Shared memory

```c
int a = 1;
int b = 0;
```

**Thread 1**

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

**Thread 2**

```c
b = 42;
printf("%d\n", b);
```
Shared memory

int a = 1;
int b = 0;

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);
int a = 1;
int b = 0;

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

int a = 1;
int b = 0;

Thread 1

Thread 2

b = 42;
printf("%d\n", b);
Shared memory

```c
int a = 1;
int b = 0;
```

**Thread 1**

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

**Thread 2**

```c
b = 42;
printf("%d\n", b);
```
int a = 1;
int b = 0;

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);
Shared memory

```c
int a = 1;
int b = 0;
```

**Thread 1**

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

**Thread 2**

```c
b = 42;
printf("%d\n", b);
```

Thread 1 returns without modifying b
Shared memory

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}
```

Thread 1

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}
```

Thread 1 returns without modifying b

Thread 2

```c
b = 42;
printf("%d\n", b);
```

Thread 2 is not affected by Thread 1 and vice-versa
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
int a = 1;
int b = 0;

Thread 1

Thread 2

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
b = 42;
printf("%d\n", b);

Thread 1 returns without modifying b

Thread 2 is not affected by Thread 1 and vice-versa

I expect this program to print 42
int a = 1;
int b = 0;

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);
int a = 1;
int b = 0;

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);
Shared memory

```c
int a = 1;
int b = 0;
```

**Thread 1**

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

**Thread 2**

```c
b = 42;
printf("%d\n", b);
```

gcc 4.7 -O2

...sometimes we get 0 on the screen
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

movl  a(%rip), %eax   # load a into eax
movl  b(%rip), %ebx   # load b into ebx
testl %eax, %eax      # if a==1
jne   .L2             #   jump to .L2
movl  $0, b(%rip)     # store 0 into eax
ret                   # return
.L2:
    movl  %ebx, b(%rip)   # store ebx into b
    xorl  %eax, %eax     # store 0 into eax
    ret                   # return
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}
```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}
```

```assembly
movl  a(%rip), %eax  # load a into eax
movl  b(%rip), %ebx  # load b into ebx
testl %eax, %eax     # if a==1
jne   .L2             #   jump to .L2
movl  $0, b(%rip)     # store 0 into eax
ret                   # return
.L2:
    movl  %ebx, b(%rip)  # store ebx into b
    xorl  %eax, %eax     # store 0 into eax
    ret                  # return
```

**gcc 4.7 -O2**

Monday 11 May 15
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}

movl  a(%rip), %eax  # load a into eax
movl  b(%rip), %ebx  # load b into ebx
movl  %ebx, a(%rip)  # store ebx into a
movl  $0, b(%rip)    # store 0 into b
ret

.L2:
movl  %ebx, b(%rip)  # store ebx into b
xorl  %eax, %eax     # store 0 into eax
ret                    # return
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b) 
    ;
}

movl  a(%rip), %eax   # load a into eax
movl  b(%rip), %ebx   # load b into ebx
testl %eax, %eax      # if a==1
jne   .L2             #   jump to .L2
movl  $0, b(%rip)     # store 0 into eax
ret                   # return

.L2:
    movl  %ebx, b(%rip)   # store ebx into b
    xorl  %eax, %eax      # store 0 into eax
    ret                  # return
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

movl a(%rip), %eax  # load a into eax
movl b(%rip), %ebx  # load b into ebx
testl %eax, %eax    # if a==1
jne .L2             # jump to .L2
movl $0, b(%rip)    # store 0 into b
ret                   # return

.L2:
movl %ebx, b(%rip)   # store ebx into b
xorl %eax, %eax     # store 0 into eax
ret                   # return
```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
};
```

```assembly
movl  a(%rip), %eax   # load a into eax
movl  b(%rip), %ebx   # load b into ebx
testl %eax, %eax      # if a==1
jne   .L2             #   jump to .L2
movl  $0, b(%rip)    # store ebx into b
ret                   # return
.L2:                   # store 0 into eax
    xorl  %eax, %eax
    ret             # return
```
```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b);
}
```

```c
movl  a(%rip), %eax   # load a into eax
movl  b(%rip), %ebx   # load b into ebx
testl %eax, %eax      # if a==1
jne   .L2             #   jump to .L2
movl  $0, b(%rip)     # store 0 into b
ret                   # return
```

```
.L2:
```
```c
movl  %ebx, b(%rip)   # store ebx into b
xorl  %eax, %eax      # store 0 into eax
ret                   # return
```

The compiled code saves and restores b
Correct result in a sequential setting
int a = 1;
int b = 0;

Thread 1

movl   a(%rip),%eax
movl   b(%rip),%ebx
testl  %eax, %eax
jne    .L2
movl   $0, b(%rip)
ret

.L2:
    movl   %ebx, b(%rip)
xorl   %eax, %eax
ret

Thread 2

b = 42;
printf("%d\n", b);
Shared memory

int a = 1;
int b = 0;

Thread 1

movl a(%rip),%eax
movl b(%rip),%ebx
testl %eax, %eax
jne .L2
movl $0, b(%rip)
ret
.L2:
movl %ebx, b(%rip)
xorl %eax, %eax
ret

Thread 2

b = 42;
printf("%d\n", b);

- Read a (1) into eax
Shared memory

```c
int a = 1;
int b = 0;
```

**Thread 1**

```
movl  a(%rip),%eax
movl  b(%rip),%ebx
testl %eax, %eax
jne   .L2
movl  $0, b(%rip)
ret
```

**Thread 2**

```
b = 42;
printf("%d\n", b);
```

- Read a (1) into eax
- Read b (0) into ebx
Shared memory

```c
int a = 1;
int b = 0;
```

**Thread 1**

```
movl   a(%rip),%eax  
movl   b(%rip),%ebx  
testl  %eax, %eax   
jne    .L2         
movl   $0, b(%rip)   
ret
.L2:              
    movl   %ebx, b(%rip)  
xorl   %eax, %eax   
    ret
```

**Thread 2**

```c
b = 42;
printf("%d\n", b);
```

- Read $a$ (1) into $eax$
- Read $b$ (0) into $ebx$
- Store 42 into $b$
Shared memory

```c
int a = 1;
int b = 0;
```

**Thread 1**

```assembly
movl   a(%rip),%eax
movl   b(%rip),%ebx
testl  %eax, %eax
jne    .L2
movl   $0, b(%rip)
ret
```

**Thread 2**

```assembly
b = 42;
printf("%d\n", b);
```

- Read a (1) into eax
- Read b (0) into ebx
- Store 42 into b
- Store ebx (0) into b
Shared memory

```c
int a = 1;
int b = 0;
```

**Thread 1**

```assembly
movl   a(%rip),%eax
movl   b(%rip),%ebx
testl  %eax, %eax
jne    .L2
movl   $0, b(%rip)
ret
.L2:
```

**Thread 2**

```assembly
b = 42;
printf("%d\n", b);
```

- Read a (1) into eax
- Read b (0) into ebx
- Store 42 into b
- Store ebx (0) into b
- Print b: 0 is printed
C can’t be so nasty!
Must be a subtle compiler bug.
C can't be so nasty!
Must be a subtle compiler bug.

Of course C allows this.
No news here.
What is C?

K&R

ANSI C

C99

C11

DeFacto C: whatever C compilers implement C programmers rely on
What is C?

1980 - ...: widespread use of threads, no spec, poor understanding of constraints
2005 onwards: proposals by Boehm, Adve, C++0x concurrency subgroup
2009-2011: Batty et al., draft standard ⇒ math ⇒ fixes ⇒ C/C++11 standard
Why is it so hard?
Constant propagation

A simple, and *innocuous*, optimisation:

Source code

\[
\begin{align*}
x &= 14 \\
y &= 7 - \frac{x}{2}
\end{align*}
\]

Optimised code

\[
\begin{align*}
x &= 14 \\
y &= 7 - \frac{14}{2} \\
x &= 14 \\
y &= 0
\end{align*}
\]
Shared memory concurrency

Shared memory

\[ x = y = 0 \]

Thread 1

\[ x = 1 \]
\[ \text{if } (y == 1) \]
\[ \text{print } x \]

Thread 2

\[ \text{if } (x == 1) \{ \]
\[ x = 0 \]
\[ y = 1 \} \]
Shared memory concurrency

Shared memory

\[ x = y = 0 \]

Thread 1

\[ x = 1 \]
\[ \text{if } (y == 1) \]
\[ \text{print } x \]

Thread 2

\[ \text{if } (x == 1) \{ \]
\[ x = 0 \]
\[ y = 1 \} \]

Intuitively this program always prints 0
Shared memory concurrency

But if the compiler propagates the constant $x = 1$...

\[ x = y = 0 \]

Thread 1

\[ x = 1 \]

\[ \text{if } (y == 1) \]
\[ \text{print } x \]
Shared memory concurrency

But if the compiler propagates the *constant* $x = 1$...

$$x = y = 0$$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1$</td>
<td>$\text{if } (x == 1) { \text{if } (y == 1) { \text{print } x } \text{print 1}} }$</td>
</tr>
</tbody>
</table>

...the program always writes 1 rather than 0.
This talk

0. Concurrency and optimisations, not so simple
1. The layman semantics
2. Escape lanes for the expert programmer

3. Compiler testing via a theory of sound optimisations

4. Escape lanes are a Pandora’s box
5. The way forward...
The layman solution
forbid data-races
Standard way out: prohibit data races

Two memory accesses **conflict** if they
- access the same memory location, e.g. variable
- at least one access is a store

A program has a **data race** if two data accesses
- conflict, and
- can occur simultaneously in a sequentially consistent execution.

A program **data-race-free** (on a particular input) if no sequentially consistent execution results in a data race.
For the actions performed by a program that uses shared variables, the following assumptions can always be made:

- If between two synchronization points in a task, this task reads a shared variable whose type is a scalar or access type, then the variable is not updated by any other task at any time between these two points.
- If between two synchronization points in a task, this task updates a shared variable whose task type is a scalar or access type, then the variable is neither read nor updated by any other task at any time between these two points.

The execution of the program is erroneous if any of these assumptions is violated.

**Data-races are errors**
Posix Threads Specification

[IEEE 1003.1-2008, Base Definitions 4.11] 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.

Data-races are errors
The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

Data-races are errors
C++2011 / C11

[C++ 2011 FDIS (WG21/N3290) 1.10p21] The execution of a program contains a
data race if it contains two conflicting actions in different threads, at least one of
which is not atomic, and neither happens before the other. Any such data race results
in undefined behavior.

How to use C/C++ to implement
low-level system code?

Data-races are errors
Escape lanes
for expert programmers
Low-level atomics in C11/C++11

std::atomic<int> flag0(0), flag1(0), turn(0);

void lock(unsigned index) {
    if (0 == index) {
        flag0.store(1, std::memory_order_relaxed);
        turn.exchange(1, std::memory_order_acq_rel);

        while (flag1.load(std::memory_order_acquire) && 1 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    } else {
        flag1.store(1, std::memory_order_relaxed);
        turn.exchange(0, std::memory_order_acq_rel);

        while (flag0.load(std::memory_order_acquire) && 0 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    }
}

void unlock(unsigned index) {
    if (0 == index) {
        flag0.store(0, std::memory_order_release);
    } else {
        flag1.store(0, std::memory_order_release);
    }
}
The qualifiers

MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED

LESS RELAXED

MORE RELAXED
The qualifiers

- **MO_SEQ_CST**
- **MO_RELEASE / MO_ACQUIRE**
- **MO_RELEASE / MO_CONSUME**
- **MO_RELAXED**

Sequential consistent accesses

Less relaxed

More relaxed
The qualifiers

- **MO_SEQ_CST**
  - Sequential consistent accesses
- **MO_RELEASE**
  - Efficient implementation of message passing
- **MO_RELEASE / MO_CONSUME**
- **MO_RELAXED**
  - More relaxed
The qualifiers

MO_SEQ_CST

Sequential consistent accesses

LES RELAXED

MO_RELEASE

Efficient implementation of message passing

MORE RELAXED

MO_RELAXED

Efficient implementation of message passing on ARM/Power
The qualifiers

**MO_SEQ_CST**
Sequential consistent accesses

**MO_RELEASE**
Efficient implementation of message passing

**MO_RELAXED**
Efficient implementation of message passing on ARM/Power

**NO_syn**
No synchronisation; direct access to hardware

LESS RELAXED

MORE RELAXED
Memory access synchronisation

\[ x = y = 0 \]

Thread 1

\[ y = 1 \]
\[ x.\text{store}(1, \text{MO\_RELEASE}) \]

Thread 2

\[ \text{if } (x.\text{load}(\text{MO\_ACQUIRE}) == 1) \]
\[ r2 = y \]
Memory access synchronisation

\[ x = y = 0 \]

Thread 1

\begin{align*}
  y &= 1 \\
  x &= 1, \text{MO}_\text{RELEASE}
\end{align*}

Thread 2

\begin{align*}
  \text{if } (x.\text{load}(\text{MO}_\text{ACQUIRE}) == 1) \\
  r2 &= y
\end{align*}

Non-atomic loads must return the *most recent write* in the happens-before order (unique in a DRF program)
Understanding `MO_RELAXED`

\[ x = y = 0 \]

**Thread 1**

\[ y = 1 \]
\[ x\text{.store}(1,\text{MO\_RELAXED}) \]

**Thread 2**

\[ \text{if } (x\text{.load}(\text{MO\_RELAXED}) == 1) \]
\[ r2 = y \]
Understanding MO_RELAXED

\[ x = y = 0 \]

Thread 1

\[ y = 1 \]

\[ x = y = 0 \]

\[ x.\text{store}(1,\text{MO\_RELAXED}) \]

Thread 2

\[ \text{if (x.load(MO\_RELAXED) == 1)} \]

\[ r2 = y \]

\[ \text{DATA RACE} \]

Two conflicting accesses not related by happens-before
Understanding MO RELAXED

\[ x = y = 0 \]

Thread 1

\[
\begin{align*}
y &.\text{store}(1, \text{MO RELAXED}) \\
x &.\text{store}(1, \text{MO RELAXED})
\end{align*}
\]

Thread 2

\[
\begin{align*}
\text{if} \ (x &.\text{load}(\text{MO RELAXED}) == 1) \\
r2 & = y.\text{load}(\text{MO RELAXED})
\end{align*}
\]

WELL DEFINED

but \( r2 = 0 \) is possible
**Intuition**

the compiler (or hardware) can reorder independent accesses

\[ x = y = 0 \]

Thread 1

\[
\begin{align*}
&\text{y.store(1,MO\_RELAXED)} & \text{if (x.load(MO\_RELAXED) == 1)} \\
&\text{x.store(1,MO\_RELAXED)} & \quad \text{r2 = y.load(MO\_RELAXED)}
\end{align*}
\]

**WELL DEFINED**

but r2 = 0 is possible
**Intuition**
the compiler (or hardware) can reorder independent accesses

\[ x = y = 0 \]

Thread 1

\[ y.\text{store}(1, \text{MO\_RELAXED}) \]
\[ x.\text{store}(1, \text{MO\_RELAXED}) \]

Thread 2

\[ \text{if } (x.\text{load}(\text{MO\_RELAXED}) == 1) \]
\[ r2 = y.\text{load}(\text{MO\_RELAXED}) \]

Allow a RELAXED load to see any store that:
- does not happen-after it
- is not hidden by an intervening store hb-ordered between them
The full model
The full model

We can reason about C concurrency!
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

b = 42;
printf("%d\n", b);

Thread 2 is not affected by Thread 1 and vice-versa
This program is data-race free

This program must print 42
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}

Thread 2 is not affected by Thread 1 and vice-versa

This program is data-race free

This program must print 42
Shared memory

```c
int a = 1;
int b = 0;
```

This is a concurrency compiler bug

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

This program is data-race free

Thread 2 is not affected by Thread 1 and vice-versa

This program must print 42
Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011
Compiler testing: state of the art
Yang, Chen, Eide, Regehr - PLDI 2011

Reported hundreds of bugs on various versions of gcc, clang and other compilers
Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

Reported hundreds of bugs on various versions of gcc, clang and other compilers

Cannot catch concurrency compiler bugs
Hunting concurrency compiler bugs?

How to deal with non-determinism?

How to generate non-racy interesting programs?

How to capture all the behaviours of concurrent programs?

A compiler can optimise away behaviours: how to test for correctness?

limit case: two compilers generate correct code with disjoint final states
**Idea**

C/C++ compilers support separate compilation
Functions can be called in arbitrary non-racy concurrent contexts

\[ \downarrow \]

C/C++ compilers can only apply transformations sound with respect to an arbitrary non-racy concurrent context

Hunt concurrency compiler bugs

= search for transformations of sequential code
not sound in an arbitrary non-racy context
Check: only transformations sound in any concurrent non-racy context
Soundness of compiler optimisations in the C11/C++11 memory model
What is an optimisation?

Compiler Writer

Semanticist
What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist
for (int i=0; i<2; i++) {
    z = i;
    x[i] += y+1;
}
What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

Semanticist

tmp = y+1;
for (int i=0; i<2; i++) {
    z = i;
    x[i] += tmp;
}
What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR
Operations on AST

Semanticist

Elimination of run-time events
Reordering of run-time events
Introduction of run-time events
Operations on sets of events

```c
int main() {
    tmp = y+1;
    for (int i=0; i<2; i++) {
        z = i;
        x[i] += tmp;
    }
    return 0;
}
```
What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

tmp = y+1;
for (int i=0; i<2; i++) {
    z = i;
    x[i] += tmp;
}

Semanticist

Elimination of run-time events
Reordering of run-time events
Introduction of run-time events

Operations on sets of events

Store z 0
Load y 42
Store x[0] 43
Store z 1
Load y 42
Store x[1] 43
What is an optimisation?

Compiler Writer

Sophisticated program analyses
Fancy algorithms
Source code or IR

Operations on AST

tmp = y+1;
for (int i=0; i<2; i++) {
    z = i;
    x[i] += tmp;
}

Semanticist

Elimination of run-time events
Reordering of run-time events
Introduction of run-time events

Operations on sets of events

Load y 42
Store z 0
Store x[0] 43
Store z 1
Store x[1] 43
Elimination of overwritten writes

Under which conditions is it correct to eliminate the first store?
A **same-thread release-acquire pair** is a pair of a release action followed by an acquire action in program order.

An action is a *release* if it is a possible source of a synchronisation

\[ \text{unlock mutex, release or seq_cst atomic write} \]

An action is an *acquire* if it is a possible target of a synchronisation

\[ \text{lock mutex, acquire or seq_cst atomic read} \]
Elimination of overwritten writes

It is safe to eliminate the first store if there are:

1. no intervening accesses to $g$
2. no intervening same-thread release-acquire pair
The soundness condition

\[ g = 0; \text{atomic } f1 = f2 = 0; \]

**Shared memory**

**Thread 1**

g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
The soundness condition

**Shared memory**

\[ g = 0; \text{ atomic } f1 = f2 = 0; \]

**Thread 1**

candidate overwritten write

\[
\begin{align*}
g & = 1; \\
f1.\text{store}(1, \text{RELEASE}); \\
\text{while}(f2.\text{load}(\text{ACQUIRE}) == 0); \\
g & = 2;
\end{align*}
\]
The soundness condition

**Shared memory**

\[
g = 0; \text{ atomic } f1 = f2 = 0;
\]

**Thread 1**

candidate overwritten write

\[
g = 1;
\]

\[
f1.\text{store}(1,\text{RELEASE});
\]

\[
\text{while}(f2.\text{load}(\text{ACQUIRE})==0);
\]

\[
g = 2;
\]

same-thread release-acquire pair
The soundness condition

**Shared memory**

\[ g = 0; \text{atomic } f_1 = f_2 = 0; \]

**Thread 1**

\[ g = 1; \]
\[ f_1.\text{store}(1,\text{RELEASE}); \]
\[ \text{while}(f_2.\text{load}(\text{ACQUIRE})==0); \]
\[ g = 2; \]

**Thread 2**

\[ \text{while}(f_1.\text{load}(\text{ACQUIRE})==0); \]
\[ \text{printf}("%d", g); \]
\[ f_2.\text{store}(1,\text{RELEASE}); \]
The soundness condition

Shared memory

\[ g = 0; \text{ atomic } f_1 = f_2 = 0; \]

**Thread 1**

\[
\begin{align*}
g &= 1; \\
f_1 &.\text{store}(1,\text{RELEASE}); \\
\text{while}(f_2.\text{load}(\text{ACQUIRE}) == 0); \\
g &= 2;
\end{align*}
\]

**Thread 2**

\[
\begin{align*}
\text{while}(f_1.\text{load}(\text{ACQUIRE}) == 0); \\
\text{printf}("%d", g); \\
f_2 &.\text{store}(1,\text{RELEASE});
\end{align*}
\]

Thread 2 is non-racy
The soundness condition

**Shared memory**

g = 0; atomic f1 = f2 = 0;

**Thread 1**

g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;

**Thread 2**

while(f1.load(ACQUIRE)==0);
printf(“%d”, g);
f2.store(1,RELEASE);

Thread 2 is non-racy
The program should only print 1
The soundness condition

Shared memory

\[ g = 0; \text{atomic} \ f_1 = f_2 = 0; \]

Thread 1
\[ g = 1; \]
\[ f_1.\text{store}(1, \text{RELEASE}); \]
\[ \text{while}(f_2.\text{load}(\text{ACQUIRE}) == 0); \]
\[ g = 2; \]

Thread 2
\[ \text{while}(f_1.\text{load}(\text{ACQUIRE}) == 0); \]
\[ \text{printf}("\%d", g); \]
\[ f_2.\text{store}(1, \text{RELEASE}); \]

Thread 2 is non-racy
The program should only print 1
If we perform overwritten write elimination it prints 0
The soundness condition

**Shared memory**

\[
g = 0; \text{ atomic } f_1 = f_2 = 0;
\]

**Thread 1**

\[
g = 1; \\
f_1.\text{store}(1, \text{RELEASE}); \\
\text{while}(f_2.\text{load}(\text{ACQUIRE}) == 0); \\
g = 2;
\]

**Thread 2**

\[
\text{while}(f_1.\text{load}(\text{ACQUIRE}) == 0); \\
\text{printf(“}\%d\”, g); \\
f_2.\text{store}(1, \text{RELEASE});
\]
The soundness condition

\[ g = 0; \text{atomic } f_1 = f_2 = 0; \]

**Thread 1**

\[ g = 1; \]
\[ f_1.\text{store}(1, \text{RELEASE}); \]
\[ g = 2; \]

**Thread 2**

\[ \text{while}(f_1.\text{load}(\text{ACQUIRE}) == 0); \]
\[ \text{printf}(\"%d\", g); \]
\[ f_2.\text{store}(1, \text{RELEASE}); \]

**Shared memory**
The soundness condition

\[ g = 0; \text{atomic } f1 = f2 = 0; \]

**Thread 1**

\[ g = 1; \]
\[ f1.\text{store}(1,\text{RELEASE}); \quad \text{sync} \]
\[ g = 2; \quad \text{data race} \]

**Thread 2**

\[ \text{while}(f1.\text{load}(\text{ACQUIRE})==0); \]
\[ \text{printf}("%d", g); \]
\[ f2.\text{store}(1,\text{RELEASE}); \]

If only a release (or acquire) is present, then all discriminating contexts are racy.
It is sound to optimise the overwritten write.
Eliminations: bestiary

Overwritten-Write Write-after-Write  Read-after-Read Read-after-Write  Write-after-Read

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (irrelevant reads).
Also correctness statements for reorderings, merging, and introductions of events.

Overwritten-Write Write-after-Write  Read-after-Read Read-after-Write  Write-after-Read

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (irrelevant reads).
From theory to the Cmmtest tool
Check: only transformations sound in any concurrent non-racy context
Check: only transformations sound in any concurrent non-racy context
Check: only transformations sound in any concurrent non-racy context.
CSmith extended with locks and atomics

gcc/clang -O0

EXECUTABLE

binary instrumentation

REFERENCE MEMORY TRACE

EXECUTABLE

optimising compiler under test

binary instrumentation

MEMORY TRACE

Check: only transformations sound in any concurrent non-racy context
CSmith extended with locks and atomics

**Sequential Program**

**Reference**

**Memory Trace**

**Executable**

**Instrumentation**

**Optimising Compiler under Test**

**Executable**

**Binary Instrumentation**

**gcc/clang -O0**

**OCaml Tool**

1. Analyse the traces to detect eliminable actions
2. Match reference and optimised traces
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}

Start with a randomly generated well-defined program
const unsigned int g3 = 0UL; void func_1(void){
int *l8 = &g6;
int l36 = 0x5E9D070FL;
unsigned int l107 = 0xAA37C3ACL;
g4 &= g3;
g5++;
int *l102 = &l36;
for (g6 = 4; g6 < (-3); g6 += 1);
l102 = &g6;
*l102 = ((*l8) && (l107 << 7)*(*l102));
}
void func_1(void) {
  int *l8 = &g6;
  int l36 = 0x5E9D070FL;
  unsigned int l107 = 0xAA37C3ACL;
  g4 &= g3;
  g5++;
  int *l102 = &l36;
  for (g6 = 4; g6 < (-3); g6 += 1);
  l102 = &g6;
  *l102 = ((*l8) && (l107 << 7)*(*l102));
}

Init g3 0
Init g4 1
Init g5 1
Init g6 6
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
void func_1(void) {
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = (*((l8) && (l107 << 7))*(*l102));
}

Init g3 0
Init g4 1
Init g5 1
Init g6 6

reference semantics

gcc -O2 memory trace

Load g4 1
Store g4 0
Load g5 1
Store g5 2
Store g6 4
Load g6 4
Load g6 4
Load g6 4
Store g6 1
Load g4 0
Load g5 1
Store g4 0
Store g6 1
Store g5 2
Load g4 0

Monday 11 May 15
void func_1(void) {
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}

Init g3 0
Init g4 1
Init g5 1
Init g6 6

reference semantics

gcc -O2 memory trace

RaW*  Load  g4 1
Store g4 0
RaW*  Load  g5 1
Store g5 2
OW*  Store g6 4
RaW*  Load  g6 4
RaR*  Load  g6 4
RaR*  Load  g6 4
RaW*  Load  g4 0

Load  g5 1
Store g4 0
Store g6 1
Store g5 2
Load  g4 0

Monday 11 May 15
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}

 gcc -O2 memory trace
void func_1(void){

  int *l8 = &g6;
  int l36 = 0x5E9D070FL;
  unsigned int l107 = 0xAA37C3ACL;
  g4 &= g3;
  g5++;
  int *l102 = &l36;
  for (g6 = 4; g6 < (-3); g6 += 1);
  l102 = &g6;
  *l102 = ((*l8) && (l107 << 7)*(*l102));
}

Can match applying
only correct eliminations and reorderings

gcc -O2 memory trace

Init g3 0
Init g4 1
Init g5 1
Init g6 6

Can match applying
only correct eliminations and reorderings

reference semantics

 RaW*  Load  g4  1
  Store  g4  0
 RaW*  Load  g5  1
  Store  g5  2
 OW*  Store  g6  4
 RaW*  Load  g6  4
 RaR*  Load  g6  4
 RaR*  Load  g6  4
  Store  g6  1
 RaW*  Load  g4  0
int a = 1;
int b = 0;
for (s=0; s! =4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

If we focus on the miscompiled initial example...
```c
int a = 1;
int b = 0;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```
int a = 1;
int b = 0;
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Load a 1
int a = 1;
int b = 0;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    
}

int s;

reference semantics

gcc -O2 memory trace

Load a 1

Load a 1
Load b 0
Store b 0
Cannot match some events → detect compiler bug

reference semantics

Load a 1

gcc -O2 memory trace

Load a 1
Load b 0
Store b 0
Applications
1. Testing C compilers (GCC, Clang, ICC)

Some concurrency compiler bugs found in the latest version of GCC.

Store introductions performed by loop invariant motion or if-conversion optimisations.

Remark: these bugs break the Posix thread model too.

All promptly fixed.
2. Checking compiler invariants

GCC internal invariant: never reorder with an atomic access

Baked this invariant into the tool and found a counterexample...

...not a bug, but fixed anyway

```c
atomic_uint a;
int32_t g1, g2;

int main (int, char **[]) {
  a.load() & a.load();
  g2 = g1 != 0;
}
```

```
ALoad  a   0  4  
ALoad  a   0  4  
Load   g1  0  4  
Store  g2  0  4  
Load   g1  0  4  
ALoad  a   0  4  
ALoad  a   0  4  
Store  g2  0  4  
```
3. Detecting unexpected behaviours

```c
uint16_t g
for (; g==0; g--);
```

```c
uint16_t g
for (; g==0; g--);
g=0;
```

Correct or not?
3. Detecting unexpected behaviours

```c
uint16_t g
for (; g==0; g--);
g=0;
```

If \( g \) is initialised with 0, a load gets replaced by a store:

```
Load  g  0   ?   Store  g  0
```

The introduced store cannot be observed by a non-racy context. Still, **arguable if a compiler should do this or not.**
3. Detecting unexpected behaviours

```c
uint16_t g
for (; g==0; g--);
g=0;
```

If `g` is initialised with 0, a load gets replaced by a store:

```
Load  g  0
Store g  0
```

*False positives in Thread Sanitizer*
The formalisation of the C11 memory model enables compiler testing... what else?
Proving the correctness of mappings for atomics

https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

<table>
<thead>
<tr>
<th>C/C++11 Operation</th>
<th>ARM implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Relaxed:</td>
<td>ldr</td>
</tr>
<tr>
<td></td>
<td>ldr + preserve dependencies until next kill_dependency</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>ldr; teq; beq; isb</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>ldr; dmb</td>
</tr>
<tr>
<td>Load Consume:</td>
<td>ldr; teq; beq; isb</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>ldr; dmb</td>
</tr>
<tr>
<td>Load Acquire:</td>
<td>ldr; teq; beq; isb</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>ldr; dmb</td>
</tr>
<tr>
<td>Load Seq Cst:</td>
<td>ldr; dmb</td>
</tr>
<tr>
<td>Store Relaxed:</td>
<td>str</td>
</tr>
<tr>
<td>Store Release:</td>
<td>dmb; str</td>
</tr>
<tr>
<td>Store Seq Cst:</td>
<td>dmb; str; dmb</td>
</tr>
<tr>
<td>Cmpxchg Relaxed (32 bit):</td>
<td>__loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; streseq rres, rnewval, [rptr]; teq rres, 0; bne __loop</td>
</tr>
<tr>
<td>Cmpxchg Acquire (32 bit):</td>
<td>__loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; streseq rres, rnewval, [rptr]; teq rres, 0; bne __loop; isb</td>
</tr>
<tr>
<td>Cmpxchg Release (32 bit):</td>
<td>dmb; __loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; streseq rres, rnewval, [rptr]; teq rres, 0; bne __loop;</td>
</tr>
<tr>
<td>Cmpxchg AcqRel (32 bit):</td>
<td>dmb; __loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; streseq rres, rnewval, [rptr]; teq rres, 0; bne __loop; isb</td>
</tr>
<tr>
<td>Cmpxchg SeqCst (32 bit):</td>
<td>dmb; __loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; streseq rres, rnewval, [rptr]; teq rres, 0; bne __loop; dmb</td>
</tr>
<tr>
<td>Acquire Fence:</td>
<td>dmb</td>
</tr>
<tr>
<td>Release Fence:</td>
<td>dmb</td>
</tr>
<tr>
<td>AcqRel Fence:</td>
<td>dmb</td>
</tr>
<tr>
<td>SeqCst Fence:</td>
<td>dmb</td>
</tr>
</tbody>
</table>
Inform new optimisations
e.g. the work by Robin Morisset on the Arm LLVM backend

```assembly
while (flag.load(acquire))
{
}

.loop
  ldr r0, [r1]
  dmb ish
  bnz .loop

.loop
  ldr r0, [r1]
  bnz .loop
  dmb ish
```
Inform new optimisations

e.g. the work by Robin Morisset on the Arm LLVM backend

```
while (flag.load(acquire))
{}
```

```
loop
  ldr r0, [r1]
  dmb ish
  bnz .loop
```

```
loop
  ldr r0, [r1]
  bnz .loop
  dmb ish
```
Not all of C/C++11 is good
A second look at qualifiers

MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED
A second look at qualifiers

- MO_SEQ_CST
- MO_RELEASE / MO_ACQUIRE
- MO_RELEASE / MO_CONSUME
- MO_RELAXED

Less relaxed

Reasonable

More relaxed
A second look at qualifiers

MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED

LESS RELAXED

REASONABLE

HARD TO IMPLEMENT

MORE RELAXED
A second look at qualifiers

MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED

LESS RELAXED

REASONABLE

HARD TO IMPLEMENT

SEMANTICS TOO WEAK

MORE RELAXED
Out of thin air reads
Shorthand

from now on, all the memory accesses are atomic with MO_RELAXED semantics
Relaxed atomics

Thread 1

\[ r1 = x \]
\[ y = r1 \]

Thread 2

\[ r2 = y \]
\[ x = 42 \]

\[ x = y = 0 \]
Relaxed atomics

Thread 1

\[ r_1 = x \]
\[ y = r_1 \]

Thread 2

\[ r_2 = y \]
\[ x = 42 \]

\[ r_1 = r_2 = 42 \]

is a valid execution.
Out-of-thin-air reads

Thread 1

\[ r_1 = x \]
\[ y = r_1 \]

Thread 2

\[ r_2 = y \]
\[ x = r_2 \]
Out-of-thin-air reads

Thread 1
\[ r1 = x \]
\[ y = r1 \]
\[ x = y = 0 \]

Thread 2
\[ r2 = y \]
\[ x = r2 \]
\[ r1 = r2 = 42 \]

is also an allowed execution

R x 42
\[ \text{sb} \]
W y 42

R y 42
\[ \text{sb} \]
W x 42

rf
rf
the value 42 appears *out-of-thin-air*

Thread 1

\[
\begin{align*}
  r_1 &= x \\
  y &= r_1
\end{align*}
\]

Thread 2

\[
\begin{align*}
  r_2 &= y \\
  x &= r_2
\end{align*}
\]

\[r_1 = r_2 = 42\]

is also an allowed execution

\[
\begin{align*}
  R \; x \; 42 & \quad \text{rf} \\
  sb & \quad \text{rf} \\
  W \; y \; 42 & \quad \text{sb} \\
  R \; y \; 42 & \quad \text{rf} \\
  W \; x \; 42
\end{align*}
\]
Speculation can justify out-of-thin-air reads

If the compiler states that $x$ is likely to hold 42...

```plaintext
y := 42
r1 := x
if (r1 != 42) y := r1;
print r1
```

<table>
<thead>
<tr>
<th>Initially</th>
<th>x = y = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 := x</td>
<td>r2 := y</td>
</tr>
<tr>
<td>y := r1</td>
<td>x := r2</td>
</tr>
<tr>
<td>print r1</td>
<td>print r2</td>
</tr>
</tbody>
</table>
Speculation can justify out-of-thin-air reads

If the compiler states that $x$ is likely to hold 42...

$$
\begin{align*}
y &:= 42 \\
r1 &:= x \\
\text{if } (r1 \neq 42) &\ y := r1; \\
\text{print } r1
\end{align*}
$$

\begin{tabular}{|c|c|}
\hline
$\text{initially } x = y = 0$ & $\text{r1 := x}$ \\
\hline
$\text{r2 := y}$ & $y := r1$ \\
\hline
$x := r2$ & $\text{print } r1$ \\
\hline
$\text{print } r2$
\end{tabular}

It does not happen in practice...

(a big thank you to compiler and hardware developers)

...but allowed by the standard
Consequences of out-of-thin-air reads
```c
struct foo {
    atomic<struct foo *> next;
}
struct foo *a;

Thread 1

r1 = a->next
r1->next = a
```
struct foo {
    atomic<struct foo *> next;
}
struct foo *a;

Thread 1

r1 = a->next
r1->next = a
struct foo {
    atomic<struct foo *> next;
}
struct foo *a, *b;

Thread 1

r1 = a->next
r1->next = a

Thread 2

r2 = b->next
r2->next = b
```c
struct foo {
    atomic<struct foo *> next;
};
struct foo *a, *b;

Thread 1
r1 = a->next
r1->next = a

Thread 2
r2 = b->next
r2->next = b

If a and b initially reference disjoint data-structures
we expect a and b to remain disjoint
struct foo {
    atomic<struct foo *> next;
}
struct foo *a, *b;

Thread 1
r1 = a->next
r1->next = a

Thread 2
r2 = b->next
r2->next = b
If the compiler speculates \( r1=b \) and \( r2=a \), then the store \( r1->next=a \) justifies \( r2=b->next \) assigning \( r2=a \) (and symmetrically to justify \( r1=b \))
If the compiler speculates $r1 = b$ and $r2 = a$, then the store $r1->next = a$ justifies $r2 = b->next$ assigning $r2 = a$ (and symmetrically to justify $r1 = b$)
If the compiler speculates \( r1 = b \) and \( r2 = a \), then
the store \( r1->next = a \) justifies \( r2 = b->next \) assigning \( r2 = a \)
(and symmetrically to justify \( r1 = b \))

Break our basic intuitions
about memory and sharing!
\[
x = y = a = 0
\]

if (x.load(rlx) == 42)  
y.write(42, rlx)

if (y.load(rlx) == 42)  
if (a == 1)  
x.write(42, rlx)
\( x = y = a = 0 \)

\[
\begin{align*}
\text{if (x.load(rlx)==42)} & \quad \text{if (y.load(rlx)==42)} & \quad \text{a = 1} \\
y.write(42,rlx) & \quad \text{if (a==1)} & \\
& \quad \quad \quad \quad \quad \quad \quad \quad x.write(42,rlx)
\end{align*}
\]

Remark 1

*This code is not racy!*

There is no consistent execution in which the read of \( a \) occurs.
\[ x = y = a = 0 \]

\[
\begin{align*}
\text{if } (x\text{.load}(rlx)==42) & \quad \text{if } (y\text{.load}(rlx)==42) & \quad a = 1 \\
y\text{.write}(42,rlx) & \quad \text{if } (a==1) & \quad x\text{.write}(42,rlx)
\end{align*}
\]

Remark 2

\[ a = 1 \land x = y = 0 \]

is the only possible final state
\[ x = y = a = 0 \]

\[
\begin{align*}
\text{if } (x.\text{load}(rlx)==42) & \quad \text{if } (y.\text{load}(rlx)==42) & \quad a = 1 \\
y.\text{write}(42,rlx) & \quad \text{if } (a==1) & \quad x.\text{write}(42,rlx)
\end{align*}
\]

Consider sequentialisation:

\[ C \ || \ D \ \Rightarrow \ C ; D \]

(ought to be correct)
\[ x = y = a = 0 \]

\[
\begin{align*}
\text{if } (x.\text{load}(rlx) == 42) & \quad \text{if } (y.\text{load}(rlx) == 42) & \quad a = 1 \\
y.\text{write}(42, rlx) & \quad \text{if } (a == 1) & \quad x.\text{write}(42, rlx) \\
\end{align*}
\]
\[ x = y = a = 0 \]

\[ a = 1 \]

\[
\text{if } (x.\text{load}(rlx)==42) \]

\[
y.\text{write}(42,rlx) \]

\[
\text{if } (y.\text{load}(rlx)==42) \]

\[
\text{if } (a==1) \]

\[
x.\text{write}(42,rlx) \]
\[ x = y = a = 0 \]

\[
\begin{align*}
\text{if } (x\text{.load}(rlx)==42) & \quad \text{if } (y\text{.load}(rlx)==42) \\
    y\text{.write}(42,rlx) & \quad \text{if } (a==1) \\
    & \quad x\text{.write}(42,rlx)
\end{align*}
\]

\[
[ x = y = z = 0 ]
\]

\[
\begin{align*}
W_{na}(a,1) & \quad R_{rlx}(y^{42}) \\
R_{rlx}(x^{42}) & \quad \text{rf} \\
R_{na}(a,1) & \quad W_{rlx}(x^{42}) \\
W_{rlx}(y^{42})
\end{align*}
\]

\[
a = 1 \\
x = y = 42 \\
is also possible
\]
\[ x = y = a = 0 \]

\[
\begin{align*}
\text{if } (x.\text{load}(rlx)==42) & \quad \text{if } (y.\text{load}(rlx)==42) \\
y.\text{write}(42,rlx) & \quad \text{if } (a==1) \\
& \quad \text{x.\text{write}(42,rlx)}
\end{align*}
\]

**Break common source-to-source**

(or LLVM IR - to - LLVM IR)

**compiler optimisations**

including expression linearisation and roach-motel reorderings
We still lack a really satisfactory proposal for the semantics of a general-purpose shared-memory concurrent programming language.
The way forward
Understand the effects of what compilers implement and programmers rely on

Build on that...
Beyond concurrency

Can one do `< comparison or pointer arithmetic between pointers to separately allocated objects?

Routinely done in Linux kernel

Forbidden by ISO standard
A web survey of 15 questions to investigate what C is in current practice: what behaviour is implemented by mainstream compilers and relied on by systems programmers.
Eventual outcome: clear descriptions of what people can rely on and what compilers \textit{in practice} should implement, what alias analysis and optimisation passes should (and should not) be allowed to do, etc.
Thank you.
Questions?