

# High-level languages

High-level languages are not immune to these problems.

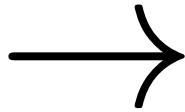
Actually, the situation is even worse:

- the compiler might reorder/remove/add memory accesses;
- and *then* the hardware will do some relaxed execution.

# Constant Propagation

$$x = 3287$$

$$y = 7 - x / 2$$



$$x = 3287$$

$$y = 7 - 3287 / 2$$

# Constant Propagation

$x = 3287$   
 $y = 7 - x / 2$        $\longrightarrow$        $x = 3287$   
 $y = 7 - 3287 / 2$

Initially $x = y = 0$	
$x = 1$	$\text{if } (x==1) \{$
$\text{if } (y==1)$	$x = 0$
$\text{print } x$	$y=1 \}$

SC: can never print 1

Sun HotSpot JVM or GCJ: always prints 1

# Non-atomic Accesses

Consider misaligned 4-byte accesses

Initially `int32_t a = 0`

---

`a = 0x44332211`

`if a = 0x00002211`

`print "oops"`

# Non-atomic Accesses

Consider misaligned 4-byte accesses

```
Initially int32_t a = 0
-----
a = 0x44332211 | if a = 0x00002211
                  print "oops"
```

(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

What about multi-word high-level language values?

# Defining PL Memory Models

# Defining PL Memory Models

## **Option 1: Don't. No Concurrency**

Poor match for current trends

# Defining PL Memory Models

## **Option 2: Don't. No Shared Memory**

A good match for *some* problems

Erlang, MPI



# Defining PL Memory Models

## **Option 3: Don't. SC Shared Memory, with Races**

(What OCaml gives you — but that's not a true concurrent impl.)

(What Haskell gives you for MVars?)

In general, it's going to be expensive...

Naive impl: barriers between *every* memory access

(smarter: analysis to approximate the thread-local or non-racy accesses, but aliasing always hard)

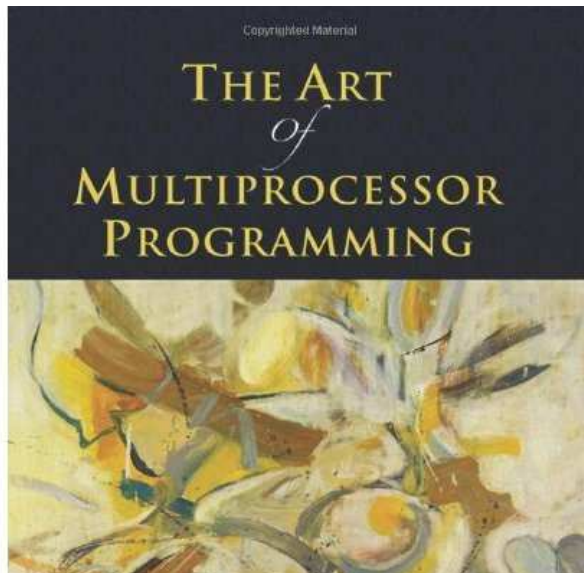
# Defining PL Memory Models

## Option 4: Don't. Shared Memory, but Language ensures Race-Free

e.g. by ensuring data accesses protected by associated locks

Possible — but inflexible... (pointer aliasing?)

What about all those fancy high-performance concurrent algorithms?



# Defining PL Memory Models

**Option 5: Don't. Shared Memory, but verify programs in concurrent separation logic and prove that implies race-freedom (and hence all executions are SC)**

Appel et al.

great — but “verify”?!

# Defining PL Memory Models

**Option 6: Don't. Leave it (sort of) up to the hardware**

Example: MLton

(high-performance ML-to-x86 compiler, with concurrency extensions)

Accesses to ML refs will exhibit the underlying x86-TSO behaviour

But, they will at least be atomic

# Defining PL Memory Models

## Option 7: **Do(!)** Use Data race freedom as a *definition*

- programs that are race-free in SC semantics have SC behaviour
- programs that have a race in some execution in SC semantics can behave in any way at all

Sarita Adve & Mark Hill, 1990



# Option 7: DRF as a definition

Core of C++0x draft. Hans Boehm & Sarita Adve, PLDI 2008

Pro:

- Simple!
- Strong guarantees for most code
- Allows lots of freedom for compiler and hardware optimisations



‘Programmer-Centric’

# Option 7: DRF as a definition

Core of C++0x draft. Hans Boehm & Sarita Adve, PLDI 2008

Con:

- programs that have a race in some execution in SC semantics *can behave in any way at all*
  - Undecidable premise.
  - Imagine debugging: either bug is X ... or there is a potential race in *some* execution
  - No guarantees for untrusted code
- restrictive. Forbids those fancy concurrent algorithms
- need to define exactly what a race is  
what about races in synchronisation and concurrent datastructure libraries?

# Defining PL Memory Models

**Option 8: Don't. Take a concurrency-oblivious language spec (e.g. C) and bolt on a thread library (e.g. Posix or Windows threads)**

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

*Threads Cannot be Implemented as a Library*, Hans Boehm,  
PLDI 2005





# Defining PL Memory Models

Recall DRF gives no guarantees for untrusted code

Would be a disaster for Java, which relies on unforgeable pointers for its security guarantees

**Option 9: Do. DRF + some out-of-thin-air guarantee for all code**

# Option 9: The Java Memory Model(s)

Java has integrated multithreading, and it attempts to specify the precise behaviour of concurrent programs.

By the year 2000, the initial specification was shown:

- to allow unexpected behaviours;
- to prohibit common compiler optimisations,
- to be challenging to implement on top of a weakly-consistent multiprocessor.

Superseded around 2004 by the JSR-133 memory model.

The Java Memory Model, Jeremy Manson, Bill Pugh & Sarita Adve, POPL05



# Option 9: JSR-133

- Goal 1: data-race free programs are sequentially consistent;
- Goal 2: all programs satisfy some memory safety and security requirements;
- Goal 3: common compiler optimisations are sound.

# Option 9: JSR-133 — Unsoundness

The model is intricate, and *fails to meet Goal 3*: Some optimisations may generate code that exhibits more behaviours than those allowed by the un-optimised source.

As an example, JSR-133 allows  $r2=1$  in the optimised code below, but forbids  $r2=1$  in the source code:

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

*HotSpot optimisation*  
→

$x = y = 0$	
$r1=x$	$x=1$
$y=r1$	$r2=y$

Jaroslav Ševčík & Dave Aspinall, ECOOP 2008



# Defining PL Memory Models

Recall DRF is restrictive, forbidding racy concurrent algorithms (also costly on Power)

And note that C and C++ don't guarantee type safety in any case.

# Defining PL Memory Models

Recall DRF is restrictive, forbidding racy concurrent algorithms (also costly on Power)

And note that C and C++ don't guarantee type safety in any case.

**Option 10: Do. DRF + low-level atomic operations with relaxed semantics**

C++0x approach.

Foundations of the C++ Memory Model, Boehm&Adve PLDI08

Working Draft, Standard for Programming Language C++, N3090, 2010-03

<http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2010/>

with Lawrence Crowl, Paul McKenney, Clark Nelson, Herb Sutter,...

# Option 10: C++0x

- normal loads and stores
- lock/unlock
- atomic operations (load, store, read-modify-write, ...)
  - `seq_cst`
  - `relaxed`, `consume`, `acquire`, `release`, `acq_rel`

Idea: if you only use SC atomics, you get DRF guarantee  
Non-SC atomics there for experts.

Informal-prose spec.

Formalisation in progress, in HOL — Mark Batty



# Problem: Untested Subtlety

For any such subtle and loose specification, how can we have any confidence that it:

- is well-defined?  
must be mathematically precise
- has the desired behaviour on key examples?  
exploration tools
- is internally self-consistent?  
formalisation and proof of metatheory
- is what is implemented by compiler+hw?  
testing tools; compiler proof
- is comprehensible to the programmer?  
must be maths explained in prose
- lets us write programs above the model?  
static analysis/dynamic checker/daemoniac emulator
- is implementable with good performance?  
implement...



# Problem/Opportunity: Legacy Complexity

Most of these talks have been dominated by complex legacy choices:

- hw: x86, Power, Alpha, Sparc, Itanium
- sw: C, C++ and Java compiler optimisations, language standards and programming idioms

We may be stuck with these - but maybe not... Can we build radically more scalable systems with a better hw/sw or lang/app interface?

# The End

# Thanks!

