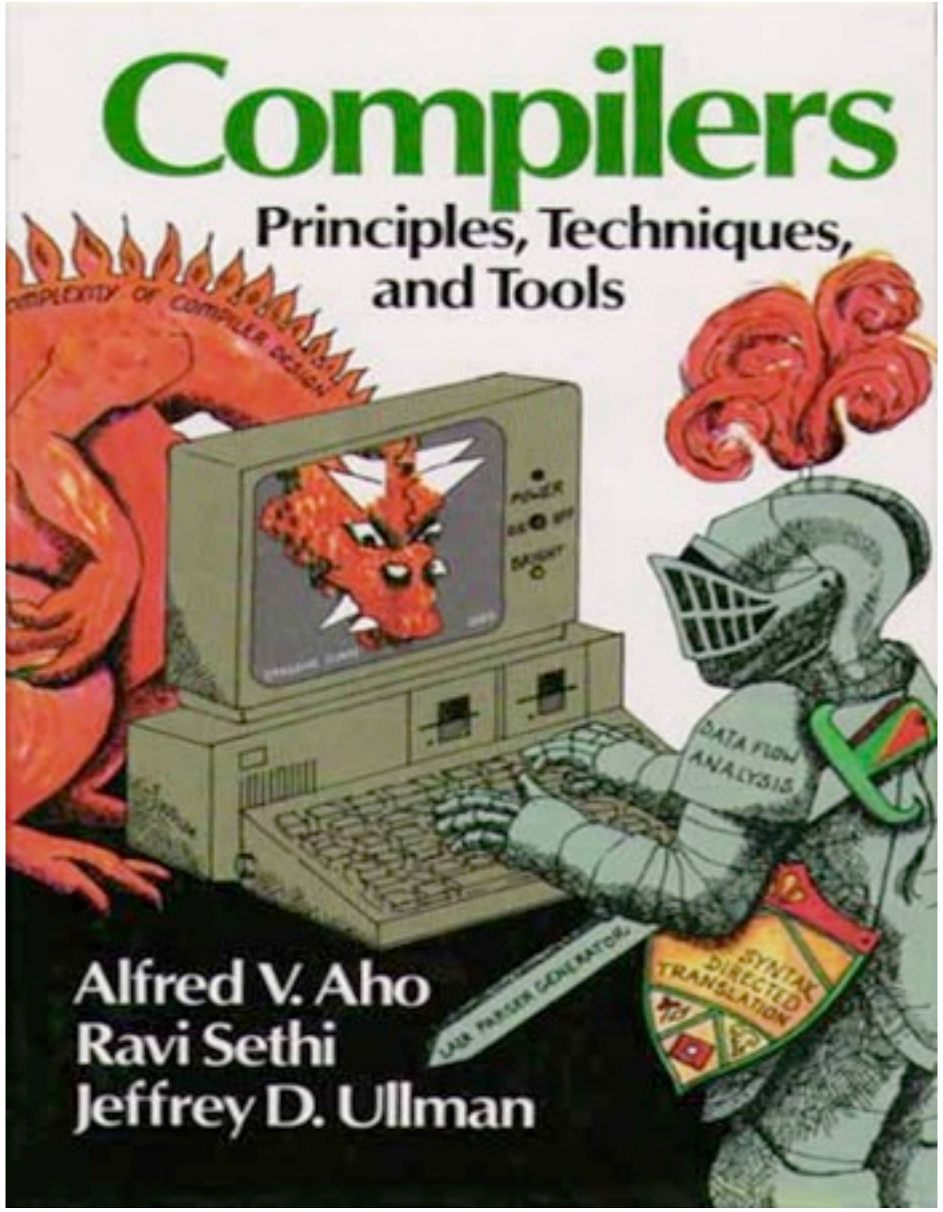
Francesco Zappa Nardelli INRIA Paris-Rocquencourt

http://moscova.inria.fr/~zappa/projects/weakmemory

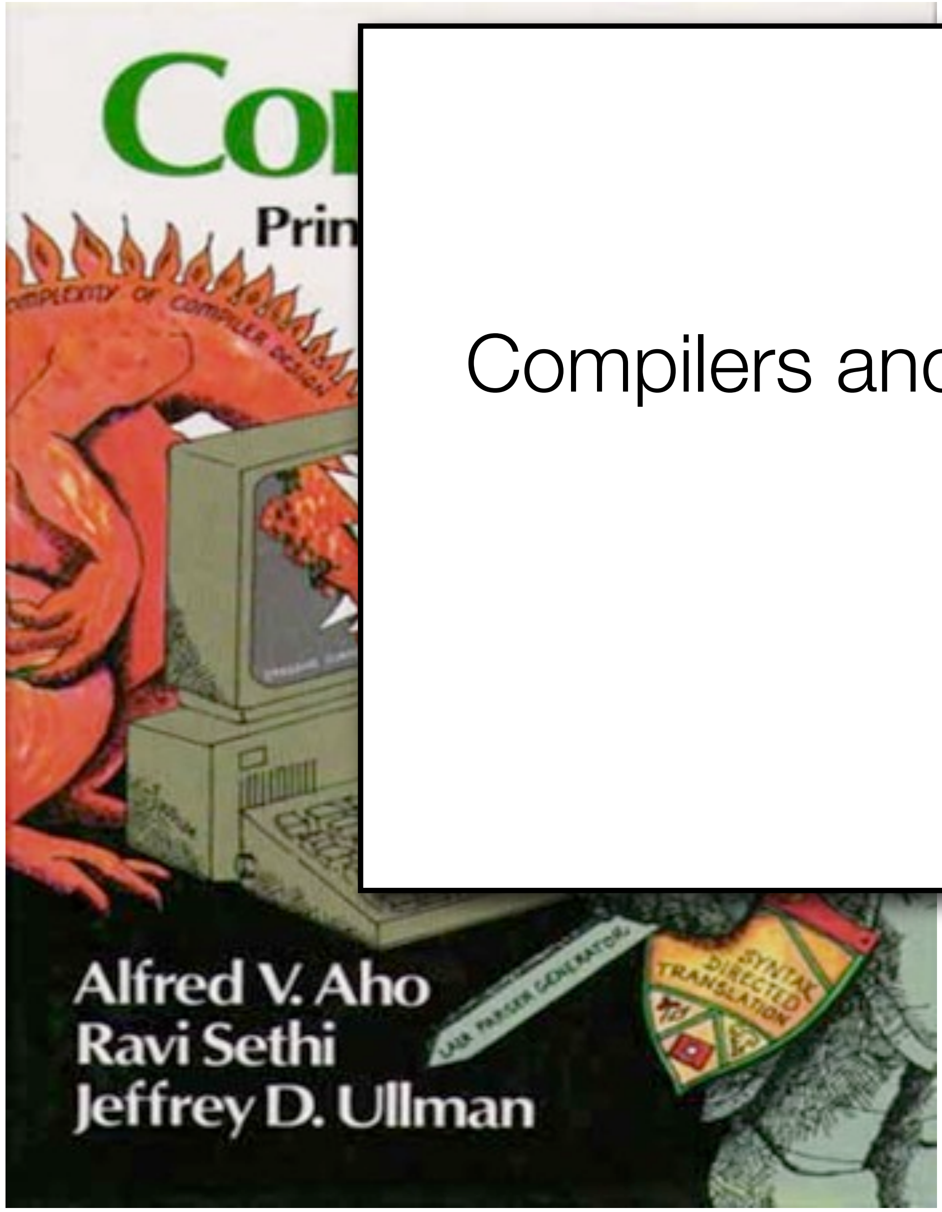
Based on work done by or with

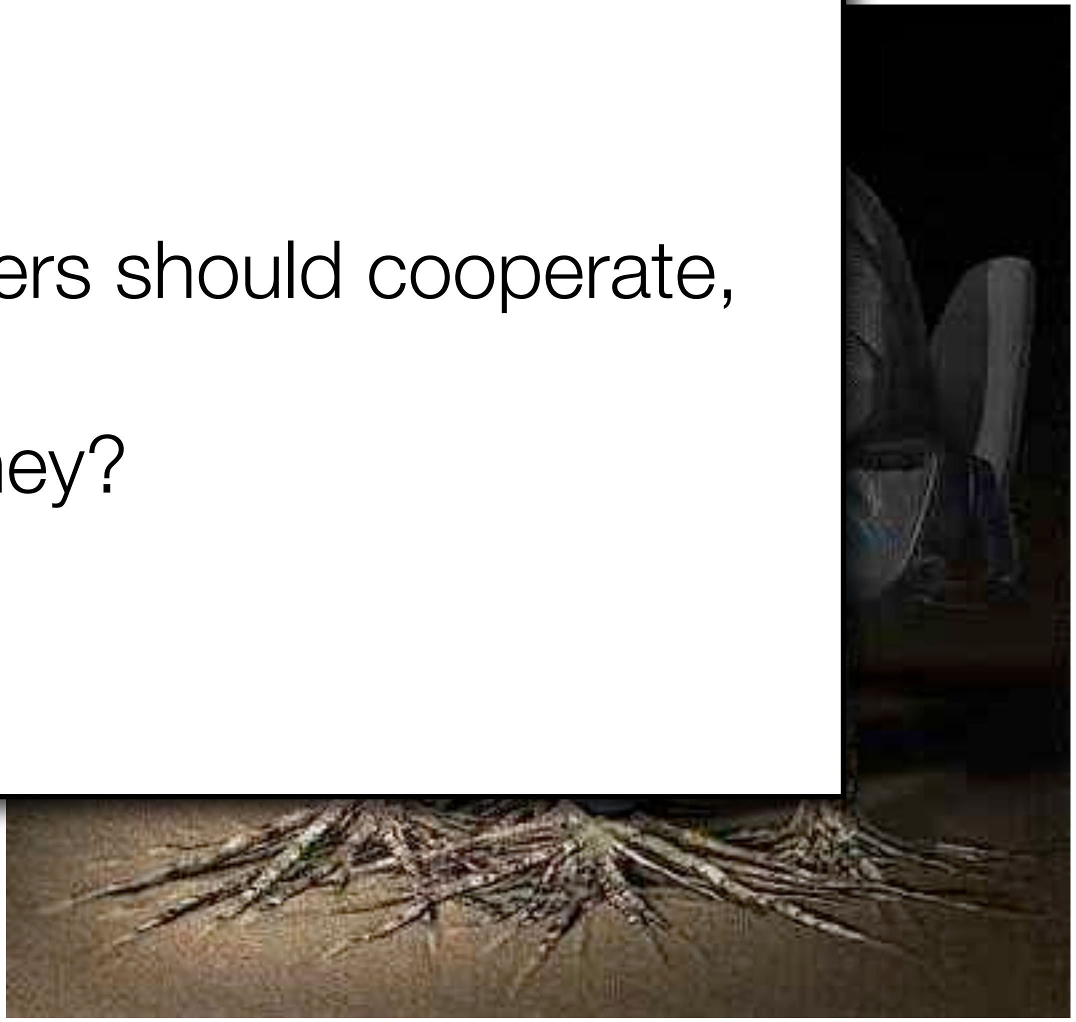
Peter Sewell, Jaroslav Ševčík, Susmit Sarkar, Tom Ridge, Scott Owens, Viktor Vafeiadis, Magnus O. Myreen, Kayvan Memarian, Luc Maranget, Derek Williams, Pankaj Pawan, Thomas Braibant, Mark Batty, Jade Alglave.

Compilers vs. programmers


Compilers
Principles, Techniques, and Tools
Complexity of Compiler Design
Data Flow Analysis
LALR Parser Generator
Syntax Directed Translation
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

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~~`	`  y = 1 }`
`    print 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;          | tmp2 = *x;
 *x = tmp1 + 1;      | *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();`	`lock();`
`tmp1 = *x;`	`tmp2 = *x;`
`*x = tmp1 + 1;`	`*x = tmp2 + 1;`
`unlock();`	`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 (2)

Hardware optimisations can be observed by concurrent code:

Thread 0	Thread 1

`x = 1`
`print y`
	`y = 1`
	`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)

Hardware optimisations can be observed by concurrent code:

Thread 0	Thread 1

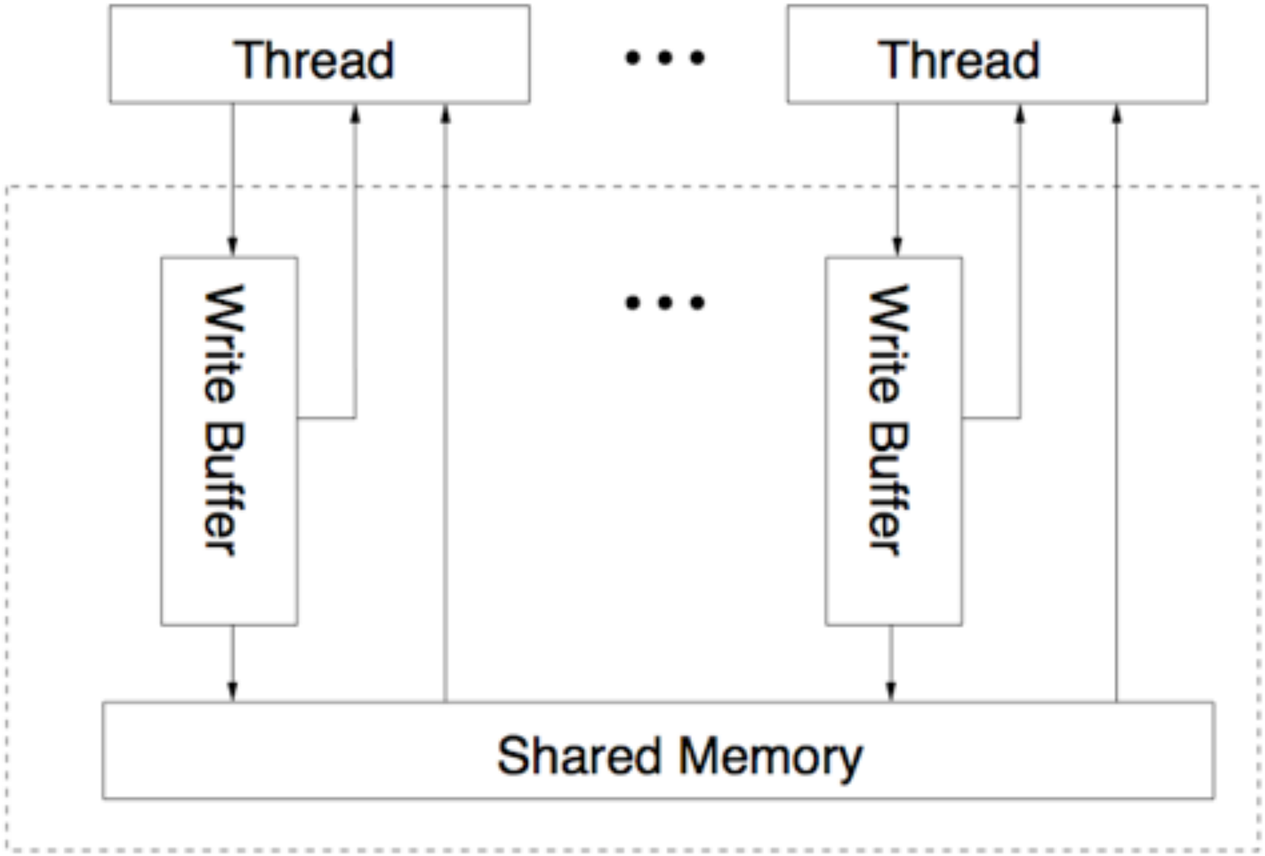
`x = 1`
`print y`
	`y = 1`
	`print x`

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...

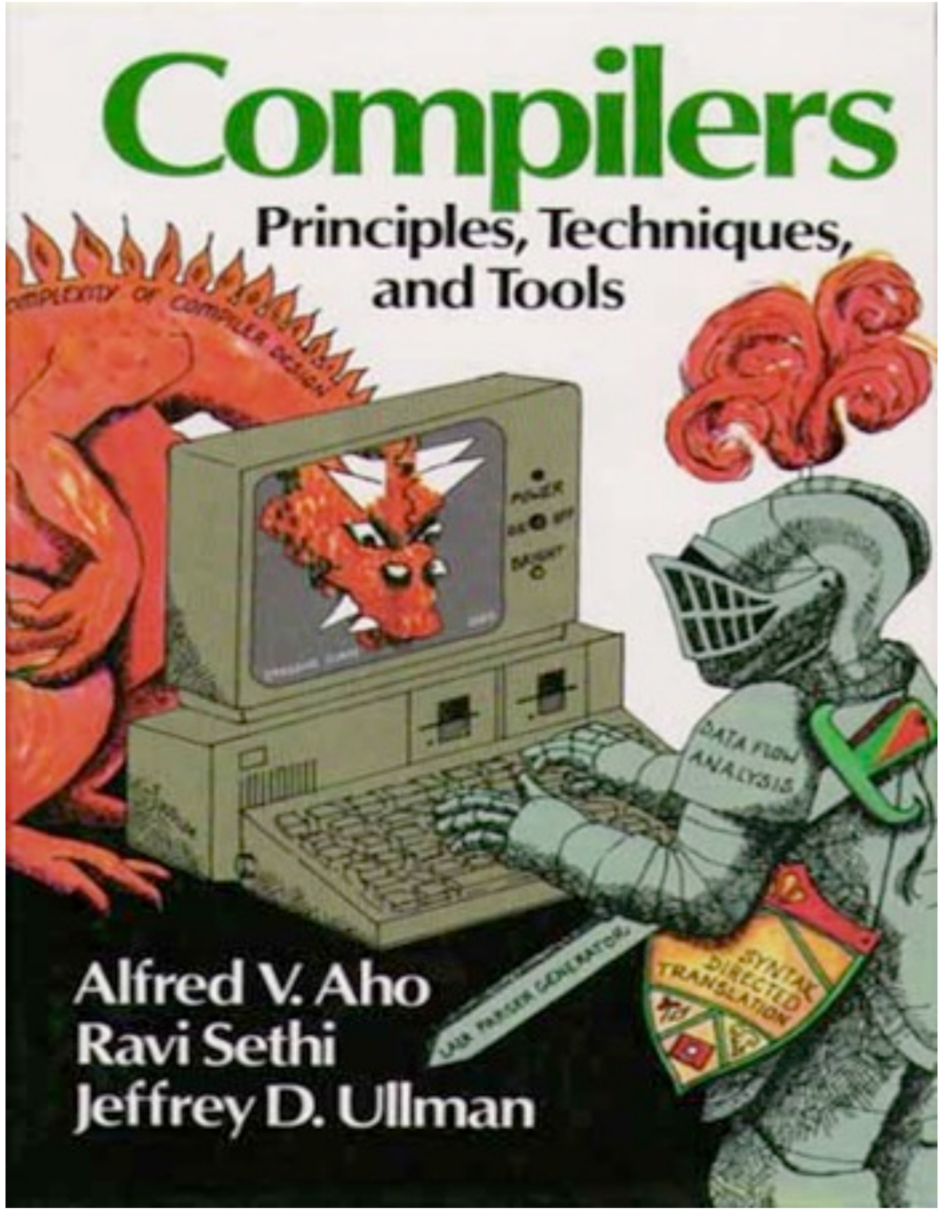
Thread 0	Thread 1
`x = 1`	`print y`
`y = 1`	`print x`

At the end of some executions:

1 0

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

Compilers vs. programmers


Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
DATA FLOW ANALYSIS
SYNTAX DIRECTED TRANSLATION
LALR PARSER GENERATOR
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

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

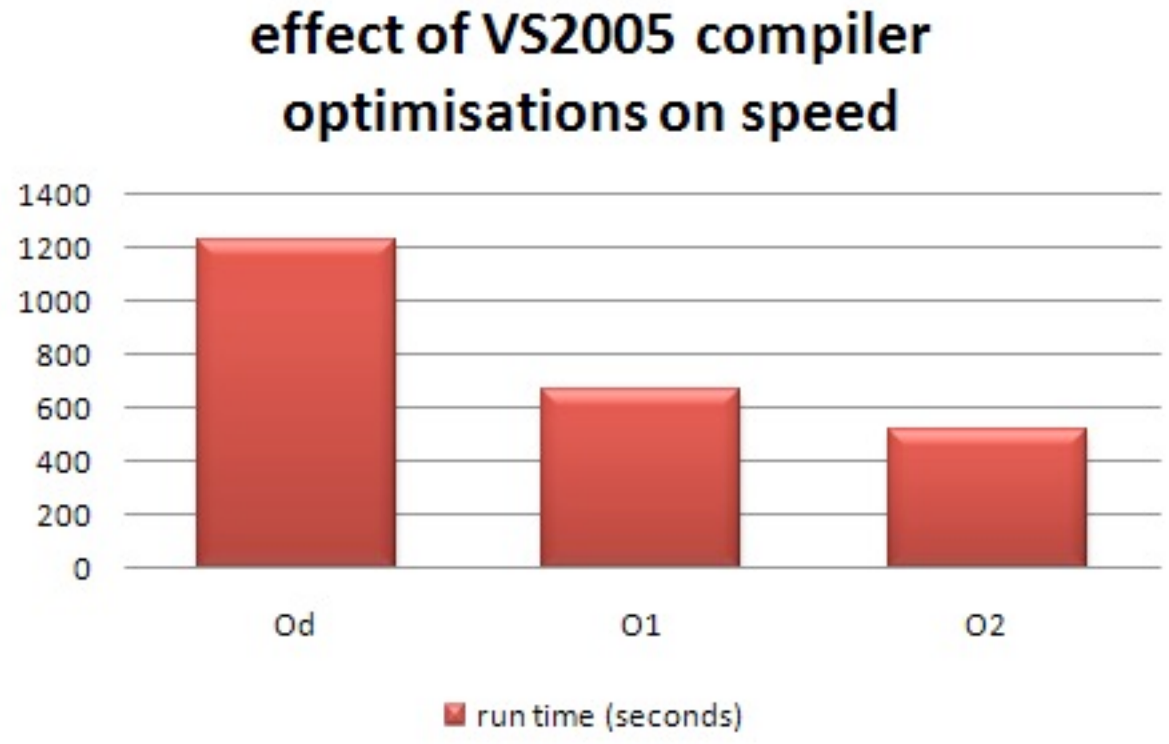
3) soundness of compiler optimisations

Previous lecture: hardware models

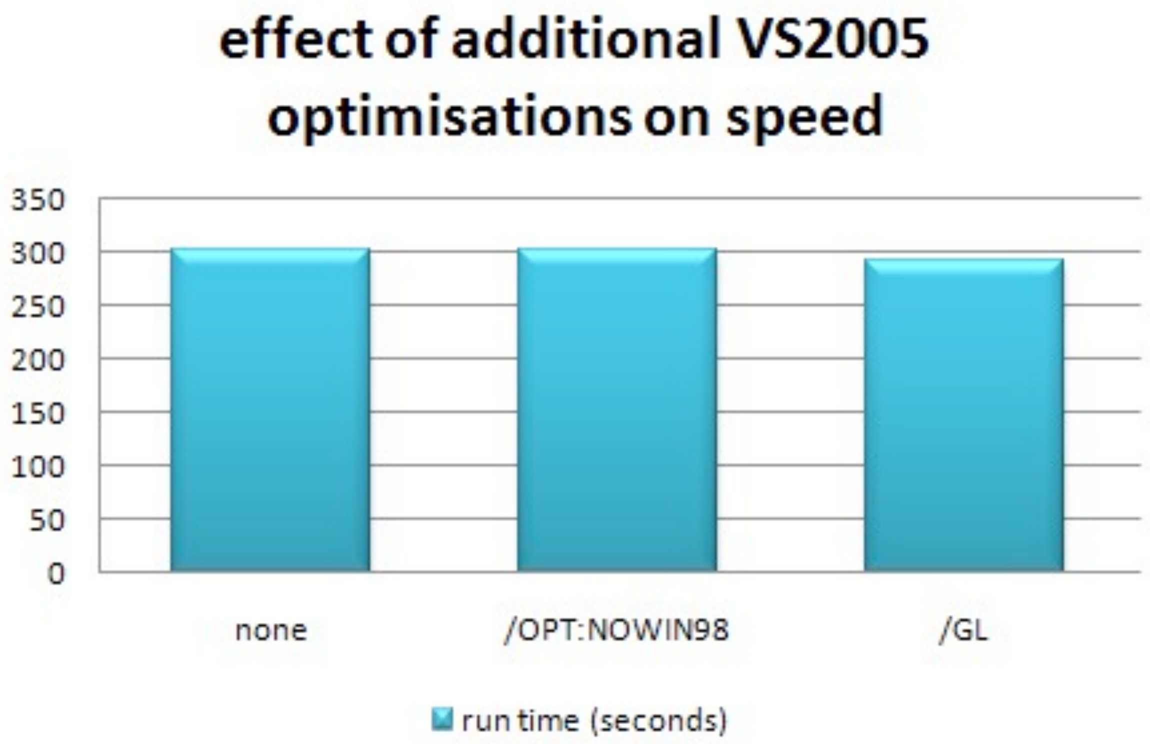
1) why are industrial specs so often flawed?

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

Memory optimisations

Optimisations of shared memory can be classified as:

Eliminations (of reads, writes, sometimes synchronisation).

Reordering (of independent non-conflicting memory accesses).

Introductions (of reads – rarely).

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

→

```
i = 0;
…
tmp = *x;
while (i != 0) {
   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.

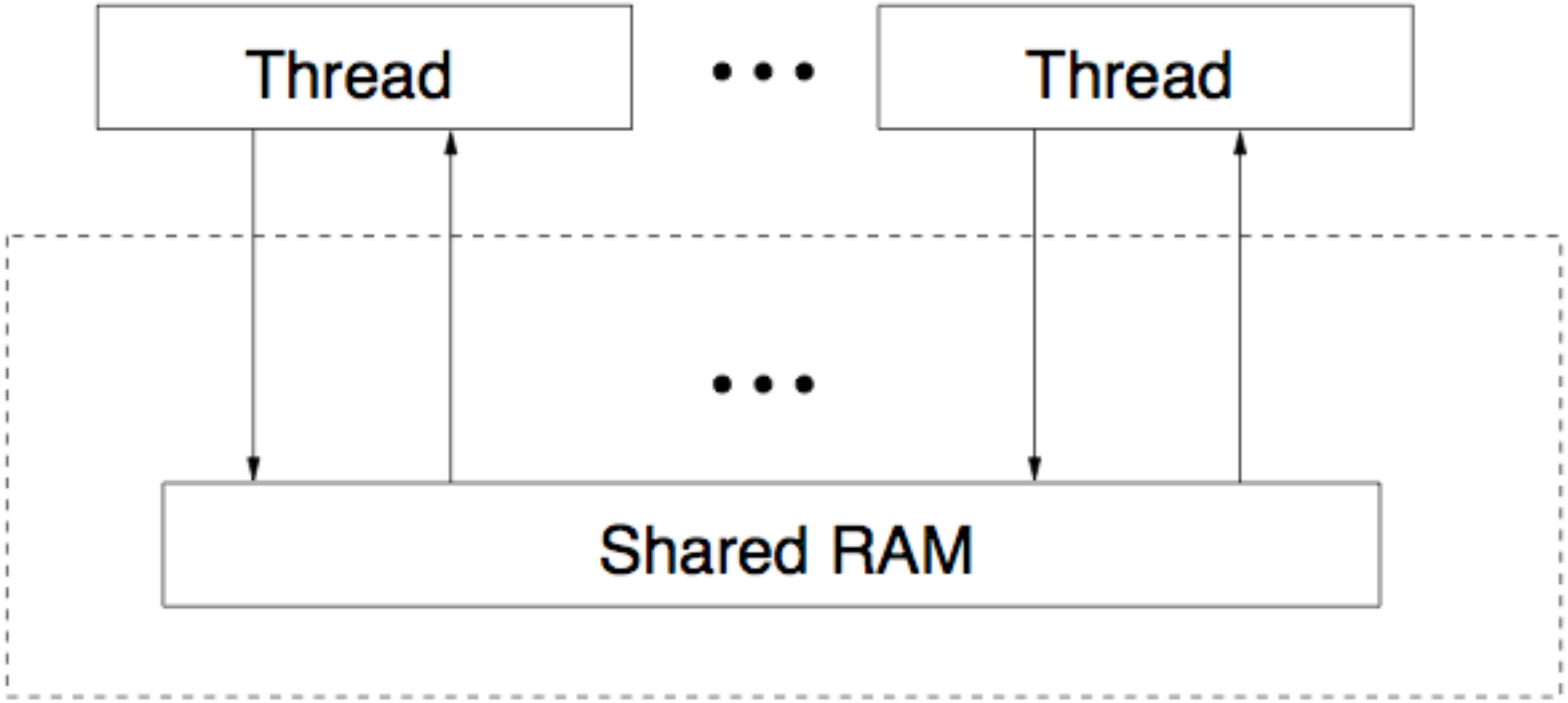
Naive answer: enforce sequential consistency

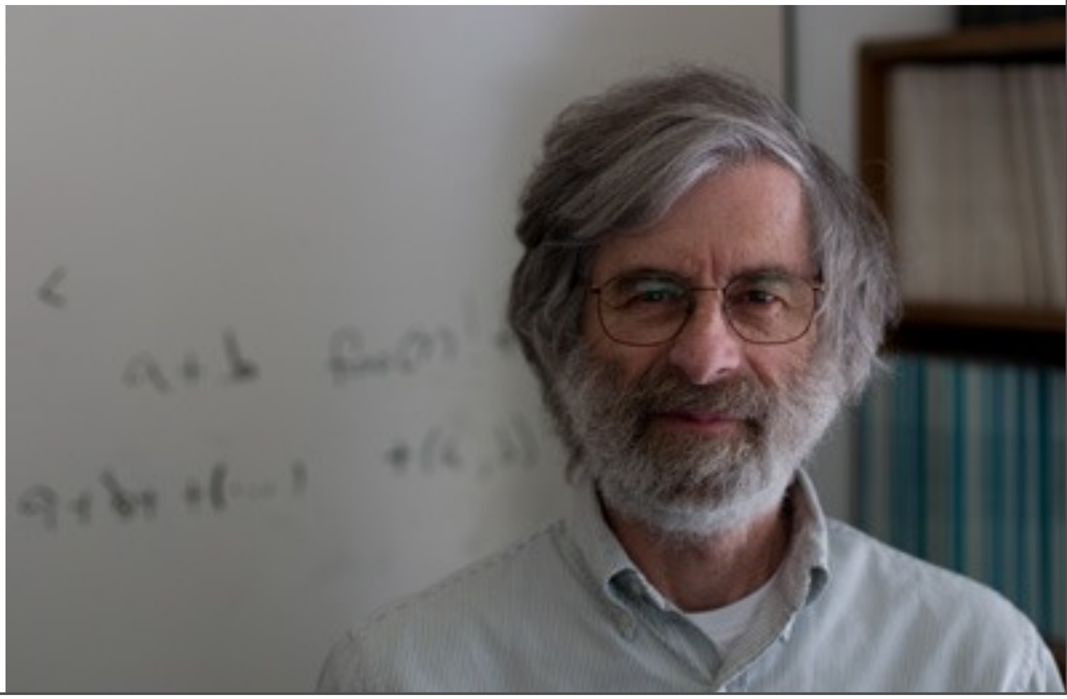
Sequential consistency

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

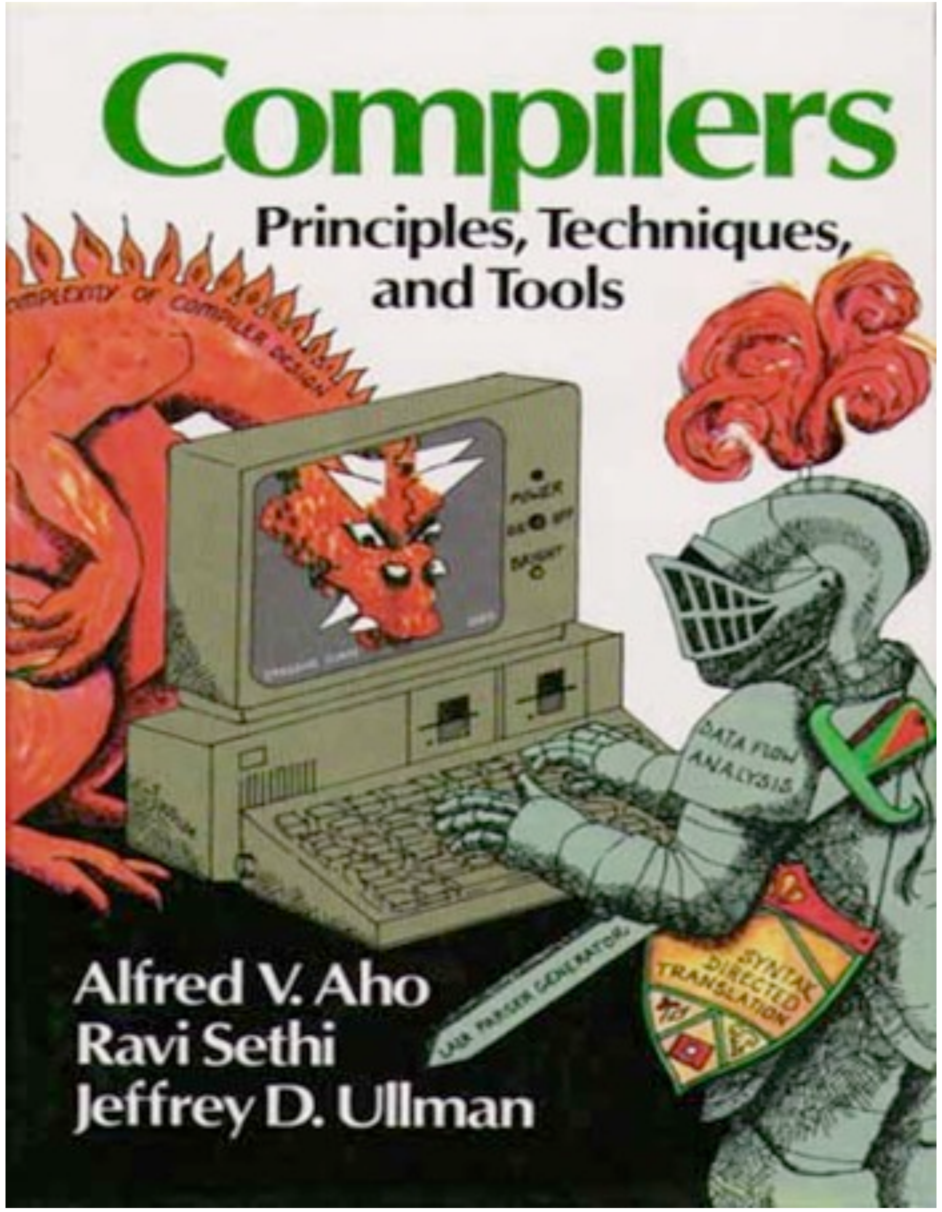
> *...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program...*

Lamport, 1979.

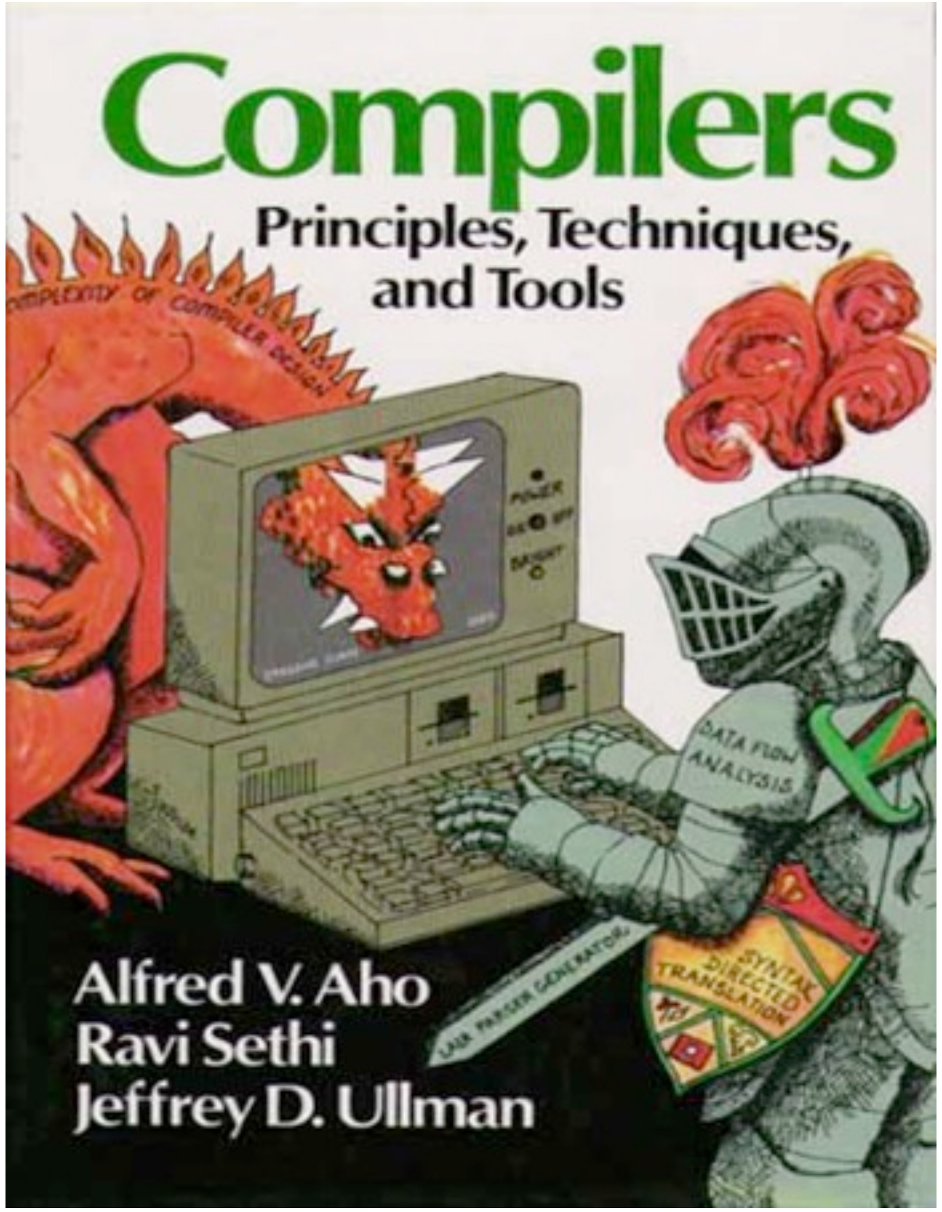


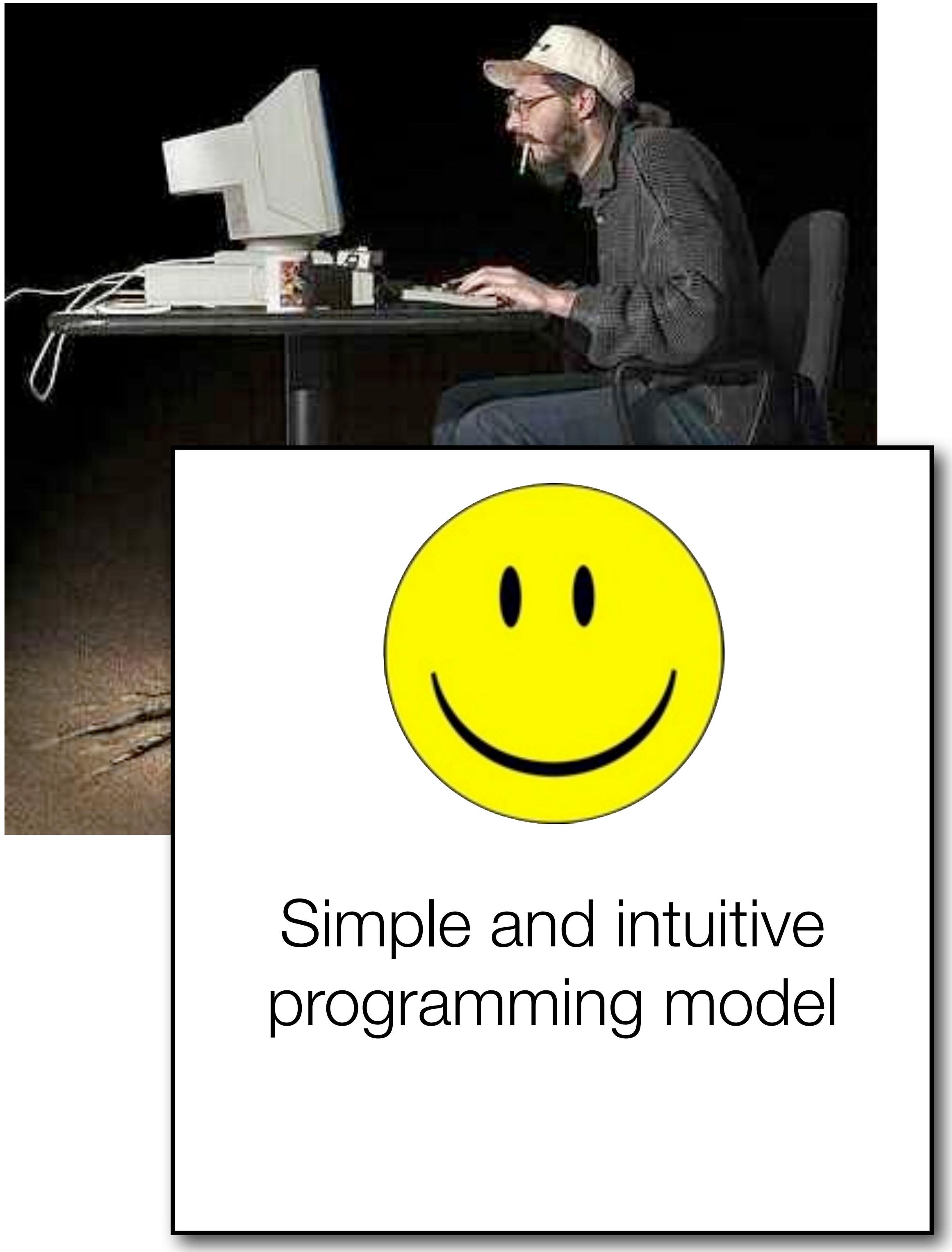


Compilers, programmers & sequential consistency



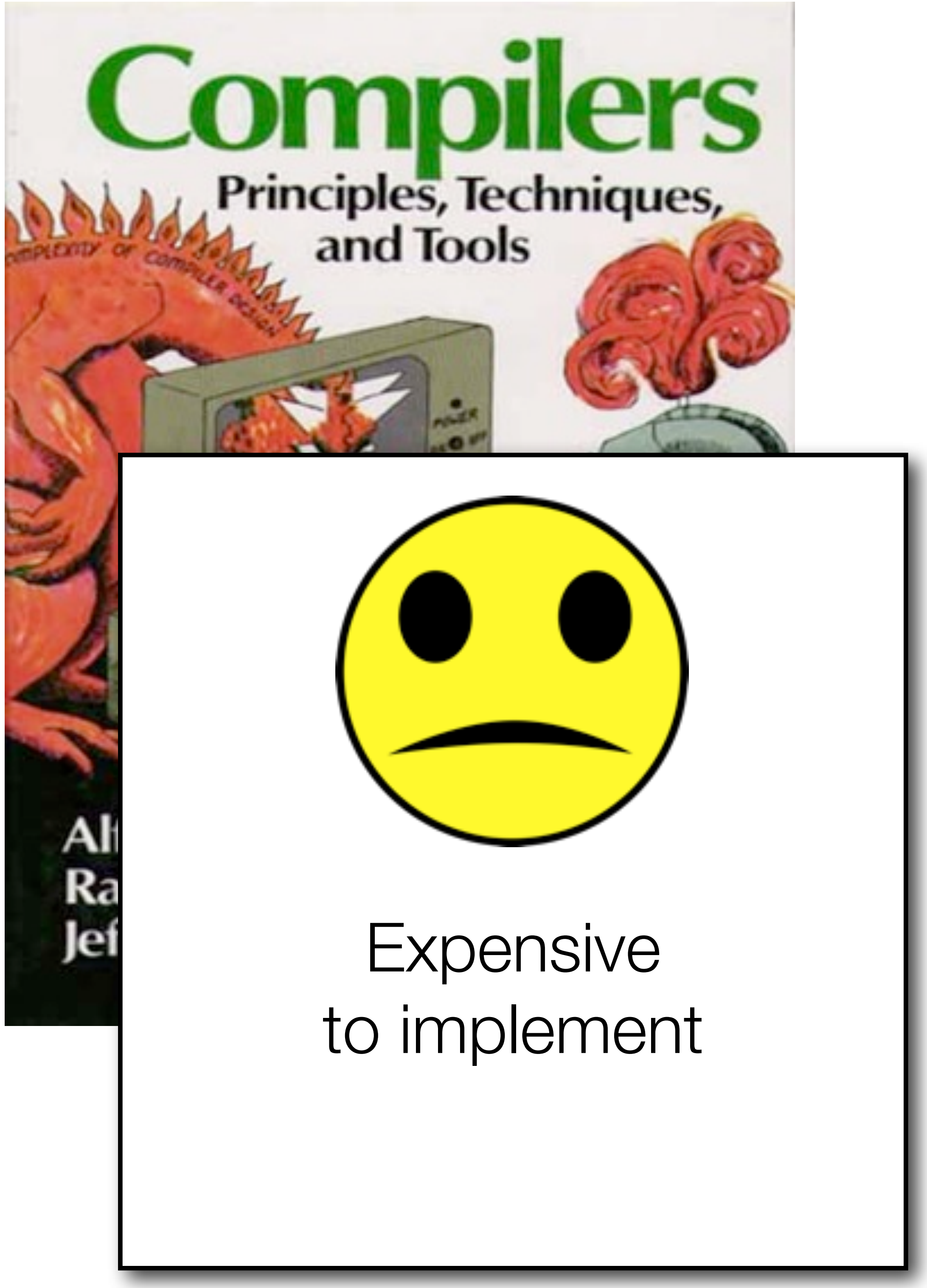
Compilers, programmers & sequential consistency


Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
DATA FLOW ANALYSIS
SYNTAX DIRECTED TRANSLATION
LALR PARSER GENERATOR
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

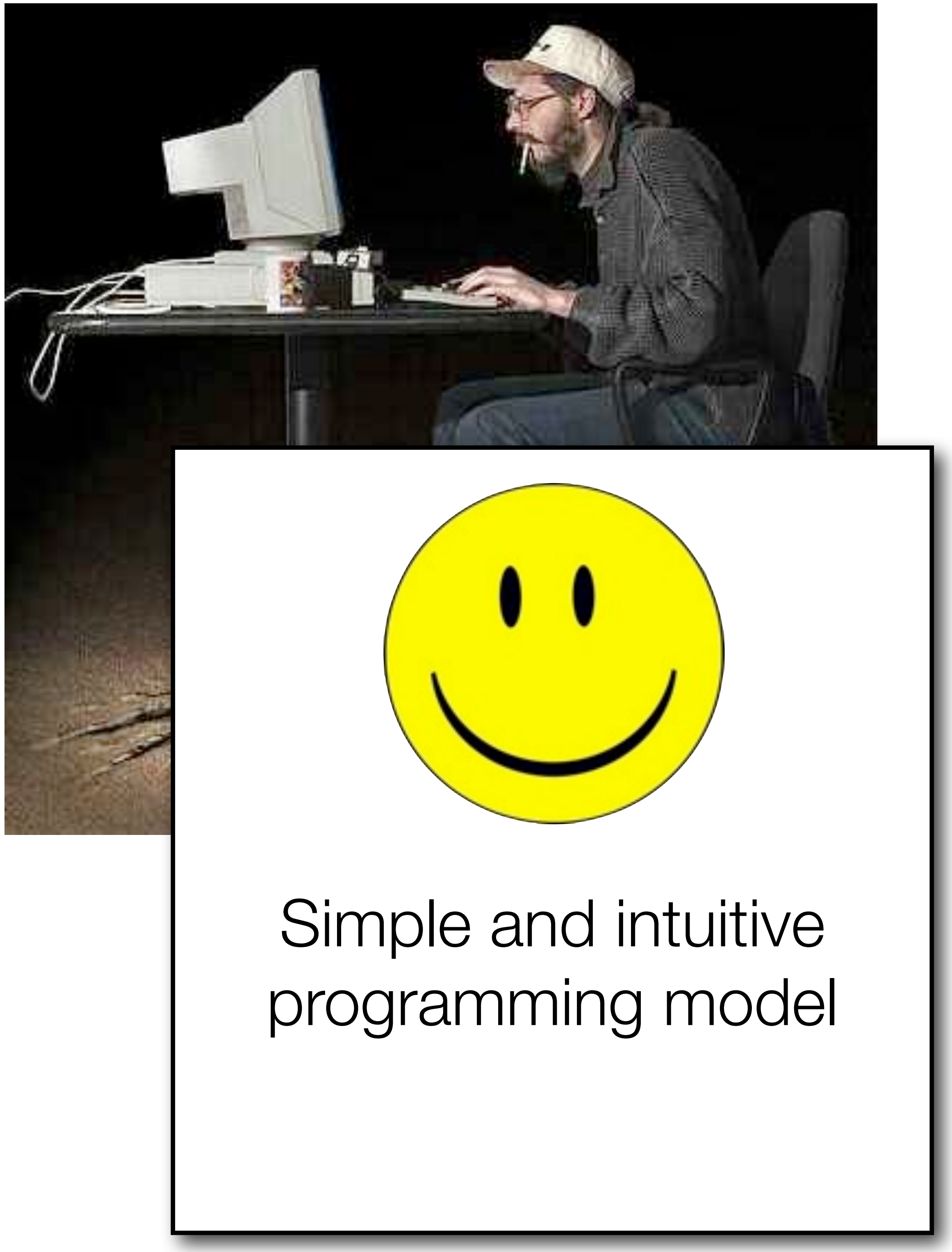


Simple and intuitive programming model

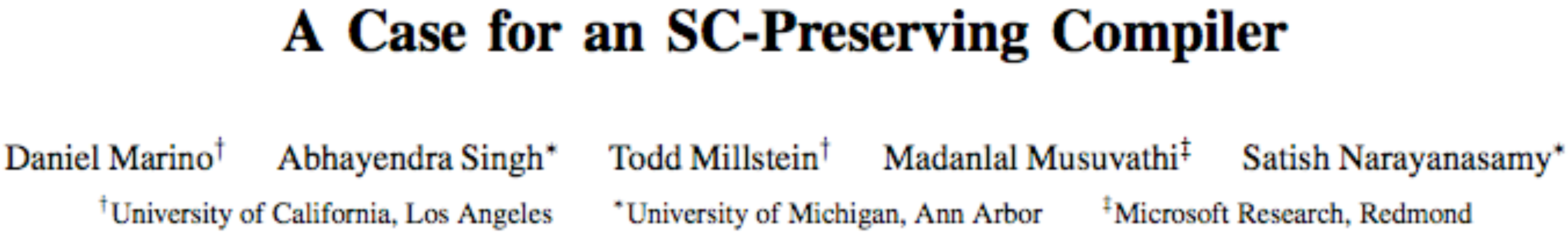
Compilers, programmers & sequential consistency



Expensive
to implement



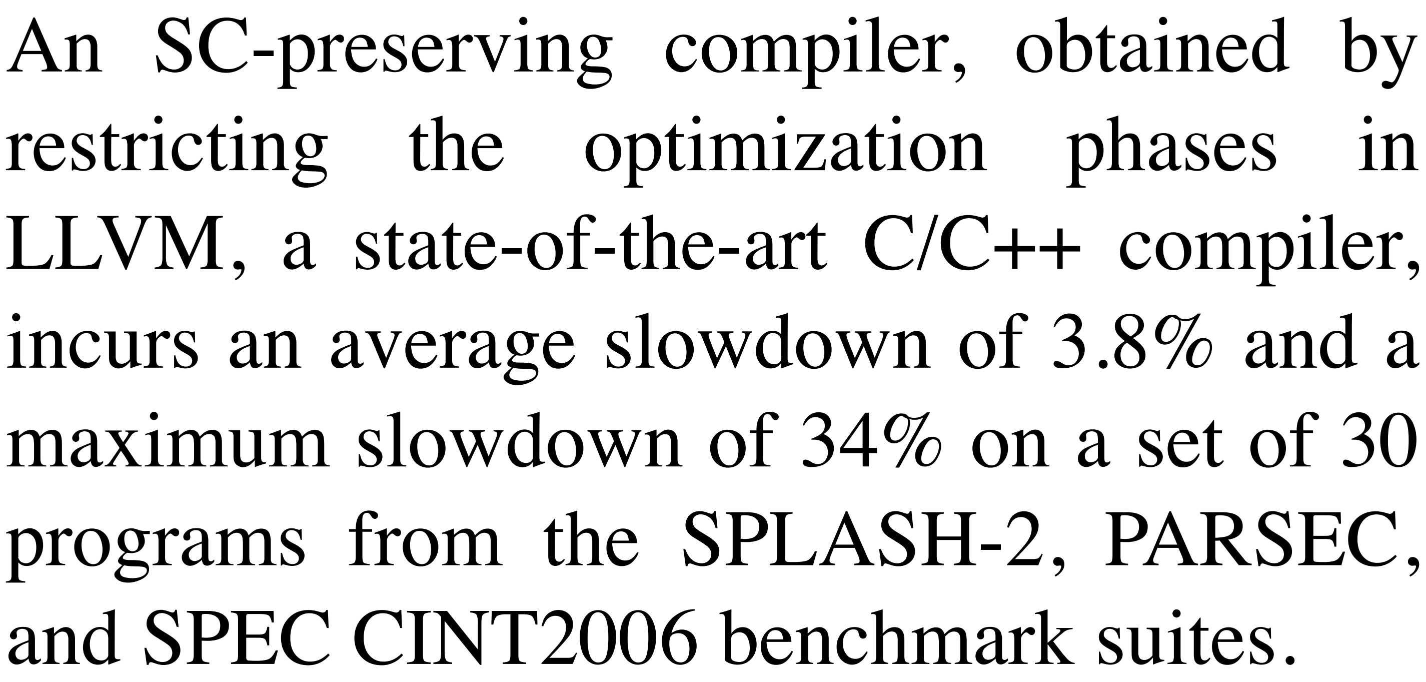
Simple and intuitive
programming model



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

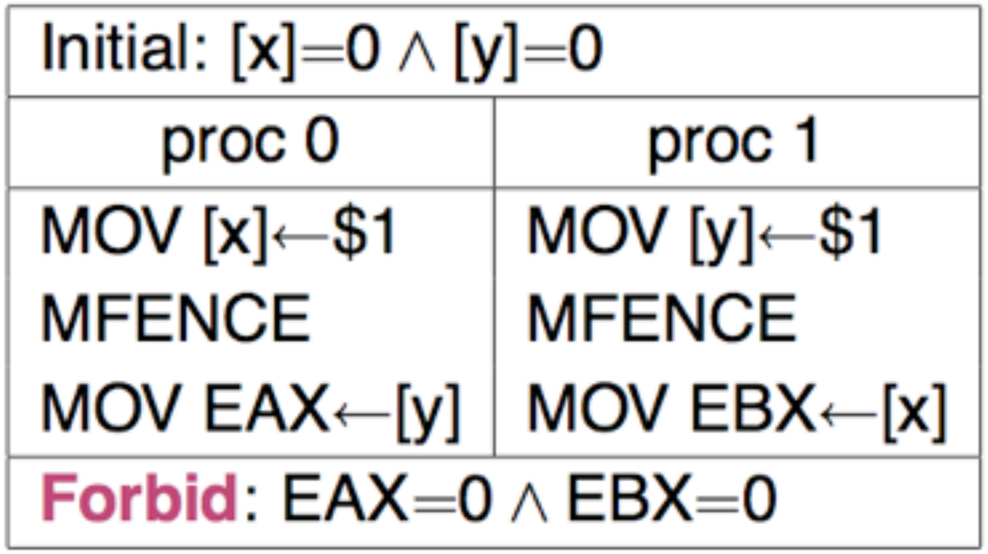
And this study supposes that the hardware is SC.

SC and hardware

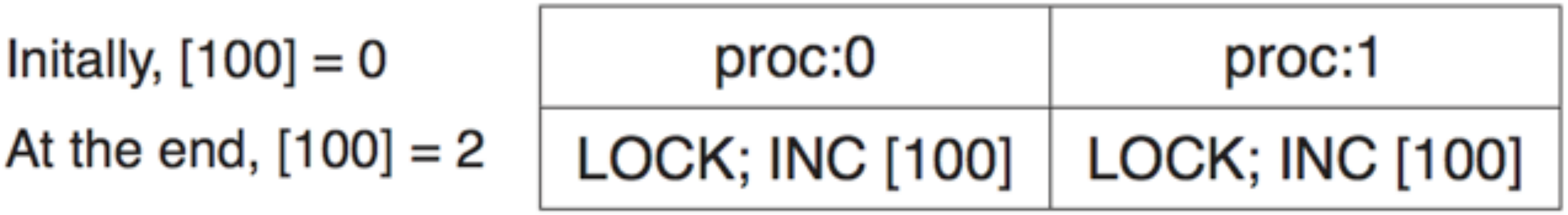
The compiler must insert enough synchronising instructions to prevent hardware reorderings. On x86 we have:

- 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]

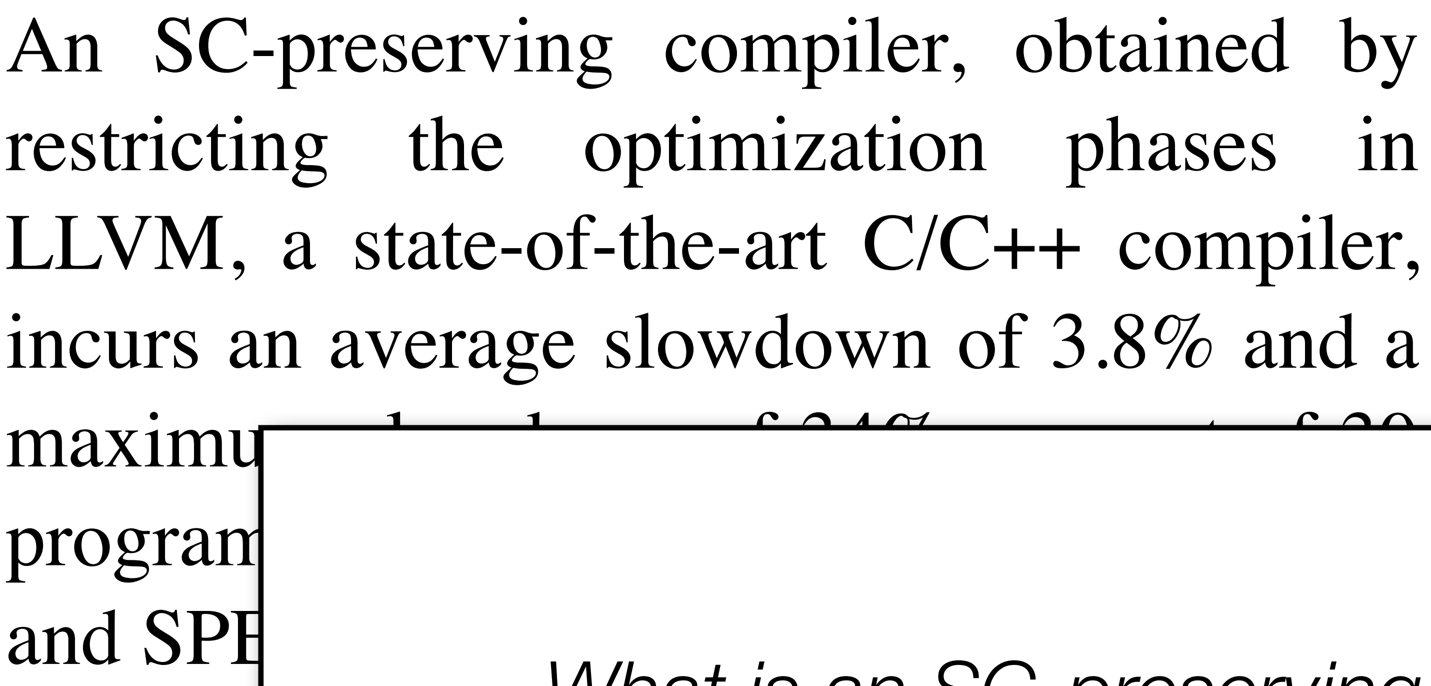
These consumes hundreds of cycles… ideally should be avoided.

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

What is an SC-preserving compiler?

When is a compiler correct?

ensive

plement

And this st

When is a compiler correct?

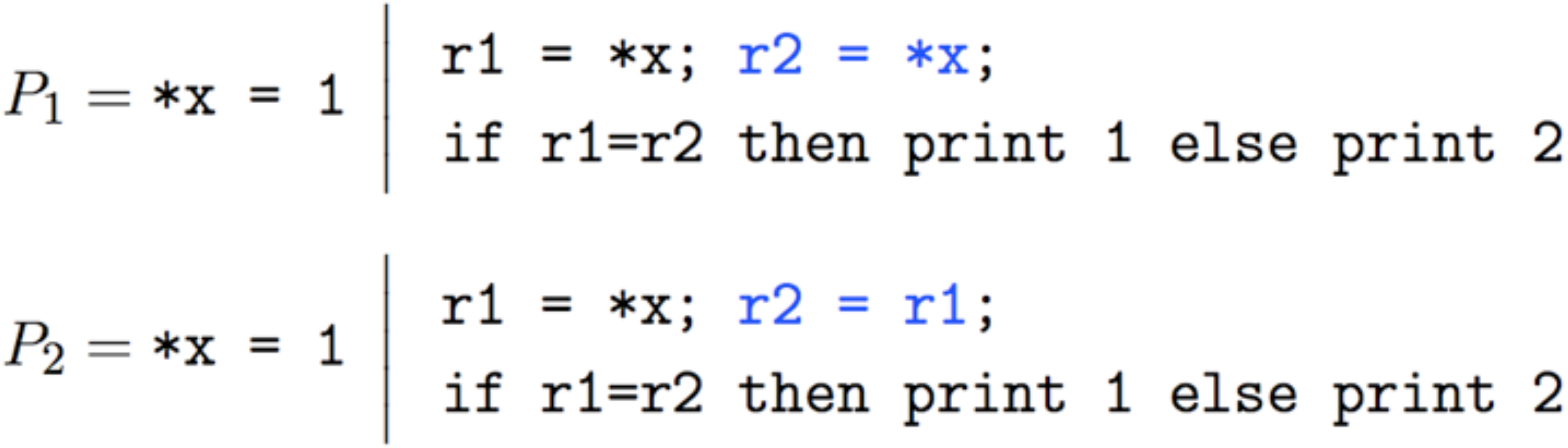
A compiler is correct if any behaviour of the compiled program could be exhibited by the original program.

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

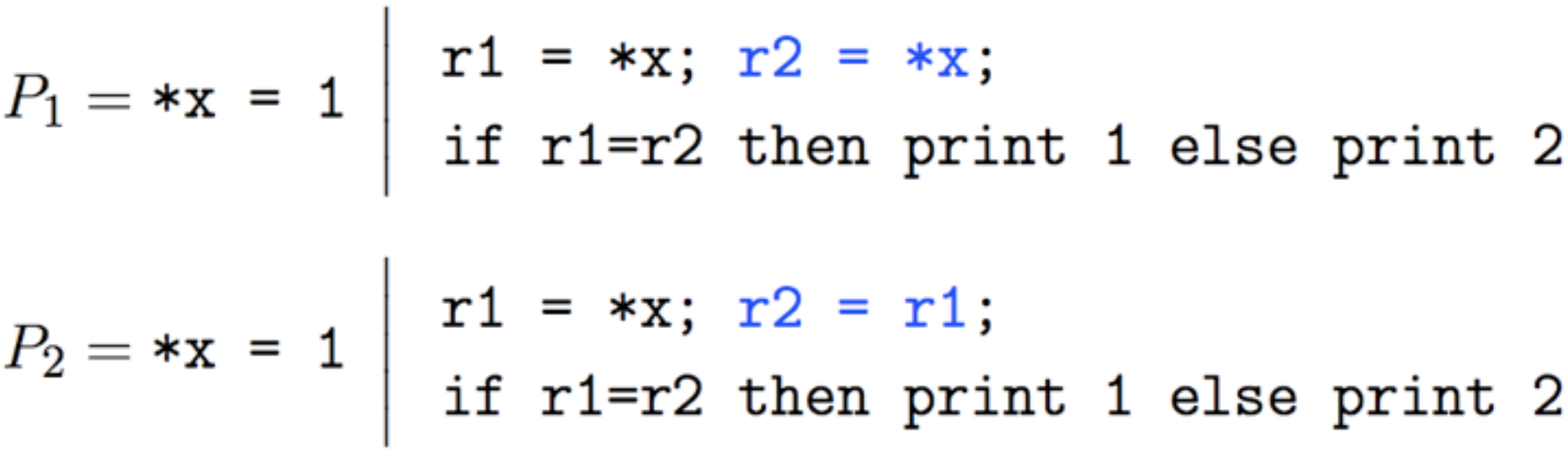


$P_1 =$ `*x = 1` | `r1 = *x; r2 = *x;`
`if r1=r2 then print 1 else print 2`

$P_2 =$ `*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

$$P_1 = \texttt{*x = 1} \;\left|\; \begin{array}{l}\texttt{r1 = *x; r2 = *x;}\\ \texttt{if r1=r2 then print 1 else print 2}\end{array}\right.$$

$$P_2 = \texttt{*x = 1} \;\left|\; \begin{array}{l}\texttt{r1 = *x; r2 = r1;}\\ \texttt{if r1=r2 then print 1 else print 2}\end{array}\right.$$

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:

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

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

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

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

Executions of P2:

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

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

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

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:

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

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

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

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

Executions of P2:

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

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

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

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

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

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

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

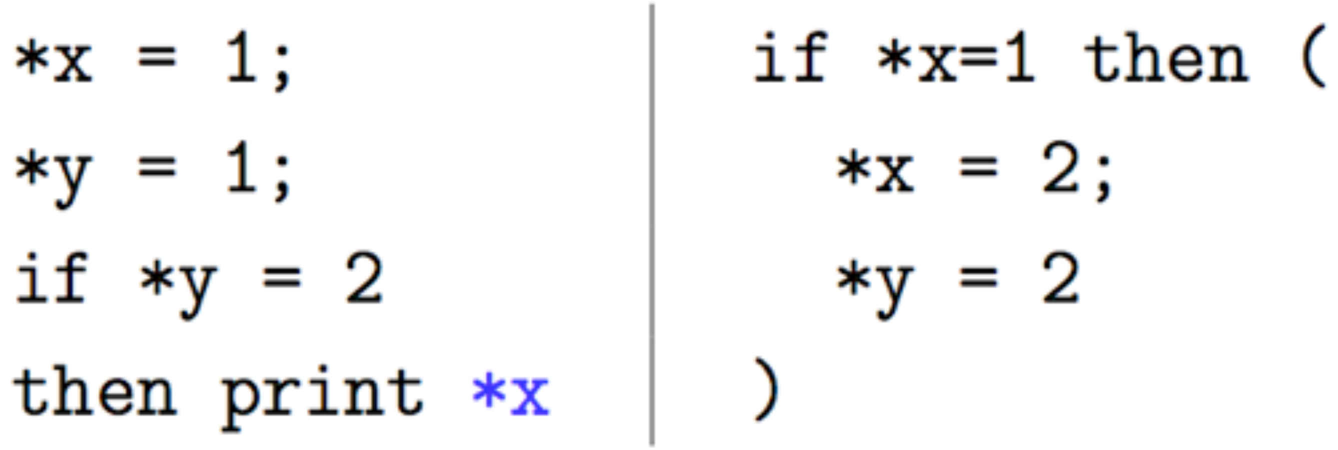
General CSE incorrect in SC

```
*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:

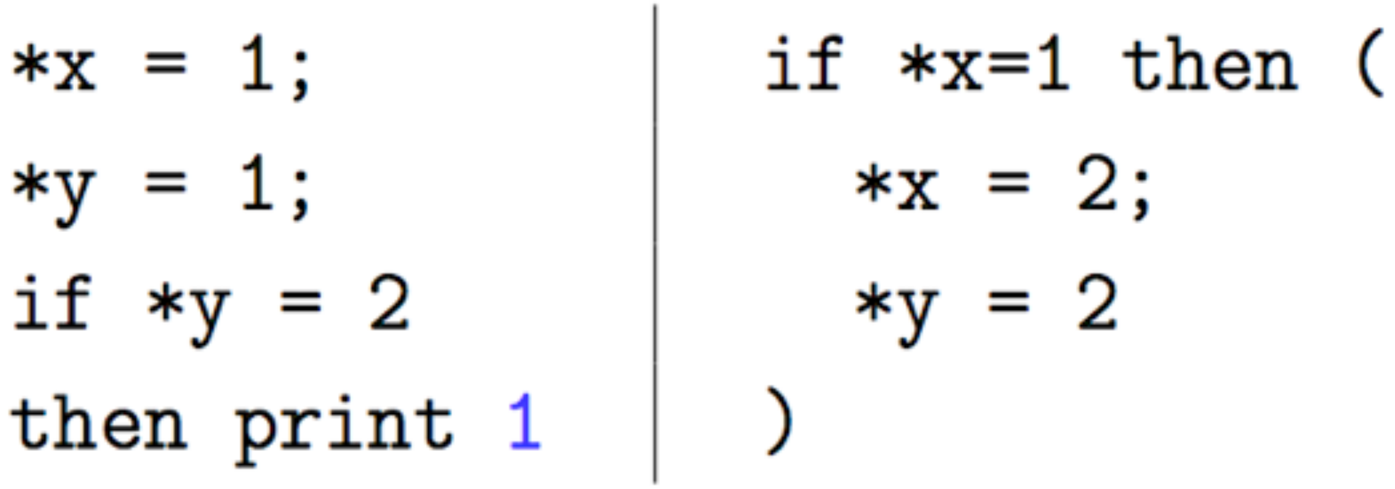
$$\mathsf{W}_{t_1}\ x{=}1, \mathsf{W}_{t_1}\ y{=}1, \mathsf{R}_{t_2}\ x{=}1, \mathsf{W}_{t_2}\ x{=}2, \mathsf{W}_{t_2}\ y{=}2, \mathsf{R}_{t_1}\ y{=}2, \mathsf{R}_{t_1}\ x{=}2, \mathsf{P}_{t_1}\ 2$$

General CSE incorrect in SC



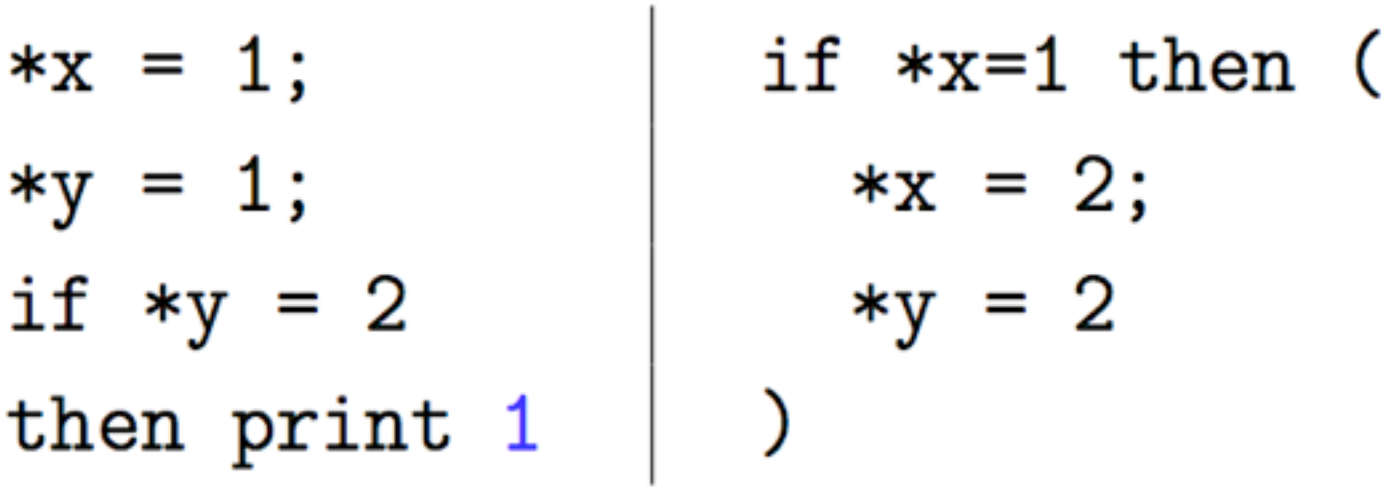
```
*x = 1;           | if *x=1 then (
*y = 1;           |   *x = 2;
if *y = 2         |   *y = 2
then print *x     | )
```

But a compiler would optimise to:



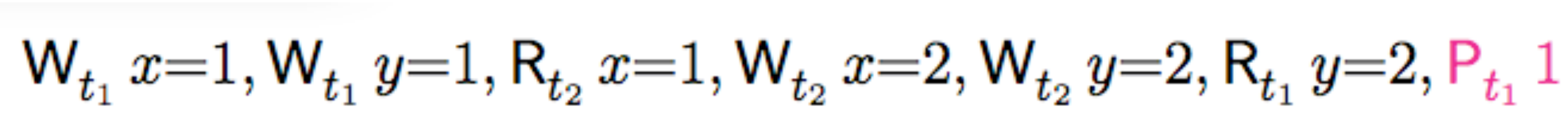
```
*x = 1;           | if *x=1 then (
*y = 1;           |   *x = 2;
if *y = 2         |   *y = 2
then print 1      | )
```

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):

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

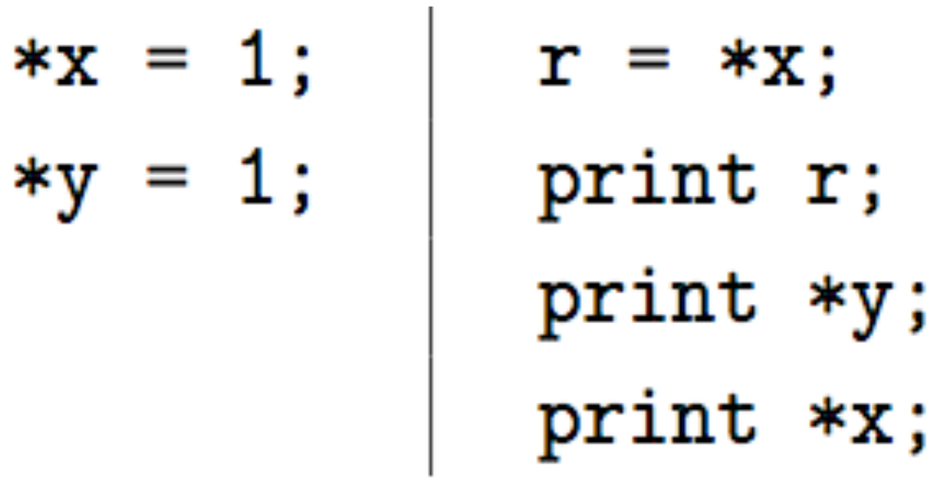
$$[\mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,1]$$

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

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

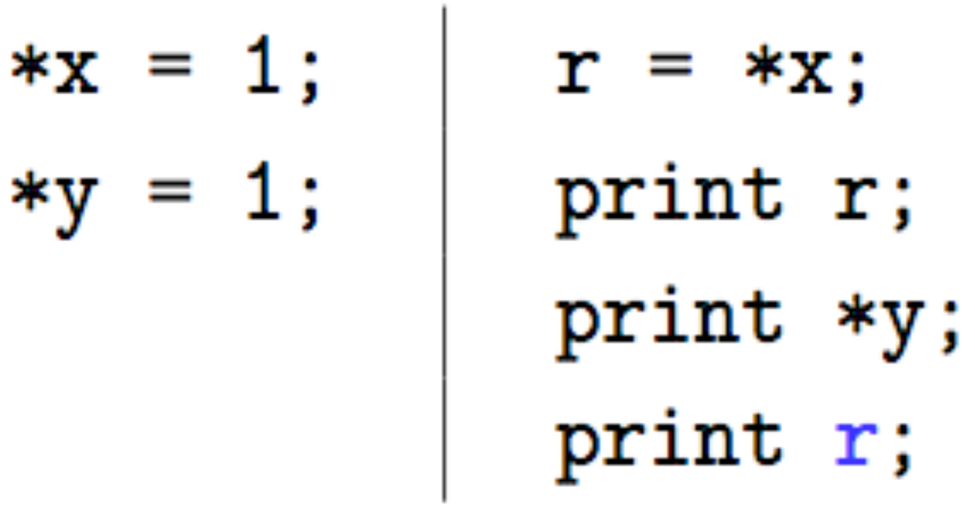
$$[\mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,0]$$

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:

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

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

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

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

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

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

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

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

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

General CSE incorrect in SC

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

```
*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:

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

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

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

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

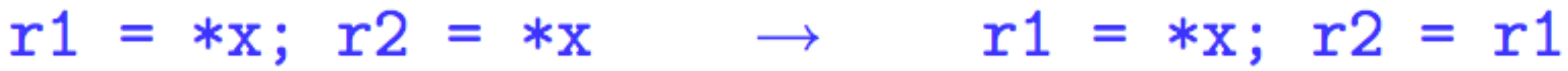
$[\mathsf{P}_{t_1}\, 1, \mathsf{P}_{t_2}\, 0]$

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

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

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

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

Our examples again:

- the probl … our: 0
  (e.g. swa
  thread 0) … rograms;

- the probl … e that
  allows tw … **y in**
  conflicting ways (e.g. one thread writes the datas read by the other).

...intuition...

Programming languages provide

synchronisation mechanisms

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

The basic solution

Thread 0	Thread 1

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

Observable behaviour: 0

Prohibit *data races*

Defined as follows:

- two mem… emory location a…
- a SC exe… flicting **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)

Thread 0	Thread 1

- `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();
```

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

Thread 0	Thread 1

- `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 acro

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

Ans

The ot

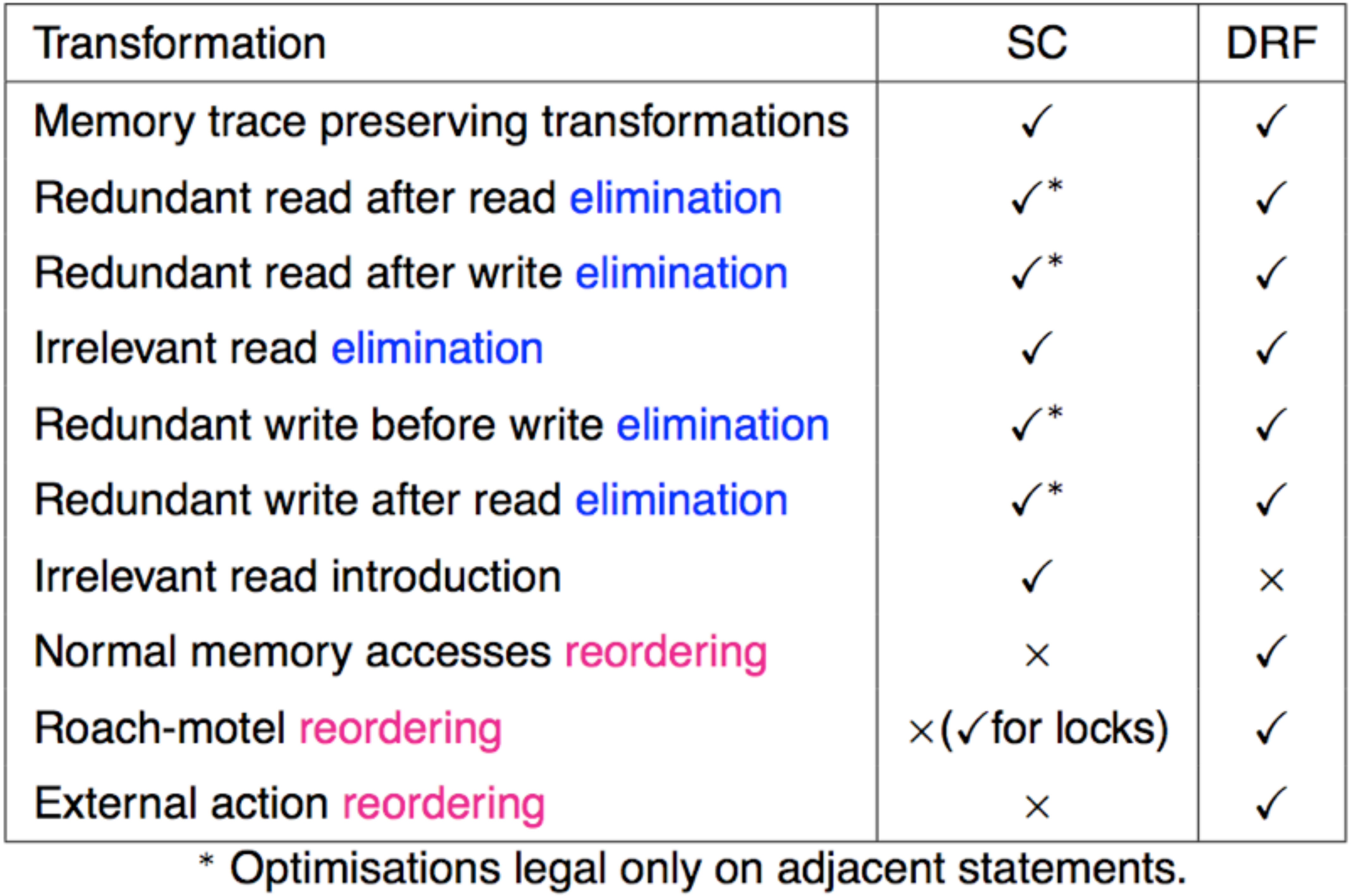
par

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	×	✓

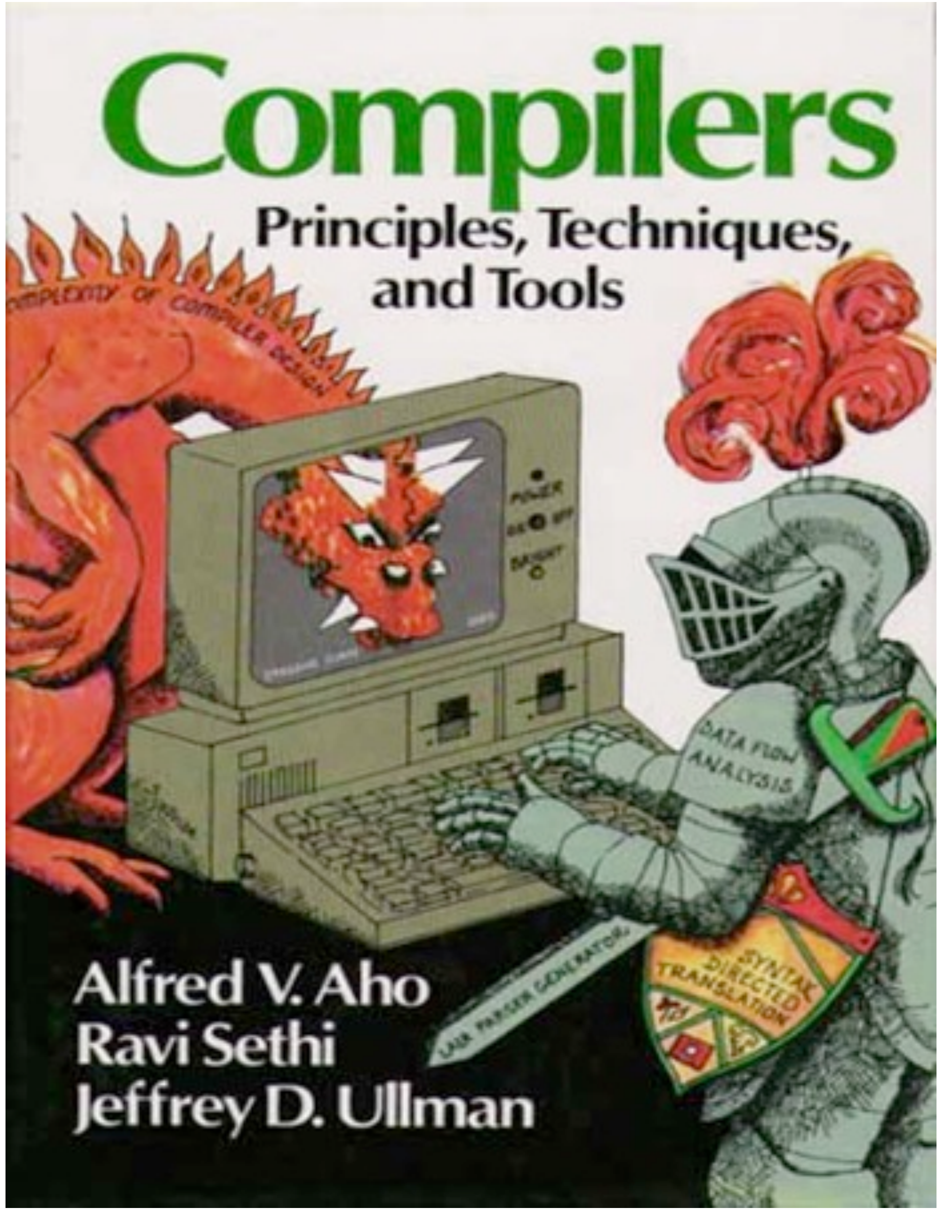
* Optimisations legal only on adjacent statements.

Jaroslav Sevcik

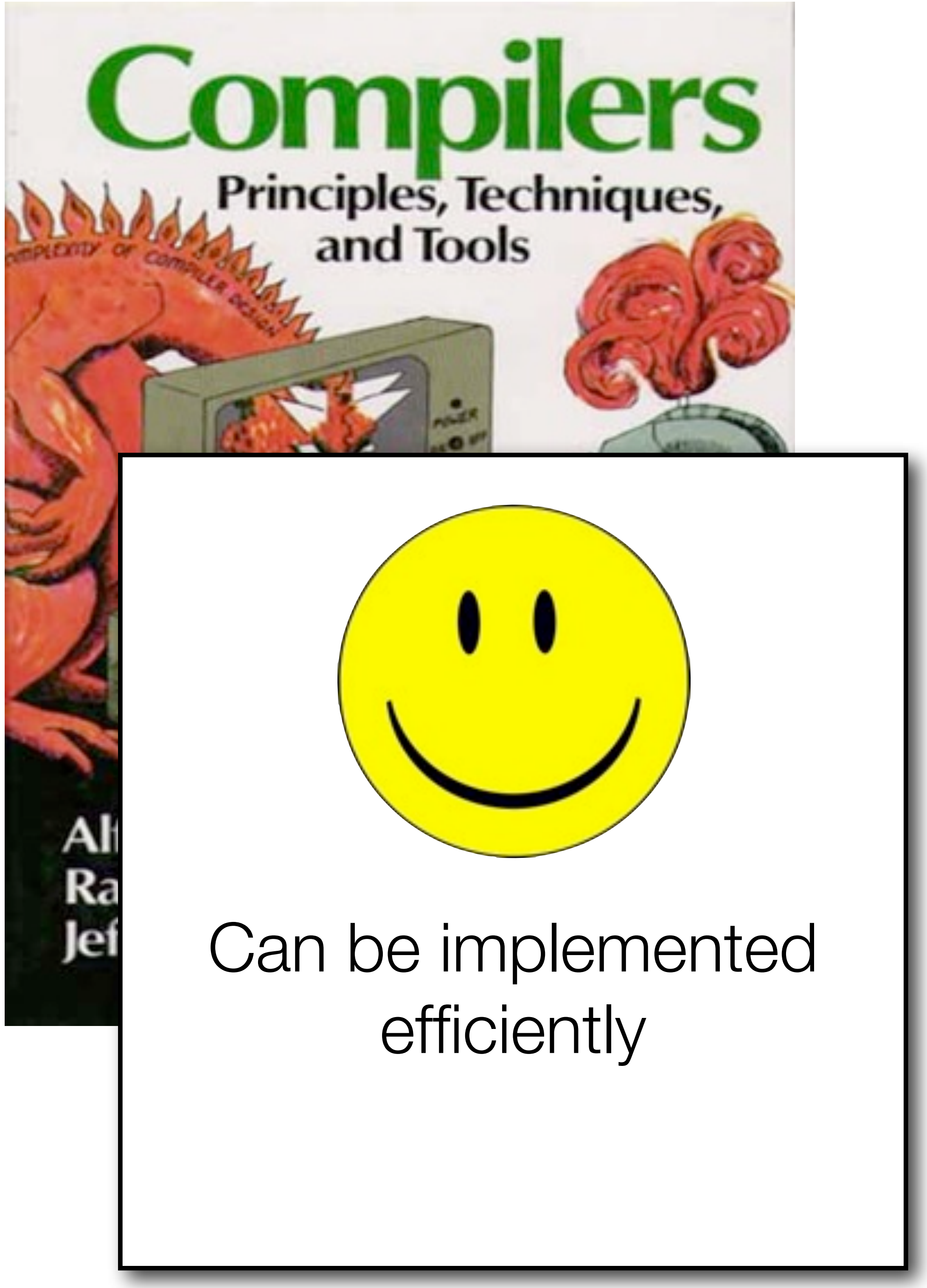
Safe Optimisations for Shared-Memory Concurrent Programs

PLDI 2011

Compilers, programmers & data-race freedom

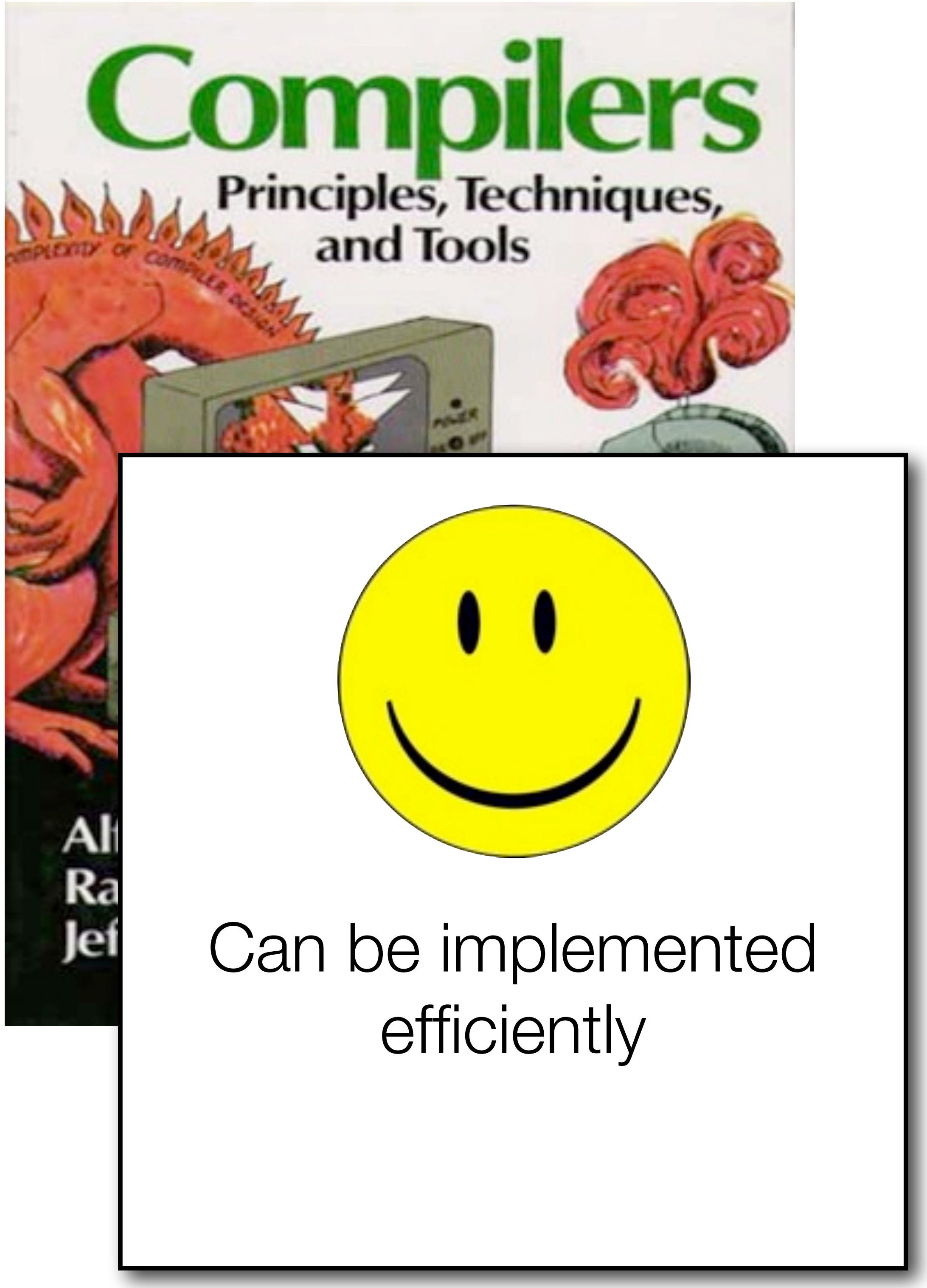

Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
DATA FLOW ANALYSIS
SYNTAX DIRECTED TRANSLATION
LALR PARSER GENERATOR
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

Compilers, programmers & data-race freedom



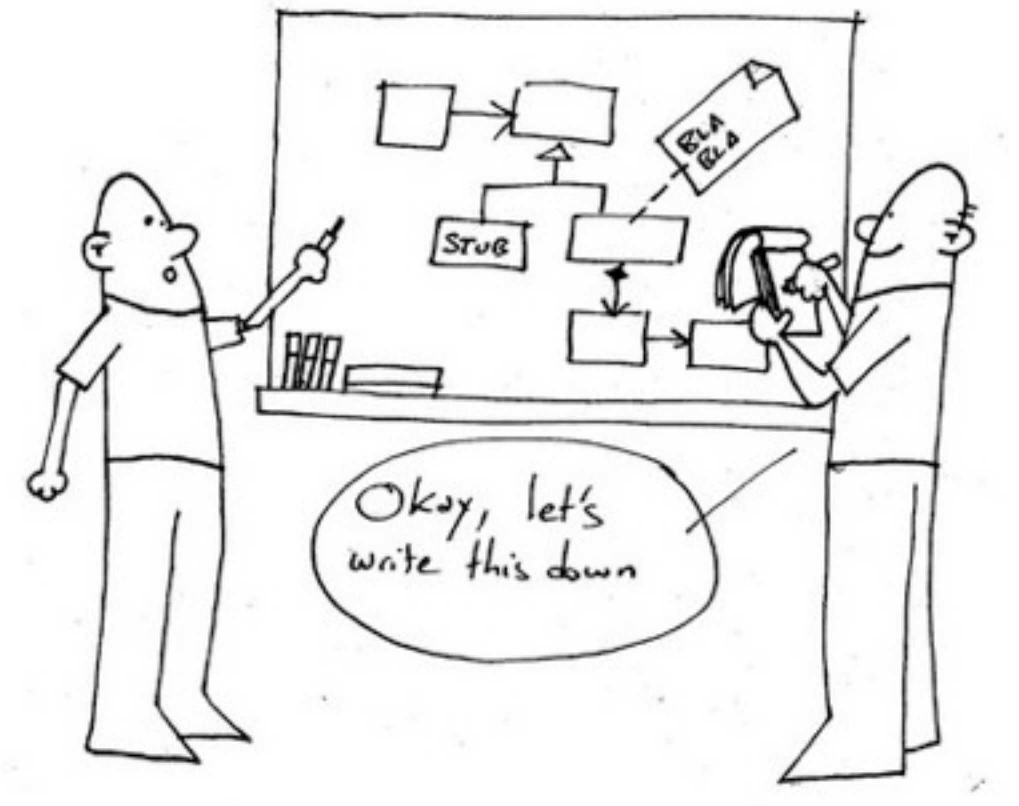
Can be implemented efficiently

Compilers, programmers & data-race freedom


Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
POWER
Al
Ra
Jef

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`	read from memory
|`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

reg

int

thr

> *We work with a toy language, but all this scales to the full Java Memory Model.*

sta

|

|

|r := x read from memory

|lock lock

|unlock unlock

|print r output

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

system ::= concurrent system

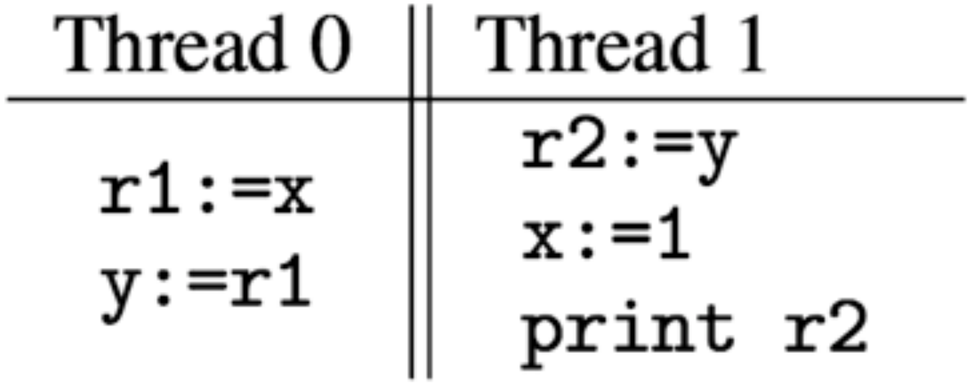
|t_0:p_0 | … | t_n:p_n parallel composition of n threads

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`

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

Tra

De ead

sta

thr

De

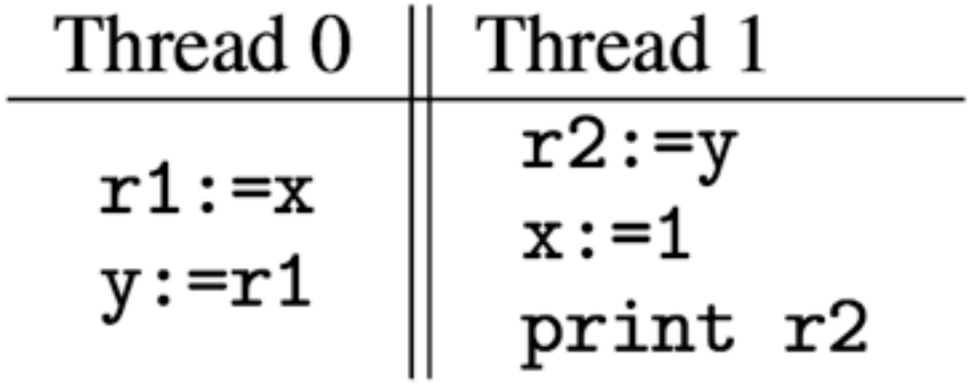
Remarks:

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

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

is also a valid traceset for the example below.

Sample traceset:



Thread 0	Thread 1

r1:=x
y:=r1
	r2:=y
	x:=1
	print r2

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

Associate tracesets to toy language programs

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

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

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

$$< S,\ \texttt{lock; s} > \xrightarrow{\texttt{L}} < S,\ \texttt{s} >$$

$$< S,\ \texttt{unlock; s} > \xrightarrow{\texttt{U}} < S,\ \texttt{s} >$$

$$< S,\ \texttt{print r; s} > \xrightarrow{\texttt{X(S(r))}} < S,\ \texttt{s} >$$

$$< S,\ t_0\texttt{:}p_0 \mid \ldots \mid t_n\texttt{:}p_n > \xrightarrow{\texttt{S(i)}} < S,\ p_i >$$

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* ∈ *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

- *tids*

- loo

- eac).

(The sses).

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

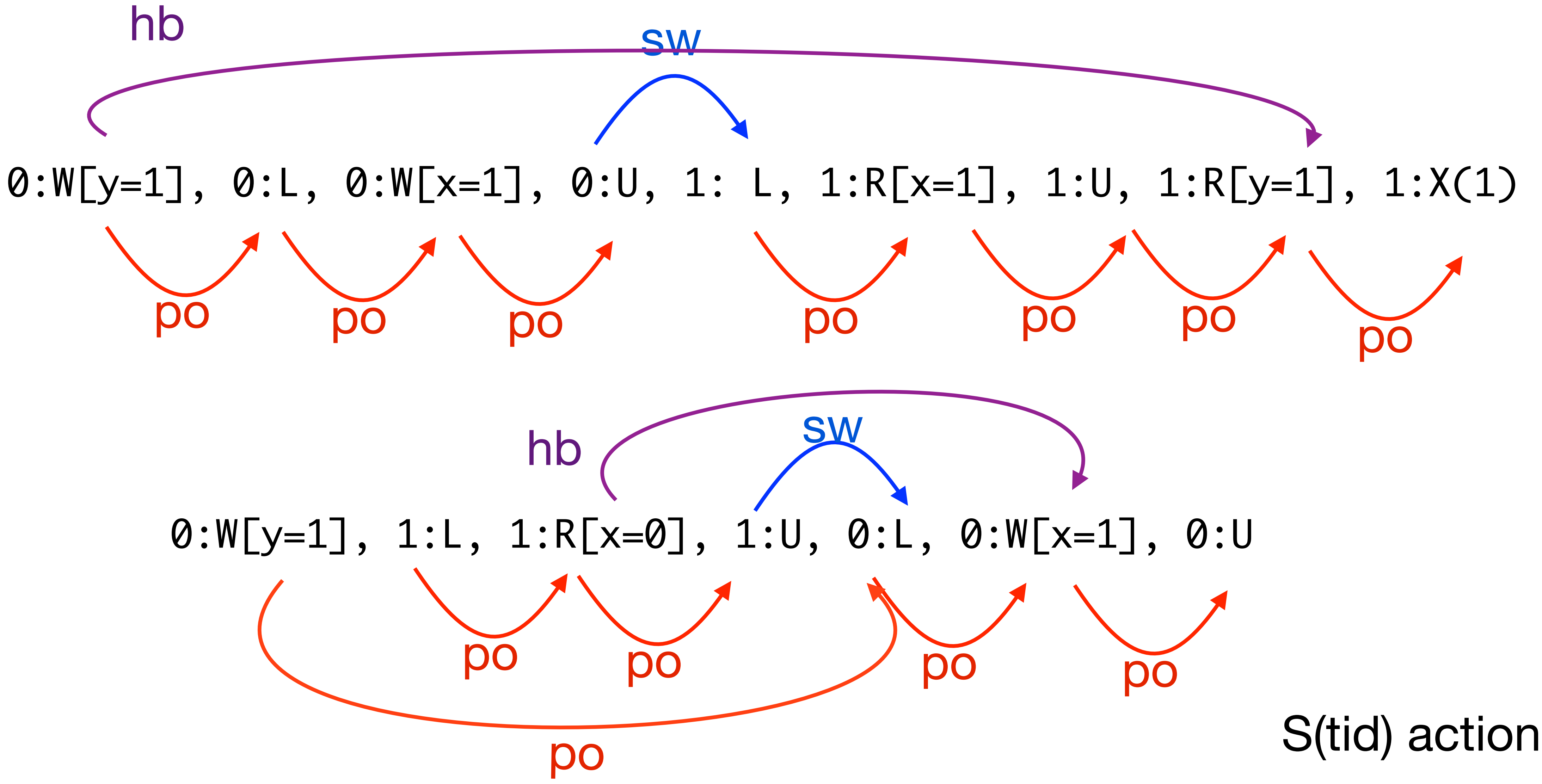
Definition [program order]: **program order**, $<_{po}$, is a total order over the actions of *the same thread in an interleaving.*

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

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

Examples of happens before

Thread 0	Thread 1
`*y = 1`	`lock();`
`lock();`	`tmp = *x;`
`*x = 1`	`unlock();`
`unlock();`	`if tmp = 1`
	` then print *y`



```
0:W[y=1], 0:L, 0:W[x=1], 0:U, 1: L, 1:R[x=1], 1:U, 1:R[y=1], 1:X(1)
```

```
0:W[y=1], 1:L, 1:R[x=0], 1:U, 0:L, 0:W[x=1], 0:U
```

S(tid) actions omitted.

Data-race freedom

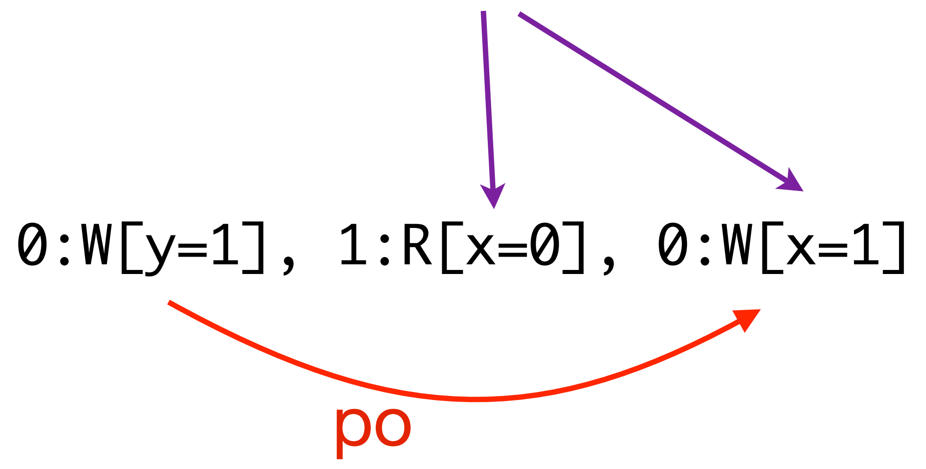
Definition [data-race-freedom]: A traceset is **data-race free** if none of its executions has two adjacent conflicting actions from different threads.

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



Two conflicting accesses
not related by happens before.

0:W[y=1], 1:R[x=0], 0:W[x=1]

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

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

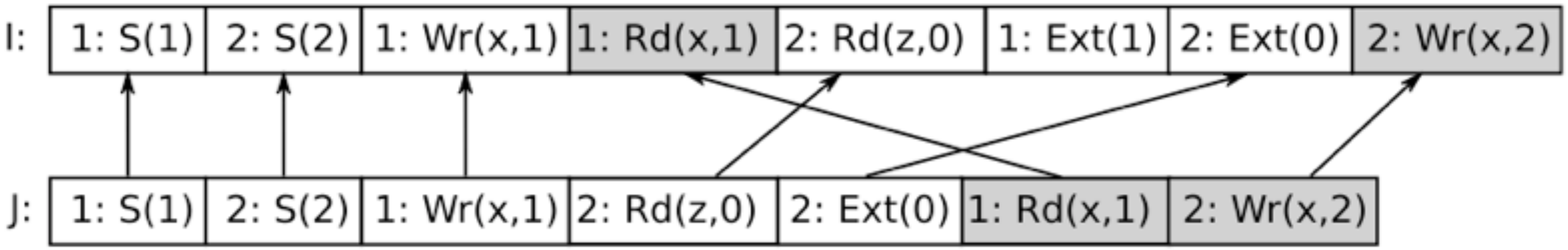
where β' are all the actions from β that strctly happen-before [b].

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.

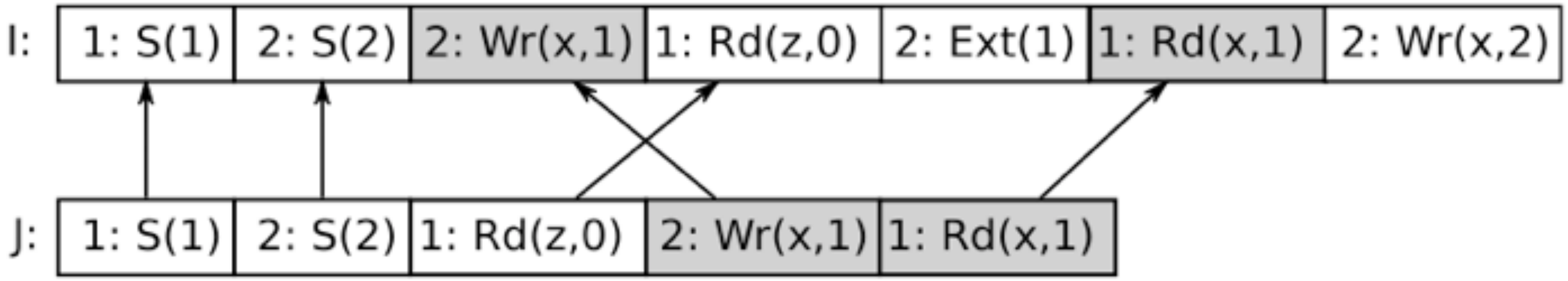
Data-race freedom: equivalence of definitions

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



I:	1: S(1)	2: S(2)	1: Wr(x,1)	1: Rd(x,1)	2: Rd(z,0)	1: Ext(1)	2: Ext(0)	2: Wr(x,2)
J:	1: S(1)	2: S(2)	1: Wr(x,1)	2: Rd(z,0)	2: Ext(0)	1: Rd(x,1)	2: Wr(x,2)	

read first



I:	1: S(1)	2: S(2)	2: Wr(x,1)	1: Rd(z,0)	2: Ext(1)	1: Rd(x,1)	2: Wr(x,2)
J:	1: S(1)	2: S(2)	1: Rd(z,0)	2: Wr(x,1)	1: Rd(x,1)		

write first

$$\begin{aligned}
x &= (\pi + 3)/2 \\
2x &= \pi + 3 \\
2x(\pi - 3) &= (\pi + 3)(\pi - 3) \\
2\pi x - 6x &= \pi^2 - 9 \\
9 - 6x &= \pi^2 - 2\pi x \\
9 - 6x + x^2 &= \pi^2 - 2\pi x + x^2 \\
(3 - x)^2 &= (\pi - x)^2 \\
3 - x &= \pi - x \\
\pi &= 3
\end{aligned}$$

(Sketch of)

proof of correctness of redundant read removal

Redundant read after read

Given a trace ***t*** we say that index $i \in \mathrm{dom}(\boldsymbol{t})$ is a *redundant read after read* if $\boldsymbol{t}_i = \boldsymbol{t}_j =$ `R[l=v]` for some `v`, `l`, and $j < i$.

Given traces ***t*** and ***t'***, the trace ***t'*** is an *elimination* of ***t*** if there is a set of indexes $S \in \mathrm{dom}(T)$ such that $\boldsymbol{t}' = \boldsymbol{t}_{|S}$ and all $i \in \mathrm{dom}(\boldsymbol{t}) \setminus S$ are redundant reads after read.

Example: [R[x=3], X(3), R[x=3], X(3)] ⇒ [R[x=3], X(3), X(3)]

Elimination lifts to tracesets: a traceset *T'* is an elimination of a traceset *T* if each trace ***t'*** in *T'* is an elimination of some trace in *T*.

Exercice: compute the traceset *T* of `r1 = x; print x; r2 = x; print x` and find a traceset which is an elimination of *T*.

Exercice

Compute the traceset *T* of

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

and find a traceset which is an elimination of *T*.

Exercice

Compute the traceset *T* of

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

and find a traceset which is an elimination of *T*.

Answer: let T be the prefix closure of

{ [S(1), R[x=v_1] , X(v_1), R[x=v_2], X(v_2)] | forall v_1,v_2 } U { [S(2), W[x=2] }

and let T' be (the prefix closure of) the following elimination of T:

{ [S(1), R[x=v_1] , X(v_1), X(v_1)] | forall v_1 } U { [S(2), W[x=2] }

Safety of redundant read after read

We show that:

1) any execution of the transformed traceset *has the same behaviour* as some execution of the original traceset, provided that the original program was data race free;

2) the transformations preserve data race freedom.

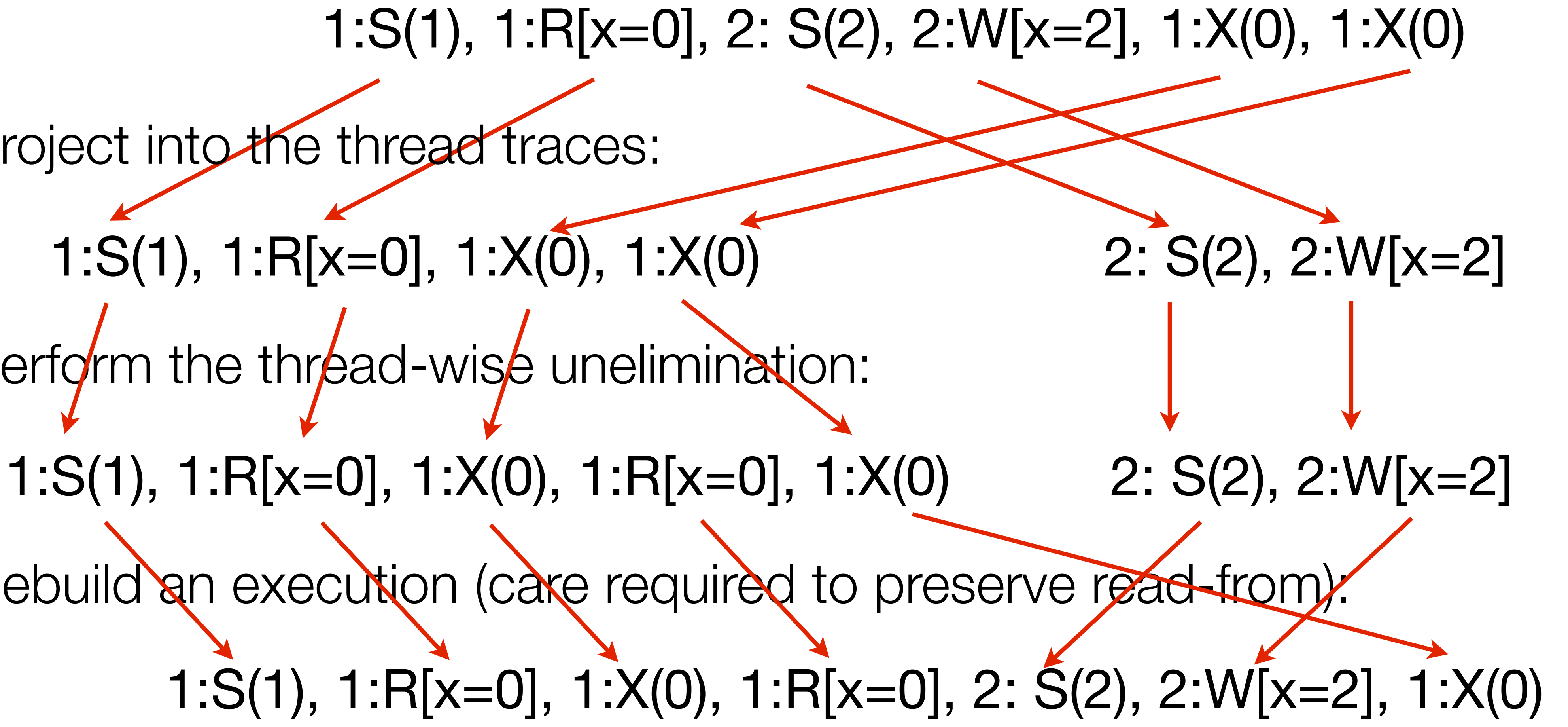
We take an arbitrary execution of the transformed program, and construct an execution of the original program that has the same behaviour. We decompose the execution of the transformed program into traces for each thread, we compute untrasformed traces of the original traceset, and then compose the untrasformed traces back into an untrasformed execution that preserve the behaviour. Now we must prove that either the constructed interleaving is an execution of the original traceset, or there must have been a data race.

Also prove that the transformed program is drf, as the happens-before orders is somewhat preserved.

Consider an arbitrary interleaving of the transformed traceset

{ [S(1), R[x=v_1] , X(v_1), X(v_1)] | forall v_1 } U { [S(2), W[x=2] }

for instance,



1:S(1), 1:R[x=0], 2: S(2), 2:W[x=2], 1:X(0), 1:X(0)

Project into the thread traces:

1:S(1), 1:R[x=0], 1:X(0), 1:X(0)   2: S(2), 2:W[x=2]

Perform the thread-wise unelimination:

1:S(1), 1:R[x=0], 1:X(0), 1:R[x=0], 1:X(0)   2: S(2), 2:W[x=2]

Rebuild an execution (care required to preserve read-from):

1:S(1), 1:R[x=0], 1:X(0), 1:R[x=0], 2: S(2), 2:W[x=2], 1:X(0)

that is an interleaving of the original traceset:

{ [S(1), R[x=v_1] , X(v_1), R[x=v_2], X(v_2)] | forall v_1,v_2 } U { [S(2), W[x=2] }

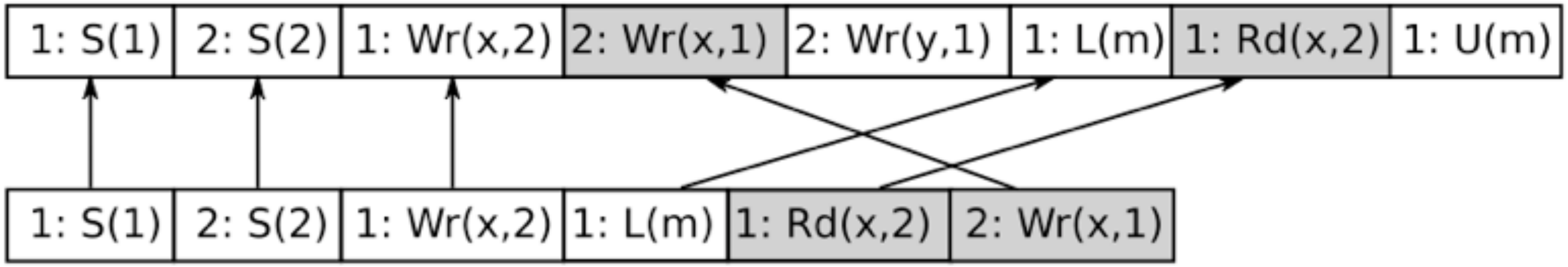
Rebuild an execution (care required to preserve read-from):

1:S(1), 1:R[x=0], 1:X(0), 1:R[x=0], 1:X(0)   2: S(2), 2:W[x=2]

that is an interleaving of the original traceset:

1:S(1), 1:R[x=0], 1:X(0), 1:R[x=0], 2: S(2), 2:W[x=2], 1:X(0)

Key property: it is always possible to rebuild the execution, reasoning as we have done to prove the equivalence of the two data race free formulations,e.g.:



1: S(1) | 2: S(2) | 1: Wr(x,2) | 2: Wr(x,1) | 2: Wr(y,1) | 1: L(m) | 1: Rd(x,2) | 1: U(m)

1: S(1) | 2: S(2) | 1: Wr(x,2) | 1: L(m) | 1: Rd(x,2) | 2: Wr(x,1)

Defining programming language memory models

Option 1

Don't.

No concurrency.

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 (e.g. by ensuring data accesses protected by associated locks, or fancy effect type systems), but likely to be inflexible.

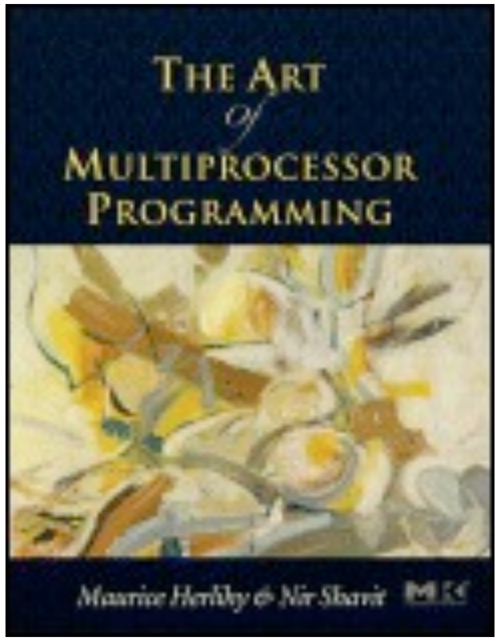
Option 3

Don't.

But language ensures data-race freedom

Possible (e.g. by ensuring data accesses protected by associated locks, or fancy effect type systems), but likely to be inflexible.

What about these fancy racy algorithms?



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 will exhibit the underlying x86-tso behaviour (at least they are atomic).

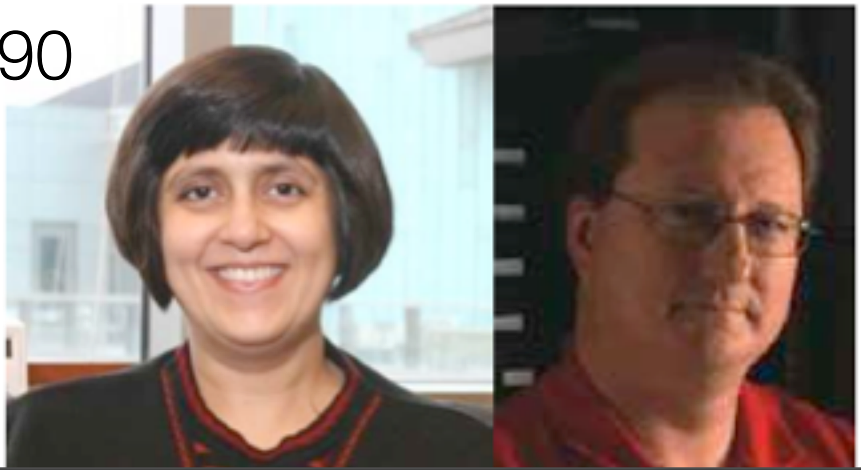
Option 5

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

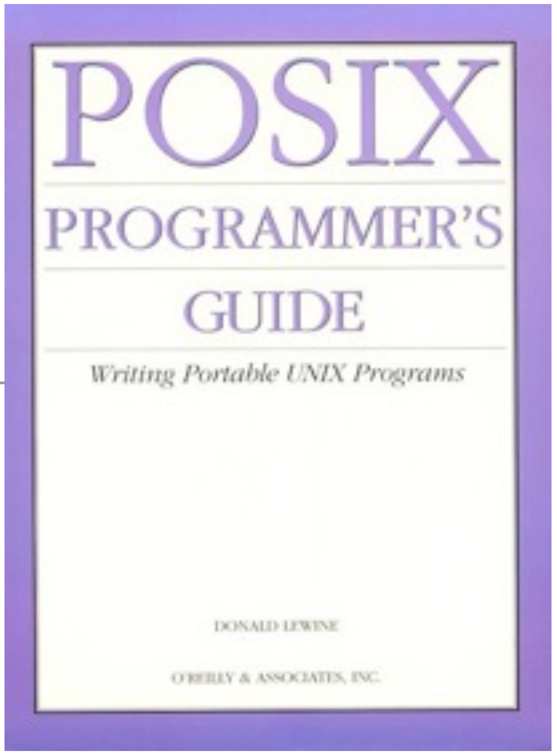
Pro:

- 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?)

Data race freedom as a definition



- Posix is sort-of DRF

> 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. Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads.

Single Unix SPEC V3 & others

Data race freedom as a definition

- Core of the C++11 standard.

Hans Boehm & Sarita Adve, PLDI 2008.

- Part of the JSR-133 standard.

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

Isn't this all obvious?

Isn't this all obvious?

Perhaps it should have been.

Isn't this all obvious?

Perhaps it should have been.

But a few things **went wrong** in the past...

1. Uncertainity about details

Initially `x = y = 0`

```
r1 := [x];            r2 := [y];
if (r1=1)       ||    if (r2=1)
    [y] := 1              [x] := 1
```

Is the outcome `r1=r2=1` allowed?

1. Uncertainity about details

Initially `x = y = 0`

```
r1 := [x];               r2 := [y];
if (r1=1)          ||    if (r2=1)
    [y] := 1                 [x] := 1
```

Is the outcome `r1=r2=1` allowed?

- If the threads *speculate* that the values of `x` and `y` are `1`, then each thread writes `1`, validating the other thread speculation;
- such execution has a data race on `x` and `y`;
- however programmers would not envisage such execution when they check if their program is data-race free…

2. Compiler transformations introduce data races

```
struct s
  { char a; char b; } x;
```

Thread 1:

```
x.a = 1;
```

Thread 2:

```
x.b = 1;
```

FORBIDDEN

Thread 1 is not equivalent to:

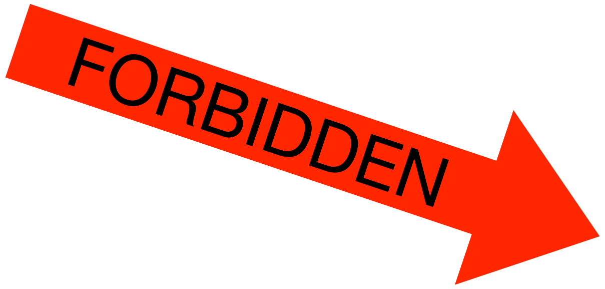
```
struct s tmp = x;
tmp.a = 1;
x = tmp;
```

- Many compilers perform transformations similar to the one above when `a` is declared as a bit field;
- May be visible to client code since the update to `x.b` by T2 may be overwritten by the store to the complete structure `x`.

And many more interesting examples...

2b. Compiler transformations introduce data races

```
for (i = 1; i < N; ++i)
  if (a[i] != 1) a[i] = 2;
```



FORBIDDEN

```
for (i = 1; i < N; ++i)
  a[i] = ((a[i] != 1)? 2 : a[i]);
```

- The vectorisation above might introduce races, but
- most compilers do things along these lines (introduce speculative stores).

3. "escape" mechanisms

Some frequently used idioms (atomic counters, flags, …) do not require sequentially consistency.

Programmers wants optimal implementations of these idioms.

Speed, much more than safety, makes programmers happier.

Data race freedom as a definition

- Core of the 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;

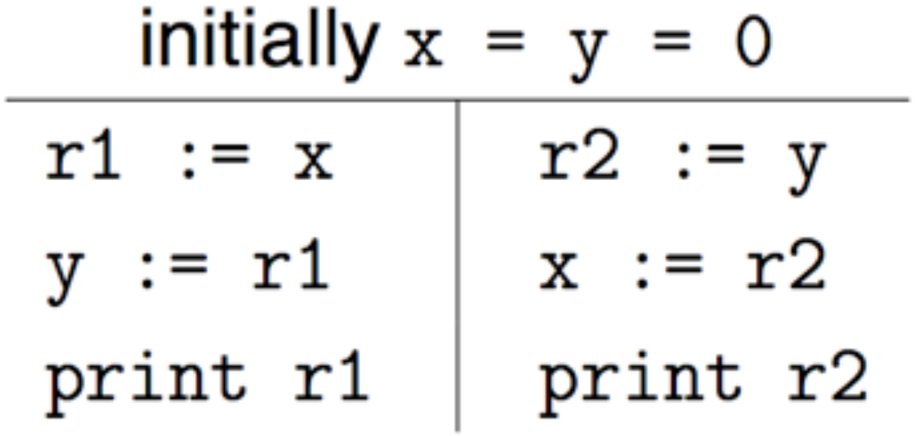
Goal 2: all programs satisfy some memory safety requirements;

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:



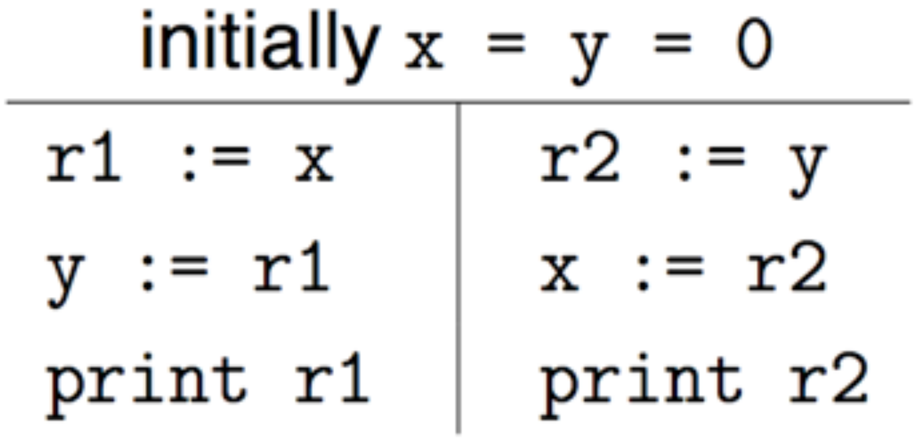
```
      initially x = y = 0
r1 := x          | r2 := y
y := r1          | x := r2
print r1         | print r2
```

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:



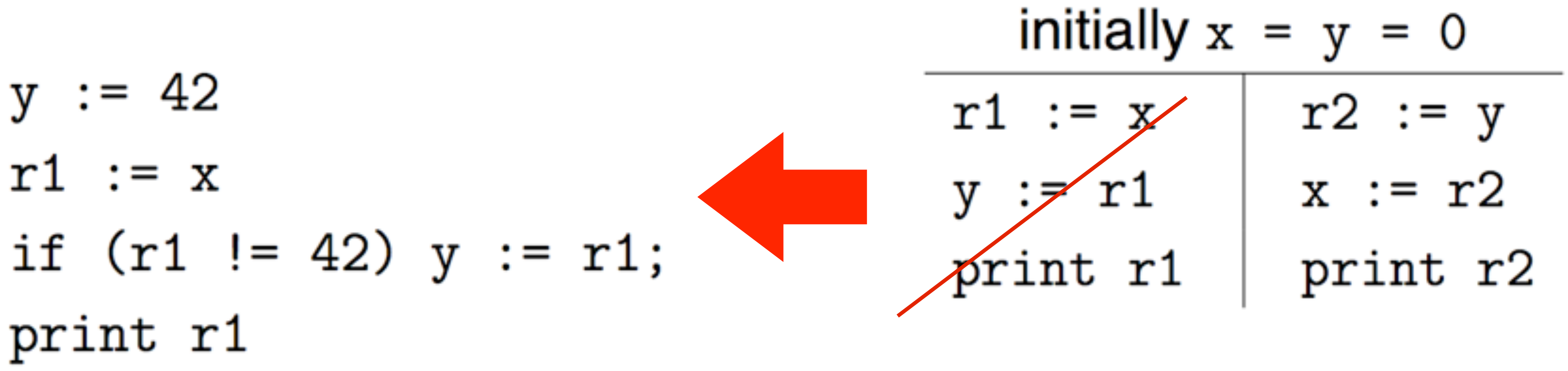
initially x = y = 0

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

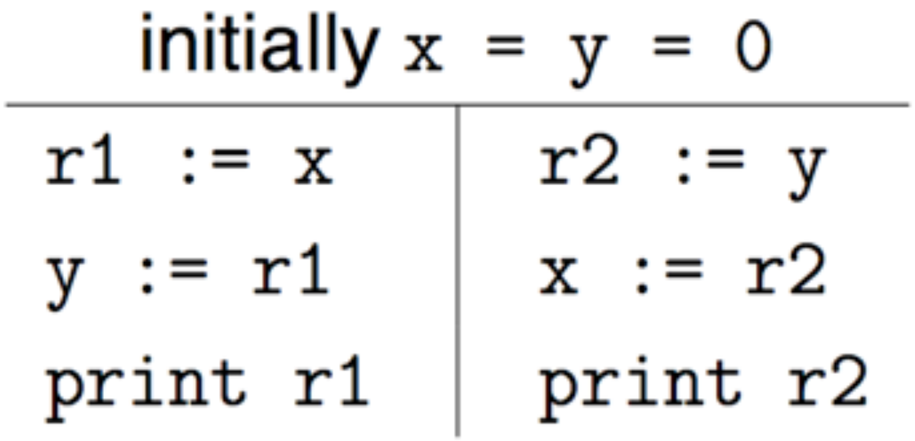
~~r1 := x~~
~~y := r1~~
~~print r1~~
	r2 := y
	x := r2
	print r2

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

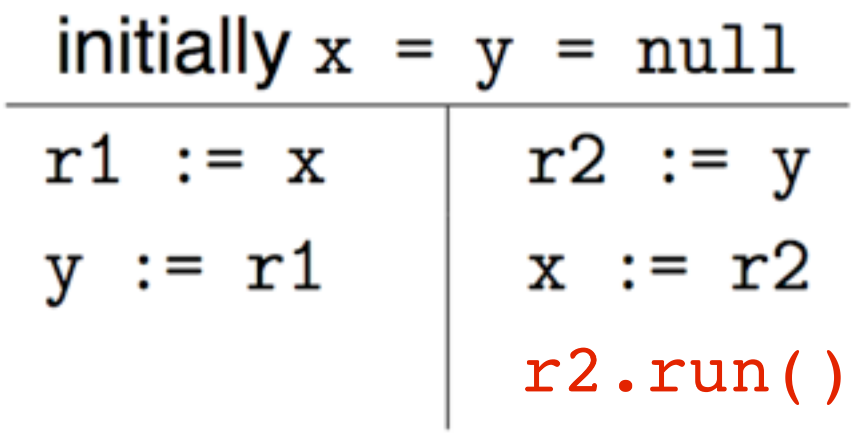
Out-of-thin-air is not so bening for references. Compare:



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   | r2 = y
y = r1   | x=(r2==1)?y:1
         | print r2
```

HotSpot Optimization →

```
        x = y = 0
-------------------------
r1 = x   | x = 1
y = r1   | r2 = y
         | print r2
```

Jaroslav Ševčík, David Aspinall, ECOOP 2008

A word on 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

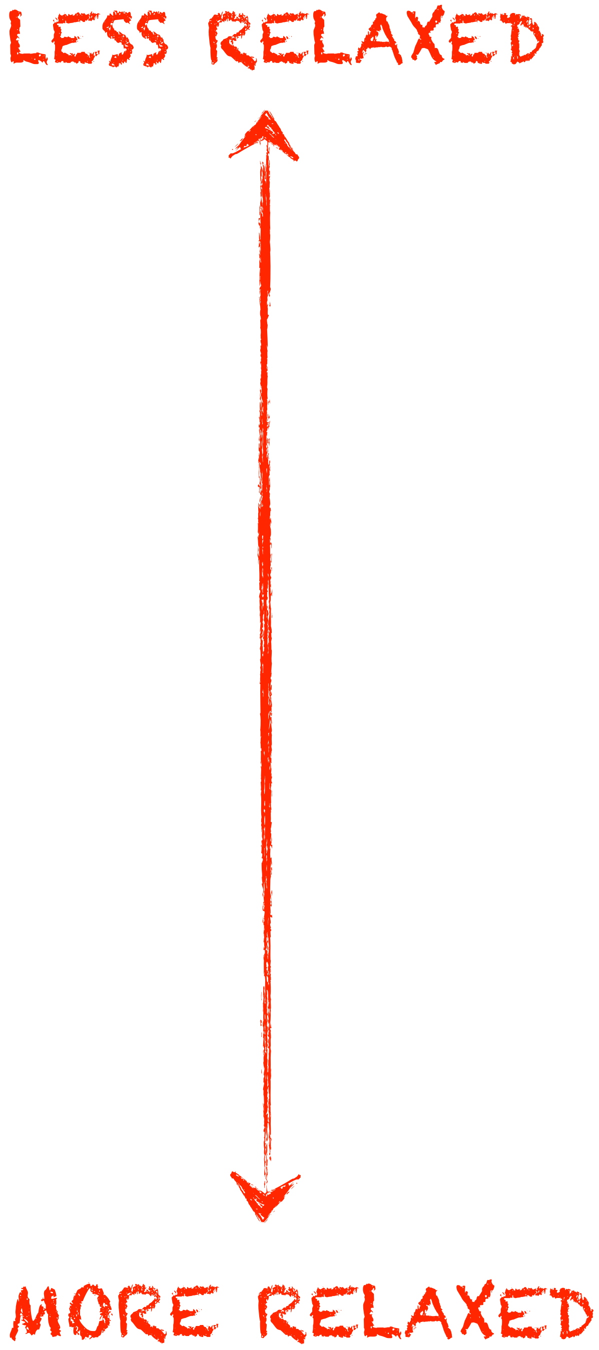
Low-level atomics

`MO_SEQ_CST`

`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

`MO_RELAXED`



LESS RELAXED

MORE RELAXED

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)`
`r1 = y.load(MO_SEQ_CST)`
	`y.store(1,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_RELEASE / MO_ACQUIRE

Supports a fast implementation of the message passing idiom:

Thread 0	Thread 1

`x.store(1,MO_RELAXED)`
`y.store(1,MO_RELEASE)`
	`r1 = y.load(MO_ACQUIRE)`
	`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 = (*x).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:

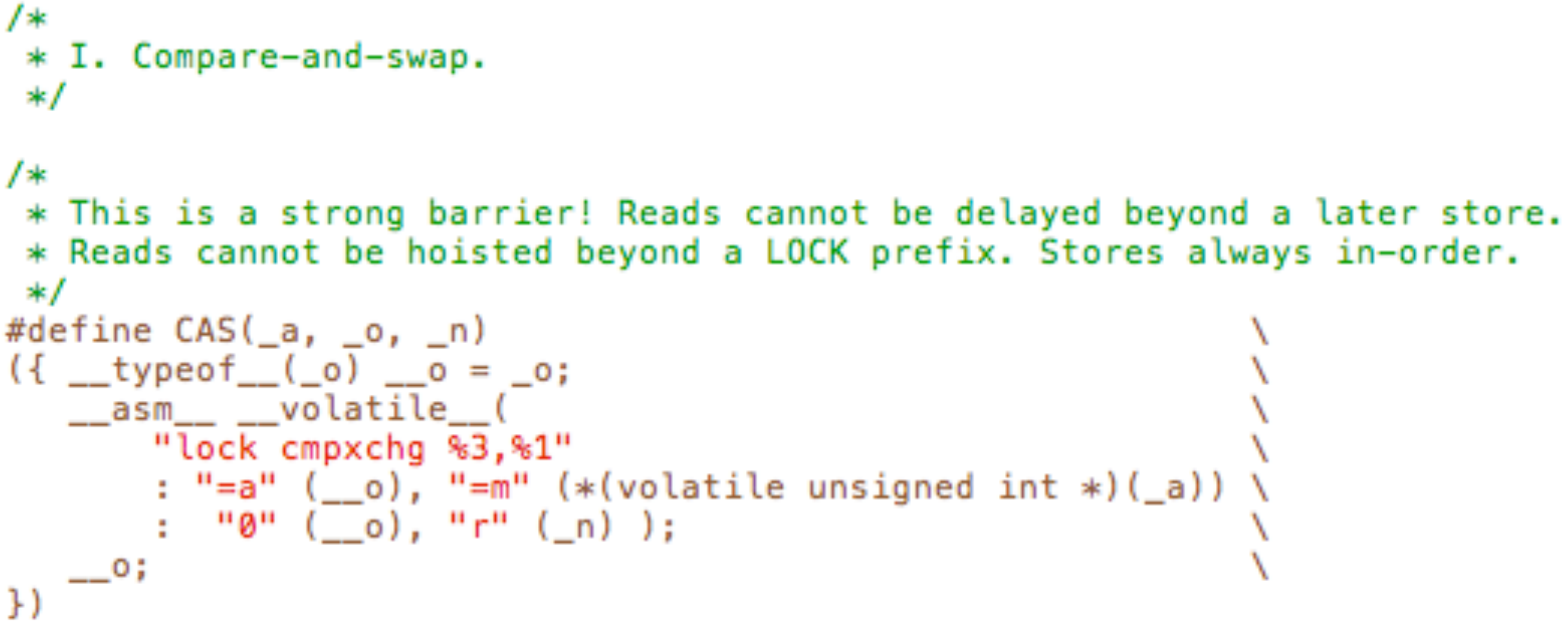
store: `LWSYNC; ST`
load: `LD`

The end?

C++11 is not yet implemented by mainstream compilers, and low-level atomics are hard to use (just google for low-level atomics).

How are interesting concurrent algorithms currently implemented? *Usually C plus asm!*

Example: `lockfree-lib`, by Keir Fraser, starts with some macro definitions...



```
/*
 * I. Compare-and-swap.
 */

/*
 * This is a strong barrier! Reads cannot be delayed beyond a later store.
 * Reads cannot be hoisted beyond a LOCK prefix. Stores always in-order.
 */
#define CAS(_a, _o, _n)                                    \
({ __typeof__(_o) __o = _o;                                \
   __asm__ __volatile__(                                   \
       "lock cmpxchg %3,%1"                                \
       : "=a" (__o), "=m" (*(volatile unsigned int *)(_a)) \
       :  "0" (__o), "r" (_n) );                           \
   __o;                                                    \
})
```

The end?

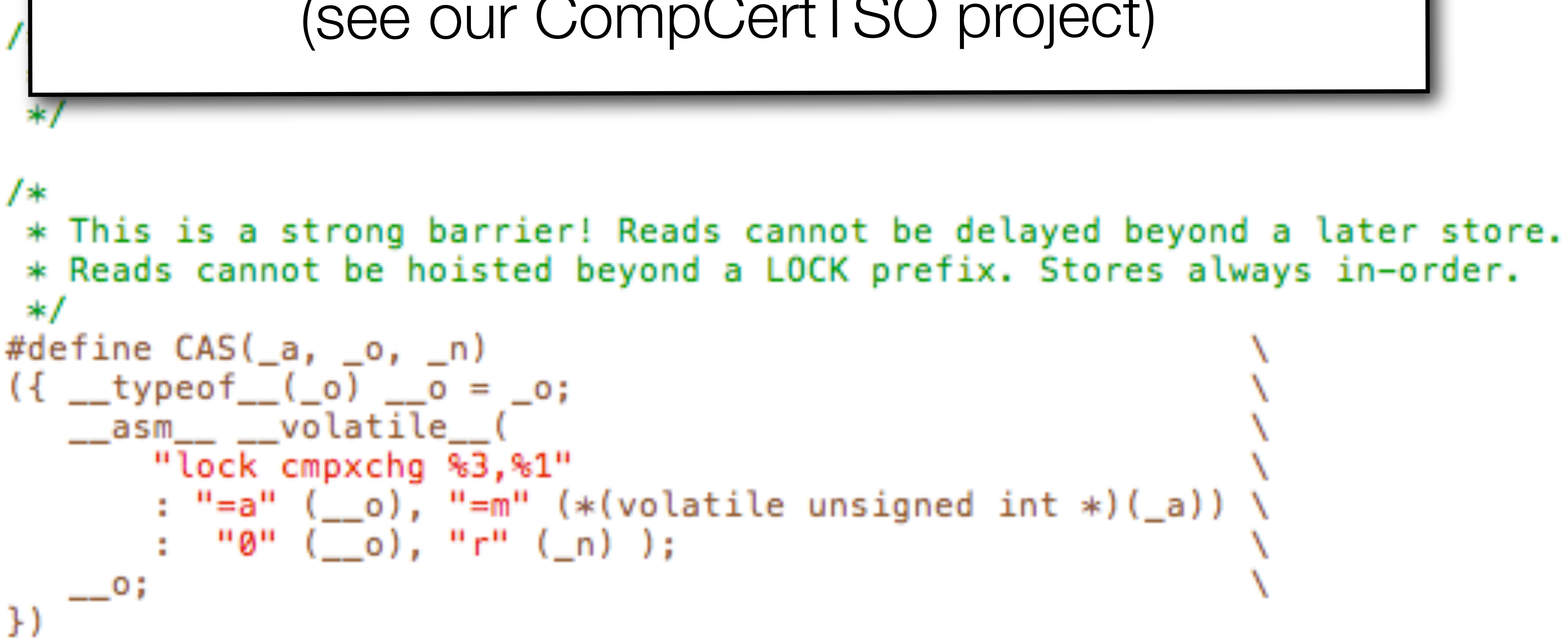
C++11 is not yet implemented by mainstream compilers, and low-level atomics are hard to use (just google for low-level atomics).

How are inte *sually C plus asm!*

Example: `lo` ons...

In some cases, it would be better to have a language whose semantics reflects the hardware reorderings, and a semantic-preserving compiler.

(see our CompCertTSO project)



```
/

 */

/*
 * This is a strong barrier! Reads cannot be delayed beyond a later store.
 * Reads cannot be hoisted beyond a LOCK prefix. Stores always in-order.
 */
#define CAS(_a, _o, _n)                                    \
({ __typeof__(_o) __o = _o;                                \
   __asm__ __volatile__(                                   \
       "lock cmpxchg %3,%1"                                \
       : "=a" (__o), "=m" (*(volatile unsigned int *)(_a)) \
       :  "0" (__o), "r" (_n) );                           \
   __o;                                                    \
})
```

A word on CompCertTSO

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

The C-TSO programming language:
a C-like language with a TSO semantics for memory accesses.

A semantic preserving compiler,
CompCertTSO

Intel processors implement the x86-TSO MM

Our we might want radically different

programming languages!

(Radically different language = radically different challenges?)

Lecture 1 (5/1/12).
Runtimes.

Lecture 2 (19/1/12).
Compilation of synchronous data-flow languages toward Java futures.

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