The C1x and C++11 concurrency model

Mark Batty

University of Cambridge

October 27, 2012

With coauthors: J. C. Blanchette, K. Memarian, S. Owens, S. Sarkar, P. Sewell, and T. Weber

Sequential consistency

ISO C1x/C++11 concurrency

Sequential consistency

Pthreads

ISO C1x/C++11 concurrency

Sequential consistency

Pthreads

Java

ISO C1x/C++11 concurrency

Sequential consistency

Pthreads

Java

Expose hardware model (e.g. ClightTSO)

ISO C1x/C++11 concurrency

Sequential consistency

Pthreads

Java

Expose hardware model (e.g. ClightTSO)

C++11/C1x: SC for data race free programs, almost...

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.

The C1x/C++11 memory model

The C1x/C++11 memory model

- top level
- sequential execution
- simple concurrency
- expert concurrency
- very expert concurrency

How may a program execute?

The memory model is factored out from a symbolic operational semantics.

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

How may a program execute?

The memory model is factored out from a symbolic operational semantics.

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

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

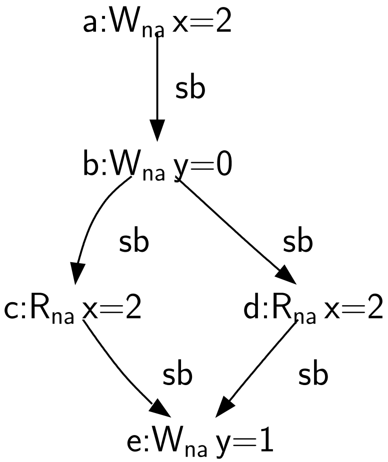
How may a program execute?

The memory model is factored out from a symbolic operational semantics.

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

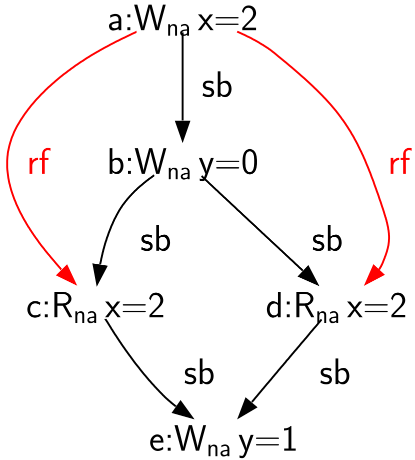
A single threaded program

```
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:

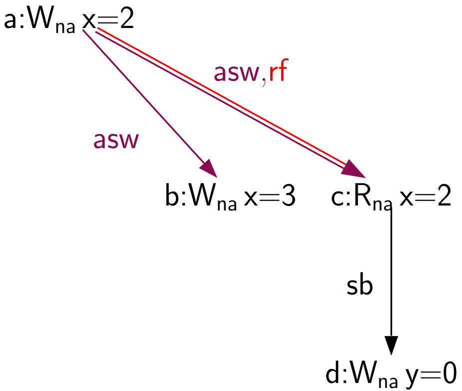
- **sb** – *sequenced before*
- **asw** – *additional synchronizes with*
- **dd** – *data-dependence*

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

- **rf** – *reads from*
- **sc** – *SC order*
- **mo** – *modification order*

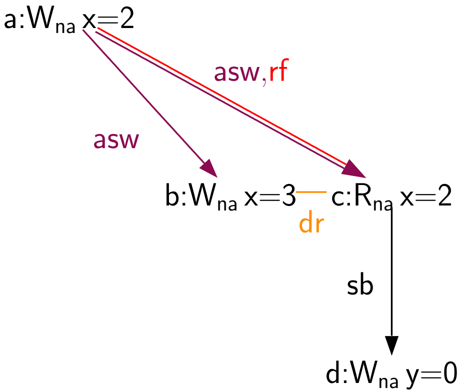
A data race

```
int y, x = 2;
x = 3;              | y = (x==3);
```



A data race

```
int y, x = 2;
x = 3;                | y = (x==3);
```



Simple concurrency: Decker's example and SC

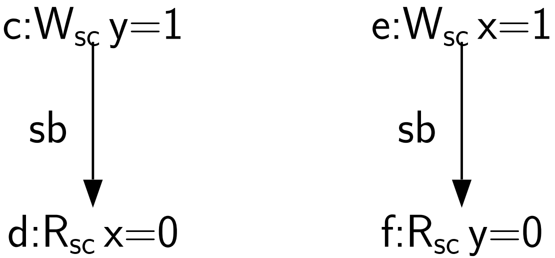
```
atomic_int x = 0;
atomic_int y = 0;
```

`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;
```

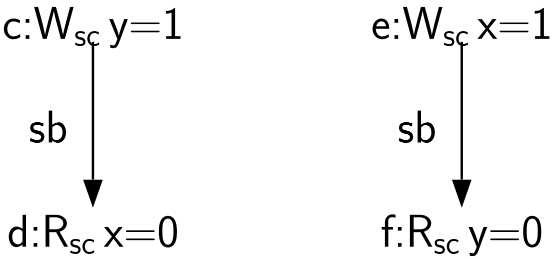
`x.store(1, seq_cst);`	`y.store(1, seq_cst);`
`y.load(seq_cst);`	`x.load(seq_cst);`



Simple concurrency: Decker's example and SC

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

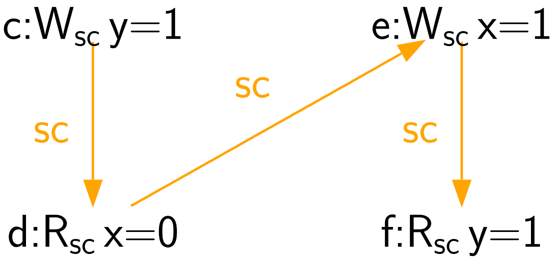
`x.store(1, seq_cst);`	`y.store(1, seq_cst);`
`y.load(seq_cst);`	`x.load(seq_cst);`



Simple concurrency: Decker's example and SC

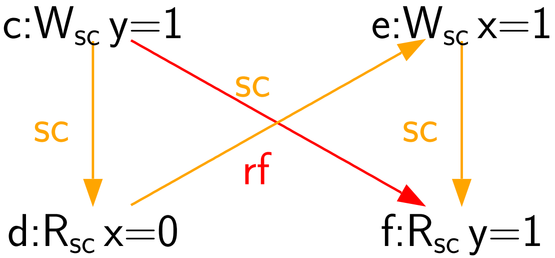
```
atomic_int x = 0;
atomic_int y = 0;

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



SC atomics

Read the last write in SC order.



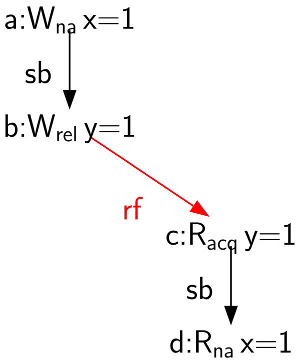
Using only `seq_cst` reads and writes gives SC.

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

Expert concurrency: The release-acquire idiom

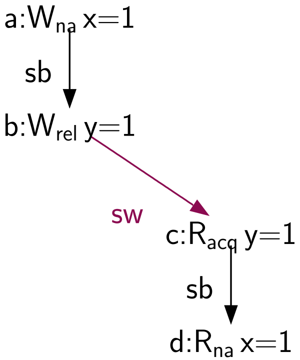
```
// sender
x = ...
y.store(1, release);
```

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



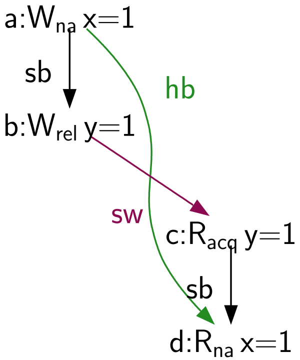
Expert concurrency: The release-acquire idiom

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



Expert concurrency: The release-acquire idiom

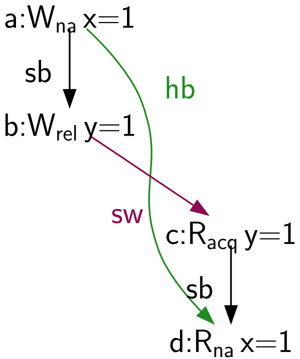
```
// sender                      // receiver
x = ...                        while (0 == y.load(acquire));
y.store(1, release);           r = x;
```



Expert concurrency: The release-acquire idiom

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

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



$$\xrightarrow{\textit{simple-happens-before}} = \left(\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}}\right)^+$$

Locks and unlocks

Unlocks and locks synchronise too:

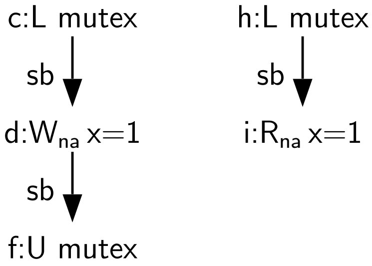
```
int x, r;
mutex m;
```

Thread 1	Thread 2
`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

Locks and unlocks

Unlocks and locks synchronise too:

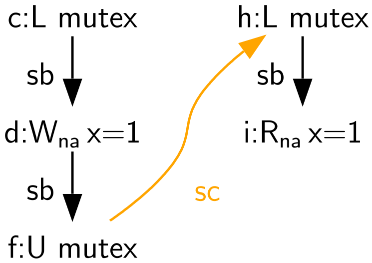
```
int x, r;
mutex m;
m.lock();          | m.lock();
x = ...            | r = x;
m.unlock();        |
```



Locks and unlocks

Unlocks and locks synchronise too:

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

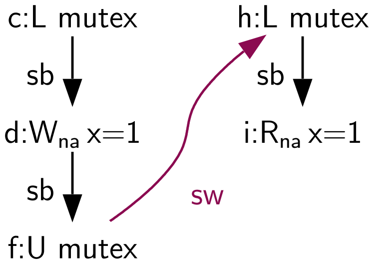


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

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

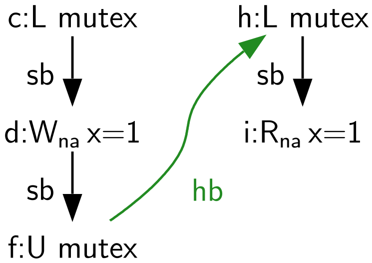


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

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

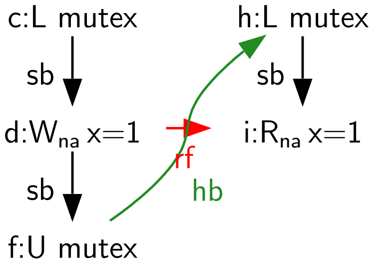


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

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



Happens before is key to the model

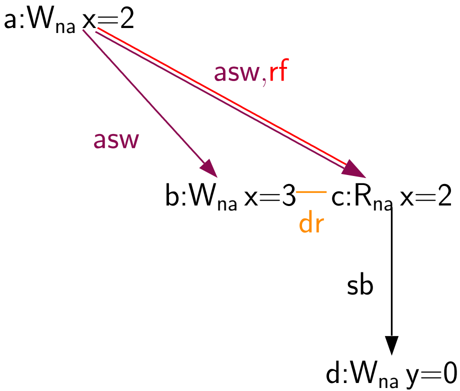
Non-atomic loads read the most recent write in happens before. (This is unique in DRF programs)

The story is more complex for atomics, as we shall see.

Data races are defined as an absence of happens before.

A data race

```
int y, x = 2;
x = 3;                | y = (x==3);
```



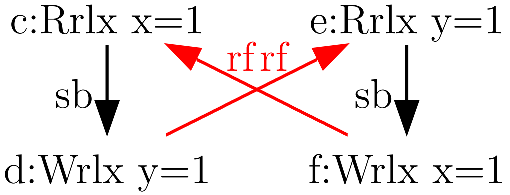
Data race definition

let data_races actions hb =
{ (a, b) — $\forall$ *a*$\in$*actions* *b*$\in$*actions* |
(a = b)
same_location a b
(is_write a is_write b)
(same_thread a b)
(is_atomic_action a is_atomic_action b)
((a, b) $\in$ *hb* (*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);
```

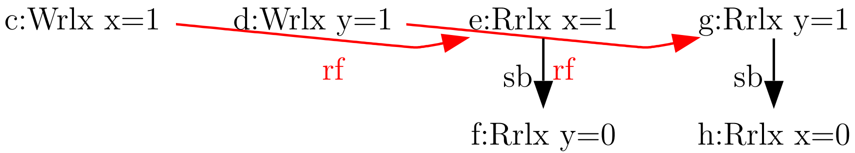


No synchronisation cost, but weakly ordered.

Relaxed writes: independent reads, independent writes

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

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



Expert concurrency: fences avoid excess synchronisation

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

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

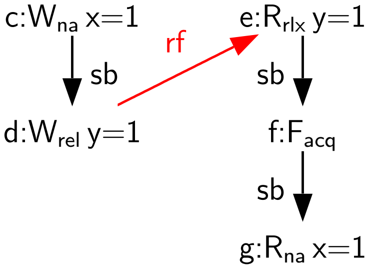
```
// sender
x = ...
y.store(1, release);
```

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

Expert concurrency: The fenced release-acquire idiom

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

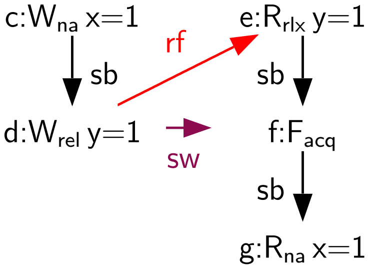
```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```



Expert concurrency: The fenced release-acquire idiom

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

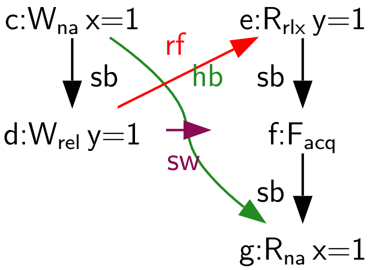
```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```



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: modification order

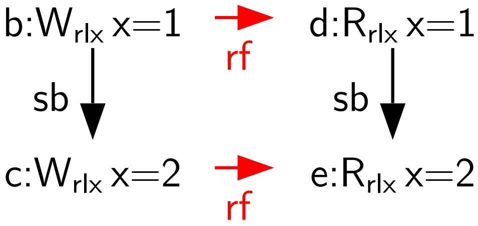
Modification order is a per-location total order over atomic writes of any memory order.

`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.

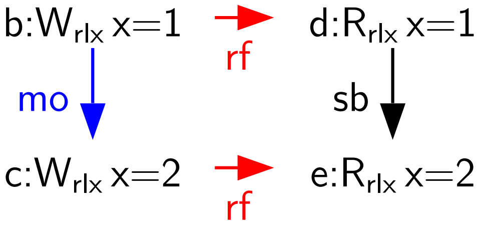
```
x.store(1, relaxed);        | x.load(relaxed);
x.store(2, relaxed);        | x.load(relaxed);
```



Expert concurrency: modification order

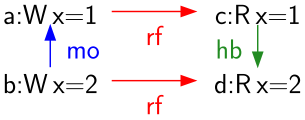
Modification order is a per-location total order over atomic writes of any memory order.

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

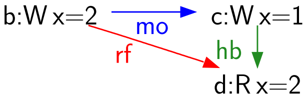


Coherence and atomic reads

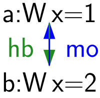
All forbidden!



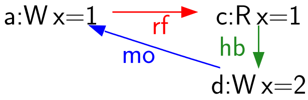
CoRR



CoWR



CoWW



CoRW

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:

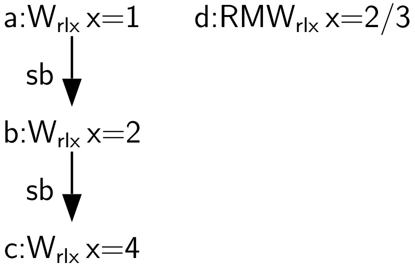
Thread 1	Thread 2
`x.store(1, relaxed);`	`compare_exchange(&x, 2, 3, relaxed, relaxed);`
`x.store(2, relaxed);`	
`x.store(4, relaxed);`	

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);  |
```

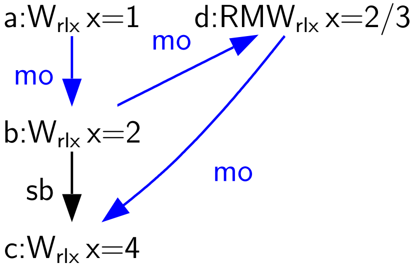


Read-modify-writes

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);   |
```

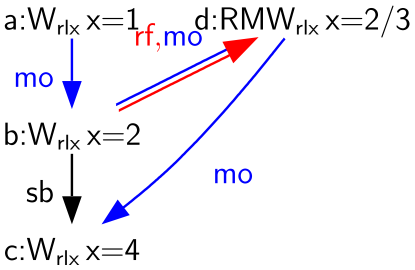


Read-modify-writes

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)

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

[illegible]

The full model

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)	lock xadd(0)
store(non-seq_cst)	mov
store(seq_cst)	lock xchg
fence(non-seq_cst)	no-op

x86-TSO is stronger and simpler.

Top level comparison

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

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

Top level comparison

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

1. $P \mapsto E_1, \ldots, E_n$, each an E_{opsem}

Top level comparison

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

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

Top level comparison

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

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

Top level comparison

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

1. $P \mapsto E_1, \ldots, E_n$, each an E_{opsem}
2. $E_i \mapsto X_{i1}, \ldots, X_{im}$, 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. $P \mapsto E_1, \ldots, E_n$, each an E_{opsem}
2. $E_i \mapsto X_{i1}, \ldots, X_{im}$, 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_{opsem}.

Top level comparison

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

1. $P \mapsto E_1, \ldots, E_n$, each an E_{opsem}
2. $E_i \mapsto X_{i1}, \ldots, X_{im}$, 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_{opsem}.
Execution witnesses, X_{x86} are analogous to X_{witness}.

Top level comparison

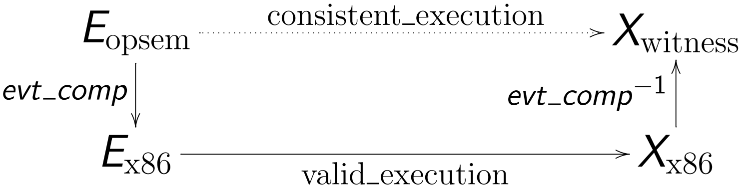
Recall the C/C++ semantics for program P:

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

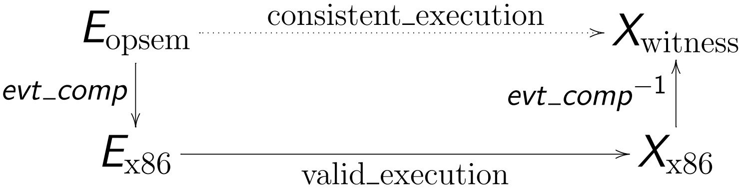
In x86-TSO:

Events and dependencies, E_{x86} are analogous to E_{opsem}.
Execution witnesses, X_{x86} are analogous to X_{witness}.
There is not a DRF semantics.

Theorem


E_{opsem}
consistent_execution
X_{witness}
evt_comp
$\textit{evt\_comp}^{-1}$
E_{x86}
valid_execution
X_{x86}

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.

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded (HOL4).

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded (HOL4).

Derivative models:

- without consume, happens-before is transitive (HOL4).
- DRF programs using only `seq_cst` atomics are SC (false).

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded (HOL4).

Derivative models:

- without consume, happens-before is transitive (HOL4).
- 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...

Usability

Provide simplified models for higher level constructs.

Formal description of mutual exclusion in terms of happens-before.

We need libraries that provide a simpler model to programmers.

Cppmem

helps explore and understand the model

Cppmem

Code in, all executions out

Confidence and speed

Communication

How may a program execute in CPPMEM?

1. $P \mapsto E_1, \ldots, E_n$ — tracking constraints

2. $E_i \mapsto X_{i1}, \ldots, X_{im}$ — automatically uses formal model

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

Refinements to the standards

The current state of the standard

Fixed:

- Happens-before
- Coherence
- `seq_cst` atomics were more broken

The current state of the standard

Fixed:

- Happens-before
- Coherence
- `seq_cst` atomics were more broken

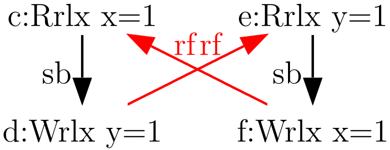
Not fixed:

- Self satisfying conditionals
- `seq_cst` atomics are still not SC

Self-satisfying conditionals

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

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



…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.

Conclusion

It's OK to like the C++0x memory model design

Our formal model lets us make fun things (go use it!)

- Optimized compilation?
- Static analysis?
- Dynamic analysis?
- Observational congruence?
- Program logics?

Conclusion

It's OK to like the C++0x memory model design

Our formal model lets us make fun things (go use it!)

- Optimized compilation?
- Static analysis?
- Dynamic analysis?
- Observational congruence?
- Program logics?

Nitpicking C++ concurrency. J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. In Proc. 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP), 2011.