

Semantics of Shared Variables & Synchronization a.k.a. Memory Models

Sarita V. Adve

University of Illinois
sadve@illinois.edu

Hans-J. Boehm

HP Labs
hans.boehm@hp.com

Acks: Vikram Adve, Rob Bocchino, Lawrence Cowl, Kourosh Gharachorloo,
Mark Hill, Doug Lea, Jeremy Manson, Paul McKenney, Clark Nelson, Bill Pugh,
Marc Snir, Herb Sutter, and many others

Shared Variable Semantics or Memory Models



Parallelism for the masses!
Shared-memory, threads most common
Memory model = Legal values for reads



Shared Variable Semantics or Memory Models



Parallelism for the masses!
Shared-memory, threads most common
Memory model = Legal values for reads



Shared Variable Semantics or Memory Models



Parallelism for the masses!
Shared-memory, threads most common
Memory model = Legal values for reads



Shared Variable Semantics or Memory Models



Parallelism for the masses!
Shared-memory, threads most common
Memory model = Legal values for reads



Shared Variable Semantics or Memory Models



Parallelism for the masses!
Shared-memory, threads most common
Memory model = Legal values for reads



Shared Variable Semantics or Memory Models



Parallelism for the masses!
Shared-memory, threads most common
Memory model = Legal values for reads

Our goals

- **Demystify**
- **Convey pitfalls**
- **Convey fundamental problems**
- **Motivate new research**



What is a Memory Model?

- Memory model defines what values a read can return

Initially A=B=Flag=0

Thread 1

A = 26

B = 90

...

Flag = 1

Thread 2

while (Flag != 1) {;}

r1 = B ← ~~90~~

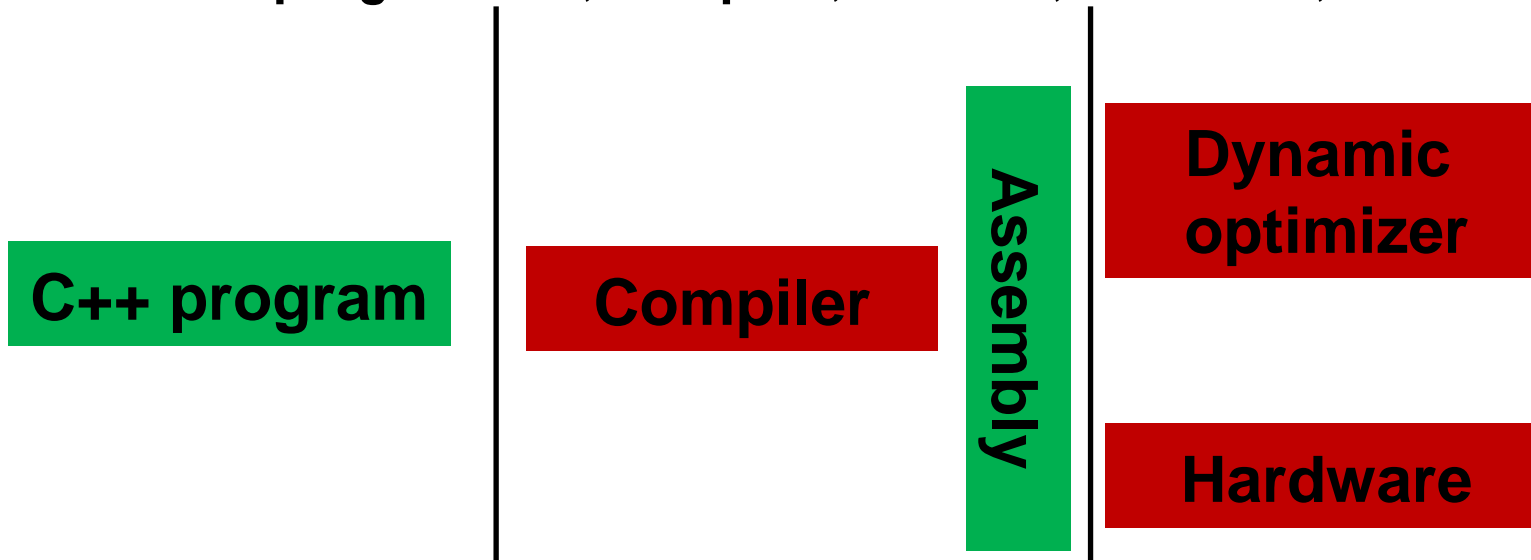
r2 = A ← ~~26~~ **0**

...

- Are concurrent accesses allowed?
- What is a concurrent access?
- When do updates become visible to other threads?
- Can an update be partially visible?
- ...

Why Should You Care?

- Interface between **program** and **transformers** of program
 - Affects programmer, compiler, runtime, hardware, ...



- Weakest system component exposed to the programmer
 - Not just a “hardware problem” nor just a “compiler problem”
 - Not just a processor issue; memory design affects system model

Desirable Properties of a Memory Model

- **3 Ps**
 - **P**rogrammability
 - **P**erformance
 - **P**ortability
- **Challenge: hard to satisfy all 3 Ps**

This Tutorial

- **The problem**
 - Sequential consistency (SC) is intuitive
 - But performance? And is SC really easy enough?
- **Recent consensus: data-race-free**
 - SC only for good (data-race-free) programs
- **Some research and practice pitfalls**
- **BUT data-race-free not the full answer**
 - Ultimate performance on current hardware?
 - Undefined race semantics complicate safety, debugging,
- **Need new approach**
 - Some ongoing research

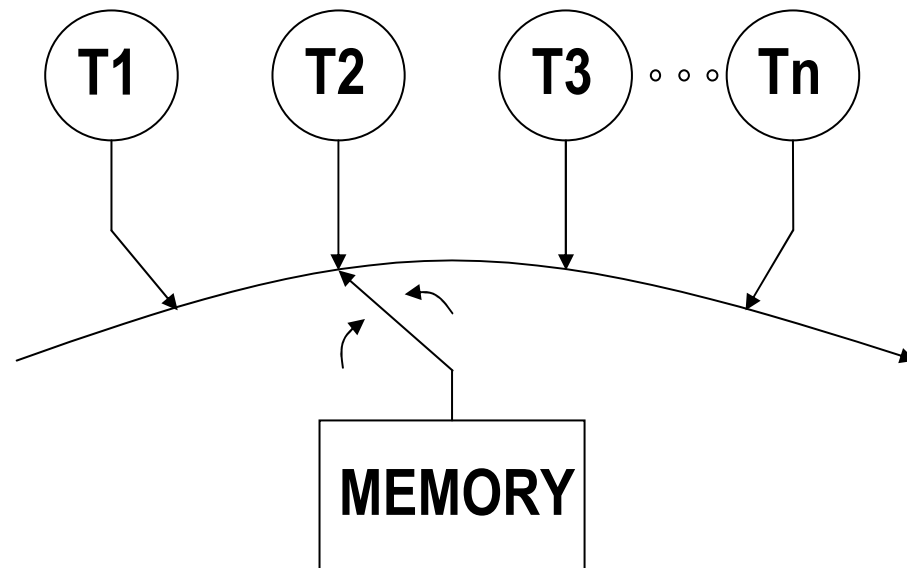


The Single Thread Model

- Program text defines total order = *program order*
 - Single thread model
 - Memory accesses execute one-at-a-time in program order
 - ⇒ Read returns value of last write
 - BUT hardware & compilers overlap, reorder accesses
 - Obey model by respecting control/data dependences
- ⇒ Easy to use + high performance

Implicit Multithreaded Model

- Sequential consistency (SC)
 - Accesses of each thread occur in **program order**
 - All accesses occur in some **sequential order (atomically)**



Still Need Synchronization

Initially $X = 0$

Thread 1

$r1 = X;$



$X = r1 + 1;$

Thread 2

$r2 = X;$



$X = r2 + 1;$

Still Need Synchronization

Initially $X = 0$

Thread 1

$r1 = X;$



$X = r1 + 1;$

Thread 2

$r2 = X;$



$X = r2 + 1;$

SC allows all program order consistent interleavings

Still Need Synchronization

Initially $X = 0$

Thread 1

$r1 = X;$



$X = r1 + 1;$

Thread 2

$r2 = X;$



$X = r2 + 1;$

SC allows all program order consistent interleavings

Still Need Synchronization

Initially $X = 0$

Thread 1

$r1 = X;$



$X = r1 + 1;$

Thread 2

$r2 = X;$



$X = r2 + 1;$

Result: $r1=r2=0, X = 1$

SC allows all program order consistent interleavings

Still Need Synchronization

Initially $X = 0$

Thread 1

lock(L)

$r1 = X;$



$X = r1 + 1;$

unlock(L)

Thread 2

lock(L)

$r2 = X;$

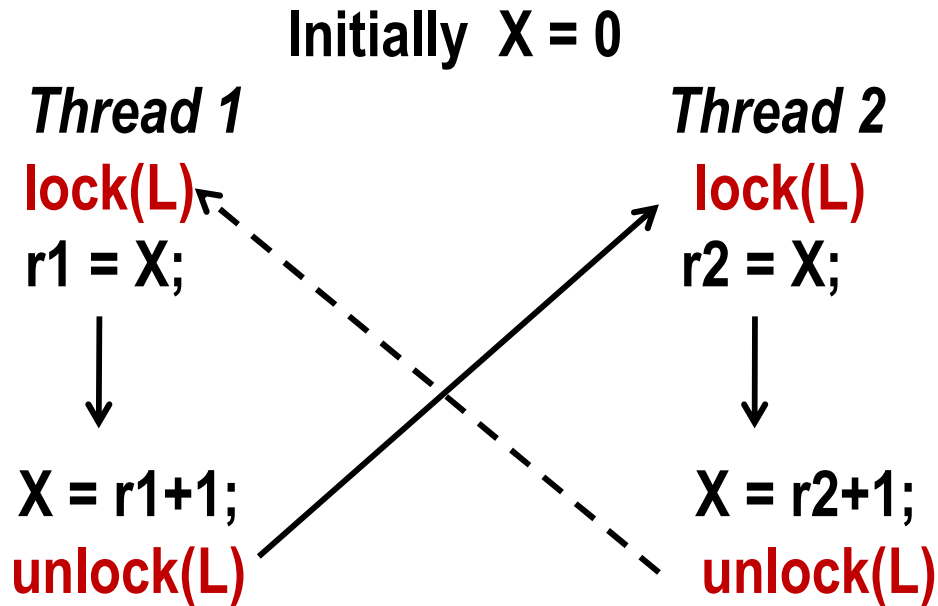


$X = r2 + 1;$

unlock(L)

Use locks (or other synch) to restrict interleavings

Still Need Synchronization



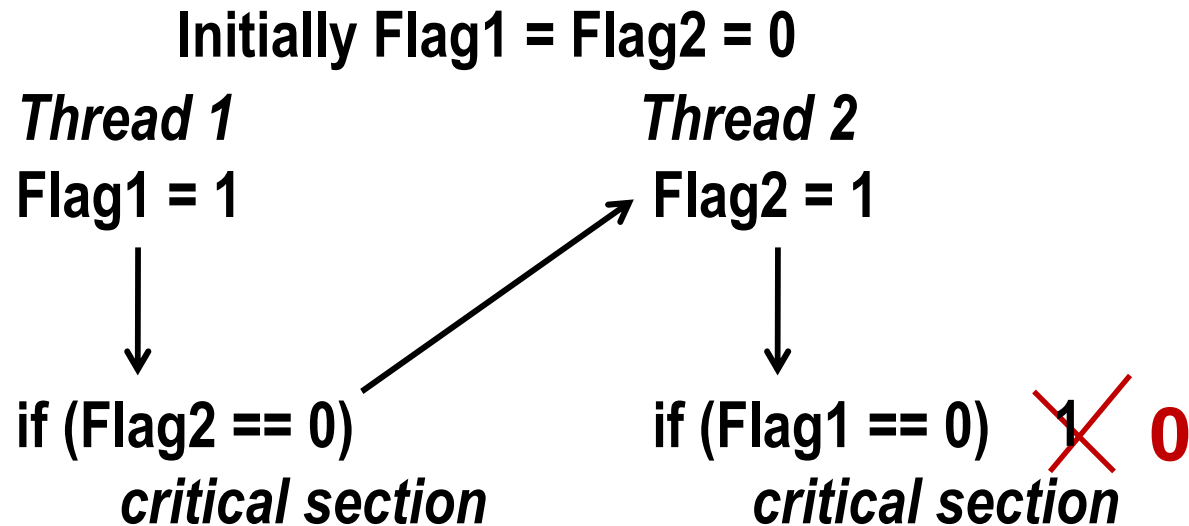
Use locks (or other synch) to restrict interleavings
Second lock must wait until first unlock

SC and Synchronization

- **SC = program order + atomicity for memory accesses**
- **What is a memory access?**
 - **Load, Store**
 - **Synchronization**
 - * **Locks/unlocks, Fetch&Add, Compare&Swap, ...**
 - * **Transaction begin/end like single lock (approximately)**
 - * **System must ensure read-modify-write is “atomic”**

Understanding Program Order – Example 1

- Dekker's algorithm for critical sections



- Can happen on most hardware
 - E.g., store buffers with load bypassing (no caches needed)
- Can happen with most compilers (more later)

Understanding Program Order - Example 2

Initially A = Flag = 0

Thread 1

A = 26;

Flag = 1;

Thread 2

while (Flag != 1) {;}

... = A; ~~26~~ **0**

- Can happen if hardware overlaps/reorders stores or loads
- Can happen with most compilers

Understanding Program Order: Summary

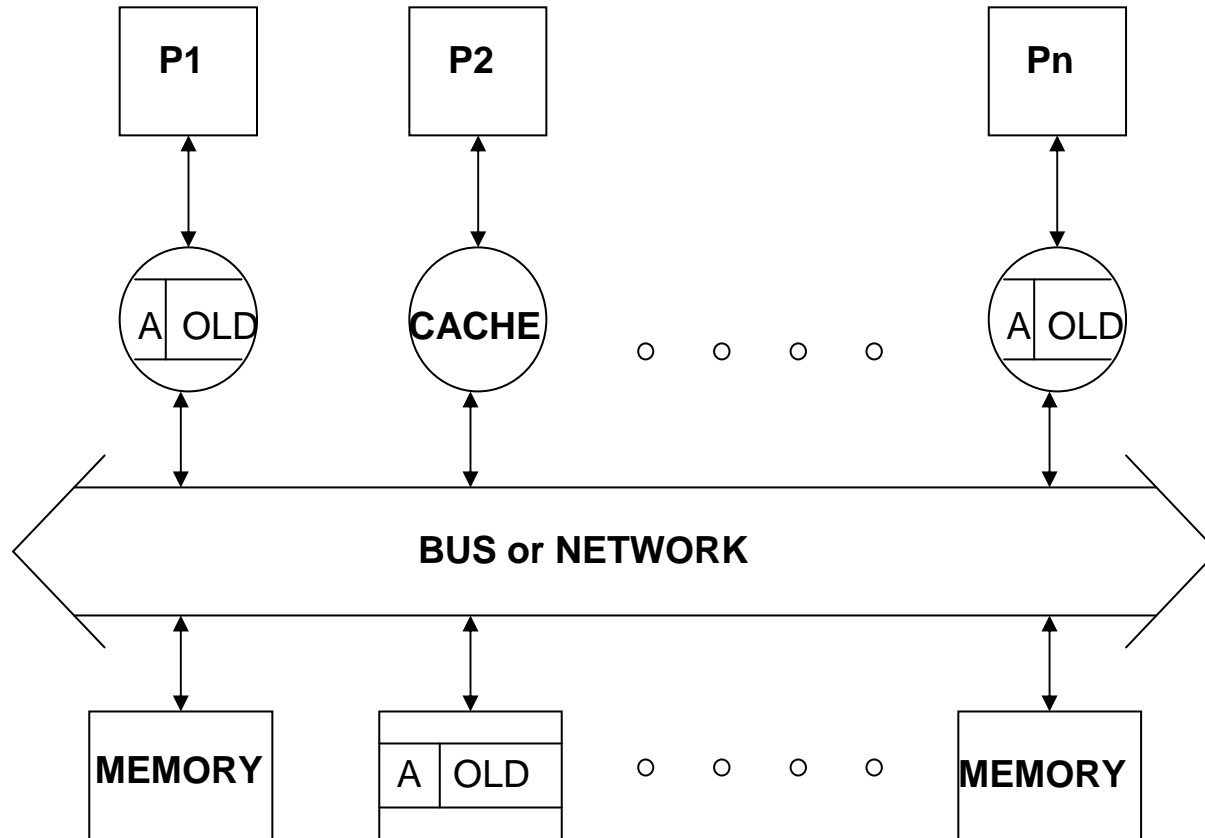
SC limits program order relaxation

- **Write → Read**
- **Write → Write**
- **Read → Read, Write**

Understanding Atomicity

- **Isolation**
 - Nobody sees half-done access; e.g., partial word update
 - Related to access granularity (more later)
- **Serializability**
 - Access appears to occur at the same time to everyone
 - Needs careful handling when multiple copies
 - Focus next

Understanding Atomicity



- A mechanism needed to propagate a write to other copies
 - *Cache coherence protocol*
 - * Invalidate or update old copies in other caches

Understanding Atomicity - Example 1

Initially A = B = C = 0

<i>Thread 1</i>	<i>Thread 2</i>	<i>Thread 3</i>	<i>Thread 4</i>
A = 1;	A = 2;	while (B != 1) {;}	while (B != 1) {;}
B = 1;	C = 1;	while (C != 1) {;}	while (C != 1) {;}
		r1 = A; 1	r2 = A; 1 2

Can happen if updates of A reach Threads 3, 4 in different order

Coherence protocol must serialize writes to same location

(Writes to same location should be seen in same order by all)

Coherence example

Understanding Atomicity - Example 2

Initially A = B = 0

Thread 1

A = 1;

Thread 2

while (A != 1) {;}

B = 1;

Thread 3

while (B != 1) {;}

r1 = A; ~~1~~ 0

Can happen if read returns new value before all copies see it

E.g., Threads 1, 2 share a cache

2 sees 1's write

3 sees 2's write before 1's write

Causality example

Understanding Atomicity – Example 3

Initially $X = Y = 0$

<i>Thread 1</i>	<i>Thread 2</i>	<i>Thread 3</i>	<i>Thread 4</i>
$X=1$	$Y=1$	$r1 = X$ 1	$r3 = Y$ 1
		$r2 = Y$ 0	$r4 = X$ 1 0

Can happen if

Threads 1 and 3 share the same cache, 3 sees 1's write early

Threads 2 and 4 share the same cache, 4 sees 2's write early

Independent reads, independent writes (IRIW) example

SC Summary for Hardware

SC limits

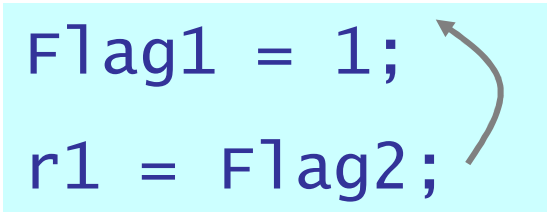
- Program order relaxation:
 - Write → Read
 - Write → Write
 - Read → Read, Write
- Unserialized writes to the same location
- Reading a write before it is seen by all

Compiler issues with sequential consistency

- Limits compiler reordering in addition to hardware reordering.
 - Compiler wants to perform loads early, stores late.
 - Hides latency, as for hardware.
 - E.g. for Dekker's example, if r1 and r2 are used immediately, it's tempting to move up loads:

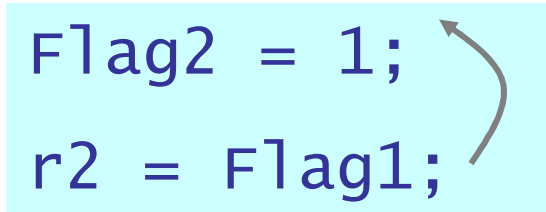
Thread 1:

```
Flag1 = 1;  
r1 = Flag2;
```



Thread 1:

```
Flag2 = 1;  
r2 = Flag1;
```



Cost of restricting compiler movement

- Intuition:
 - Moving loads out of loops is important.
 - Not generally possible with sequential consistency.
 - More important issue than hardware reordering?
- But:
 - Very sensitive to whole program analysis.
 - Empirical results are much more mixed.
 - E.g. Sura et al PPOPP 05, DRFx paper in PLDI '10
- Not comfortable outlawing these optimizations.

```
for (i = ...) {  
    a[i] += *p;  
}
```

reload?



Usability issue with sequential consistency

- Sensitive to memory access granularity:

Thread 1

`x = 300;`

Thread 2

`x = 100;`

- may result in $x = 356$ with sequentially consistent byte accesses.
- *Need to* reason at level of memory accesses.
 - Not programmer meaningful.
- *Want to* reason about interleaving of “communication-free” source code sections.

This Tutorial

- The problem
 - Sequential consistency (SC) is intuitive
 - But performance? And is SC really easy enough?
- **Recent consensus: data-race-free**
 - SC only for good (data-race-free) programs
- Some research and practice pitfalls
- **BUT data-race-free not the full answer**
 - Ultimate performance on current hardware?
 - Undefined race semantics complicate safety, debugging,
- Need new approach
 - Some ongoing research



Real threads programming model (1)

- Two memory accesses **conflict** if they
 - access the same scalar variable*
 - at least one access is a store.
- Two ordinary memory accesses participate in a **data race** if they
 - conflict, and
 - can occur simultaneously
 - i.e. appear as adjacent operations in interleaving.
- A program is **data-race-free** (on a particular input) if no sequentially consistent execution results in a data race.

*to be refined for bit-fields

Real threads programming model (2)


- Sequential consistency only for data-race-free programs!
 - Avoid anything else.
- Data races are prevented by
 - locks (or atomic sections) to restrict interleaving
 - declaring synchronization variables (stay tuned ...)

Alternate data-race-free definition: happens-before

- Memory access *a* happens-before *b* if
 - *a* precedes *b* in program order.
 - *a* and *b* are synchronization operations, and *b* observes the results of *a*, thus enforcing ordering between them.
 - e.g. *a* unlocks *m*, *b* subsequently locks *m*.
 - Or there is a *c* such that *a* happens-before *c* and *c* happens-before *b*.
- Two ordinary memory operations *a* and *b* participate in a data race in a particular execution, if neither
 - *a* happens-before *b*,
 - nor *b* happens-before *a*.
- Set of data-race-free programs usually the same.

Thread 1:

```
lock(m);  
t = x + 1;  
x = t;  
unlock(m)
```



Thread 2:

```
lock(m);  
t = x + 1;  
x = t;  
unlock(m)
```

Data Races

- Are defined in terms of sequentially consistent executions.
- If x and y are initially zero, this does *not* have a data race:

Thread 1
if (x)
 y = 1;

Thread 2
if (y)
 x = 1;

Bit-fields

```
struct { int a:3; int b:2;} x;
```

Thread 1:

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

Thread 2:

```
x.b = 2;
```

- Is this a data race?
- Yes.
 - Real hardware can't update them independently.
 - Stay tuned for illustration of why this matters.
 - Races in C++0x are defined in terms of “memory locations”.
 - “Memory location” is either
 - scalar variable, or
 - contiguous bit-field sequence.

SC for DRF programming model advantages over SC

- Supports important hardware & compiler optimizations.
- DRF restriction → Independence from memory access granularity.
 - Hardware independence.
 - Synchronization-free library calls are atomic.
 - Really a different and better programming model than SC.

Library implemented objects often behave like built-in data

- HashSet operations participate in a data-race if two threads simultaneously access the same HashSet, and one is an update.
- In race-free programs HashSet operations are atomic.
- HashSets may be shared or private.
 - If shared, caller arranges for synchronization.

Synchronization variables

- Java: `volatile`, `java.util.concurrent.atomic`.
- C++0x: `atomic<int>`
- C1x, C++0x: `atomic_int`, `_Atomic(int)`
- Guarantee indivisibility of operations.
- “Don’t count” in determining whether there is a data race:
 - Programs with “races” on synchronization variables are still sequentially consistent.
 - Though there may be “escapes”.
- Dekker’s algorithm “just works” with synchronization variables.

As expressive as races

Double-checked locking:

Wrong!

```
bool x_init;

if (!x_init) {
    l.lock();
    if (!x_init) {
        initialize x;
        x_init = true;
    }
    l.unlock();
}
use x;
```

Double-checked locking:

Correct (C++0x):

```
atomic<bool> x_init;

if (!x_init) {
    l.lock();
    if (!x_init) {
        initialize x;
        x_init = true;
    }
    l.unlock();
}
use x;
```

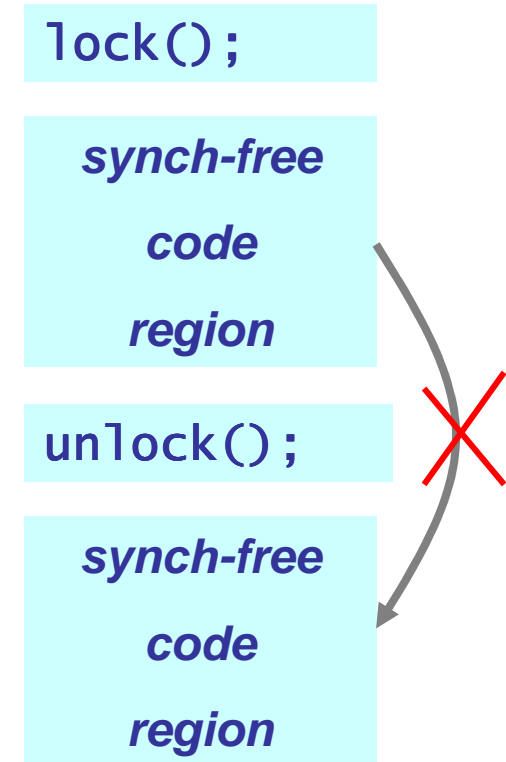
Some variants

C++ draft (C++0x) C draft (C1x)	SC for DRF*, Data races are errors
Java	SC for DRF**, More details later.
Posix threads	SC for drf (sort of, no atomics)
Ada 83	SC for drf (sort of)
OpenMP, Fortran 2008	SC for drf (except atomics, sort of)
.Net	Getting there, we hope ☺

* Except explicitly specified memory ordering. ** Except some j.u.c.atomic.

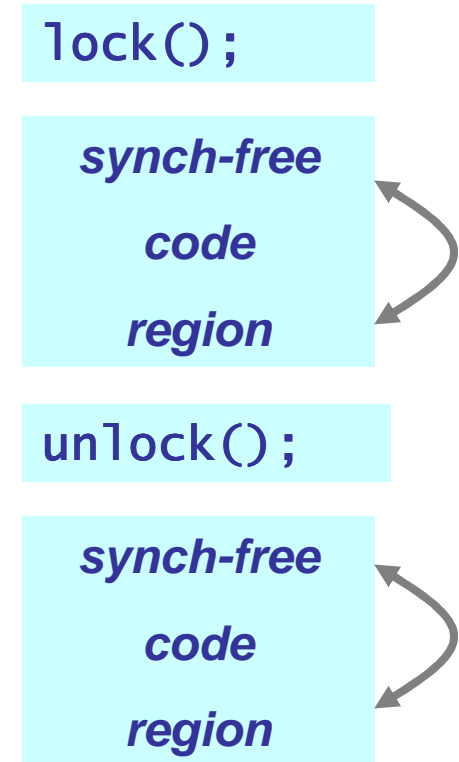
SC for DRF implementation model (1)

- Very restricted reordering of memory operations around synchronization operations:
 - Compiler either understands these, or treats them as opaque, potentially updating any location.
 - Synchronization operations include instructions to limit or prevent hardware reordering (“memory fences”).
 - e.g. lock acquisition, release, atomic store, might contain memory fences.



SC for DRF implementation model (2)

- Code may be reordered between synchronization operations.
 - Another thread can only tell if it accesses the same data between reordered operations.
 - Such an access would be a data race.
- If data races are *disallowed* (e.g. Posix, Ada, C++0x, *not* Java), compiler may assume that variables don't change asynchronously.



Possible effect of “no asynchronous changes” compiler assumption:

```
unsigned x;  
  
If (x < 3) {  
    ... // async x change  
    switch(x) {  
        case 0: ...  
        case 1: ...  
        case 2: ...  
    }  
}
```

- Assume switch statement compiled as branch table.
- May assume x is in range.
- Asynchronous change to x causes wild branch.
 - Not just wrong value.
- Rare, but possible in current compilers?

SC for DRF implementation constraints: Compilers

- Compilers must not introduce visible data races.
- Unfortunately, sequentially correct optimization can.
- Most current compilers do. ☹️
- Some examples:

Introducing Races: Struct field update

```
struct {char a; int b:5; int c:11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
x.c = 1;
```

Thread2:

```
x.d = 1;
```

commonly implemented as

Thread1:

```
tmp = x;
```

```
tmp.c = 1;
```

```
x = tmp;
```

Thread2:

```
x.d = 1;
```


Struct field update (contd 1):

```
struct {char a; int b:5; int c:11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
tmp = x;  
tmp.c = 1;  
x = tmp;
```

Thread2:

```
x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 0;

Struct field update (contd 2):

```
struct {char a; int b:5; int c:11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
➔ tmp = x;  
tmp.c = 1;  
x = tmp;
```

Thread2:

```
x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 0;

tmp: a: 0; b: 0; c: 0; d: 0;

Struct field update (contd 3):

```
struct {char a; int b:5; int c:11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
tmp = x;  
➔ tmp.c = 1;  
x = tmp;
```

Thread2:

```
x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 0;

tmp: a: 0; b: 0; c: 1; d: 0;

Struct field update (contd 4):

```
struct {char a; int b:5; int c:11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
tmp = x;  
➔ tmp.c = 1;  
x = tmp;
```

Thread2:

```
➔ x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 1;

tmp: a: 0; b: 0; c: 1; d: 0;

Struct field update (contd 5):

```
struct {char a; int b:5; int c:11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
tmp = x;  
➔ tmp.c = 1;  
x = tmp;
```

Thread2:

```
x.d = 1;
```



x: a: 0; b: 0; c: 0; d: 1;

tmp: a: 0; b: 0; c: 1; d: 0;

Struct field update (contd 6):

```
struct {char a; int b:5; int c:11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
tmp = x;  
tmp.c = 1;  
x = tmp;
```

Thread2:

```
x.d = 1;
```



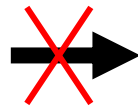
x: a: 0; b: 0; c: 1; d: 0;

- This behavior is currently allowed and common.
 - No two fields can safely be independently updated.

Introducing Races: Register Promotion 1

[g is global]

```
for(...) {  
    if(mt) lock();  
    use/update g;  
    if(mt) unlock();  
}
```



```
r = g;  
for(...) {  
    if(mt) {  
        g = r; lock(); r = g;  
    }  
    use/update r instead of g;  
    if(mt) {  
        g = r; unlock(); r = g;  
    }  
}  
g = r;
```

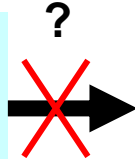
Introducing Races: Register Promotion 2

```
int count;    // global, possibly shared
...
for (p = q; p != 0; p = p -> next)
    if (p -> data > 0) ++count;
```



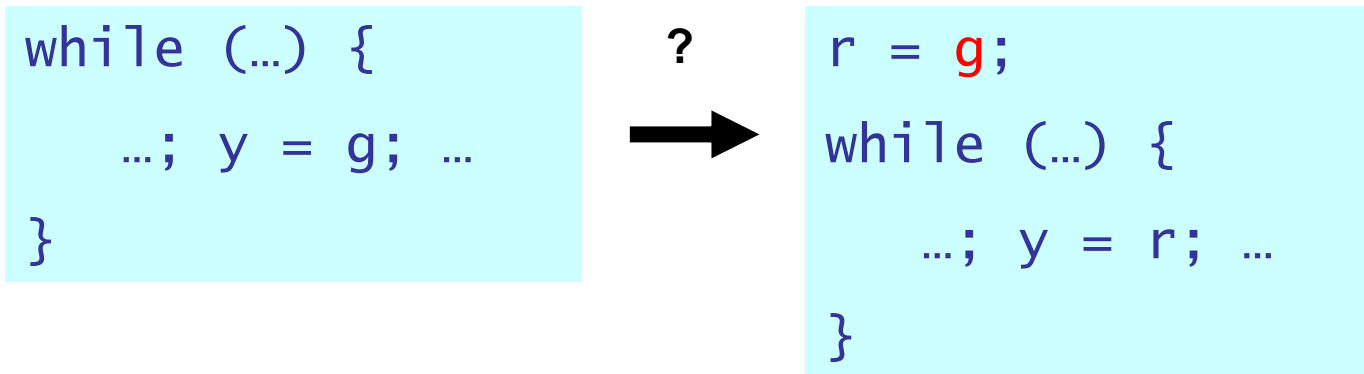
```
int count;    // global, possibly shared
...
reg = count;
for (p = q; p != 0; p = p -> next)
    if (p -> data > 0) ++reg;
count = reg;  // may spuriously assign to count
```


Introducing data races: Potentially infinite loops

```
while (...) { a[i] = 1; }  
x = 1;    
x = 1;   
while (...) { a[i] = 1; }
```

- `x` may not be accessed!
- Prohibited in Java.
- Allowed by giving undefined behavior to such infinite loops in C++0x.
(Controversial.)

Some added data races are sometimes OK



- May add data race if loop iterates 0 times.
- *Disallowed* as C++0x source transformation.
- But racing load with unused result is benign in most target ISAs, and in Java.
- Generally allowed as compiler transformation.
- Incorrect if hardware were to detect data races.
 - Use prefetch instead, or avoid load in zero-iteration case.

Synchronization primitives need careful definition

- `lock()`, `unlock()`, like all other synchronization accesses, cannot be visibly reordered w.r.t. other accesses.
- But moving memory accesses *into* critical section is not detectable by a data-race-free program using only `lock()`, `unlock()`. (Cf. Boehm, PPOPP 07)
- Implementations that allow one-way reordering are common, and faster.
- But what about `trylock()/timedlock()`?

Trylock restricts `lock()` reordering:

- Some really awful code:

Thread 1: Thread 2: *Don't try this at home!!*

```
x = 42;
lock();
```

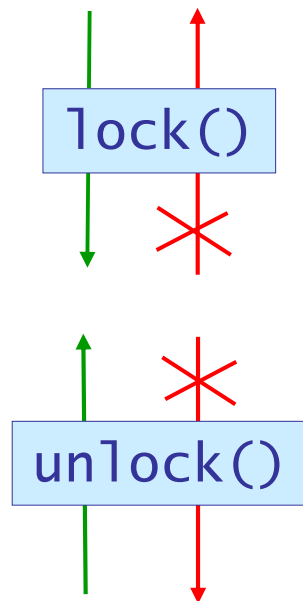
```
while (trylock() == SUCCESS)
    unlock();
assert (x == 42);
```

- Can't move `x = 42` into “critical section”!
- Note:
 - Example requires tweaking to be pthreads-compliant.
 - In some happens-before formulations, this has a data race

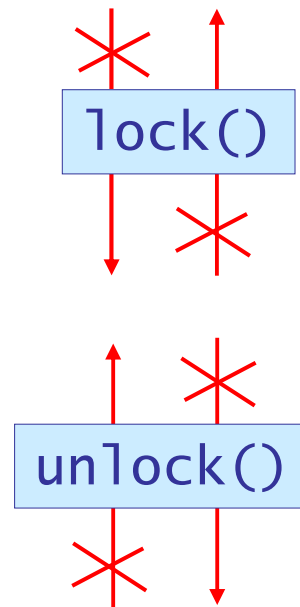
With Trylock: Critical section reordering?

- Reordering of memory operations with respect to critical sections:

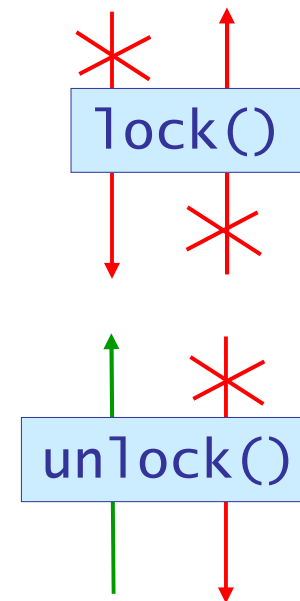
Expected (& Java):



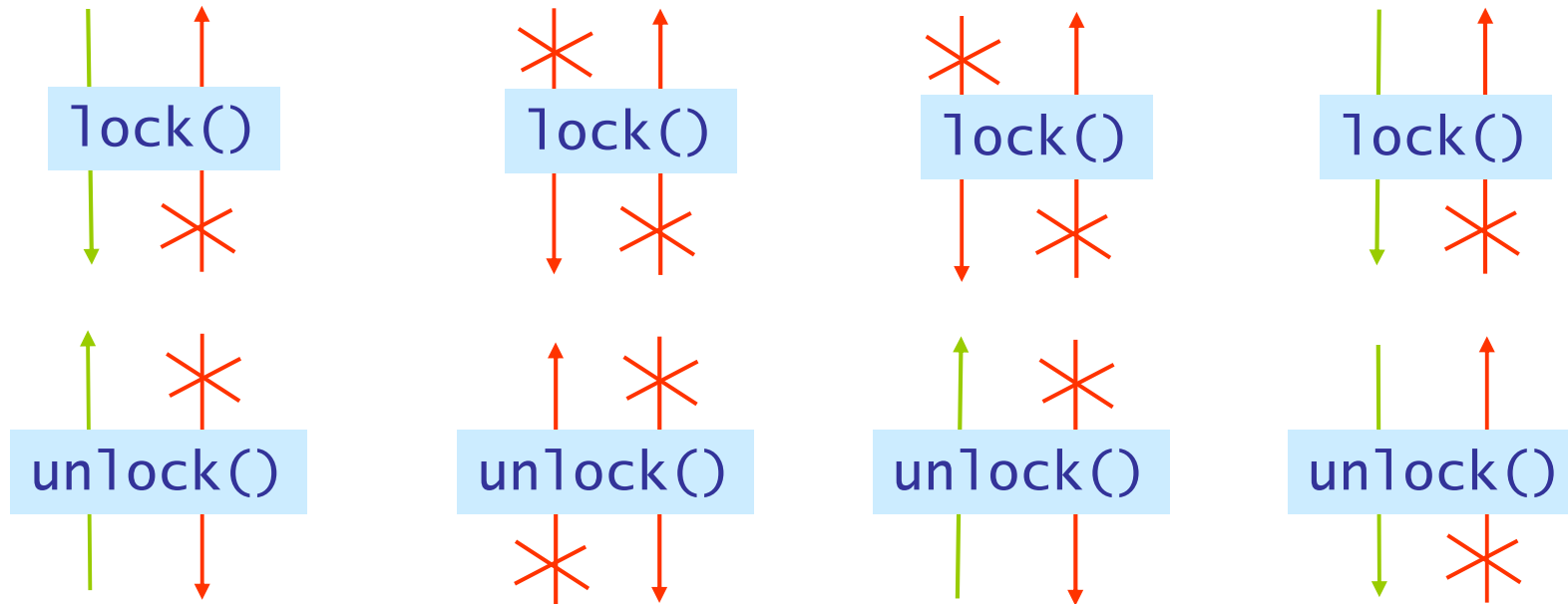
Naïve pthreads:



Optimized pthreads



Some open source pthread lock implementations (2006):



[technically incorrect]

NPTL

{Alpha, PowerPC}

{mutex, spin}

[Correct, slow]

NPTL

Itanium (&X86)

mutex

[Correct]

NPTL

{ Itanium, X86 }

spin

[Incorrect]

FreeBSD

Itanium

spin

SC for DRF Implementation: Hardware

- **Hardware models took different trajectory from DRF**
- **Largely motivated by hardware optimizations**
- **Next**
 - **Brief overview of hardware models**
 - **Mapping DRF to hardware models**
 - **Limitations with current hardware**
 - **Problems with fences**

Classification for Relaxed Hardware Models

- Typically described as hardware optimizations
 - Program order relaxation:
 - Write → Read
 - Write → Write
 - Read → Read, Write
 - Read others' write early
 - Read own write early
- All models provide
 - Some form of write serialization
 - Some form of single thread data/control dependences
 - Subtleties for models that relax Read → Read (not covered here)
 - Safety net (e.g., fences, memory barriers) to prohibit optimization
 - Atomic RMW, with some subtle differences (not covered here)
- Many subtleties, ambiguities not covered here

Major Relaxed Hardware Models*

Relaxation	W →R Order	W →W Order	R →RW Order	Read Others' Write Early	Read Own Write Early	Safety Net
IBM 370	✓					serialization instructions
TSO (x86)	✓				✓	RMW, fences
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	synchronization
RCsc	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	various MEMBARs
Itanium	✓	✓	✓		✓	LD.acq, ST.rel, mf
PowerPC	✓	✓	✓	✓	✓	sync, lwsync

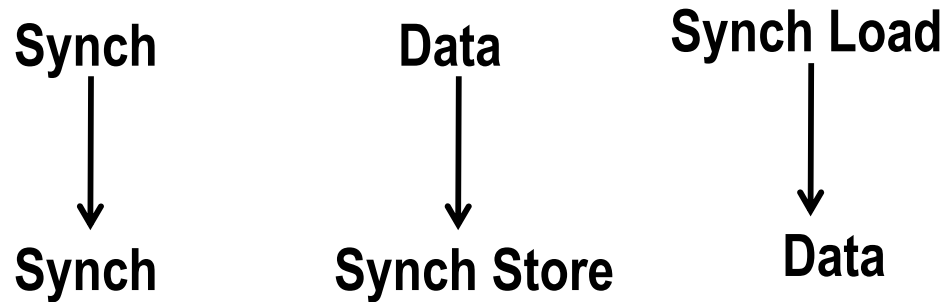
* HP PA-RISC and MIPS R10000 processors potentially SC (depending on rest of system)

Mapping DRF to Hardware Models

- **Use safety net to prohibit hardware model optimizations**
 - But most focus on program order, not atomicity
 - DRF requires SC synch \Rightarrow program order + atomicity
 - Many subtle issues
 - For some hardware, fences not sufficient
 - For others, fences over-constrain
 - Unintended hardware/software mismatch
- **Don't forget access granularity**

Mapping DRF to Hardware: Program Order

- For program order, sufficient to enforce



- E.g., TSO/x86: need care for **Synch Store** → **Synch Load**
 - Insert mfence/membar after each **Synch Store**
 - Or convert **Synch Store** to RMW (xchg)
- E.g., PowerPC: also need care for **Synch Load** → **X**
 - Insert additional fence after each **Synch Load**
(many subtleties not covered here)

Mapping DRF to Hardware: Atomicity

Independent reads, independent writes (IRIW):

X and Y are synch and initially 0

<i>Thread 1</i>	<i>Thread 2</i>	<i>Thread 3</i>	<i>Thread 4</i>
X=1	Y=1	r1 = X 1	r3 = Y 1
		fence	fence
		r2 = Y 0	r4 = X 1 0

- **Must prohibit reading others' write too early**
 - Can happen if threads share caches (see previous slide)
 - Some hardware specs were vague on IRIW
- **But programmers don't use IRIW!**
 - Why pay cost of SC if programmers won't use it?
- **Much pressure to change SC semantics for synch**

Change SC Synchronisation Requirement of DRF?

- **Much pressure to change SC for synchronisation to allow IRIW opt**
 - **Showed IRIW opt gives unacceptable behavior for other codes**
 - **Violates composition of write causality + coherence**
 - **No simple alternate model found**
- **Problem with IRIW**
 - **Mainly hardware-software mismatch**
 - **Ensuring we don't read synchronisation writes too early is not hard**
 - **Ensure for all stores or**
 - **Ensure only for clearly marked synchronisation stores (e.g., xchg for x86)**
 - **Be careful with fences**
- **Current status**
 - **X86 spec is now changed so all stores atomic**
 - **PowerPC still requires heavyweight fence**

More Problems with Fences

- **IRIW discussion showed fences often under-constrained**
- **Next: Fences often over-constrained**

Fences May Over-constrain (1 of 2)

- **Fences typically order all prior LD/ST w.r.t. all later LD/ST**
 - We want ordering w.r.t. specific synchronization accesses
 - s is a synchronization variable**
 - x = 1;**
 - s = 2; // includes fence**
 - r1 = y;**
 - Unnecessarily prevents reordering of **x = 1** and **r1 = y;**
- **One solution: distinguish synch LD/ST in hardware**

Fences May Over-constrain (2 of 2)

- Don't really need program order

Thread1

A = 26;

B = 90;

Flag = 1;

Thread2

while (Flag != 1) {;

... = B;

... = A;

- Can postpone stores of A, B to load, Flag, 1
- Can postpone stores of A, B to loads of A, B
- Can exploit last two observations with
 - Lazy invalidations
 - Lazy release consistency on software DSMs

Fences Summary

- Memory ordering prescribes interactions **between** threads
- Most fences constrain interactions **within** a thread
 - ⇒ Fundamental mismatch

Don't Forget Access Granularity

- Data-race-freedom guaranteed for byte (or higher) granularity
- `x.c = 'a'`; may not visibly read and rewrite adjacent fields
- Byte stores must be implemented with
 - Byte store instruction, or
 - Atomic read-modify-write
 - Typically expensive on multiprocessors

DRF Summary So Far

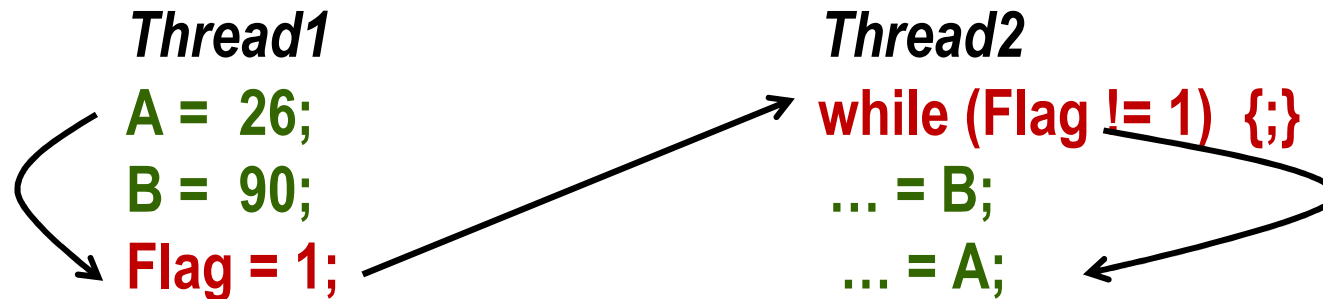
- **SC for DRF programs is minimal programming model**
- **But some hardware-software mismatch**
 - Many attempts for alternatives, but unsuccessful
- **But what about programs with races?**
 - Key open problem in language semantics

Initial Alternatives to DRF (1 of 2)

- **Use hardware-centric models**
 - **But not suitable for programming models**
 - **No common model covers all desirable optimizations**
- **Sequential Consistency**
 - **Use hardware speculation, prefetching to overcome perf limits**
 - **But complex hardware (interface must last through trends)**
 - **But compilers still limited by SC (previous slides)**
 - **But SC not good enough model (previous slides)**

Initial Alternatives to DRF (2 of 2)

- Happens-before consistency
 - If X, Y conflict & X happens-before Y, then X executes before Y



- But does not give SC for data-race-free programs

Initially A=B=0

Thread1

If (A==1) B=1

Thread2

if (B==1) A=1

Happens-before consistency allows A=B=1

- More alternatives coming up later

This Tutorial

- **The problem**
 - Sequential consistency (SC) is intuitive
 - But performance? And is SC really easy enough?
- **Recent consensus: data-race-free**
 - SC only for good (data-race-free) programs
- **Some research and practice pitfalls**
- **BUT data-race-free not the full answer**
 - Ultimate performance on current hardware?
 - Undefined race semantics complicate safety, debugging,
- **Need new approach**
 - Some ongoing research



Research and practice pitfalls

- Our goal is *not* to embarrass authors of prior work.
- Much of what we cite predates clarity about memory models, or was intended for sequential code.
- And we might even be confused about some of this.
- But: **Beware of “established practice” in this field!**

Research and practice pitfalls

- We've already seen a few mis-steps:
 - Posix (since 1995), Ada 83, ... are very clear that data races are disallowed.
 - But we found substantial disagreement as to what a data race is.
 - Including among committee members, researchers, ...
 - Confusion about lock ordering semantics
 - Similar issues with e.g. gcc `__sync` operations.
 - Nobody gets explicit memory ordering right?
 - Memory fences are pervasive.
 - But are they the right mechanism?
- Some more:

Uncertainty about hardware memory models

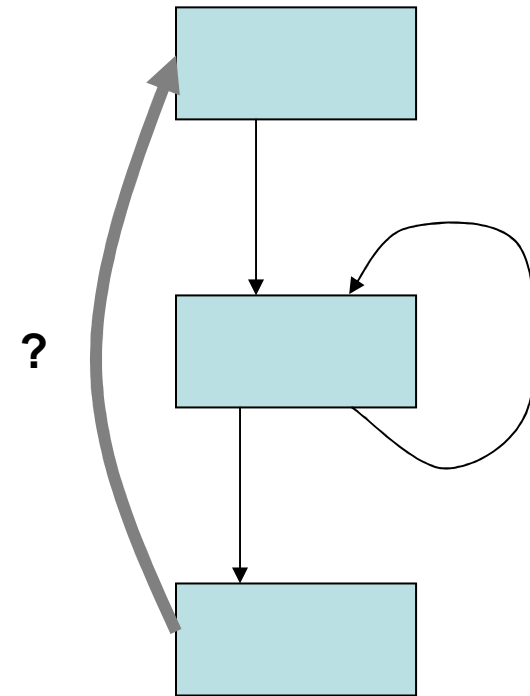
- Some processor vendors published precise memory models.
- Others did not (e.g. X86 before 2007).
- Hard to fix with multiple processor & chipset vendors (e.g. X86, MIPS)
- Recent efforts by academics, notably Peter Sewell's group, help.
- Don't believe e.g. pre-2006 x86 hardware specs! (If you figure out what they mean.)

Sequential consistency by fence insertion

- Many papers, starting with Shasha & Snir, April 88, TOPLAS, look at sequential consistency enforcement with fences, but
- Too little attention was paid to whether real hardware fences could do this at all.
 - Unclear for X86 and PowerPC until 2007+.
 - Fences + ordinary ld/st insufficient for Itanium.
 - Need st.rel for store atomicity (serializability).

Infinite loops

- Published compiler optimizations (e.g. “Lazy Code Motion”, by Knoop et al, PLDI '92 most influential paper) usually consider only paths from start node to end node in CFG?
 - Not always safe in Java.
 - Issue not addressed in literature?



Assuming that libraries are “thread-safe” or “thread-unsafe”.

- Most libraries should ensure that:
 - Simultaneous “read” accesses are safe.
 - No hidden updates, e.g. unprotected reference counts!
 - Simultaneous accesses to logically disjoint objects are safe.
 - No hidden shared data structures.
- Client locks when there is a logical data race.
 - Just like built-in scalar types.
- Minimal overhead for thread-local use.
- This is not “thread-safe” by most definitions.
- This is not completely “thread-unsafe”.

Assuming sequential consistency when unwarranted

- Common in programming language research literature.
- One example:
 - “Atomic set serializability” (Hammer et al., 2008) looks for atomicity bugs by finding certain access *sequences* .
 - Detects a subset of what we call data races.
 - *Accesses do not form a sequence* unless they are data-race-free (our definition).
- But sequential consistency should continue to play an important analysis. Can verify safety by:
 - Ensuring DRF using SC analysis.
 - Verifying properties using SC analysis.

Sequential consistency in compiler program analysis

- Program analysis based purely on sequential consistency is unsound for Java:
 - Actual executions allow more behaviors.
- Program analysis based on sequential consistency is sound, but weak for C++0x:
 - Programs without races are SC.
 - Programs with races may be “miscompiled”.
 - But the absence of races provides a lot of additional information:
 - e.g. atomicity of sync-free regions.

Assuming shared objects must be accessed in critical sections

Thread 1:

```
tmp1 = new foo();  
tmp1.value = 17;  
q.put(tmp1);
```

Thread 2:

```
tmp2 = q.take();  
... tmp2.value ...
```

- E.g. ignoring “privatization safety” for transactional memory.

Incorrect modeling of synchronization primitives

- Condition variable waits may return spuriously in Java 5+/pthreads/C++0x.
- Cannot assume that `wait()` blocks until `notify()`. (E.g. MHP analysis)
- For partial correctness purposes, `wait()` is equivalent to `unlock(); lock()`.
- Analyzing `wait()/notify()` requires reasoning about underlying predicate!

Dubious definition of synchronization primitives

- Example: Boost threads defines thread destructor to detach associated thread.

```
int f () {  
    int x, y;  
    thread t(compute, &x); // compute value of x  
    y = do_something_else();  
    t.join();  
    return x + y;  
}
```

- What if `do_something_else()` throws?

Some common features don't combine well

- Detached threads (e.g. Posix)
 - Run until they finish or process ends
 - e.g. a service for a library
- C++ destructors for static objects
 - Called just before process shutdown
- Detached threads run with invalid global variables just before shutdown?!

This Tutorial

- The problem
 - Sequential consistency (SC) is intuitive
 - But performance? And is SC really easy enough?
- Recent consensus: data-race-free
 - SC only for good (data-race-free) programs
- Some research and practice pitfalls
- **BUT data-race-free not the full answer**
 - Ultimate performance on current hardware?
 - Undefined race semantics complicate safety, debugging,
- Need new approach
 - Some ongoing research



But data-race-free not the full answer

- Sequential Consistency, even for data-race-free programs, may be expensive on current hardware.
 - Too expensive for some highly tuned performance critical applications?
- Undefined semantics for data races are not always acceptable.
 - For debugging.
 - For sand-boxed code.

Sequential consistency may be expensive

- We require sequential consistency for volatile/atomic operations.
- Many lock-free algorithms don't need full sequential consistency, e.g.
 - Simple event counter, read only after all threads complete.
 - Need atomic increment. Visibility to other threads unimportant.
 - “Done” flag can usually become visible late.
 - PLDI 05 Parallel Sieve or Eratosthenes example is immune to all memory access reordering.

Good reasons for enforcing sequential consistency anyway:

- It's really hard to explicitly specify memory visibility ordering correctly.
- In our experience, such specifications are usually wrong.
- It's hard to keep violations of sequential consistency local:
 - Library routines using non-sequentially consistent behavior internally are often visibly not sequentially consistent.

But can we afford that much sequential consistency?

- Answer varies depending on hardware and algorithm.
 - On X86, major cost is that atomic stores turn into atomic XCHG instruction.
 - (1 cycle → dozens of cycles)
 - Often negligible compared to coherence misses
 - On PowerPC, atomic loads require (?) heavy weight fence.
 - Probably less acceptable.
 - Most others somewhere in between.

Bottom Line

- Languages grow “loopholes” to avoid overhead for sequentially consistent atomics.
- For C++0x/C1x, the “loopholes” drove the memory model specification
- Possibly a temporary issue?
- C++0x/C1x: Explicit `memory_order_` specifications.
- Java: A growing list of more ad hoc exceptions.
 - Worse alternative: growing use of unsafe code.

C++0x Approach(1)

- Pairs of atomic operations cannot form a data race.
- Operations that do not specify `memory_order_seq_cst` (the default) are not guaranteed to execute in a single total order.
- A `memory_order_release` store still happens-before a `memory_order_acquire` load that reads the value.
- Atomic load may see any store that doesn't happen after it, and is not hidden by another store that "happens between" the two.

C++0x Approach(2)

- A `memory_order_relaxed` operation also drops that requirement.
- But operations on a single variable still behave as though they were interleaved (cache coherent).
- A `memory_order_consume` operation behaves like `memory_order_acquire`, but only with respect to subsequent data-dependent operations.
- (And there are explicit fences if you really want them.)

Dekker's with C++0x low-level atomics

```
atomic<int> x, y;
```

Thread 1:

```
x.store(1, memory_order_release);  
r1 = y.load(memory_order_acquire);
```

Thread 2:

```
y.store(1, memory_order_release);  
r2 = x.load(memory_order_acquire);
```

- $r1 = r2 = 0$ is possible outcome.
- No cross-thread happens-before relationships → no constraints.
- Same as `memory_order_relaxed`.
- Allows ordinary `MOV` on X86, much cheaper on PowerPC.

C++0x fine-tuned double-checked locking

```
atomic<bool> x_init;

if (!x_init.load(memory_order_acquire) {
    l.lock();
    if (!x_init.load(memory_order_relaxed) {
        initialize x;
        x_init.store(true, memory_order_release);
    }
    l.unlock();
}
use x;
```

Non-sequentially-consistent constructs in Java

- `j.u.c.atomic.AtomicInteger.lazySet()` is roughly equivalent to `store(..., memory_order_release)`
- `weakCompareAndSet()` behaves roughly like `memory_order_relaxed`.
- Ordinary variables can be used roughly like `memory_order_relaxed`.
 - But even weaker: No cache coherence:

```
x = 1;
```

```
x = 2;
```

```
r1 = x;
```

```
r2 = x;
```

- Allows `r1 = 2` and `r2 = 1`.

Semantics for Programs with Data Races

- DRF doesn't define semantics of programs with data races
 - Legal for computer to catch fire
- How to debug programs with data races?
- How to deal with safe languages like Java?
 - Java cannot have undefined semantics for ANY program
 - Must limit damage from data races in untrusted code
 - Goal: Safety w/ maximum system flexibility
 - Problem: “safety, limited damage” w/ threads very vague

Java Memory Model Highlights (1 of 5)

- Quick consensus for SC for DRF programs
- About 5 years for semantics for data races!

Initially $X=Y=0$

	<i>Thread 1</i>	<i>Thread 2</i>
speculate 42 →	$r1 = X$	$r2 = Y$
	$Y = r1$	$X = r2$

Can $X = Y = r1 = r2 = 42$? ~~42~~ <your bank password>?

Out-of-thin-air example

- Data races produce causality loops!
 - Definition of a causality loop was surprisingly hard
 - Common compiler optimizations seem to violate “causality”

Java Memory Model Highlights (2 of 5)

- **Common compiler optimizations violate causality**

Initially X=0, Y=1

Thread 1

r1 = X

r2 = X

if (r1==r2) Y=2

Thread 2

r3 = Y

X = r3

- **r1=r2=r3=2 seems to violate causality**
- **But compiler can eliminate redundant reads, move Y=2**
- **Challenge: Allow above but not thin-air causality loop**

Java Memory Model Highlights (3 of 5)

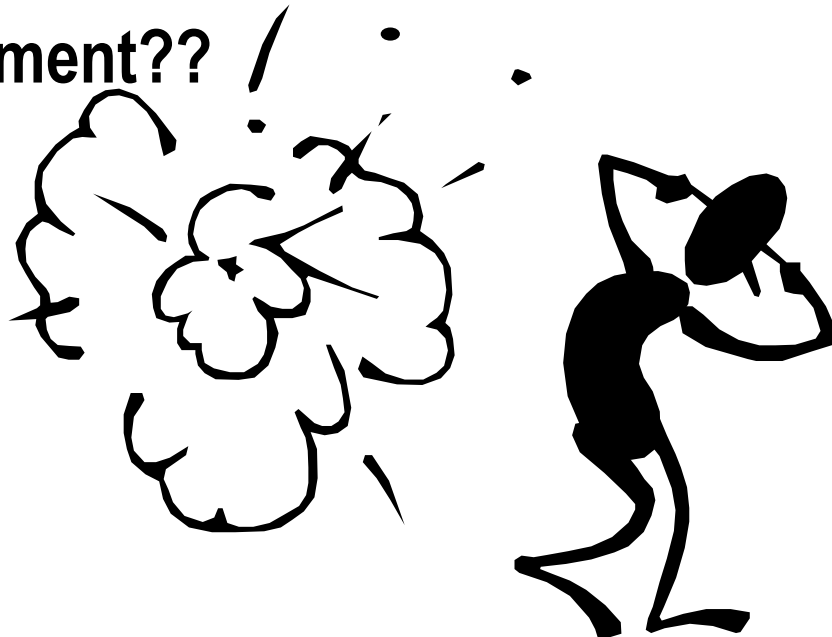
- **Key insights:**
 - **Out-of-thin-air value comes from “thin air”**
 - **Values in other causality violation come from some reasonable execution of program**
 - ⇒ **Somehow allow only speculative values from executions that are somehow reasonable**
- **Took 5 years!**

Java Memory Model Highlights (4 of 5)

- **Non-operational spec: given an execution, is it legal?**
 - Commit several accesses at a time until all committed
- **New commit candidates from well-formed executions containing currently committed accesses**
- **But what is well-formed?**
 - SC is one possibility, but allows some controversial executions
 - Final choice
 - Happens-before consistent
 - Uncommitted read does not return value of a racing write
 - Always returns write that happens-before it

Java Memory Model Highlights (5 of 5)

- **Problem: Incredibly complex!**
- **Problem: Adding synch allows more legal values**
 - Gave some surprising but acceptable behaviors
- **Worse: Aspinall & Sevcik discovered a bug [VAMP'07]!**
 - Invalidates key theorem for reordering independent instructions
- **Fix causality treatment??**



Not me!!!

Lessons Learned from Java and C++

- **SC for data-race-free minimal baseline**
 - **Specifying semantics for programs with data races is HARD**
 - **But “no semantics for data races” also has problems**
 - Not an option for safe languages; debugging; ...
 - **Hardware-software mismatch for some code**
 - **“Simple” optimizations have unintended consequences**
- ⇒ **State-of-the-art is fundamentally broken**

Lessons Learned from Java and C++

- **SC for data-race-free minimal baseline**
 - **Specifying semantics for programs with data races is HARD**
 - But “no semantics for data races” also has problems
 - Not an option for safe languages; debugging; ...
 - **Hardware-software mismatch for some code**
 - “Simple” optimizations have unintended consequences
- ⇒ **State-of-the-art is fundamentally broken**
- **Next: two ongoing research approaches that we are (separately) working on**
 - Final solution may be a mix of these, one of these, or neither

Lessons Learned from Java and C++

- **SC for data-race-free minimal baseline**
 - **Specifying semantics for programs with data races is HARD**
 - **But “no semantics for data races” also has problems**
 - Not an option for safe languages; debugging; ...
 - **Hardware-software mismatch for some code**
 - **“Simple” optimizations have unintended consequences**
- ⇒ **State-of-the-art is fundamentally broken**

Lessons Learned

- SC for data-race-free minimal baseline
 - Specifying semantics for programs with data races is HARD
 - But “no semantics for data races” also has problems
 - Not an option for safe languages; debugging; correctness checking tools
 - Hardware-software mismatch for some code
 - “Simple” optimizations have unintended consequences
- ⇒ State-of-the-art is fundamentally broken

Lessons Learned

- SC for data-race-free minimal baseline
 - Specifying semantics for programs with data races is HARD
 - But “no semantics for data races” also has problems
 - **Banish wild shared-memory!**
 - Not an option for safe languages; debugging; correctness checking tools
 - Hardware-software mismatch for some code
 - “Simple” optimizations have unintended consequences
- ⇒ State-of-the-art is fundamentally broken

Lessons Learned

- SC for data-race-free minimal baseline
- Specifying semantics for programs with data races is HARD
 - But “no semantics for data races” also has problems

Banish wild shared-memory!

- Hardware-software mismatch for some code
 - “Simple” optimizations have unintended consequences

⇒ State-of-the-art is fundamentally broken

- **We need**
 - Higher-level disciplined models that **enforce** discipline
 - Hardware co-designed with high-level models

Lessons Learned

- SC for data-race-free minimal baseline
- Specifying semantics for programs with data races is HARD
 - But “no semantics for data races” also has problems

Banish wild shared-memory!

- Hardware-software mismatch for some code
 - “Simple” optimizations have unintended consequences

⇒ State-of-the-art is fundamentally broken

- **We need**
 - Higher-level disciplined models that **enforce** discipline
 - Deterministic Parallel Java [V. Adve et al.]**
 - Hardware co-designed with high-level models
 - DeNovo hardware [S. Adve et al.]**

Key: What Discipline, How to Enforce?

- **Obvious discipline: Data-race-free**
 - Enforcement: Ideally, language prohibits by design
Else, runtime catches as exception
- **But even data-race-free parallel programs are too hard**
 - Multiple interleavings due to unordered synchronization (or races)
 - Makes reasoning and testing hard
- **But many algorithms are deterministic**
 - Fixed input gives fixed output
 - Standard model for sequential programs
 - Also holds for many transformative parallel programs

**Why write such an algorithm in non-deterministic style,
then struggle to understand and control its behavior?**

Deterministic-by-Default Programming Model

- **Parallel programs should be deterministic-by-default**
 - Sequential semantics (easier than SC!)
- **If non-determinism is needed**
 - Should be explicitly requested
 - Should be isolated from deterministic parts
- **Enforcement:**
 - Ideally, language prohibits by design
 - Else, runtime

State-of-the-art

- **Many deterministic languages today**
 - Functional, pure data parallel, some domain-specific, ...
 - Much recent work on runtime, library-based approaches
- **Our work: Language approach for modern O-O methods**
 - Deterministic Parallel Java (DPJ) [V. Adve et al.]

Deterministic Parallel Java (DPJ)

- **Object-oriented type and effect system**
 - Use “named” **regions** to partition the heap
 - Annotate methods with **effect** summaries: regions read or written
 - **If program type-checks, guaranteed deterministic**
 - * Simple, modular compiler checking
 - * No run-time checks today, may add in future
 - Side benefit: **regions, effects are valuable documentation**
- **Extended sequential subset of Java (DPC++ ongoing)**
 - Initial evaluation for expressivity, performance [Oops1a09]
 - Integrating disciplined non-determinism
 - Encapsulating frameworks and unchecked code
 - Semi-automatic tool for effect annotations [ASE09]

DeNovo Hardware Project [HotPar'10]

- **Design hardware to exploit disciplined parallelism**
 - **Simpler hardware**
 - **Scalable performance**
 - **Power/energy efficiency**
- **Working with DPJ as example disciplined model**
 - **Exploit data-race-freedom, region/effect information**
 - * **Simpler coherence**
 - * **Efficient communication: point to point, bulk, ...**
 - * **Efficient data layout: region vs. cache line centric memory**
 - * **Best of message passing and shared memory**

Dynamic race avoidance

- **A less drastic alternative:**
 - Stay with (more or less) existing programming languages.
 - Outlaw data races everywhere, even in Java.
 - Detect all violations.
 - Raise exception so race outcome cannot be observed.
 - No need to specify race semantics:
 - SC for data-race-free suffices.

Making dynamic race avoidance real

- We know how to precisely detect data races (e.g. Goldilocks (Elmas et al., PLDI 07), FastTrack (Flanagan & Freund, PLDI 09) work).
- Slow for always-on, large worst-case space overhead.
- Alternative:
 - Don't detect all races. (Detect as many as possible for debugging purposes.)
 - But guarantee at least SC if race is not detected.
 - DRFx (Marino et al., PLDI 10): Guarantee only SC.
 - Conflict Exceptions (Lucia et al., ISCA 10): Guarantee also atomicity for synchronization-free-regions.

PL semantics with dynamic data-race detection

- **Programs with data-races may abort.**
- **Programs that don't abort have (at least) sequentially consistent semantics.**

Personal opinion

1. Sequential consistency for data-race-free programs, with race detection

is a much better programming model than

2. Sequential consistency

Let's work on the former, not the latter!

Conclusions

- **Increasing consensus for “sequential consistency for data-race-free” as the fundamental model.**
- **This is a big improvement, but:**
 - This probably reflects existing software practice better than existing research.
 - This doesn't sufficiently solve the problem for Java.
 - It exposes hardware/software mismatches.
- **Less confusion, but still enough.**



Open Issues

- **We need a better story for shared variable semantics in languages like Java!**
- **We need a better debugging story for all programming languages.**
- **We need to avoid choice between poor performance or incomprehensible memory ordering primitives.**
 - **Better hardware?**