

The C1x and C++11 concurrency model

Mark Batty

University of Cambridge

January 16, 2013

C11 and C++11 Memory Model

A DRF model with the option to expose relaxed behaviour in exchange for high performance.

C11 takes it's model directly from C++11.

Allows for relaxed behaviour on target architectures, and compiler optimisation.

C++11: the next C++

C++11: the next C++

1300 page prose specification defined by the ISO.

C++11: the next C++

1300 page prose specification defined by the ISO.

The design is a detailed compromise:

- hardware/compiler implementability
- useful abstractions
- broad spectrum of programmers

C++11: the next C++

1300 page prose specification defined by the ISO.

The design is a detailed compromise:

- hardware/compiler implementability
- useful abstractions
- broad spectrum of programmers

We fixed serious problems in both C++11 and C1x, both now finalised.

What does C++11 look like?

```
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);
    }
}
```

Atomic accesses take a n ordering parameter

From most relaxed to most like DRF-SC:

`memory_order_relaxed`

`memory_order_release/memory_order_acquire`

`memory_order_release/memory_order_consume`

`memory_order_seq_cst`

mo_seq_cst

The compiler must ensure that `mo_seq_cst` atomics have SC semantics.

```
x.store(1, mo_seq_cst);   | y.store(1, mo_seq_cst);  
r1 = y.load(mo_seq_cst); | r2 = x.load(mo_seq_cst);
```

The program above cannot end with $r1 = r2 = 0$.

mo_seq_cst

The compiler must ensure that `mo_seq_cst` atomics have SC semantics.

```
x.store(1, mo_seq_cst);   | y.store(1, mo_seq_cst);  
r1 = y.load(mo_seq_cst); | r2 = x.load(mo_seq_cst);
```

The program above cannot end with `r1 = r2 = 0`.

...so, MP is forbidden over `mo_seq_cst`. So are all other relaxed behaviours.

mo_release / mo_acquire

Supports fast implementation of the message passing idiom.

```
x = 1;                               | r1 = y.load(mo_acquire);  
y.store(1, mo_release);             | r2 = x;
```

The program above cannot end with $r1 = 1$ and $r2 = 0$.

mo_release / mo_acquire

Supports fast implementation of the message passing idiom.

```
x = 1;                               | r1 = y.load(mo_acquire);  
y.store(1, mo_release);             | r2 = x;
```

The program above cannot end with $r1 = 1$ and $r2 = 0$.

...so, MP is forbidden using `mo_release` and `mo_acquire`. SB and IRIW are allowed though.

mo_release / mo_consume

Supports faster implementation of the message passing idiom on Power.

```
x = 1;                               | r1 = y.load(mo_consume);  
y.store(&x, mo_release);             | r2 = *r1;
```

The program above cannot end with $r1 = \&x$ and $r2 = 0$.

The two loads must have an address dependency.

mo_relaxed

Very fast access, but also lots of strange behaviour.

```
r1 = x.load(mo_relaxed); | r2 = y.load(mo_relaxed);  
y.store(1, mo_relaxed); | x.store(1, mo_relaxed);
```

The program above can end with $r1 = 1$ and $r2 = 1$.

mo_relaxed

Very fast access, but also lots of strange behaviour.

```
r1 = x.load(mo_relaxed); | r2 = y.load(mo_relaxed);  
y.store(1, mo_relaxed); | x.store(1, mo_relaxed);
```

The program above can end with $r1 = 1$ and $r2 = 1$.

...so, LB is allowed using `mo_relaxed`. We will see that these accesses are more relaxed than Power even.

The C1x/C++11 memory model

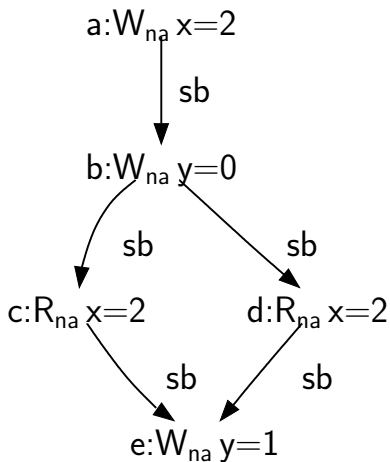
The C1x/C++11 memory model

- sequential execution
- simple concurrency
- expert concurrency
- very expert concurrency

A single threaded program

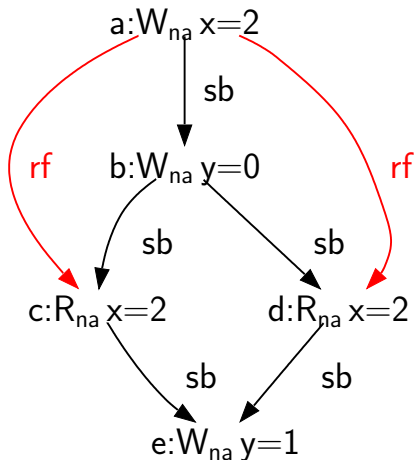
```
int main() {  
    int x = 2;  
    int y = 0;  
    y = (x==x);  
    return 0; }  

```



A single threaded program

```
int main() {  
    int x = 2;  
    int y = 0;  
    y = (x==x);  
    return 0; }
```



The relations of a pre-execution

Each symbolic execution, E_i , contains:

sb – *sequenced before*

asw – *additional synchronizes with*

dd – *data-dependence*

The relations of a pre-execution

Each symbolic execution, E_i , contains:

sb – *sequenced before*

asw – *additional synchronizes with*

dd – *data-dependence*

Each full execution, X_{ij} , also has:

rf – *reads from*

sc – *SC order*

mo – *modification order*

A data race

```
int y, x = 2;
```

```
x = 3;          | y = (x==3);
```

a:W_{na} x=2

asw,rf

asw

b:W_{na} x=3

c:R_{na} x=2

sb

d:W_{na} y=0

A data race

```
int y, x = 2;
```

```
x = 3;          | y = (x==3);
```

a:W_{na} x=2

asw,rf

asw

b:W_{na} x=3

dr

c:R_{na} x=2

sb

d:W_{na} y=0

Simple concurrency: Decker's example and SC

```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst); | y.store(1, seq_cst);
y.load(seq_cst);     | x.load(seq_cst);
```


Simple concurrency: Decker's example and SC

```
atomic_int x = 0;
```

```
atomic_int y = 0;
```

```
x.store(1, seq_cst);
```

```
y.load(seq_cst);
```

```
| y.store(1, seq_cst);
```

```
| x.load(seq_cst);
```

c:W_{sc} y=1

sb



d:R_{sc} x=0

e:W_{sc} x=1

sb



f:R_{sc} y=0

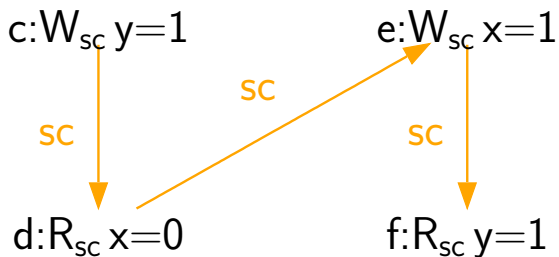
Simple concurrency: Decker's example and SC

```
atomic_int x = 0;
```

```
atomic_int y = 0;
```

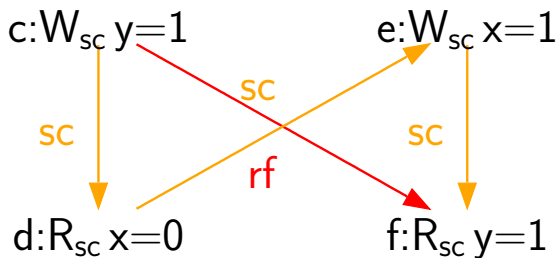
```
x.store(1, seq_cst); | y.store(1, seq_cst);
```

```
y.load(seq_cst);    | x.load(seq_cst);
```



SC atomics

Read the last write in SC order.

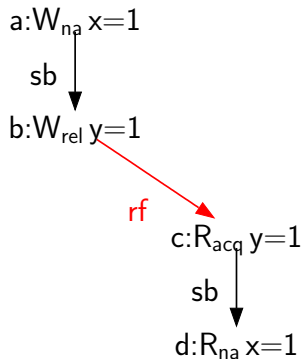


Using only `seq_cst` reads and writes gives SC.

(Initialization is not `seq_cst` though...)

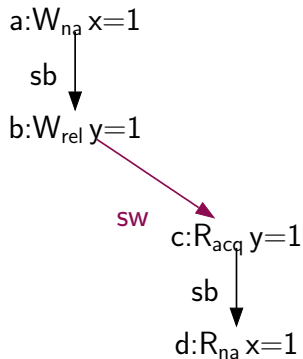
Expert concurrency: The release-acquire idiom

```
// sender | // receiver  
x = ...  | while (0 == y.load(acquire));  
y.store(1, release); | r = x;
```



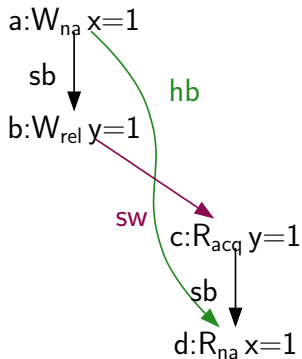
Expert concurrency: The release-acquire idiom

```
// sender | // receiver  
x = ...  | while (0 == y.load(acquire));  
y.store(1, release); | r = x;
```



Expert concurrency: The release-acquire idiom

```
// sender | // receiver  
x = ...  | while (0 == y.load(acquire));  
y.store(1, release); | r = x;
```



Expert concurrency: The release-acquire idiom

```
// sender
```

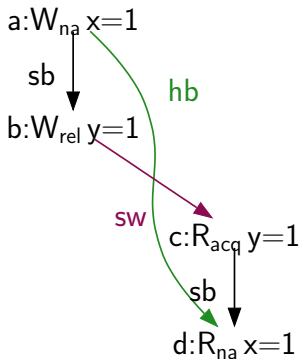
```
x = ...
```

```
y.store(1, release);
```

```
// receiver
```

```
while (0 == y.load(acquire));
```

```
r = x;
```



$$\begin{array}{l} \xrightarrow{\text{simple-happens-before}} = \\ \left(\xrightarrow{\text{sequenced-before}} \cup \xrightarrow{\text{synchronizes-with}} \right)^+ \end{array}$$

Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;

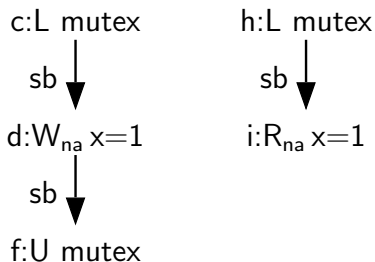
m.lock();           | m.lock();
x = ...            | r = x;
m.unlock();        |
```


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;

m.lock();           | m.lock();
x = ...             | r = x;
m.unlock();         |
```

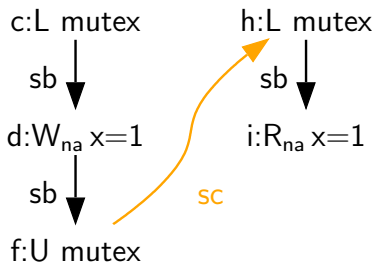


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;

m.lock();           | m.lock();
x = ...            | r = x;
m.unlock();        |
```

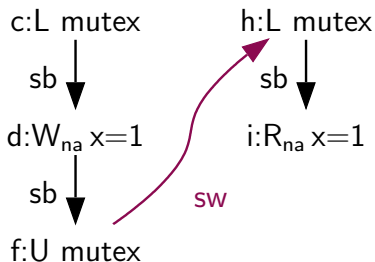


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;

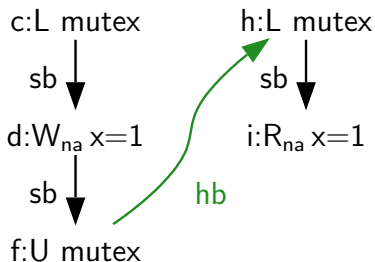
m.lock();           | m.lock();
x = ...            | r = x;
m.unlock();        |
```



Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;  
mutex m;  
  
m.lock();           | m.lock();  
x = ...            | r = x;  
m.unlock();        |
```

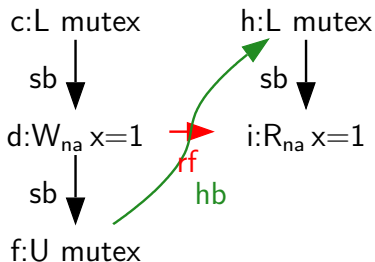


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;

m.lock();           | m.lock();
x = ...            | r = x;
m.unlock();        |
```



Happens-before is key to the model

Non-atomic loads read the most recent write in happens-before. (This is unique in DRF programs)

The story is more complex for atomics, as we shall see, but we cannot read from the future, in happens-before.

Data races are defined as an absence of happens-before.

A data race

```
int y, x = 2;
```

```
x = 3;          | y = (x==3);
```

a:W_{na} x=2

asw,rf

asw

b:W_{na} x=3

dr

c:R_{na} x=2

sb

d:W_{na} y=0

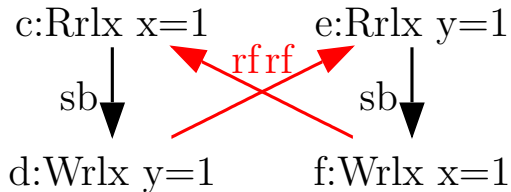
Data race definition

let $data_races\ actions\ hb =$
 { $(a, b) \mid \forall a \in actions\ b \in actions \mid$
 $\neg (a = b) \wedge$
 $same_location\ a\ b \wedge$
 $(is_write\ a \vee is_write\ b) \wedge$
 $\neg (same_thread\ a\ b) \wedge$
 $\neg (is_atomic_action\ a \wedge is_atomic_action\ b) \wedge$
 $\neg ((a, b) \in hb \vee (b, a) \in hb) \}$

A program with a data race has undefined behaviour.

Relaxed writes: load buffering

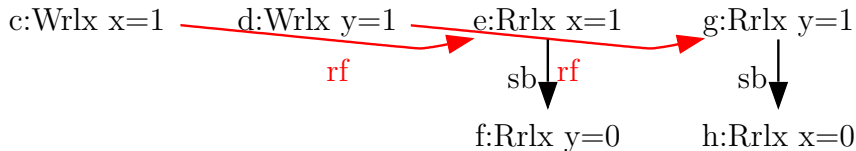
```
x.load(relaxed);      | y.load(relaxed);  
y.store(1, relaxed); | x.store(1, relaxed);
```



No synchronisation cost, but weakly ordered.

Relaxed writes: independent reads, independent writes

```
atomic_int x = 0;
atomic_int y = 0;
x.store(1, relaxed); | y.store(21, relaxed); | x.load(relaxed); | y.load(relaxed);
                    | y.load(relaxed); | x.load(relaxed);
```



Expert concurrency: fences avoid excess synchronisation

```
// sender          | // receiver  
x = ...           | while (0 == y.load(acquire));  
y.store(1, release); | r = x;
```

Expert concurrency: fences avoid excess synchronisation

```
// sender          | // receiver
x = ...           | while (0 == y.load(acquire));
y.store(1, release); | r = x;
```

```
// sender          | // receiver
x = ...           | while (0 == y.load(relaxed));
y.store(1, release); | fence(acquire);
                   | r = x;
```

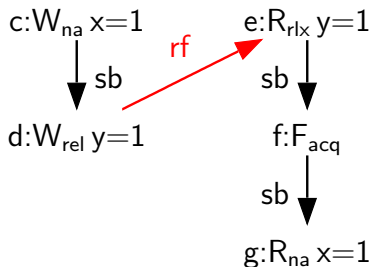
Expert concurrency: The fenced release-acquire idiom

```
// sender  
x = ...  
y.store(1, release);  
  
// receiver  
while (0 == y.load(relaxed));  
fence(acquire);  
r = x;
```

Expert concurrency: The fenced release-acquire idiom

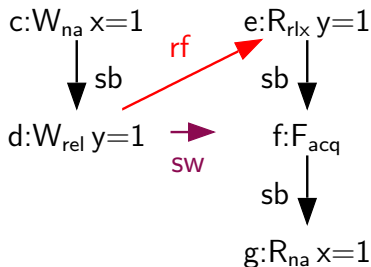
```
// sender  
x = ...  
y.store(1, release);
```

```
// receiver  
while (0 == y.load(relaxed));  
fence(acquire);  
r = x;
```



Expert concurrency: The fenced release-acquire idiom

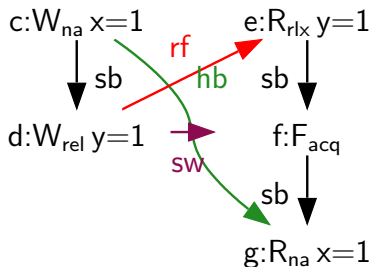
```
// sender          // receiver
x = ...           while (0 == y.load(relaxed));
y.store(1, release); fence(acquire);
                  r = x;
```



Expert concurrency: The fenced release-acquire idiom

```
// sender  
x = ...  
y.store(1, release);
```

```
// receiver  
while (0 == y.load(relaxed));  
fence(acquire);  
r = x;
```



Expert concurrency: modification order

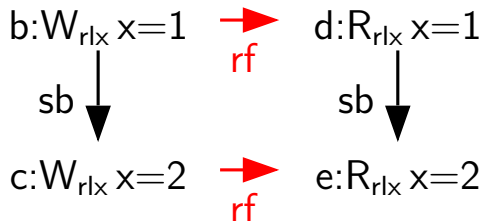
Modification order is a per-location total order over atomic writes of any memory order.

| | | |
|-----------------------------------|--|-------------------------------|
| <code>x.store(1, relaxed);</code> | | <code>x.load(relaxed);</code> |
| <code>x.store(2, relaxed);</code> | | <code>x.load(relaxed);</code> |

Expert concurrency: modification order

Modification order is a per-location total order over atomic writes of any memory order.

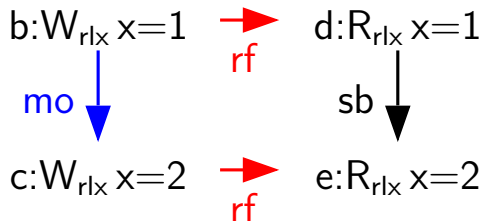
```
x.store(1, relaxed);      | x.load(relaxed);  
x.store(2, relaxed);      | x.load(relaxed);
```



Expert concurrency: modification order

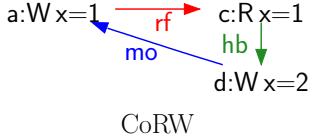
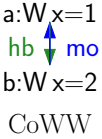
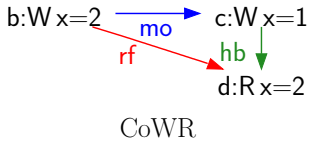
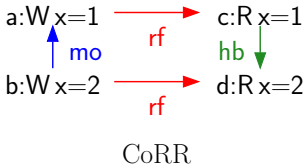
Modification order is a per-location total order over atomic writes of any memory order.

```
x.store(1, relaxed);      | x.load(relaxed);  
x.store(2, relaxed);      | x.load(relaxed);
```



Coherence and atomic reads

All forbidden!



Atomics cannot read from later writes in happens before.

Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

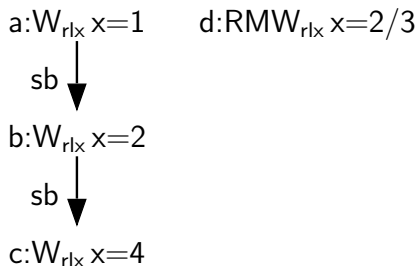
```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);  
x.store(2, relaxed); |  
x.store(4, relaxed); |
```

Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);  
x.store(2, relaxed); |  
x.store(4, relaxed); |
```

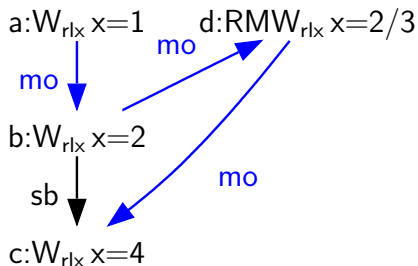


Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);  
x.store(2, relaxed); |  
x.store(4, relaxed); |
```

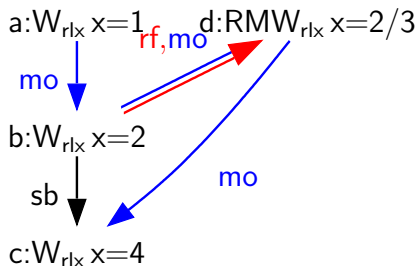


Read-modify-writes

A successful compare_exchange is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);  
x.store(2, relaxed); |  
x.store(4, relaxed); |
```



Very expert concurrency: consume

Weaker than acquire

Stronger than relaxed

Non-transitive happens before! (only fully transitive through data dependence, dd)

How may a program execute in the model?

1. $P \mapsto E_1, \dots, E_n$

How may a program execute in the model?

1. $P \mapsto E_1, \dots, E_n$

— find memory accesses with thread local semantics

How may a program execute in the model?

1. $P \mapsto E_1, \dots, E_n$

— find memory accesses with thread local semantics

2. $E_j \mapsto X_{i1}, \dots, X_{im}$

How may a program execute in the model?

1. $P \mapsto E_1, \dots, E_n$

— find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, \dots, X_{im}$

— calculate happens before, check the rules

How may a program execute in the model?

1. $P \mapsto E_1, \dots, E_n$

— find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, \dots, X_{im}$

— calculate happens before, check the rules

3. is there an X_{ij} with a race?

How may a program execute in the model?

1. $P \mapsto E_1, \dots, E_n$

— find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, \dots, X_{im}$

— calculate happens before, check the rules

3. is there an X_{ij} with a race?

— if so then have undefined behaviour

CPPMEM - demo!

Code in, all executions out

CPPMEM - demo!

Code in, all executions out

How may a program execute in CPPMEM?

1. $P \mapsto E_1, \dots, E_n$ — tracking constraints
2. $E_i \mapsto X_{i1}, \dots, X_{im}$ — automatically uses formal model
3. is there an X_{ij} with a race?

The model as a whole

C1x and C++11 support many modes of programming:

- sequential

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire
- with relaxed, fences and the rest

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire
- with relaxed, fences and the rest
- with all of the above plus consume

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire
- with relaxed, fences and the rest
- with all of the above plus consume

Mathematizing C++ concurrency. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. In Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 2011.

The full model

| | | |
|----------------------|----------------------|----------------------|
| Model 1: 1990-2000 | Model 2: 2000-2010 | Model 3: 2010-2020 |
| Model 4: 2020-2030 | Model 5: 2030-2040 | Model 6: 2040-2050 |
| Model 7: 2050-2060 | Model 8: 2060-2070 | Model 9: 2070-2080 |
| Model 10: 2080-2090 | Model 11: 2090-2100 | Model 12: 2100-2110 |
| Model 13: 2110-2120 | Model 14: 2120-2130 | Model 15: 2130-2140 |
| Model 16: 2140-2150 | Model 17: 2150-2160 | Model 18: 2160-2170 |
| Model 19: 2170-2180 | Model 20: 2180-2190 | Model 21: 2190-2200 |
| Model 22: 2200-2210 | Model 23: 2210-2220 | Model 24: 2220-2230 |
| Model 25: 2230-2240 | Model 26: 2240-2250 | Model 27: 2250-2260 |
| Model 28: 2260-2270 | Model 29: 2270-2280 | Model 30: 2280-2290 |
| Model 31: 2290-2300 | Model 32: 2300-2310 | Model 33: 2310-2320 |
| Model 34: 2320-2330 | Model 35: 2330-2340 | Model 36: 2340-2350 |
| Model 37: 2350-2360 | Model 38: 2360-2370 | Model 39: 2370-2380 |
| Model 40: 2380-2390 | Model 41: 2390-2400 | Model 42: 2400-2410 |
| Model 43: 2410-2420 | Model 44: 2420-2430 | Model 45: 2430-2440 |
| Model 46: 2440-2450 | Model 47: 2450-2460 | Model 48: 2460-2470 |
| Model 49: 2470-2480 | Model 50: 2480-2490 | Model 51: 2490-2500 |
| Model 52: 2500-2510 | Model 53: 2510-2520 | Model 54: 2520-2530 |
| Model 55: 2530-2540 | Model 56: 2540-2550 | Model 57: 2550-2560 |
| Model 58: 2560-2570 | Model 59: 2570-2580 | Model 60: 2580-2590 |
| Model 61: 2590-2600 | Model 62: 2600-2610 | Model 63: 2610-2620 |
| Model 64: 2620-2630 | Model 65: 2630-2640 | Model 66: 2640-2650 |
| Model 67: 2650-2660 | Model 68: 2660-2670 | Model 69: 2670-2680 |
| Model 70: 2680-2690 | Model 71: 2690-2700 | Model 72: 2700-2710 |
| Model 73: 2710-2720 | Model 74: 2720-2730 | Model 75: 2730-2740 |
| Model 76: 2740-2750 | Model 77: 2750-2760 | Model 78: 2760-2770 |
| Model 79: 2770-2780 | Model 80: 2780-2790 | Model 81: 2790-2800 |
| Model 82: 2800-2810 | Model 83: 2810-2820 | Model 84: 2820-2830 |
| Model 85: 2830-2840 | Model 86: 2840-2850 | Model 87: 2850-2860 |
| Model 88: 2860-2870 | Model 89: 2870-2880 | Model 90: 2880-2890 |
| Model 91: 2890-2900 | Model 92: 2900-2910 | Model 93: 2910-2920 |
| Model 94: 2920-2930 | Model 95: 2930-2940 | Model 96: 2940-2950 |
| Model 97: 2950-2960 | Model 98: 2960-2970 | Model 99: 2970-2980 |
| Model 100: 2980-2990 | Model 101: 2990-3000 | Model 102: 3000-3010 |
| Model 103: 3010-3020 | Model 104: 3020-3030 | Model 105: 3030-3040 |
| Model 106: 3040-3050 | Model 107: 3050-3060 | Model 108: 3060-3070 |
| Model 109: 3070-3080 | Model 110: 3080-3090 | Model 111: 3090-3100 |
| Model 112: 3100-3110 | Model 113: 3110-3120 | Model 114: 3120-3130 |
| Model 115: 3130-3140 | Model 116: 3140-3150 | Model 117: 3150-3160 |
| Model 118: 3160-3170 | Model 119: 3170-3180 | Model 120: 3180-3190 |
| Model 121: 3190-3200 | Model 122: 3200-3210 | Model 123: 3210-3220 |
| Model 124: 3220-3230 | Model 125: 3230-3240 | Model 126: 3240-3250 |
| Model 127: 3250-3260 | Model 128: 3260-3270 | Model 129: 3270-3280 |
| Model 130: 3280-3290 | Model 131: 3290-3300 | Model 132: 3300-3310 |
| Model 133: 3310-3320 | Model 134: 3320-3330 | Model 135: 3330-3340 |
| Model 136: 3340-3350 | Model 137: 3350-3360 | Model 138: 3360-3370 |
| Model 139: 3370-3380 | Model 140: 3380-3390 | Model 141: 3390-3400 |
| Model 142: 3400-3410 | Model 143: 3410-3420 | Model 144: 3420-3430 |
| Model 145: 3430-3440 | Model 146: 3440-3450 | Model 147: 3450-3460 |
| Model 148: 3460-3470 | Model 149: 3470-3480 | Model 150: 3480-3490 |
| Model 151: 3490-3500 | Model 152: 3500-3510 | Model 153: 3510-3520 |
| Model 154: 3520-3530 | Model 155: 3530-3540 | Model 156: 3540-3550 |
| Model 157: 3550-3560 | Model 158: 3560-3570 | Model 159: 3570-3580 |
| Model 160: 3580-3590 | Model 161: 3590-3600 | Model 162: 3600-3610 |
| Model 163: 3610-3620 | Model 164: 3620-3630 | Model 165: 3630-3640 |
| Model 166: 3640-3650 | Model 167: 3650-3660 | Model 168: 3660-3670 |
| Model 169: 3670-3680 | Model 170: 3680-3690 | Model 171: 3690-3700 |
| Model 172: 3700-3710 | Model 173: 3710-3720 | Model 174: 3720-3730 |
| Model 175: 3730-3740 | Model 176: 3740-3750 | Model 177: 3750-3760 |
| Model 178: 3760-3770 | Model 179: 3770-3780 | Model 180: 3780-3790 |
| Model 181: 3790-3800 | Model 182: 3800-3810 | Model 183: 3810-3820 |
| Model 184: 3820-3830 | Model 185: 3830-3840 | Model 186: 3840-3850 |
| Model 187: 3850-3860 | Model 188: 3860-3870 | Model 189: 3870-3880 |
| Model 190: 3880-3890 | Model 191: 3890-3900 | Model 192: 3900-3910 |
| Model 193: 3910-3920 | Model 194: 3920-3930 | Model 195: 3930-3940 |
| Model 196: 3940-3950 | Model 197: 3950-3960 | Model 198: 3960-3970 |
| Model 199: 3970-3980 | Model 200: 3980-3990 | Model 201: 3990-4000 |

Theorems

Are C1x and C++11 hopelessly complicated?

Programmers cannot be given this model!

With a formal definition, we can do proof, and even mechanise it.

What do we need to prove?

Are C1x and C++11 hopelessly complicated?

Programmers cannot be given this model!

With a formal definition, we can do proof, and even mechanise it.

What do we need to prove?

- implementability
- simplifications
- libraries

Implementability

Can we compile to x86?

Implementability

Can we compile to x86?

| Operation | x86 Implementation |
|--------------------|--------------------|
| load(non-seq_cst) | mov |
| load(seq_cst) | mov |
| store(non-seq_cst) | mov |
| store(seq_cst) | mov; mfence |
| fence(non-seq_cst) | no-op |
| fence(seq_cst) | mfence |

x86-TSO is stronger and simpler.

Top level comparison

Recall the C/C++ semantics for program P :

1. $P \mapsto E_1, \dots, E_n,$

Top level comparison

Recall the C/C++ semantics for program P :

1. $P \mapsto E_1, \dots, E_n$, each an E_{thread}

Top level comparison

Recall the C/C++ semantics for program P :

1. $P \mapsto E_1, \dots, E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, \dots, X_{im}$,

Top level comparison

Recall the C/C++ semantics for program P :

1. $P \mapsto E_1, \dots, E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, \dots, X_{im}$, collectively X_{witness}

Top level comparison

Recall the C/C++ semantics for program P :

1. $P \mapsto E_1, \dots, E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, \dots, X_{im}$, collectively X_{witness}
3. is there an X_{ij} with a race? (actually, several kinds...)

Top level comparison

Recall the C/C++ semantics for program P :

1. $P \mapsto E_1, \dots, E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, \dots, X_{im}$, collectively X_{witness}
3. is there an X_{ij} with a race? (actually, several kinds...)

In x86-TSO:

Events and dependencies, E_{x86} are analogous to E_{thread} .

Top level comparison

Recall the C/C++ semantics for program P :

1. $P \mapsto E_1, \dots, E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, \dots, X_{im}$, collectively X_{witness}
3. is there an X_{ij} with a race? (actually, several kinds...)

In x86-TSO:

Events and dependencies, E_{x86} are analogous to E_{thread} .
Execution witnesses, X_{x86} are analogous to X_{witness} .

Top level comparison

Recall the C/C++ semantics for program P :

1. $P \mapsto E_1, \dots, E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, \dots, X_{im}$, collectively X_{witness}
3. is there an X_{ij} with a race? (actually, several kinds...)

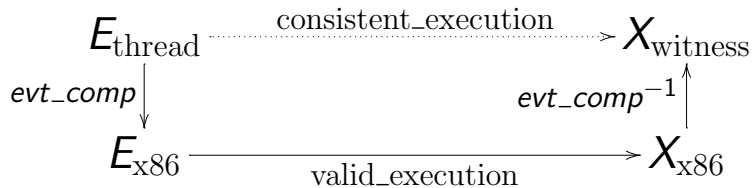
In x86-TSO:

Events and dependencies, E_{x86} are analogous to E_{thread} .

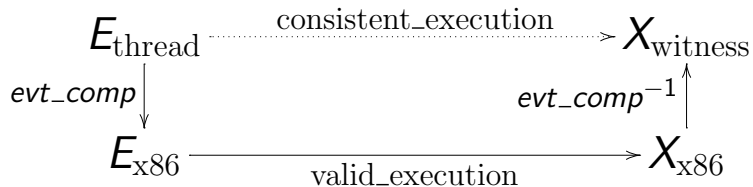
Execution witnesses, X_{x86} are analogous to X_{witness} .

There is not a DRF semantics.

Theorem



Theorem



We have a mechanised proof that C1x/C++11 behaviour is preserved.

Implementability

Can we compile to IBM Power?

Implementability

Can we compile to IBM Power?

| C++0x Operation | POWER Implementation |
|------------------|------------------------------|
| Non-atomic Load | ld |
| Load Relaxed | ld |
| Load Consume | ld (and preserve dependency) |
| Load Acquire | ld; cmp; bc; isync |
| Load Seq Cst | sync; ld; cmp; bc; isync |
| Non-atomic Store | st |
| Store Relaxed | st |
| Store Release | lwsync; st |
| Store Seq Cst | sync; st |

We have a hand proof that C1x/C++11 behaviour is preserved.

Implementability

Can we compile to IBM Power?

| C++0x Operation | POWER Implementation |
|------------------|------------------------------|
| Non-atomic Load | ld |
| Load Relaxed | ld |
| Load Consume | ld (and preserve dependency) |
| Load Acquire | ld; cmp; bc; isync |
| Load Seq Cst | sync; ld; cmp; bc; isync |
| Non-atomic Store | st |
| Store Relaxed | st |
| Store Release | lwsync; st |
| Store Seq Cst | sync; st |

We have a hand proof that C1x/C++11 behaviour is preserved.

Clarifying and compiling C/C++ concurrency: from C++0x to POWER. M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. In Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2012.

mo_seq_cst

The compiler must ensure that `mo_seq_cst` atomics have SC semantics.

```
x.store(1, mo_seq_cst); | y.store(1, mo_seq_cst);  
r1 = y.load(mo_seq_cst); | r2 = x.load(mo_seq_cst);
```

The program above cannot end with $r1 = r2 = 0$.

Sample compilation on x86:

```
store: mov; mfence  
load: mov
```

Sample compilation on Power:

```
store: sync; st  
load: sync; ld; cmp; bc; isync
```

mo_release / mo_acquire

Supports fast implementation of the message passing idiom.

```
x = 1;                               | r1 = y.load(mo_acquire);  
y.store(1, mo_release);             | r2 = x;
```

The program above cannot end with $r1 = 1$ and $r2 = 0$.

Accesses to the data could be reordered/optimised with `mo_relaxed`.

Sample compilation on x86: **Sample compilation on Power:**

store: mov
load: mov

store: lwsync; st
load: ld; cmp; bc; isync

mo_release / mo_consume

Supports faster implementation of the message passing idiom on Power.

```
x = 1;                               | r1 = y.load(mo_consume);  
y.store(&x, mo_release);             | r2 = *r1;
```

The program above cannot end with $r1 = \&x$ and $r2 = 0$.

The two loads have an address dependency - Power won't reorder them.

Sample compilation on x86:

```
store: mov  
load:  mov
```

Sample compilation on Power:

```
store: lwsync; st  
load:  ld
```

Refinements to the model and standards

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded.

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded.

Derivative models:

- without consume, happens-before is transitive.
- DRF programs using only `seq_cst` atomics are SC (false).

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded.

Derivative models:

- without consume, happens-before is transitive.
- DRF programs using only seq_cst atomics are SC (false).

```
atomic_int x = 0;
atomic_int y = 0;
if (1 == x.load(seq_cst)) | if (1 == y.load(seq_cst))
    atomic_init(&y, 1);    |    atomic_init(&x, 1);
```

atomic_init is a non-atomic write, and in C1x/C++11 they race...

The current state of the standard

Fixed:

- Happens-before
- Coherence
- seq_cst atomics were broken

The current state of the standard

Fixed:

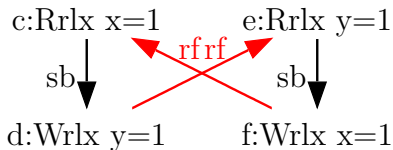
- Happens-before
- Coherence
- seq_cst atomics were broken

Not fixed:

- Self satisfying conditionals

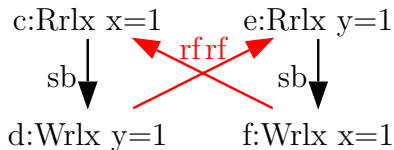
Self-satisfying conditionals

```
r1 = x.load(mo_relaxed);   |   r2 = y.load(mo_relaxed);  
if (r1 == 42)              |   if (r2 == 42)  
    y.store(r1, mo_relaxed); |   x.store(42, mo_relaxed);
```



Self-satisfying conditionals

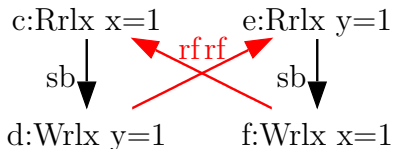
```
r1 = x.load(mo_relaxed);    |    r2 = y.load(mo_relaxed);  
if (r1 == 42)              |    if (r2 == 42)  
    y.store(r1, mo_relaxed); |    x.store(42, mo_relaxed);
```



"However, implementations **should** not allow such behavior."

Self-satisfying conditionals

```
r1 = x.load(mo_relaxed);    | r2 = y.load(mo_relaxed);  
if (r1 == 42)              | if (r2 == 42)  
    y.store(r1, mo_relaxed); |    x.store(42, mo_relaxed);
```



"However, implementations **should** not allow such behavior."

"should not" means "is allowed to" in the standard!

...but it's not all bad!

Syntactic divide supported by simpler memory models.

Increasingly reasonable, consistent specification.

Remaining problems far less serious than Java.

Implementable above key architectures.

Thanks!

