The C1x and C++11 concurrency model

Mark Batty

University of Cambridge
Sequential consistency
Sequential consistency

Pthreads
Sequential consistency

Pthreads

Java
Sequential consistency

Pthreads

Java

Expose hardware model (e.g. ClightTSO)
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...)
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 single threaded program

```c
int main() {
    int x = 2;
    int y = 0;
    y = (x==x);
    return 0; }
```
A single threaded program

```c
int main() {
    int x = 2;
    int y = 0;
    y = (x==x);
    return 0; }
```
A data race

```c
int y, x = 2;
x = 3;    // y = (x==3);
```

Diagram:

- **a**: Write to `x` after `x = 2`
- **b**: Write to `x` after `x = 3`
- **c**: Read from `x` before `y = (x==3)`
- **d**: Write to `y` after `y = (x==3)`

State transitions:
- `a` to `b`: `x` is updated to 3
- `b` to `c`: `y` is updated to 0
- `c` to `d`: `y` is updated to 0

States:
- **W<sub>na</sub> x=2**: Initial state
- **W<sub>na</sub> x=3**: After `x` is updated to 3
- **R<sub>na</sub> x=2**: Before `y` is updated to 0
- **W<sub>na</sub> y=0**: After `y` is updated to 0
int y, x = 2;
x = 3;
| y = (x==3);

A data race
Simple concurrency: Decker’s example and SC

```c
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);
```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  // c: W_{sc} y = 1
y.load(seq_cst);      // d: R_{sc} x = 0

y.store(1, seq_cst);  // e: W_{sc} x = 1
x.load(seq_cst);      // f: R_{sc} y = 0
Simple concurrency: Decker’s example and SC

```c
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

FORBIDDEN

e:W_{sc} x=1
sb
f:R_{sc} y=0
```
Simple concurrency: Decker’s example and SC

```c
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  
d:R_{sc} x=0  
e:W_{sc} x=1  
f:R_{sc} y=1
```
let sc_reads_restricted actions rf sc mo hb =
\[ \forall (a, b) \in rf. \]
\[ \text{is_seq_cst } b \rightarrow \]
\[ ((\text{adjacent_less_than_such_that} \]
\[ (\text{fun } c \rightarrow \text{is_write } c \land \text{same_location } b c) \]
\[ \text{sc actions } a b) \]
\[ \lor \ldots) \]

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;

\[
\begin{array}{c}
\text{a: } W_{na} x=1 \\
\text{b: } W_{rel} y=1 \\
\text{c: } R_{acq} y=1 \\
\text{d: } R_{na} x=1 \\
\end{array}
\]
Expert concurrency: The release-acquire idiom

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

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

a: \text{W}_{\text{na}} x=1 \\
\text{sb} \\
b: \text{W}_{\text{rel}} y=1 \\
\text{sw} \\
c: \text{R}_{\text{acq}} y=1 \\
\text{sb} \\
d: \text{R}_{\text{na}} x=1
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
x = ...  
y.store(1, release);

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

\[
\text{simple-happens-before} = \text{sequenced-before} \cup \text{synchronizes-with}^+
\]
Locks and unlocks

Unlocks and locks synchronise too:

```c
int x, r;
m_mutex m;

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

```c
m.lock();
r = x;
```
Locks and unlocks

Unlocks and locks synchronise too:

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

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

```
c:L mutex
  sb
  d:W_{na} x=1
  sb
  f:U mutex

h:L mutex
  sb
  i:R_{na} x=1
```
Locks and unlocks

Unlocks and locks synchronise too:

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

```
c:L mutex
  sb
  d:W_{na} x=1
  sb
  f:U mutex
  sb
  sb
  n_a
  sb
  i:R_{na} x=1
  sb
  sc
  h:L mutex
```
Locks and unlocks

Unlocks and locks synchronise too:

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

\[ c:\text{L mutex} \quad \text{h:\ L mutex} \]
\[ d:\text{W}_{na} x=1 \quad i:\text{R}_{na} x=1 \]
\[ f:\text{U mutex} \quad \text{sw} \]

\[ \text{sb} \quad \text{sb} \]
Unlocks and locks synchronise too:

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

m.lock();
x = ...  // Critical section
m.unlock();

m.lock();  // Lock acquired again
r = x;
```

![Diagram showing the synchronization of unlocks and locks](image-url)
Locks and unlocks

Unlocks and locks synchronise too:

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

```c
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

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

```
a:W_{na} x=2
```

```
b:W_{na} x=3  
c:R_{na} x=2
```

```
d:W_{na} y=0
```

A program with a data race has undefined behaviour.
Relaxed writes: load buffering

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

```
c:Rrlx x=1
d:Wrlx y=1
e:Rrlx y=1
f:Wrlx x=1
```

No synchronisation cost, but weakly ordered.
Relaxed writes: independent reads, independent writes

```c
atomic_int x = 0;
atomic_int y = 0;
x.store(1, relaxed);    y.store(2, relaxed);
x.load(relaxed);       y.load(relaxed);
```

```
c:Wrlx x=1    d:Wrlx y=1    e:Rrlx x=1    g:Rrlx y=1
```

```
rf
```

```
sb
```

```
f:Rrlx y=0    h:Rrlx x=0
```
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
x = ...
y.store(1, release);

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

// 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;

c:W_{na} x=1
sb

\[\text{rf}\]

d:W_{rel} y=1

\[\text{sw}\]
e:R_{rlx} y=1
sb

\[\text{f}\]
f:R_{acq}

sb

g:R_{na} x=1
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.

```plaintext
x.store(1, relaxed);
x.store(2, relaxed);
x.load(relaxed);
x.load(relaxed);
```
**Modification order** is a per-location total order over atomic writes of any memory order.

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

```
b:W_{rlx} x=1
  sb
  c:W_{rlx} x=2
  d:R_{rlx} x=1
    rf
    sb
  e:R_{rlx} x=2
    rf
```
**Modification order** is a per-location total order over atomic writes of any memory order.

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

```
b:W_{rlx} x=1
  mo

c:W_{rlx} x=2

d:R_{rlx} x=1
  rf
  sb

e:R_{rlx} x=2
  rf
```
Coherence and atomic reads

All forbidden!

Atoms 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`:

```c
x.store(1, relaxed);  // compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed);
x.store(4, relaxed);
```
A successful `compare_exchange` is a read-modify-write.

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

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

```
  a:W_{rlx} x=1  d:RMW_{rlx} x=2/3
      sb
    sb
  b:W_{rlx} x=2
      sb
    sb
  c:W_{rlx} x=4
```
A successful `compare_exchange` is a read-modify-write.

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

```plaintext
x.store(1, relaxed);  // a: W_{rlx} x=1
x.store(2, relaxed);  // b: W_{rlx} x=2
x.store(4, relaxed);  // c: W_{rlx} x=4

compare_exchange(&x, 2, 3, relaxed, relaxed);  // d: RMW_{rlx} x=2/3
```
A successful `compare_exchange` is a read-modify-write.

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

```c
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)
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
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
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
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 full model
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?

<table>
<thead>
<tr>
<th>Operation</th>
<th>x86 Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load(non-seq_cst)</td>
<td>mov</td>
</tr>
<tr>
<td>load(seq_cst)</td>
<td>lock xadd(0)</td>
</tr>
<tr>
<td>store(non-seq_cst)</td>
<td>mov</td>
</tr>
<tr>
<td>store(seq_cst)</td>
<td>lock xchg</td>
</tr>
<tr>
<td>fence(non-seq_cst)</td>
<td>no-op</td>
</tr>
</tbody>
</table>

x86-TSO is stronger and simpler.
Recall the C/C++ semantics for program $P$:

1. $P \mapsto E_1, \ldots, E_n$. 
Recall the C/C++ semantics for program $P$:

1. $P \mapsto E_1, \ldots, E_n$, each an $E_{\text{opsem}}$
Recall the C/C++ semantics for program $P$:

1. $P \mapsto E_1, ..., E_n$, each an $E_{opsem}$
2. $E_i \mapsto X_{i1}, ..., X_{im}$,
Recall the C/C++ semantics for program \( P \):

1. \( P \mapsto E_1, \ldots, E_n \), each an \( E_{\text{opsem}} \)
2. \( E_i \mapsto X_{i1}, \ldots, X_{im} \), collectively \( X_{\text{witness}} \)
Recall the C/C++ semantics for program \( P \):

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

In x86-TSO:

Events and dependencies, $E_{\text{x86}}$ are analogous to $E_{\text{opsem}}$. 
Recall the C/C++ semantics for program $P$:

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

In x86-TSO:

Events and dependencies, $E_{\text{x86}}$ are analogous to $E_{\text{opsem}}$.
Execution witnesses, $X_{\text{x86}}$ are analogous to $X_{\text{witness}}$. 
Recall the C/C++ semantics for program $P$:

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

In x86-TSO:

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

\[ E_{\text{opsem}} \xrightarrow{\text{consistent\_execution}} X_{\text{witness}} \]

\[ \downarrow \text{evt\_comp} \]

\[ E_{\text{x86}} \xrightarrow{\text{valid\_execution}} X_{\text{x86}} \]
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?

<table>
<thead>
<tr>
<th>C++0x Operation</th>
<th>POWER Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-atomic Load</td>
<td>ld</td>
</tr>
<tr>
<td>Load Relaxed</td>
<td>ld</td>
</tr>
<tr>
<td>Load Consume</td>
<td>ld (and preserve dependency)</td>
</tr>
<tr>
<td>Load Acquire</td>
<td>ld; cmp; bc; isync</td>
</tr>
<tr>
<td>Load Seq Cst</td>
<td>sync; ld; cmp; bc; isync</td>
</tr>
<tr>
<td>Non-atomic Store</td>
<td>st</td>
</tr>
<tr>
<td>Store Relaxed</td>
<td>st</td>
</tr>
<tr>
<td>Store Release</td>
<td>lwsync; st</td>
</tr>
<tr>
<td>Store Seq Cst</td>
<td>sync; st</td>
</tr>
</tbody>
</table>

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

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

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

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).

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

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

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