Multicore Programming: C++0x

Mark Batty

University of Cambridge

in collaboration with

Scott Owens, Susmit Sarkar, Peter Sewell, Tjark Weber

November, 2010
C++0x: the next C++

Specified by the C++ Standards Committee

Defined in The Standard, a 1300 page prose document

The design is a detailed compromise:

- performance, optimisations and hardware
- usability
- compatibility with the next C, C1X
- legacy code
Our mathematical model is faithful to the intent of, and has influenced The Standard

The model:

- syntactically separates out expert features
- has a weak memory
- defines a happens-before relation
- requires non-atomic reads and writes to be DRF
- provides atomic reads and writes for racy programs
The syntactic divide

An example of the syntax

// for regular programmers:
atomic_int x = 0;
x.store(1);
y = x.load();

// for experts:
x.store(2, memory_order);
y = x.load(memory_order);
atomic_thread_fence(memory_order);

With a choice of memory_order

mo_seq_cst mo_release mo_acquire
mo_acq_rel mo_consume mo_relaxed
A model of two parts

An operational semantics:

Processes programs, identifying *memory actions*

Constructs candidate executions, $E_{\text{opsem}}$

An axiomatic memory model:

Judges $E_{\text{opsem}}$ paired with a memory ordering, $X_{\text{witness}}$

Searches the consistent executions for races and unconstrained reads
cpp_memory_model $opsem (p : \text{program}) =$

let pre_executions = \{(E_{opsem}, X_{\text{witness}}). \}

\begin{align*}
& \text{opsem } p \ E_{opsem} \land \\
& \text{consistent_execution } (E_{opsem}, X_{\text{witness}}) \} \text{ in} \\
& \text{if } \exists X \in \text{pre_executions} . \\
& \quad (\text{indeterminate} \_ \text{reads } X \neq \{\}) \lor \\
& \quad (\text{unsequenced} \_ \text{races } X \neq \{\}) \lor \\
& \quad (\text{data} \_ \text{races } X \neq \{\}) \\
& \text{then NONE} \\
& \text{else SOME } \text{pre_executions} \\
\end{align*}
The relations of a pre-execution

An $E_{\text{opsem}}$ part containing:

- \text{sb} — sequenced before, program order
- \text{asw} — additional synchronizes with, inter-thread ordering
- \text{dd} — data-dependence

An $X_{\text{witness}}$ part containing:

- \text{rf} — relates a write to any reads that take its value
- \text{sc} — a total order over $\text{mo\_seq\_cst}$ and mutex actions
- \text{mo} — modification order, per location total order of writes
int main() {
    int x = 2;
    int y = 0;
    y = (x == x);
    return 0;
}
Memory actions

action ::= 
  a:R_{na} x=v \quad \text{non-atomic read} \\
  a:W_{na} x=v \quad \text{non-atomic write} \\
  a:R_{mo} x=v \quad \text{atomic read} \\
  a:W_{mo} x=v \quad \text{atomic write} \\
  a:RMW_{mo} x=v_1/v_2 \quad \text{atomic read-modify-write} \\
  a:L x \quad \text{lock} \\
  a:U x \quad \text{unlock} \\
  a:F_{mo} \quad \text{fence}
Memory orders

Memory orders are shown as follows:

\[ mo ::= \]
\[
\begin{align*}
  SC & \quad \text{memory\_order\_seq\_cst} \\
  | & \quad \text{memory\_order\_relaxed} \\
  | & \quad \text{memory\_order\_release} \\
  | & \quad \text{memory\_order\_acquire} \\
  | & \quad \text{memory\_order\_consume} \\
  | & \quad \text{memory\_order\_acq\_rel}
\end{align*}
\]
Location kinds

\[
\text{location\_kind} = \\
\text{MUTEX} \\
| \text{NON\_ATOMIC} \\
| \text{ATOMIC}
\]

\[
\text{actions\_respect\_location\_kinds} = \\
\forall a. \\
\text{case location } a \text{ of SOME } l \rightarrow \\
(\text{case location\_kind } l \text{ of} \\
\text{MUTEX } \rightarrow \text{is\_lock\_or\_unlock } a \\
| \text{NON\_ATOMIC } \rightarrow \text{is\_load\_or\_store } a \\
| \text{ATOMIC } \rightarrow \text{is\_load\_or\_store } a \lor \text{is\_atomic\_action } a \\
| \text{NONE } \rightarrow T)
\]
int main() {
    int x = 2;
    int y = 0;
    y = (x == x);
    return 0;
}
Unsequenced race

unsequenced_races = \{(a, b)\}

\begin{align*}
\text{is\_load\_or\_store} \ a & \land \text{is\_load\_or\_store} \ b \land \\
(a \neq b) & \land \text{same\_location} \ a \ b \land (\text{is\_write} \ a \lor \text{is\_write} \ b) \land \\
\text{same\_thread} \ a \ b & \land \\
\neg & (a \xrightarrow{\text{sequenced\_before}} b \lor b \xrightarrow{\text{sequenced\_before}} a) \}\
\end{align*}
int main() {
    int x = 2;
    int y = 0;
    y = (x == (x=3));
    return 0;
}
void foo(int* p) {*p=3;}

int main() {
    int x = 2;
    int y;
    thread t1(foo, &x);
    y = 3;
    t1.join();
    return 0; }

becomes:

int main() {
    int x = 2;
    int y;
    {{
        x = 3;
        y = 3;
    }}
    return 0; }

../examples/t3-parallel.c
Synchronizes-with and happens-before

The parent thread has synchronization edges, labeled asw, to its child threads. There are other ways to synchronize.

We will define the happens-before relation later. It contains the transitive closure of all synchronization edges and all sequenced before edges (amongst other things).
Data race

data_races = \{ (a, b). \\
(a \neq b) \land \text{same\_location} \ a \ b \land (\text{is\_write} \ a \lor \text{is\_write} \ b) \land \\
\neg \text{same\_thread} \ a \ b \land \\
\neg (\text{is\_atomic\_action} \ a \land \text{is\_atomic\_action} \ b) \land \\
\neg (a \xrightarrow{\text{happens\_before}} b \lor b \xrightarrow{\text{happens\_before}} a)\}
int main() {
    int x = 2;
    int y;
    {{{ x=3;
        ||| y=(x==3);
    }}}};
    return 0; }

A data race
Modification order

A total order of the writes at each atomic location, similar to coherence order on Power

```c
int main() {
    atomic_int x = 0;
    int y = 0;
    {{
      { x.store(1);
        x.store(2); }
    ||| { y = 1; }
    }}
    return 0; }
```
There is a total order over all sequentially consistent atomic actions. SC atomics read the last prior write in SC order (or a non SC write).

\[
\text{consistent\_sc\_order =}
\begin{align*}
\text{let} & \quad \text{sc\_happens\_before} = \frac{\text{happens\_before}}{\text{all\_sc\_actions}} \in \\
\text{let} & \quad \text{sc\_mod\_order} = \frac{\text{modification\_order}}{\text{all\_sc\_actions}} \in \\
\text{strict\_total\_order\_over} & \quad \text{all\_sc\_actions} (\xrightarrow{\text{sc}}) \land \\
\text{sc\_happens\_before} & \quad \subseteq \xrightarrow{\text{sc}} \land \\
\text{sc\_mod\_order} & \quad \subseteq \xrightarrow{\text{sc}} \\
\end{align*}
\]
Atomic actions do not race

```c
int main() {
    atomic_int x;
    x.store(2, mo_seq_cst);
    int y = 0;
    {{
        x.store(3);
        y = ((x.load()) == 3);
    }};
    return 0; }
```
The release-acquire idiom

// sender
x = ...  
y = 1;

// receiver
while (0 == y);
r = x;

../examples/t15.c
Release-acquire synchronization

a: $W_{na} \ x=1$

b: $W_{REL} \ y=1$

c: $W_{RLX} \ y=2$

d: $R_{ACQ} \ y=2$

e: $R_{na} \ x=1$

../examples/t8a.c
The release sequence

The release sequence is a sub-sequence of the modification order following a release

\[
\begin{align*}
\text{rs_element } rs\_head \ a &= \\
\text{same_thread } a \ rs\_head \lor \text{is\_atomic\_rmw } a
\end{align*}
\]

\[
\begin{align*}
a_{rel} \xrightarrow{\text{release-sequence}} b &= \\
\text{is\_at\_atomic\_location } b \land \\
\text{is\_release } a_{rel} \land ( \\
\ (b = a_{rel}) \lor \\
\ (\text{rs\_element } a_{rel} b \land a_{rel} \xrightarrow{\text{modification-order}} b \land \\
\ (\forall c. a_{rel} \xrightarrow{\text{modification-order}} c \xrightarrow{\text{modification-order}} b) \implies \\
\ (\text{rs\_element } a_{rel} c)))
\end{align*}
\]
An execution with a release sequence

a: \(W_{na} x=1\)

b: \(W_{REL} y=1\)

c: \(W_{RLX} y=2\)

d: \(R_{ACQ} y=2\)

e: \(R_{na} x=1\)

..//examples/t8a-no-sw.c
Synchronizes-with

\[ a \xrightarrow{\text{synchronizes-with}} b = \]

(* – additional synchronization, from thread create etc. – *)
\[ a \xrightarrow{\text{additional-synchronized-with}} b \lor \]

(same_location \(a, b\) \(\land\) \(a \in \text{actions}\) \(\land\) \(b \in \text{actions}\) \(\land\) ( (* – mutex synchronization – *)
(is_unlock \(a\) \(\land\) is_lock \(b\) \(\land\) \(a \xrightarrow{sc} b\)) \lor

(* – release/acquire synchronization – *)
(is_release \(a\) \(\land\) is_acquire \(b\) \(\land\) \(\neg\) same_thread \(a, b\) \(\land\)
\((\exists c. a \xrightarrow{\text{release-sequence}} c \xrightarrow{rf} b))) \lor
[[...]])
Release-acquire synchronization

a: \( W_{na} \ x=1 \)

\[ \text{sb} \]

b: \( W_{REL} \ y=1 \)

\[ \text{sb, mo, rs} \]

c: \( W_{RLX} \ y=2 \)

\[ \text{sw} \]

d: \( R_{ACQ} \ y=2 \)

\[ \text{rf} \]

e: \( R_{na} \ x=1 \)

\[ \text{sb} \]

.. \text{./examples/t8a.c}
Happens-before (without consume)

\[\text{simple\_happens\_before} = \text{sequenced\_before} \cup \text{synchronizes\_with}^+\]

\[\text{consistent\_simple\_happens\_before} = \text{irreflexive (simple\_happens\_before)}\]
Happens-before

\[
\begin{align*}
\text{let } r &= \text{synchronizes-with} \cup \text{dependency-ordered-before} \\
&\quad \cup (\text{synchronizes-with} \circ \text{sequenced-before}) \\
&\quad \cup (\text{sequenced-before} \circ r) \\
\end{align*}
\]

consistent_inter_thread_happens_before =
irreflexive \( \text{inter-thread-happens-before} \)

\[
\begin{align*}
\text{happens-before} &= \text{sequenced-before} \cup \text{inter-thread-happens-before} \\
\end{align*}
\]
Visible side effect

Non-atomic reads read from one of their visible side effects

\[
\begin{align*}
& a \xrightarrow{\text{visible-side-effect}} b = \\
& a \xrightarrow{\text{happens-before}} b \land \\
& \text{is\_write } a \land \text{is\_read } b \land \text{same\_location } a b \land \\
& \neg(\exists c. (c \neq a) \land (c \neq b) \land \\
& \quad \text{is\_write } c \land \text{same\_location } c b \land \\
& \quad a \xrightarrow{\text{happens-before}} c \xrightarrow{\text{happens-before}} b)
\end{align*}
\]
Visible sequence of side effects

Atomic reads read from a write in one of their visible sequences of side effects.

\[
\text{visible_sequence_of_side_effects_tail} \ vsse\_head\ b = \\
\{ \ c. \ vsse\_head \xrightarrow{\text{modification-order}} c \land \\
\neg (b \xrightarrow{\text{happens-before}} c) \land \\
(\forall a. \ vsse\_head \xrightarrow{\text{modification-order}} a \xrightarrow{\text{modification-order}} c \\
\implies \neg (b \xrightarrow{\text{happens-before}} a)) \}\}
\]
An atomic read

a: $W_{\text{na}} x=1$

b: $W_{\text{REL}} y=1$

c: $W_{\text{RLX}} y=2$

d: $R_{\text{ACQ}} y=2$

e: $R_{\text{na}} x=1$

../examples/t8a.c
Consistent reads-from mapping

\[
\text{consistent\_reads\_from\_mapping} = \\
(\forall b. (\text{is\_read } b \land \text{is\_at\_non\_atomic\_location } b) \implies \\
(\text{if } (\exists a_{\text{vse}}. a_{\text{vse}} \xrightarrow{\text{visible\_side\_effect}} b) \\
\text{then } (\exists a_{\text{vse}}. a_{\text{vse}} \xrightarrow{\text{visible\_side\_effect}} b \land a_{\text{vse}} \xrightarrow{rt} b) \\
\text{else } \neg (\exists a. a \xrightarrow{rt} b)) \land \\
(\forall b. (\text{is\_read } b \land \text{is\_at\_atomic\_location } b) \implies \\
(\text{if } (\exists b'. \text{vsse}) \in \text{visible\_sequences\_of\_side\_effects}. (b' = b)) \\text{then } (\exists (b', \text{vsse}) \in \text{visible\_sequences\_of\_side\_effects}. \\
(b' = b) \land (\exists c \in \text{vsse}. c \xrightarrow{rt} b)) \\
\text{else } \neg (\exists a. a \xrightarrow{rt} b)) \land \\
(\forall (x, a) \in \xrightarrow{rt}. \\
\forall (y, b) \in \xrightarrow{rt}. \\
a \xrightarrow{\text{happens\_before}} b \land \\
\text{same\_location } a b \land \text{is\_at\_atomic\_location } b \\
\implies (x = y) \lor x \xrightarrow{\text{modification\_order}} y) \land \\
(\forall (a, b) \in \xrightarrow{rt}. \text{is\_atomic\_rmw } b \\
\implies a \xrightarrow{\text{modification\_order}} b) \land \\
(\forall (a, b) \in \xrightarrow{rt}. \text{is\_seq\_cst } b \\
\implies \neg \text{is\_seq\_cst } a \lor \\
a \xrightarrow{sc} \lambda c. \text{is\_write } c \land \text{same\_location } b c) \land \\
[\ldots]
\]
Coherence

Coherence is defined as the absence of four execution fragments:

a: \(W_{RLX} x=1\)  \(\rightarrow\)  \(c: R_{RLX} x=1\)

b: \(W_{RLX} x=2\)  \(\rightarrow\)  \(d: R_{RLX} x=2\)

a: \(W_{RLX} x=1\)  \(\rightarrow\)  \(b: R_{RLX} x=2\)

b: \(W_{RLX} x=2\)  \(\rightarrow\)  \(c: R_{RLX} x=2\)

\text{./examples/coherence-axiom-1.exc}

\text{./examples/coherence-axiom-2.exc}

\text{./examples/coherence-axiom-4.exc}

\text{./examples/coherence-axiom-3.exc}
Concurrency examples that can be observed

The model allows the following non-SC behaviour:

- message passing (RLX, REL-CON)
- store buffering (REL-ACQ, RLX, REL-CON)
- load buffering (RLX, CON)
- write-to-read causality (RLX, CON)
- IRIW (REL-ACQ, RLX, REL-CON)

...but DRF programs that use only the
memory_order_seq_cst atomics should be sequentially consistent
### An execution compiler

<table>
<thead>
<tr>
<th>Operation</th>
<th>x86 Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load non-SC</td>
<td>mov</td>
</tr>
<tr>
<td>Load Seq_cst</td>
<td>lock xadd(0)</td>
</tr>
<tr>
<td>Store non-SC</td>
<td>mov</td>
</tr>
<tr>
<td>Store Seq_cst</td>
<td>lock xchg</td>
</tr>
<tr>
<td>Fence non-SC</td>
<td>no-op</td>
</tr>
<tr>
<td>Fence Seq_cst</td>
<td>mfence</td>
</tr>
</tbody>
</table>

OR: mfence, mov

OR: mov, mfence
Theorem

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

\[ E_{\text{x86}} \xleftarrow{\text{valid\_execution}} X_{\text{x86}} \]

\[ \xrightarrow{\text{evt\_comp}} \quad \xleftarrow{\text{evt\_comp}^{-1}} \]
Conclusion

C++0x offers a simple model to normal programmers while experts get a highly configurable language that abstracts the hardware memory model.

We have arrived just in time to point out a few bugs, and many changes have been made as a result of our work.

The intricacy of such models makes tools important, CPPMEM helps in exploring and understanding the model.

Formal models provide an opportunity to provide guarantees about programs based on the specification, like our compiler correctness result.