Semantics and tools for low-level concurrent programming
Semantics and tools for low-level concurrent programming

Assembler is *has-been*. Why should we care?
Compilers vs. programmers
Compilers vs. programmers

Compilers and programmers should cooperate, don't they?
Constant propagation  (an optimising compiler breaks your program)

A simple and innocent looking optimization:

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

```c
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 (y == 1)
print x
```

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

Intuitively, this program always prints 0
Constant propagation (an optimising compiler breaks your program)

A simple and innocent looking optimization:

\[
\begin{align*}
\text{int } x &= 14; \\
\text{int } y &= 7 - x / 2; \\
\end{align*}
\]

Consider the two threads below:

\[
\begin{align*}
x &= y = 0 \\
\text{if } (y == 1) \{ \\
\text{print } x \\
\text{ } \text{if } (x == 1) \{ \\
\text{print } y = 1 \\
\text{\} } \\
\end{align*}
\]

\textit{Sun HotSpot JVM or GCJ: always prints 1.}
Background: lock and unlock

• Suppose that two threads increment a shared memory location:

\[
x = 0
\]

\[
\begin{align*}
\text{tmp1} &= \*x; \\
\*x &= \text{tmp1} + 1;
\end{align*}
\]

\[
\begin{align*}
\text{tmp2} &= \*x; \\
\*x &= \text{tmp2} + 1;
\end{align*}
\]

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

\[
\begin{align*}
\text{tmp1} &= \*x; \\
\text{tmp2} &= \*x; \\
\*x &= \text{tmp1} + 1; \\
\*x &= \text{tmp2} + 1
\end{align*}
\]
Background: lock and unlock

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

\[ x = 0 \]

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

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

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

```java
int x = computeInitValue(); // eager initialization
...
// clients refer to x
```

with:

```java
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:

```cpp
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

```cpp
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

```cpp
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

```c
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:

```c
instance_ =
    operator new(sizeof(Singleton)); // 1
new (instance_) Singleton // 2
```

If this code is generated, the order is 1,3,2.
Broken...

```cpp
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

```c
int32_t a = 0
```

```c
a = 0x44332211
```

```c
if (a == 0x00002211)
  print "error"
```

(Disclaimer: compiler will normally ensure alignment)

Intel SDM x86 atomic accesses:

- $n$-bytes on an $n$-byte boundary ($n = 1, 2, 4, 16$)
- P6 or later: … or if unaligned but within a cache line

**Question:** what about multi-word high-level language values?
That pesky hardware (1)

Consider misaligned 4-byte accesses

\[ \text{int32_t } a = 0 \]

\[ a = 0x44332211 \quad \text{if } (a == 0x00002211) \]

\[ \text{print } "\text{error}" \]

(Disclaimer: compiler will normally ensure alignment)

Intel SDM x86 atomic accesses:

- \(n\)-bytes on an \(n\)-byte boundary (\(n = 1, 2, 4, 16\))
- P6 or later: … or if unaligned but within a cache line

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

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:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = 1)</td>
<td>(y = 1)</td>
</tr>
<tr>
<td><code>print y</code></td>
<td><code>print x</code></td>
</tr>
</tbody>
</table>

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...

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>print y</td>
</tr>
<tr>
<td>y = 1</td>
<td>print x</td>
</tr>
</tbody>
</table>

At the end of some executions:

```
1 0
```

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

Tension:
• the programmer wants to understand the code he writes
• the compiler and the hardware want to optimise it.

Which are the valid optimisations that the compiler or the hardware can perform without breaking the expected semantics of a concurrent program?

Which is the semantics of a concurrent program?
This lecture

Programming language models

1) soundness of compiler optimisations
2) data-race freedom
3) defining the semantics of a concurrent programming language

Tomorrow:

The C11/C++11 model in detail.
A brief tour of compiler optimisations
World of optimisations

A typical compiler performs many optimisations.

gcc 4.4.1. with -O2 option goes through 147 compilation passes.

computed using -fdump-tree-all and -fdump-rtl-all

Sun Hotspot Server JVM has 18 high-level passes with each pass composed of one or more smaller passes.

World of optimisations

A typical compiler performs many optimisations.

- Common subexpression elimination
  (copy propagation, partial redundancy elimination, value numbering)
- (conditional) constant propagation
- dead code elimination
- loop optimisations
  (loop invariant code motion, loop splitting/peeling, loop unrolling, etc.)
- vectorisation
- peephole optimisations
- tail duplication removal
- building graph representations/graph linearisation
- register allocation
- call inlining
- local memory to registers promotion
- spilling
- instruction scheduling
World of optimisations

However only some optimisations change shared-memory traces:

- **Common subexpression elimination**
  (copy propagation, partial redundancy elimination, value numbering)
- (conditional) constant propagation
- dead code elimination
- loop optimisations
  (loop invariant code motion, loop splitting/peeling, loop unrolling, etc.)
- vectorisation
- **peephole optimisations**
- tail duplication removal
- building graph representations/graph linearisation
- register allocation
- call inlining
- **local memory to registers promotion**
- **spilling**
- instruction scheduling
Memory optimisations

Optimisations of shared memory can be classified as:

*Eliminations* (of reads, writes, sometimes synchronisation).

*Reordering* (of independent non-conflicting memory accesses).

*Introductions* (of reads – rarely).
Eliminations

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

*Irrelevant read elimination:* 

\[ r = *x; \ C \rightarrow \ C \]

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:**

\[
\begin{align*}
  r_1 &= *x; \ r_2 = *y \rightarrow r_2 = *y; \ r_1 = *x \\
  *x &= r_1; \ *y = r_2 \rightarrow *y = r_2; \ *x = r_1 \\
  r_1 &= *x; \ *y = r_2 \leftrightarrow *y = r_2; \ r_1 = *x
\end{align*}
\]

**Roach motel reordering:**

\[
\begin{align*}
  \text{memop; lock } m \leftrightarrow \text{lock } m; \ \text{memop} \\
  \text{unlock } m; \ \text{memop} \rightarrow \text{memop; unlock } m
\end{align*}
\]

where memop is \(*x=r_1\) or \(r_1=*x\)
Memory access introduction

Can an optimisation introduce memory accesses?

Yes, but rarely:

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

→

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

Note that the loop body is not executed.
Memory access introduction

Can an optimisation introduce memory accesses?

Yes, but rarely:

Note that the loop body is not executed.

Back to our question now:

Which is the semantics of a concurrent program?

Note that the loop body is not executed.
Naive answer: enforce sequential consistency
Sequential consistency

Multiprocessors have a \textit{sequentially consistent} shared memory when:

\begin{quote}
...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...
\end{quote}

Compilers, programmers & sequential consistency
Compilers, programmers & sequential consistency

Simple and intuitive programming model
Compilers, programmers & sequential consistency

- Simple and intuitive programming model
- Expensive to implement

Expensive to implement

Simple and intuitive programming model
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.

This study supposes that the hardware is SC.

A recent paper at ISCA mentions a 6.2% slowdown wrt TSO to enforce end-to-end SC on dedicated hardware.
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. This study supposes that the hardware is SC. A recent paper at ISCA mentions a 6.2% slowdown wrt TSO to enforce end-to-end SC on dedicated hardware.

What is an SC-preserving compiler?

When is a compiler correct?
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

\[ P_1 = \star x = 1 \quad | \quad r_1 = \star x; \ r_2 = \star x; \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{if } r_1=r_2 \text{ then print 1 else print 2} \]

\[ P_2 = \star x = 1 \quad | \quad r_1 = \star x; \ r_2 = r_1; \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{if } r_1=r_2 \text{ then print 1 else print 2} \]

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

\[ P_1 = \star x = 1 \quad | \quad r_1 = \star x; \ r_2 = \star x; \]
\[ \quad \text{if } r_1=r_2 \text{ then print 1 else print 2} \]

\[ P_2 = \star x = 1 \quad | \quad r_1 = \star x; \ r_2 = r_1; \]
\[ \quad \text{if } r_1=r_2 \text{ then print 1 else print 2} \]
Example

\[ P_1 = *x = 1 \quad r1 = *x; \quad r2 = *x; \]
if \( r1=r2 \) then print 1 else print 2

\[ P_2 = *x = 1 \quad r1 = *x; \quad r2 = r1; \]
if \( r1=r2 \) then print 1 else print 2

Executions of P1:

\[ W_{t_1} x=1, R_{t_2} x=1, R_{t_2} x=1, P_{t_2} 1 \]
\[ R_{t_2} x=0, W_{t_1} x=1, R_{t_2} x=1, P_{t_2} 2 \]
\[ R_{t_2} x=0, R_{t_2} x=0, W_{t_1} x=1, P_{t_2} 1 \]
\[ R_{t_2} x=0, R_{t_2} x=0, P_{t_2} 1, W_{t_1} x=1 \]
Example

\[ P_1 = *x = 1 \]

\[ P_2 = *x = 1 \]

\[ r1 = *x; \quad r2 = *x; \]
\[ \text{if } r1 = r2 \text{ then print 1 else print 2} \]

\[ r1 = *x; \quad r2 = r1; \]
\[ \text{if } r1 = r2 \text{ then print 1 else print 2} \]

Executions of P1:

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

Executions of P2:

\[ W_{t_1} x=1, R_{t_2} x=1, P_{t_2} 1 \]
\[ R_{t_2} x=0, W_{t_1} x=1, P_{t_2} 1 \]
\[ R_{t_2} x=0, P_{t_2} 1, W_{t_1} x=1 \]
Example

\[ P_1 = \ast \text{x} = 1 \]

r1 = \ast \text{x};  
\text{if r1=r2 then print 1 else print 2}

\[ P_2 = \ast \text{x} = 1 \]

r1 = \ast \text{x};  
\text{r2 = r1;}
\text{if r1=r2 then print 1 else print 2}

Executions of P1:

\begin{align*}
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
\end{align*}

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

Executions of P2:

\begin{align*}
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
\end{align*}

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

It is correct to rewrite $P_1$ into $P_2$, but not the opposite!

Behaviours of $P_1$: $[P_{t_2} 1], [P_{t_2} 2]$  
Behaviours of $P_2$: $[P_{t_2} 1]$
General CSE incorrect in SC

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

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

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

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

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

But a compiler would optimise to:

*\(x = 1;\)
*\(y = 1;\)
if *\(y = 2\)
then print 1
General CSE incorrect in SC

\[
\begin{align*}
*x &= 1; \\
*y &= 1; \\
\text{if } *y &= 2 \\
\text{then print } 1
\end{align*}
\quad
\begin{align*}
\text{if } *x=1 \text{ then (} \\
*x &= 2; \\
*y &= 2
\end{align*}
\]

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.
Reordering incorrect

\[ \begin{align*}
*x &= 1; & *y &= 1; & r1 &= *y & *y &= 1; \\
r1 &= *y & r2 &= *x; & \Rightarrow & *x &= 1; & r2 &= *x; \\
\text{print } r1 & & \text{print } r2 & & \text{print } r1 & & \text{print } r2
\end{align*} \]

Again, the optimised program exhibits a new behaviour:

\[
\begin{bmatrix}
P_{t_1} & 0 & P_{t_2} & 1 \\
P_{t_1} & 1 & P_{t_2} & 0 \\
P_{t_1} & 1 & P_{t_2} & 1
\end{bmatrix}
\quad\quad
\begin{bmatrix}
P_{t_1} & 0 & P_{t_2} & 1 \\
P_{t_1} & 1 & P_{t_2} & 0 \\
P_{t_1} & 1 & P_{t_2} & 1 \\
P_{t_1} & 0 & P_{t_2} & 0
\end{bmatrix}
\]
Elimination of adjacent accesses

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

\[ r1 = \ast x; r2 = \ast x \quad \rightarrow \quad r1 = \ast x; r2 = r1 \]

*The basic idea:* whenever we perform the read \( r1 = \ast x \) in the optimised program, we perform *both* reads in the source program.
Elimination of adjacent accesses

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

\[ r1 = *x; r2 = *x \quad \rightarrow \quad r1 = *x; r2 = r1 \]

Can we define a model that:
1) enables more optimisations than SC, and
2) retains the simplicity of SC?
Alternative answer: data-race freedom
Data-race freedom

Our examples again:

- the problematic transformations (e.g. swapping the two writes in thread 0) do not change the meaning of single-threaded programs;

- the problematic transformations are detectable only by code that allows two threads to access the same data simultaneously in conflicting ways (e.g. one thread writes the data read by the other).

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>*y = 1</td>
<td>if *x == 1</td>
</tr>
<tr>
<td>*x = 1</td>
<td>then print *y</td>
</tr>
</tbody>
</table>

Observable behaviour: 0
Data-race freedom

Our examples again:

• the problematic transformations (e.g. swapping the two writes in thread 0)
  do not change the meaning of single-threaded programs;

• the problematic transformations are detectable only by code that allows two threads to access the same data simultaneously in conflicting ways (e.g. one thread writes the data read by the other).

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>*y = 1</td>
<td>if *x == 1</td>
</tr>
<tr>
<td></td>
<td>print *y</td>
</tr>
</tbody>
</table>

...intuition...
Programming languages provide synchronisation mechanisms
if these are used (and implemented) correctly, we might avoid the issues above...
The basic solution

Prohibit *data races*

Defined as follows:

- two memory operations **conflict** if they access the same memory location and at least one is a store operation;
- a SC execution (interleaving) contains a data race if **two conflicting operations corresponding to different threads are adjacent** (maybe executed concurrently).

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

\[ W_{t_1} y = 1, W_{t_1} x = 1, R_{t_2} x = 1, R_{t_2} y = 1, P_{t_2} 1 \]
Prohibit *data races*

Defined as follows:

- two memory operations conflict if they access the same memory location and at least one is a store operation;
- a SC execution (interleaving) contains a data race if two conflicting operations corresponding to different threads are adjacent (maybe executed concurrently).

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

\[ W_{t_1} y = 1, W_{t_1} x = 1, R_{t_2} x = 1, R_{t_2} y = 1, P_{t_2} 1 \]
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:**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>*y = 1</td>
<td><code>lock();</code></td>
</tr>
<tr>
<td>lock();</td>
<td><code>tmp = *x;</code></td>
</tr>
<tr>
<td>*x = 1</td>
<td>unlock();</td>
</tr>
<tr>
<td>unlock();</td>
<td>if tmp = 1</td>
</tr>
<tr>
<td></td>
<td>then print *y</td>
</tr>
</tbody>
</table>
How do we avoid data races? (focus on high-level languages)

This program is data-race free:

```
*y = 1; lock(); *x = 1; unlock();
```

**Thread 0**

```
*y = 1
lock();
*x = 1
unlock();
```

**Thread 1**

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

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

• **lock()**, **unlock()** are opaque for the compiler: viewed as potentially modifying any location, memory operations cannot be moved past 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
*y* = 1; lock(); tmp = *x*; unlock(); if tmp=1; lock(); *x* = 1; unlock();
lock(); tmp = *x*; unlock();
lock(); tmp = *x*; unlock(); if tmp=1; *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(); if tmp=1; *y* = 1; if tmp=1; lock(); *x* = 1; unlock();
How do we avoid data races? (focus on high-level languages)

• \texttt{lock()}, \texttt{unlock()} are opaque for the compiler: viewed as potentially modifying any location, memory operations cannot be moved past them

• \texttt{lock()}, \texttt{unlock()} contain "sufficient fences" to prevent hardware reordering across them

\begin{verbatim}
*y = 1; lock(); tmp = *x; unlock(); lock(); *x = 1; unlock(); if tmp=1
\end{verbatim}

Compiler/hardware can continue to reorder accesses

\textit{Intuition:}
compiler/hardware do not know about threads, but only racing threads can tell the difference!
Another example of DRF program

Exercise: is this program DRF?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>if *x == 1</td>
<td>if *y == 1</td>
</tr>
<tr>
<td>then *y = 1</td>
<td>then *x = 1</td>
</tr>
</tbody>
</table>
Another example of DRF program

Exercise: is this program DRF?

<table>
<thead>
<tr>
<th></th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>*x == 1</td>
<td>if *y == 1</td>
</tr>
<tr>
<td>then</td>
<td>*y = 1</td>
<td>then *x = 1</td>
</tr>
</tbody>
</table>

Answer: yes!

The writes cannot be executed in any SC execution, so they cannot participate in a data race.
Another example of DRF program

*Exercise*: is this program DRF?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if *x == 1</code></td>
<td><code>if *y == 1</code></td>
</tr>
<tr>
<td><code>then *y = 1</code></td>
<td><code>then *x = 1</code></td>
</tr>
</tbody>
</table>

**Answer**: yes!

The writes cannot be executed in any SC execution, so they cannot participate in a data race.

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

<table>
<thead>
<tr>
<th>Transformation</th>
<th>SC</th>
<th>DRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory trace preserving transformations</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Redundant read after read elimination</td>
<td>✓*</td>
<td>✓</td>
</tr>
<tr>
<td>Redundant read after write elimination</td>
<td>✓*</td>
<td>✓</td>
</tr>
<tr>
<td>Irrelevant read elimination</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Redundant write before write elimination</td>
<td>✓*</td>
<td>✓</td>
</tr>
<tr>
<td>Redundant write after read elimination</td>
<td>✓*</td>
<td>✓</td>
</tr>
<tr>
<td>Irrelevant read introduction</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Normal memory accesses reordering</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Roach-motel reordering</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>External action reordering</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

* Optimisations legal only on adjacent statements.
Validity of compiler optimisations, summary

<table>
<thead>
<tr>
<th>Transformation</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory trace preserving transformations</td>
<td>✓</td>
</tr>
</tbody>
</table>

Jaroslav Sevcik

*Safe Optimisations for Shared-Memory Concurrent Programs*

*Optimisations legal only on adjacent statements.*
Compilers, programmers & data-race freedom
Compilers, programmers & data-race freedom

Can be implemented efficiently
Compilers, programmers & data-race freedom

Can be implemented efficiently

Intuitive programming model (but detecting races is tricky!)
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 e.g. Erlang and 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

*Examples: MLton*

MLton is a high performance ML-to-x86 compiler, with concurrency extensions. Accesses to ML refs will exhibit the underlying x86-TSO behaviour (compiler guarantees atomicity).
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
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

...again, model in informal prose...
Data race freedom as a definition

• Core of the C11/C++11 standard.


with some escape mechanism called "low level atomics".

Mark Batty & al., POPL 2011.

• Part of the JSR-133 standard.


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.
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?)
Isn't this all obvious?
Isn't this all obvious?

Perhaps it should have been.
Isn't this all obvious?

Perhaps it should have been.

But a few things went wrong in the past...
1. Uncertainty about details

Initially \( x = y = 0 \)

\[
\begin{align*}
\text{r1} & := [x]; & \text{r2} & := [y]; \\
\text{if } (\text{r1}=1) & \quad \| \quad \text{if } (\text{r2}=1) & \\
[y] & := 1 & [x] & := 1
\end{align*}
\]

Is the outcome \( r1=r2=1 \) allowed?
1. Uncertainty about details

Initially $x = y = 0$

\[
\begin{align*}
  r1 & := [x]; & r2 & := [y]; \\
  \text{if (r1=1)} & \parallel \text{if (r2=1)} \\
  [y] & := 1 & [x] & := 1
\end{align*}
\]

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

```c
struct s
    { char a; char b; } x;
Thread 1:    Thread 2:  
x.a = 1;       x.b = 1; FORBIDDEN
```

- 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.*
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:

\[
\begin{array}{c|c}
\text{initially} & x = y = 0 \\
r1 := x & r2 := y \\
y := r1 & x := r2 \\
\text{print } r1 & \text{print } r2 \\
\end{array}
\]

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
```

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

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

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

Initially $x = y = 0$
\begin{align*}
r1 &:= x \\
y &:= r1 \\
\text{print } r1
\end{align*}

and

Initially $x = y = \text{null}$
\begin{align*}
r1 &:= x \\
y &:= r1 \\
\text{print } r2
\end{align*}

What should $r2.\text{run()}$ call?

If we allow out-of-thin-air, then it could do anything!
A word on JSR-133

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

**Goal 2**: all programs satisfy some memory safety requirements;

**Goal 3**: common compiler optimisations are sound.

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

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

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

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

Jaroslav Ševčík, David Aspinall, ECOOP 2008
The end?

C11/C++11 is not yet implemented by mainstream compilers, and low-level atomics are hard to use (just google for low-level atomics).

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

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

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

/*
 * This is a strong barrier! Reads cannot be delayed beyond a later store.
 * Reads cannot be hoisted beyond a LOCK prefix. Stores always in-order.
 */
#define CAS(_a, _o, _n)  
  {{ __typeof__((o) __o = _o;  
    __asm__ __volatile__(  
      "lock cmpxchg %3,%1"
      : "=a" (__o), "=m"(*(volatile unsigned int *)(a))  
      : "0" (__o), "r" (_n);  
    __o;  
  )
```
Next lecture: the C11/C++11 memory model

This afternoon, 2pm: exercises...