

# High-level languages, compilers, multiprocessors... Part 1. Shared memory: an elusive abstraction an elusive mix?

---

Francesco Zappa Nardelli

INRIA Paris-Rocquencourt

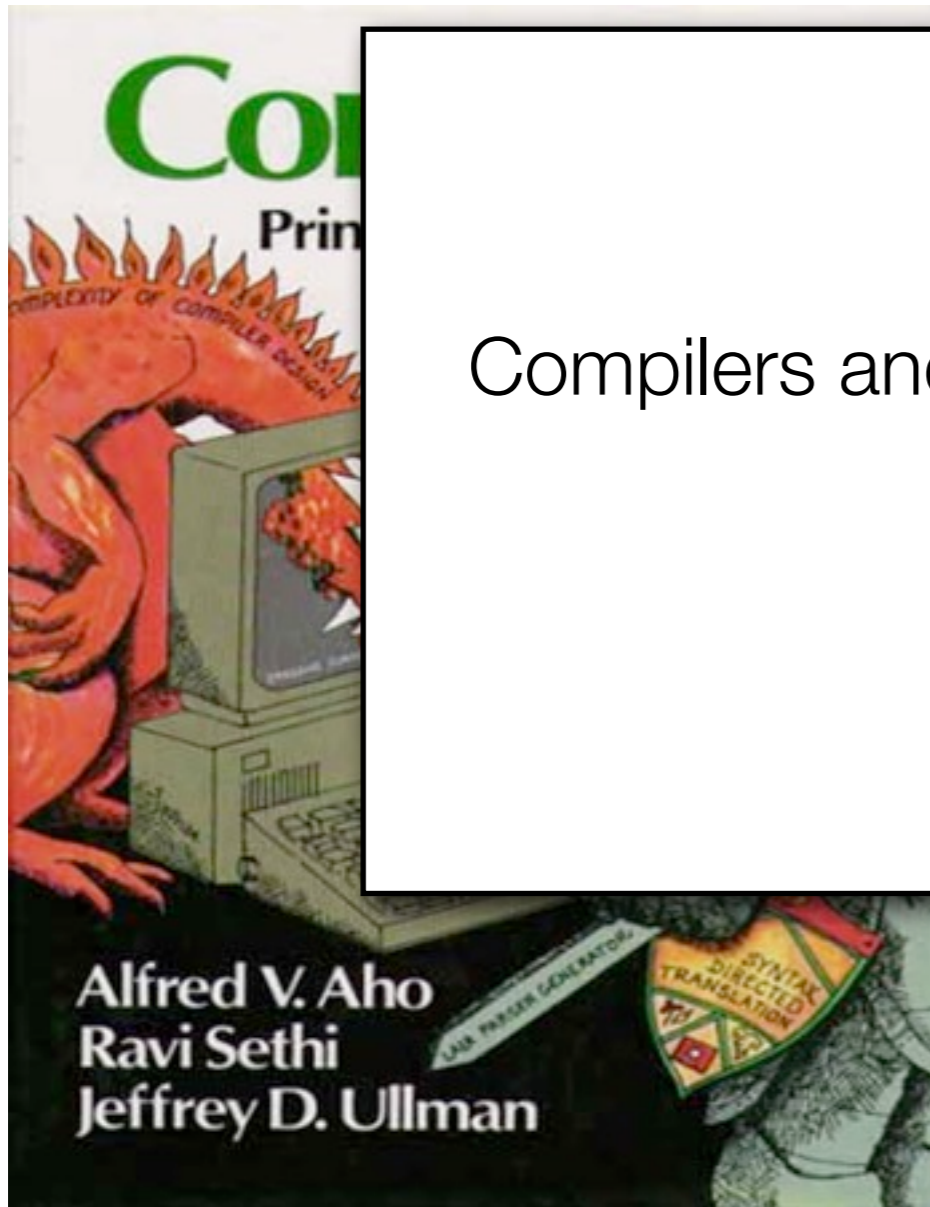
<http://moscova.inria.fr/~zappa/projects/weakmemory>

Based on work done by or with

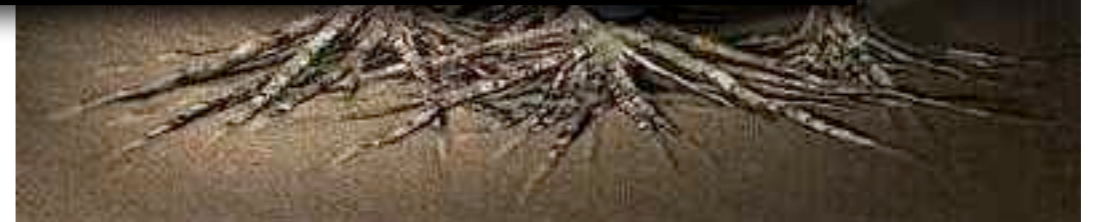
Peter Sewell, Jaroslav Ševčík, Susmit Sarkar, Tom Ridge, Scott Owens,  
Viktor Vafeiadis, Magnus O. Myreen, Kayvan Memarian, Luc Maranget,  
Derek Williams, Pankaj Pawan, Thomas Braibant, Mark Batty, Jade Alglave.

# Compilers vs. programmers

---




Compilers 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;  
*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`

<pre>lock(); tmp1 = *x; *x = tmp1 + 1; unlock();</pre>	<pre>lock(); tmp2 = *x; *x = tmp2 + 1; unlock();</pre>
--	--

- 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 until 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)

Intel SDM

- *n*-bytes

- P6 or later

*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)

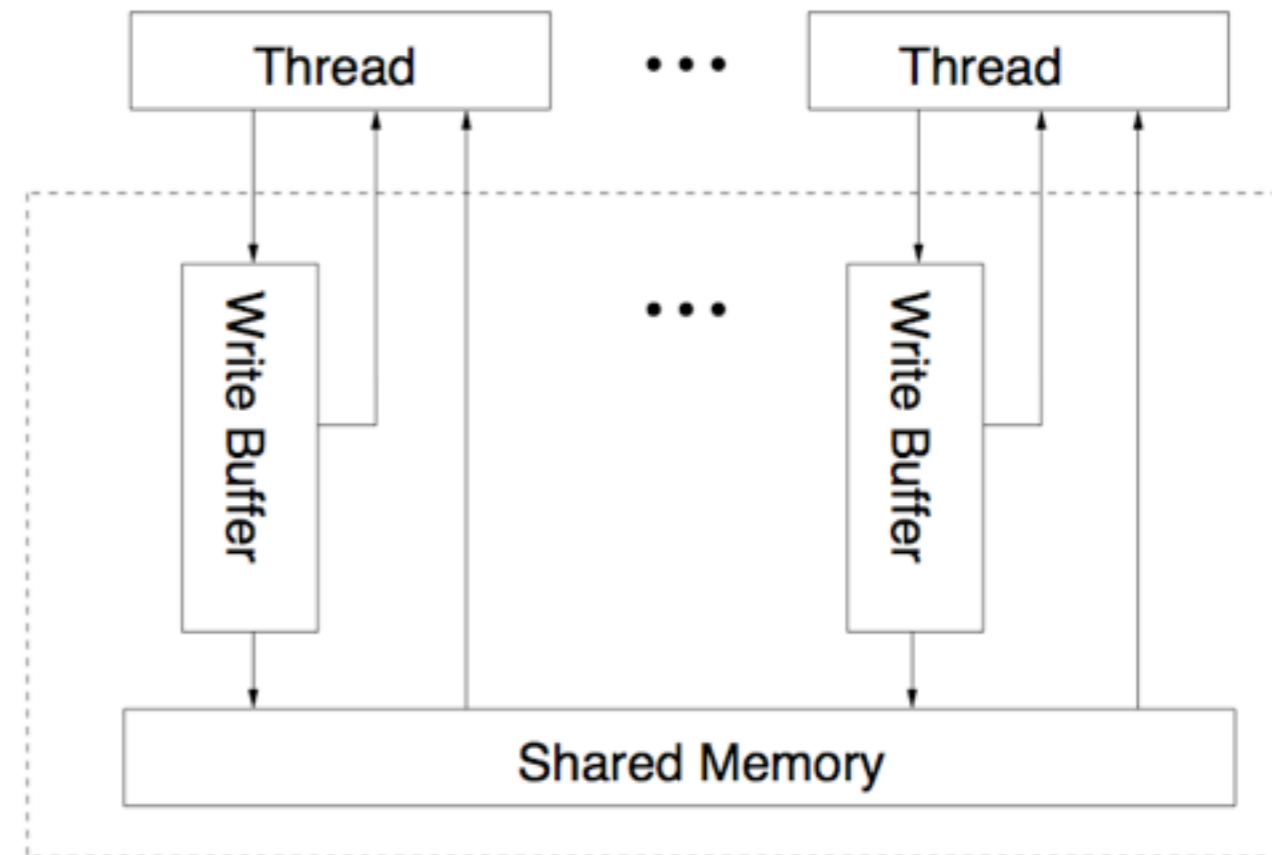
Hardware optimisations can be observed by concurrent code:

Thread 0	Thread 1
<code>x = 1</code> <code>print y</code>	<code>y = 1</code> <code>print x</code>

At the end of some executions:

0 0

is printed on the screen,  
both on x86 and Power/ARM).



# That pesky hardware (2)

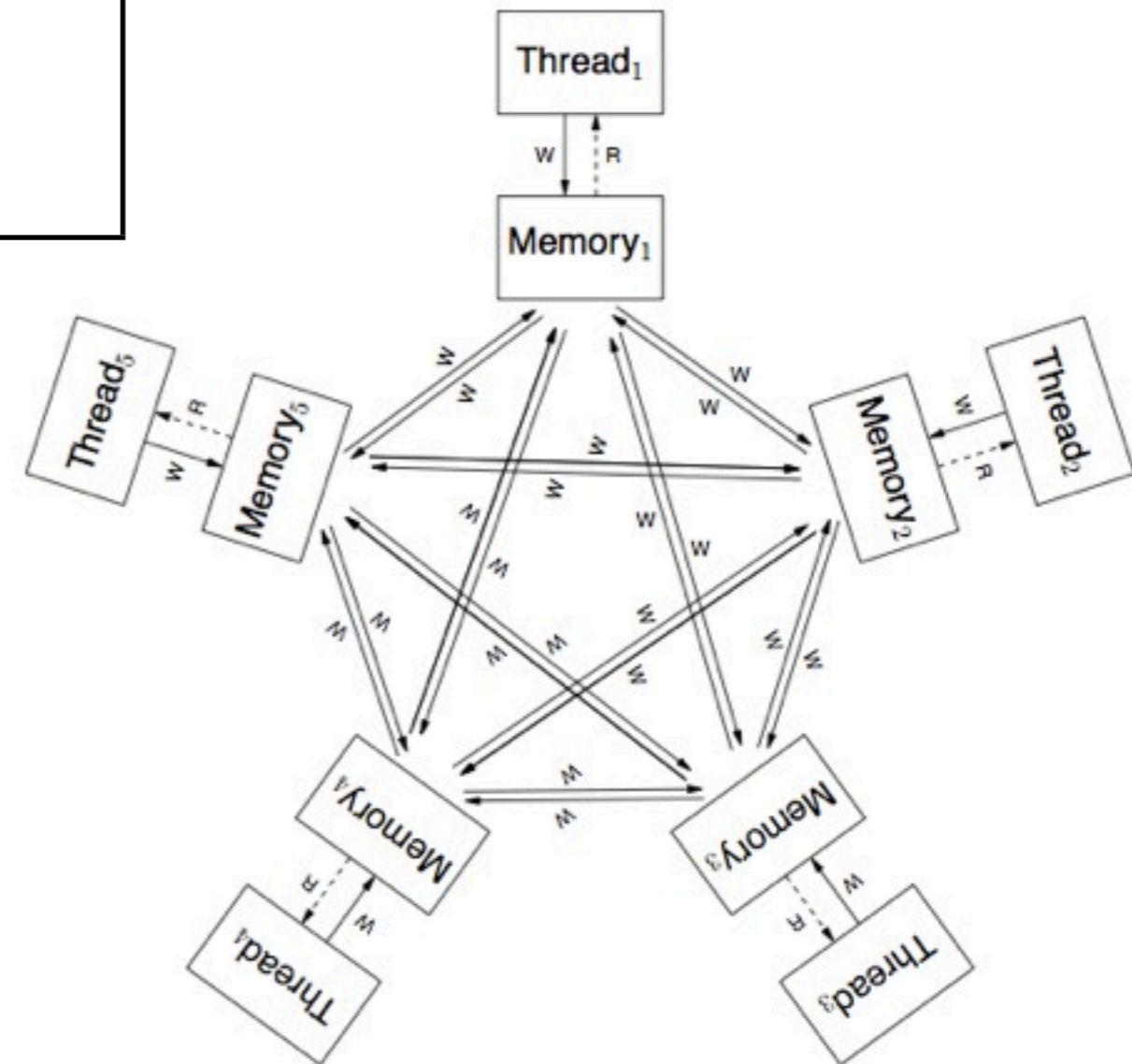
...and differ between architectures...

Thread 0	Thread 1
<code>x = 1</code>	<code>print y</code>
<code>y = 1</code>	<code>print x</code>

At the end of some executions:

1 0

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



# 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

## effect of VS2005 compiler optimisations on speed



## A brief tour of compiler optimisations

### effect of additional VS2005 optimisations on speed



# 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 and of writes – rarely).

# Eliminations

---

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

*Irrelevant read elimination:*

$$r=*x; C \rightarrow C$$

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

*Redundant read after read elimination:*

$$r1=*x; r2=*x \rightarrow r1=*x; r2=r1$$

*Redundant read after write elimination:*

$$*x=r1; r2=*x \rightarrow *x=r1; r2=r1$$

# 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

---

Back to our question now:

*Which is the semantics of a concurrent program?*

Note that the loop body is not executed.

# Vote: topics for my next lecture

---

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



Naive answer: enforce sequential consistency

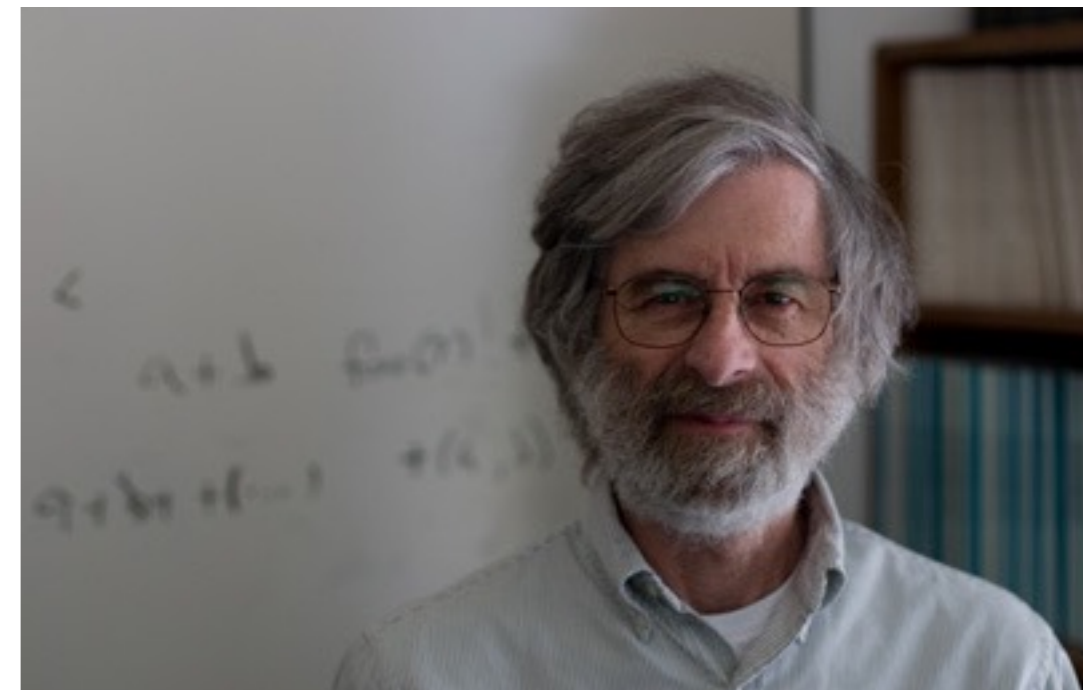
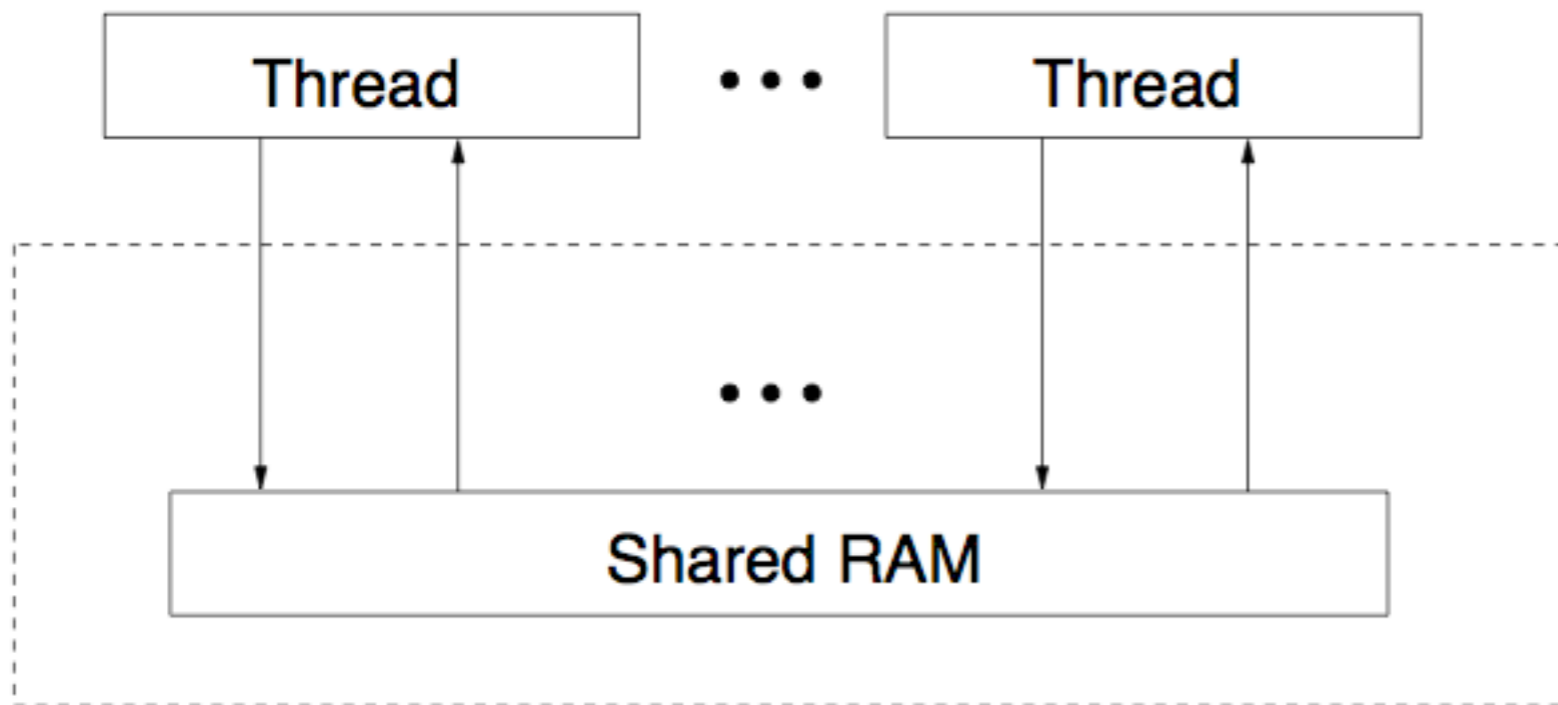
# Sequential consistency

---

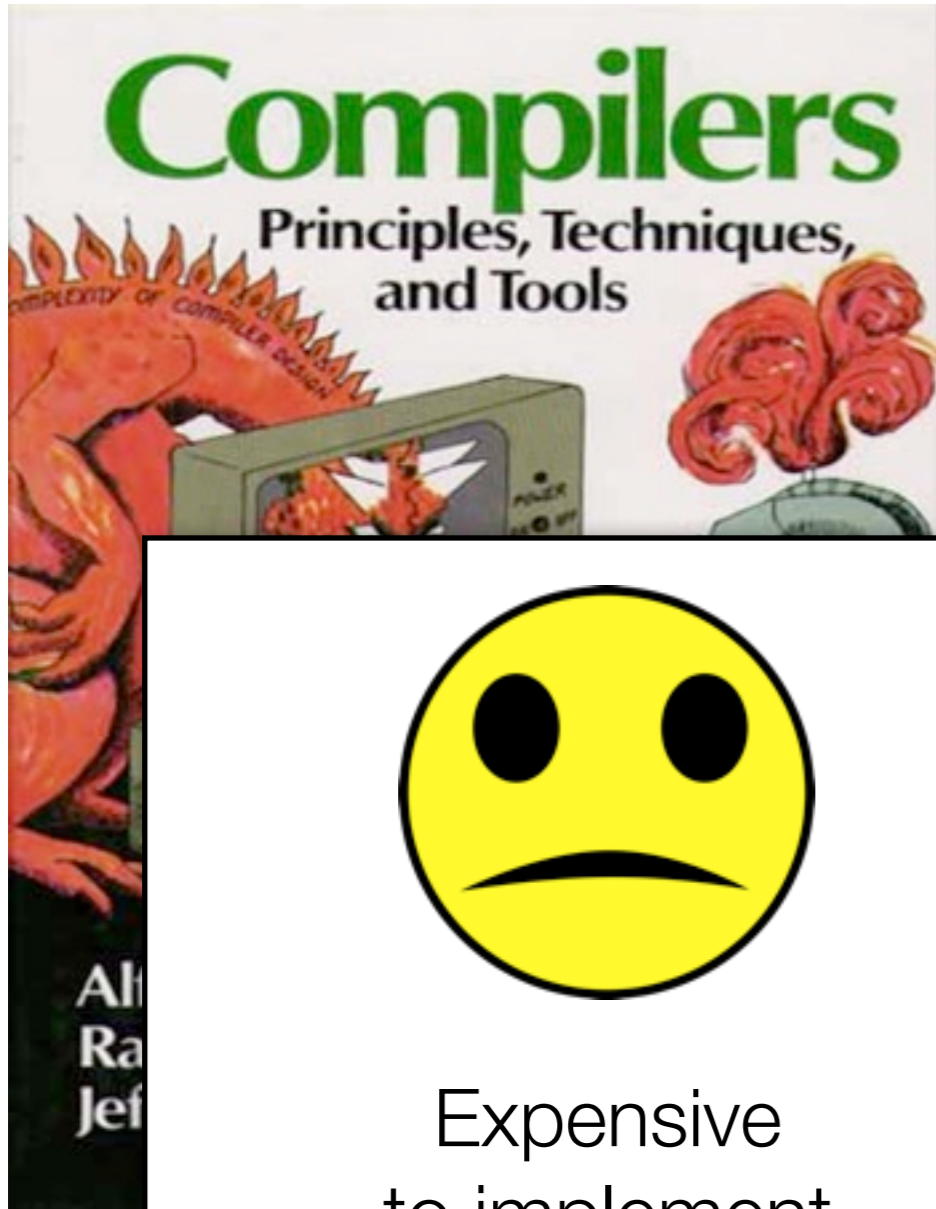
Multiprocessors have a *sequentially consistent* shared memory when:

*...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



Expensive  
to implement



Simple and intuitive  
programming model



# A Case for an SC-Preserving Compiler

Daniel Marino<sup>†</sup>   Abhayendra Singh\*   Todd Millstein<sup>†</sup>   Madanlal Musuvathi<sup>‡</sup>   Satish Narayanasamy\*

<sup>†</sup>University of California, Los Angeles

\*University of Michigan, Ann Arbor

<sup>‡</sup>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.

And this study supposes that the hardware is SC.



Expensive  
to implement

# 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: $[x]=0 \wedge [y]=0$	
proc 0	proc 1
MOV $[x] \leftarrow \$1$	MOV $[y] \leftarrow \$1$
MFENCE	MFENCE
MOV $EAX \leftarrow [y]$	MOV $EBX \leftarrow [x]$
<b>Forbid:</b> $EAX=0 \wedge EBX=0$	

Initially,  $[100] = 0$

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<sup>†</sup>   Abhayendra Singh\*   Todd Millstein<sup>†</sup>   Madanlal Musuvathi<sup>‡</sup>   Satish Narayanasamy\*

<sup>†</sup>University of California, Los Angeles

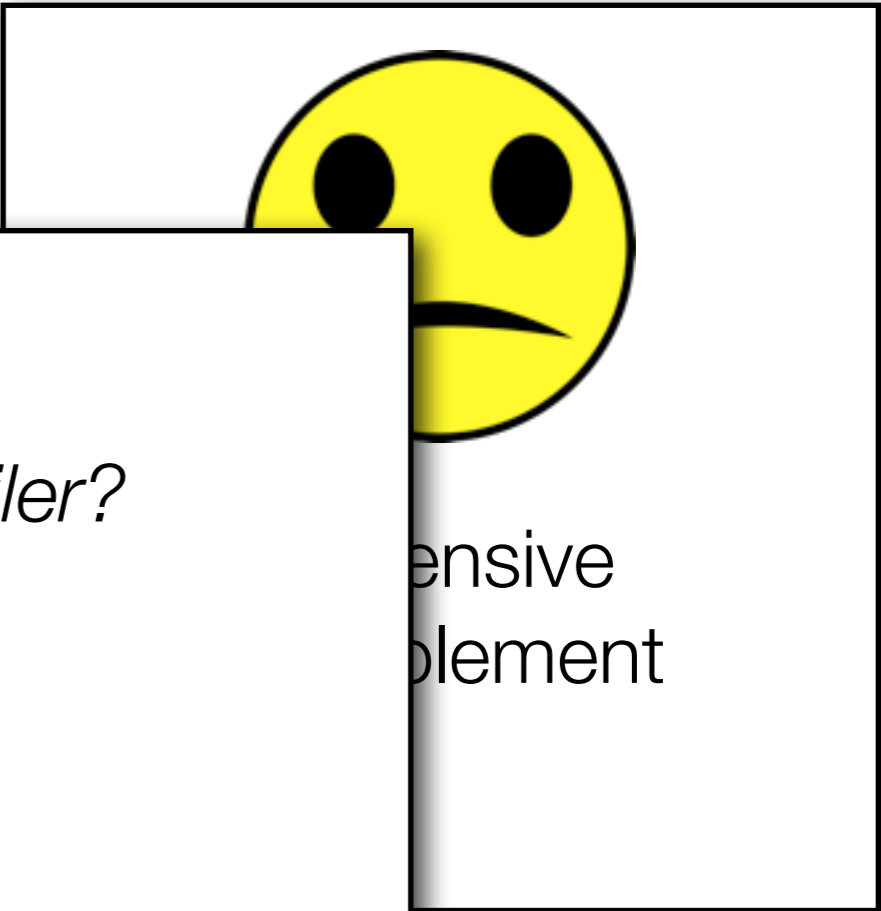
\*University of Michigan, Ann Arbor

<sup>‡</sup>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 6.04% on a set of 600 programs. This compiler also improves performance and SPEAK scores.

*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

---

$$P_1 = *x = 1 \quad \left| \quad \begin{array}{l} r1 = *x; \quad r2 = *x; \\ \text{if } r1=r2 \text{ then print 1 else print 2} \end{array} \right.$$
$$P_2 = *x = 1 \quad \left| \quad \begin{array}{l} r1 = *x; \quad r2 = r1; \\ \text{if } r1=r2 \text{ then print 1 else print 2} \end{array} \right.$$

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

# Example

---

$P_1 = *x = 1$		<code>r1 = *x; r2 = *x;</code> <code>if r1=r2 then print 1 else print 2</code>
$P_2 = *x = 1$		<code>r1 = *x; r2 = r1;</code> <code>if r1=r2 then print 1 else print 2</code>

Executions of P1:

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

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

Executions of P2:

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

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

# Example

---

$P_1 = *x = 1$	<pre>r1 = *x; r2 = *x; if r1=r2 then print 1 else print 2</pre>
$P_2 = *x = 1$	<pre>r1 = *x; r2 = r1; if r1=r2 then print 1 else print 2</pre>

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

---

<pre>*x = 1; *y = 1; if *y = 2 then print *x</pre>		<pre>if *x=1 then (     *x = 2;     *y = 2 )</pre>
--	--	--

There is only one execution with a printing behaviour:

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



# General CSE incorrect in SC

---

<pre>*x = 1; *y = 1; if *y = 2 then print *x</pre>	<pre>if *x=1 then (     *x = 2;     *y = 2 )</pre>
--	--

But a compiler would optimise to:

<pre>*x = 1; *y = 1; if *y = 2 then print 1</pre>	<pre>if *x=1 then (     *x = 2;     *y = 2 )</pre>
---	--

# General CSE incorrect in SC

---

<pre>*x = 1; *y = 1; if *y = 2 then print 1</pre>		<pre>if *x=1 then (     *x = 2;     *y = 2 )</pre>
---	--	--

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

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

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.

$[P_{t_2} 1, P_{t_2} 0, P_{t_2} 1]$

$[P_{t_2} 0, P_{t_2} 1, P_{t_2} 1]$

$[P_{t_2} 0, P_{t_2} 0, P_{t_2} 1]$

$[P_{t_2} 0, P_{t_2} 0, P_{t_2} 0]$

# General CSE incorrect in SC

---

<code>*x = 1;</code>	<code>r = *x;</code>
<code>*y = 1;</code>	<code>print r;</code>
	<code>print *y;</code>
	<code>print *x;</code>

But a compiler would optimise as:

<code>*x = 1;</code>	<code>r = *x;</code>
<code>*y = 1;</code>	<code>print r;</code>
	<code>print *y;</code>
	<code>print <b>r</b>;</code>

# 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.

$[P_{t_2} 1, P_{t_2} 1, P_{t_2} 1]$

$[P_{t_2} 1, P_{t_2} 0, P_{t_2} 1]$

$[P_{t_2} 0, P_{t_2} 1, P_{t_2} 1]$

$[P_{t_2} 0, P_{t_2} 0, P_{t_2} 1]$

$[P_{t_2} 0, P_{t_2} 0, P_{t_2} 0]$

$[P_{t_2} 1, P_{t_2} 1, P_{t_2} 1]$

$[P_{t_2} 1, P_{t_2} 0, P_{t_2} 1]$

$[P_{t_2} 0, P_{t_2} 1, P_{t_2} 0]$

$[P_{t_2} 0, P_{t_2} 0, P_{t_2} 0]$

# Reordering incorrect

---

<code>*x = 1;</code>		<code>*y = 1;</code>		<code>r1 = *y</code>		<code>*y = 1;</code>
<code>r1 = *y</code>		<code>r2 = *x;</code>	$\Rightarrow$	<code>*x = 1;</code>		<code>r2 = *x;</code>
<code>print r1</code>		<code>print r2</code>		<code>print r1</code>		<code>print r2</code>

Again, the optimised program exhibits a new behaviour:

$[P_{t_1}$	0,	$P_{t_2}$	1]
$[P_{t_1}$	1,	$P_{t_2}$	0]
$[P_{t_1}$	1,	$P_{t_2}$	1]

$[P_{t_1}$	0,	$P_{t_2}$	1]
$[P_{t_1}$	1,	$P_{t_2}$	0]
$[P_{t_1}$	1,	$P_{t_2}$	1]
$[P_{t_1}$	0,	$P_{t_2}$	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`

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 exam

- the prob  
(e.g. sw  
thread C

- the prob

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

**...intuition...**

Programming languages provide  
synchronisation mechanisms

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

Thread 1

```
x == 1  
n print *y
```

viour: 0

programs;

de that

# The basic solution

## Prohibit *data races*

Thread 0	Thread 1
<code>*y = 1</code> <code>*x = 1</code>	<code>if *x == 1</code> <code>then print *y</code>

Observable behaviour: 0

Defined as follows:

- two memory locations
- a SC execution (maybe executed concurrently).

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

memory  
conflicting  
t (maybe

*Example:* a data race in the example above:

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

# 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
<pre>*y = 1 lock(); *x = 1 unlock();</pre>	<pre>lock(); tmp = *x; unlock(); if tmp = 1 then print *y</pre>

# How do w

Compiler/hardware can continue to reorder accesses

*Intuition:*

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

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

```
*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
<pre>if *x == 1 then *y = 1</pre>	<pre>if *y == 1 then *x = 1</pre>

**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
Memory trace preserving transformations	✓



Jaroslav Sevcik

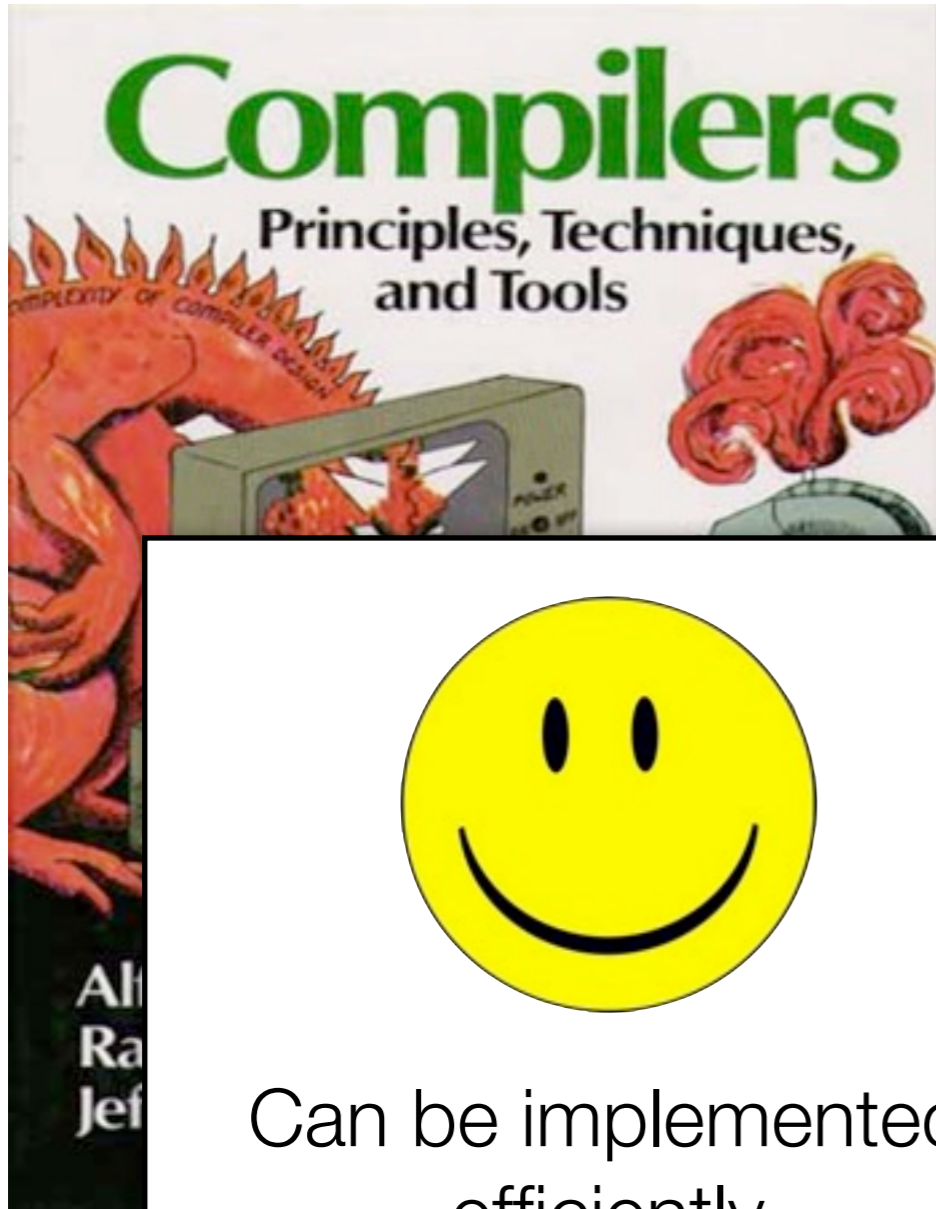
*Safe Optimisations for Shared-Memory Concurrent Programs*

*PLDI 2011*

Roach-motel <b>reordering</b>	× (✓ for locks)	✓
External action <b>reordering</b>	×	✓

\* Optimisations legal only on adjacent statements.

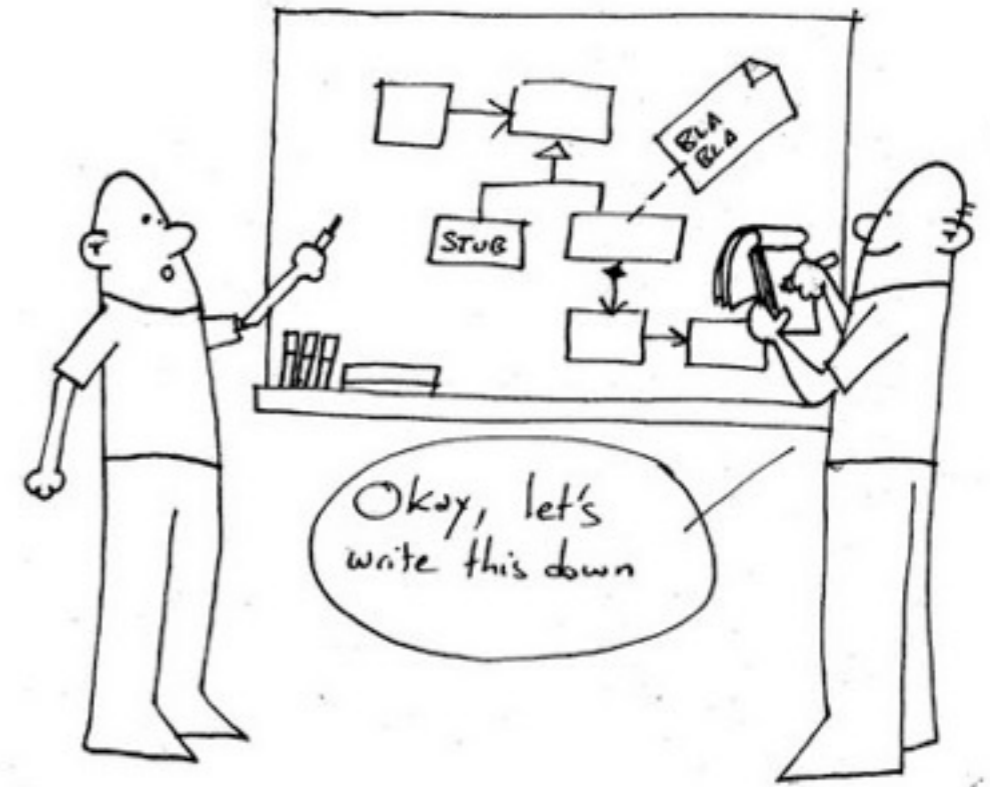
# Compilers, programmers & data-race freedom



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

*in*  
*th*  
*sta*

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

| lock                    lock

| unlock                unlock

| print r                output

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

*system ::=*            concurrent system

| *t<sub>0</sub> : p<sub>0</sub>* | ... | *t<sub>n</sub> : p<sub>n</sub>*            parallel composition of n threads

*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:

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

is also a valid traceset for the example below.

*Sample traceset:*

Thread 0	Thread 1
<code>r1:=x</code>	<code>r2:=y</code>
<code>y:=r1</code>	<code>x:=1</code>
	<code>print r2</code>

$$\begin{aligned} & \{[S(0), R[x=v], W[y=v]] \mid v \in V\} \\ \cup & \{[S(1), R[y=v], W[x=1], X(v)] \mid v \in V\} \end{aligned}$$

# Associate tracesets to toy language programs

---

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

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

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

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

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

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

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

# Tracesets and interleavings

---

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

*Example:*                    `y:=1;    ||    r2:=v;print r2;`

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

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

$$\text{trace } 1 \ I' = [ S(1), R[v=0], X(0) ].$$

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

- *tids*

- loc

- each

(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.

).

sses).

# Happens-before

---

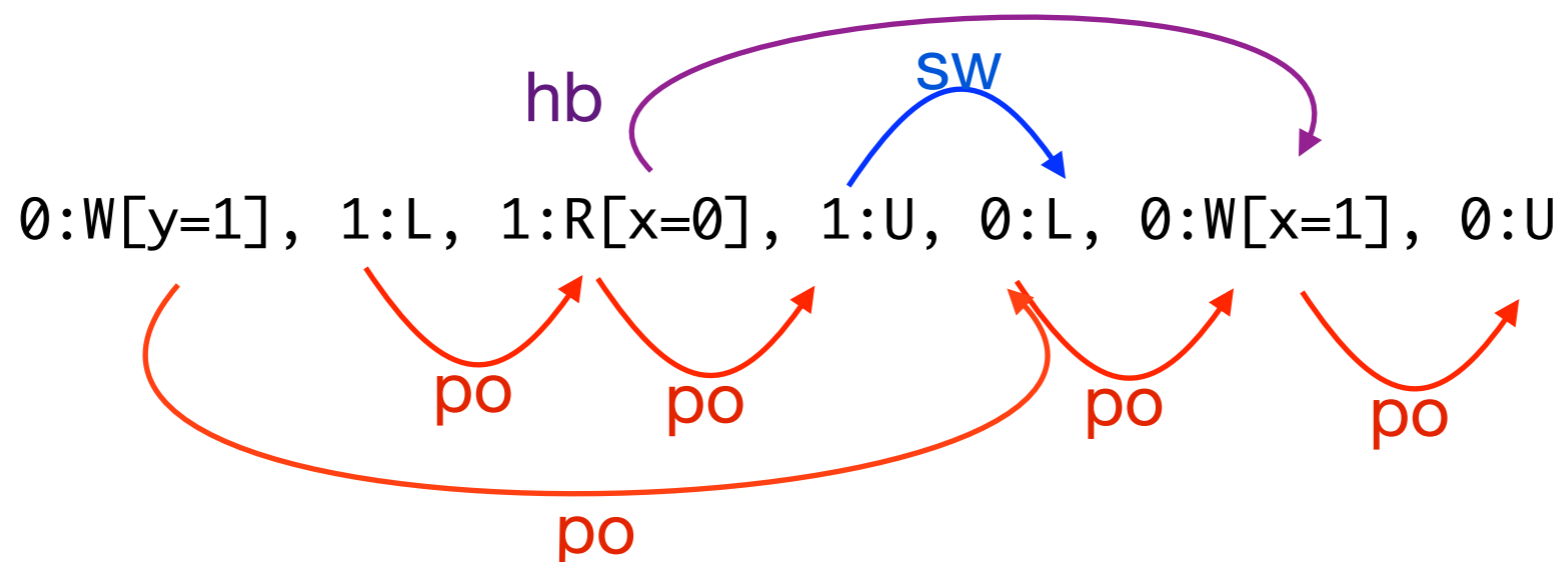
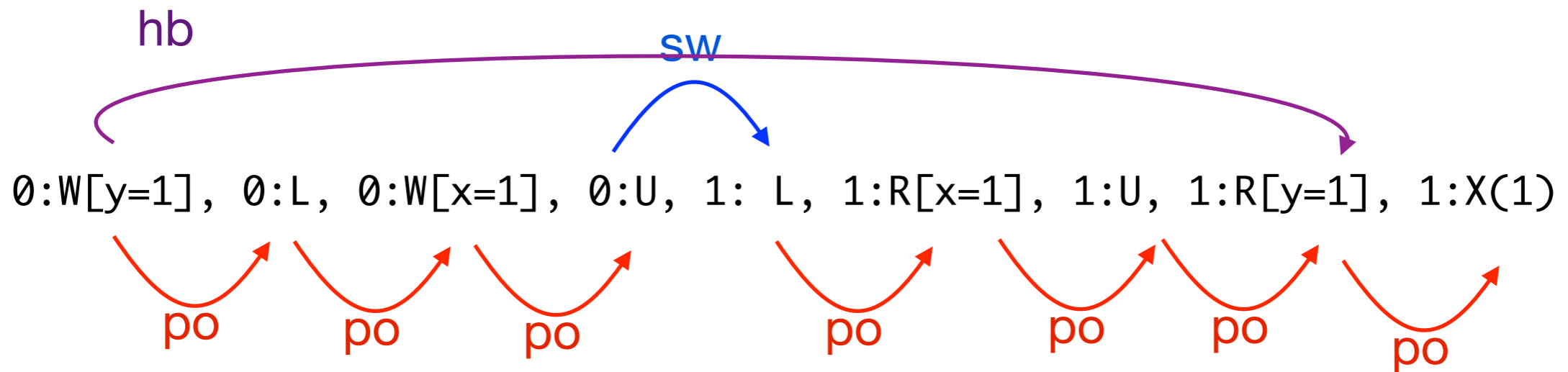
*Definition [program order]:* **program order**,  $<_{po}$ , is a total order over the actions of *the same thread in an interleaving*.

*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
<code>*y = 1</code> <code>lock();</code> <code>*x = 1</code> <code>unlock();</code>	<code>lock();</code> <code>tmp = *x;</code> <code>unlock();</code> <code>if tmp = 1</code> <code>then print *y</code>



S(tid) actions omitted.

# Data-race freedom

---

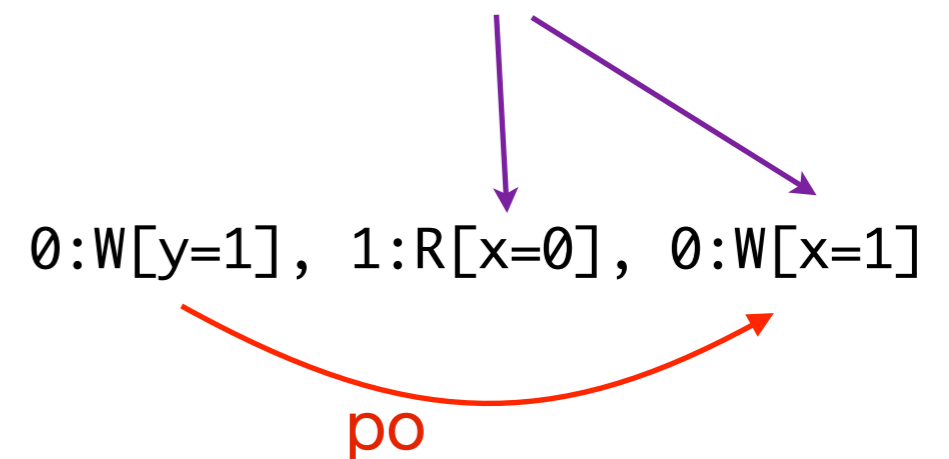
*Definition [data-race-freedom]:* A traceset is **data-race free** if none of its executions has two adjacent conflicting actions from different threads.

*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
<code>*y = 1</code>	<code>if *x == 1</code>
<code>*x = 1</code>	<code>then print *y</code>

Two conflicting accesses  
not related by happens before.





Peter Thiemann, University of Freiburg, Germany

1. Gradual typing for session types;
2. manifestation for types and effects.

Mooly Sagiv, Tel Aviv University, Israel

Applying formal methods to Network Verification

Lars Birkedal, Aarhus University, Denmark

Programming logics for reasoning about concurrent, higher-order, imperative programs

Peter Mueller, Swiss Federal Institute of Technology, Zurich, Switzerland

1. Techniques and tools for the specification and verification of concurrent programs.
2. Static analysis of mobile apps

Cesare Tinelli, University of Iowa, USA

Formal Methods For Critical Systems: Mutation-based Testing and Model Checking

Formal Verification Of Critical Systems: Property-Directed Invariant Discovery By Backward Analysis

# Data-race freedom: equivalence of definitions

---

Given an execution

$$\alpha \text{ ++ } [a] \text{ ++ } \beta \text{ ++ } [b]$$

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

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

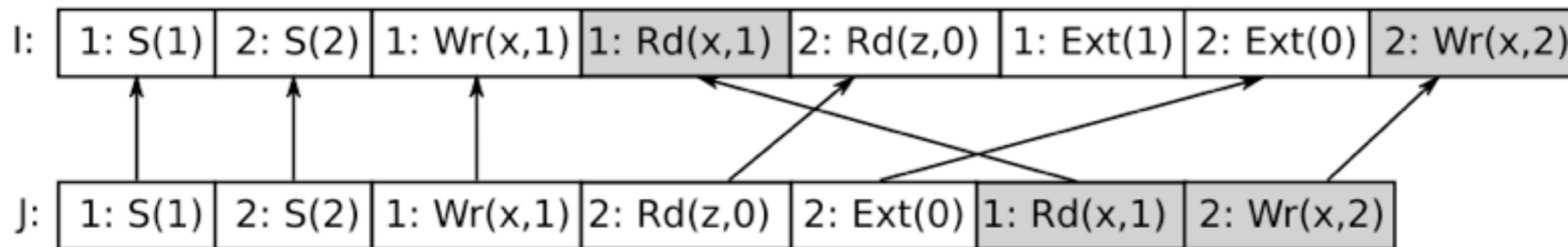
where  $\beta'$  are all the actions from  $\beta$  that strictly happen-before  $[b]$ .

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

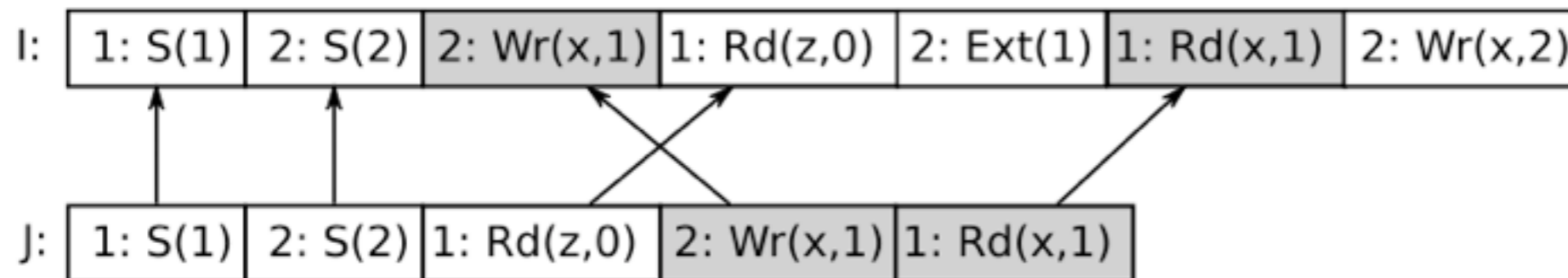
The formal proof is tedious and not easy (see Boyland 2008, Bohem & Adve 2008, Sevcik ), here will give the intuitions of the construction on an example.

# 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.

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

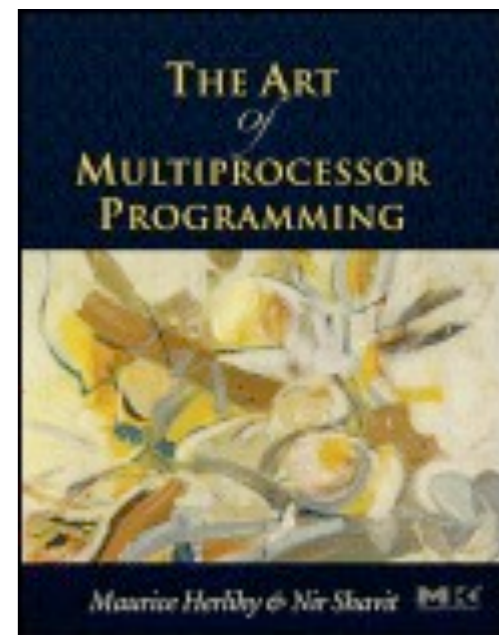
---

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

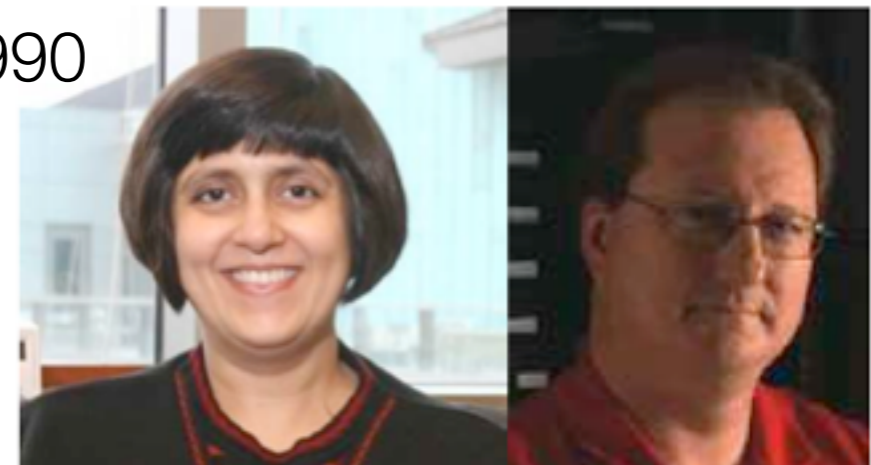
---

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

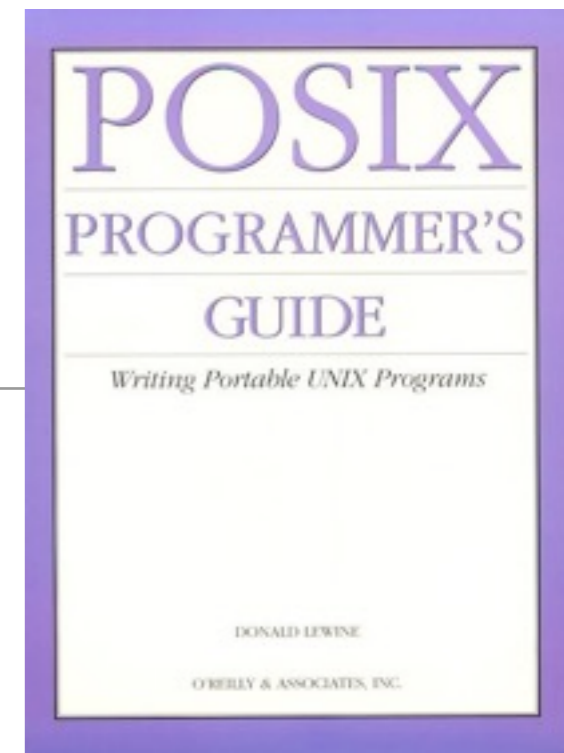
*Cons:*

- 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 C11/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?

---

Perhaps it should have been.

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



# 1. Uncertainty 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:      Thread 2:
x.a = 1;      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;
```



```
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 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:

<b>initially <math>x = y = 0</math></b>	
<code>r1 := x</code>	<code>r2 := y</code>
<code>y := r1</code>	<code>x := r2</code>
<code>print r1</code>	<code>print r2</code>

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:

<b>initially <math>x = y = 0</math></b>	
<code>r1 := x</code>	<code>r2 := y</code>
<code>y := r1</code>	<code>x := r2</code>
<code>print r1</code>	<code>print r2</code>

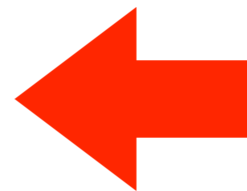
the only possible result should be printing two zeros because no other value appears in or can be created by the program.

# 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	
<del>r1 := x</del>	r2 := y
<del>y := r1</del>	x := r2
<del>print r1</del>	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:

<code>initially x = y = 0</code>			<code>initially x = y = null</code>	
<code>r1 := x</code>	<code>r2 := y</code>	and	<code>r1 := x</code>	<code>r2 := y</code>
<code>y := r1</code>	<code>x := r2</code>		<code>y := r1</code>	<code>x := r2</code>
<code>print r1</code>	<code>print r2</code>			<code>r2.run()</code>

What should `r2.run()` call?

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



# A word on JSR-133



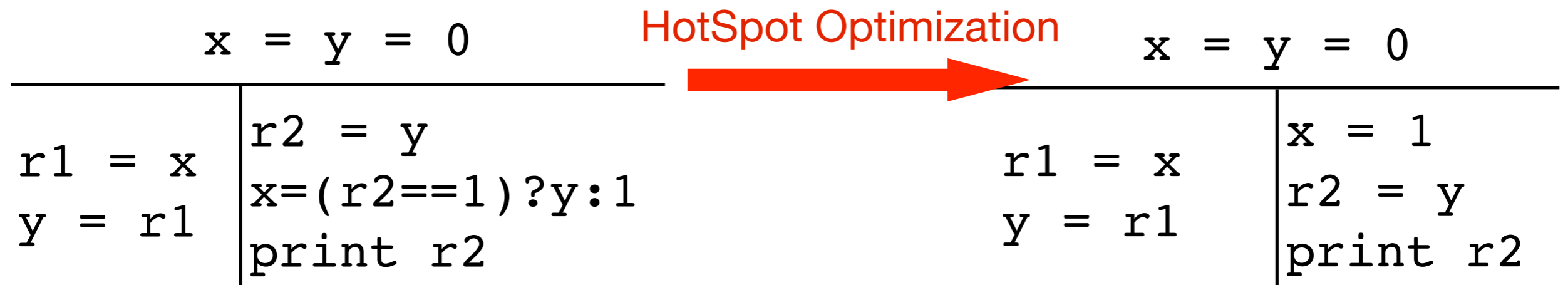
Goal 1: data-race free programs are sequentially consistent;

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?



Jaroslav Ševčík, David Aspinall, ECOOP 2008

# 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