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

C++0x: the next C++

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_{opsem}

An axiomatic memory model:

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

Searches the consistent executions for races and unconstrained reads

Judgement of the axiomatic model

$$
\begin{array}{l}
\text{cpp\_memory\_model } \mathit{opsem}\ (p : \textsf{program}) = \\
\quad \textbf{let}\ \ \text{pre\_executions} = \{(E_{\text{opsem}}, X_{\text{witness}}). \\
\qquad\qquad \mathit{opsem}\ p\ E_{\text{opsem}} \wedge \\
\qquad\qquad \text{consistent\_execution}\ (E_{\text{opsem}}, X_{\text{witness}})\}\ \textbf{in} \\
\qquad \textbf{if}\ \exists X \in\ \text{pre\_executions}\,. \\
\qquad\qquad (\text{indeterminate\_reads}\ X \neq \{\}) \vee \\
\qquad\qquad (\text{unsequenced\_races}\ X \neq \{\}) \vee \\
\qquad\qquad (\text{data\_races}\ X \neq \{\}) \\
\qquad \textbf{then}\ \textsc{None} \\
\qquad \textbf{else}\ \textsc{Some}\ \text{pre\_executions}
\end{array}
$$

The relations of a pre-execution

An E_{opsem} part containing:

sb — *sequenced before*, program order

asw — *additional synchronizes with*, inter-thread ordering

dd — *data-dependence*

An X_{witness} part containing:

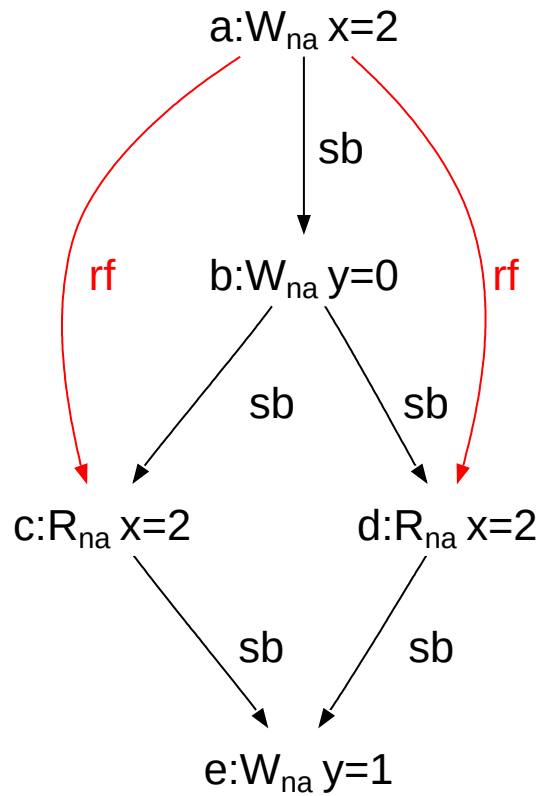
rf — relates a write to any reads that take its value

sc — a total order over `mo_seq_cst` and mutex actions

mo — *modification order*, per location total order of writes

A single threaded program

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



../examples/t1.c

Memory actions

action ::=

$a{:}R_{na}\ x{=}v$	non-atomic read
$a{:}W_{na}\ x{=}v$	non-atomic write
$a{:}R_{mo}\ x{=}v$	atomic read
$a{:}W_{mo}\ x{=}v$	atomic write
$a{:}RMW_{mo}\ x{=}v1/v2$	atomic read-modify-write
$a{:}L\ x$	lock
$a{:}U\ x$	unlock
$a{:}F_{mo}$	fence

Memory orders

Memory orders are shown as follows:

mo ::=

SC	`memory_order_seq_cst`
RLX	`memory_order_relaxed`
REL	`memory_order_release`
ACQ	`memory_order_acquire`
CON	`memory_order_consume`
A/R	`memory_order_acq_rel`

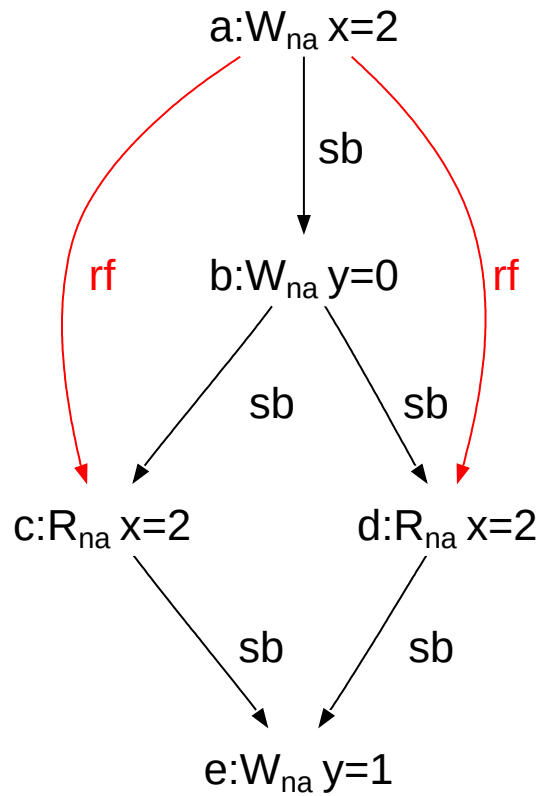
Location kinds

location_kind =
  MUTEX
| NON_ATOMIC
| ATOMIC

actions_respect_location_kinds =
  $\forall a.$
    case location a **of** SOME $l \rightarrow$
      (**case** *location-kind* l **of**
        MUTEX $\rightarrow$ is_lock_or_unlock a
      || NON_ATOMIC $\rightarrow$ is_load_or_store a
      || ATOMIC $\rightarrow$ is_load_or_store $a \vee$ is_atomic_action a)
    || NONE $\rightarrow$ **T**

That single threaded program again

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



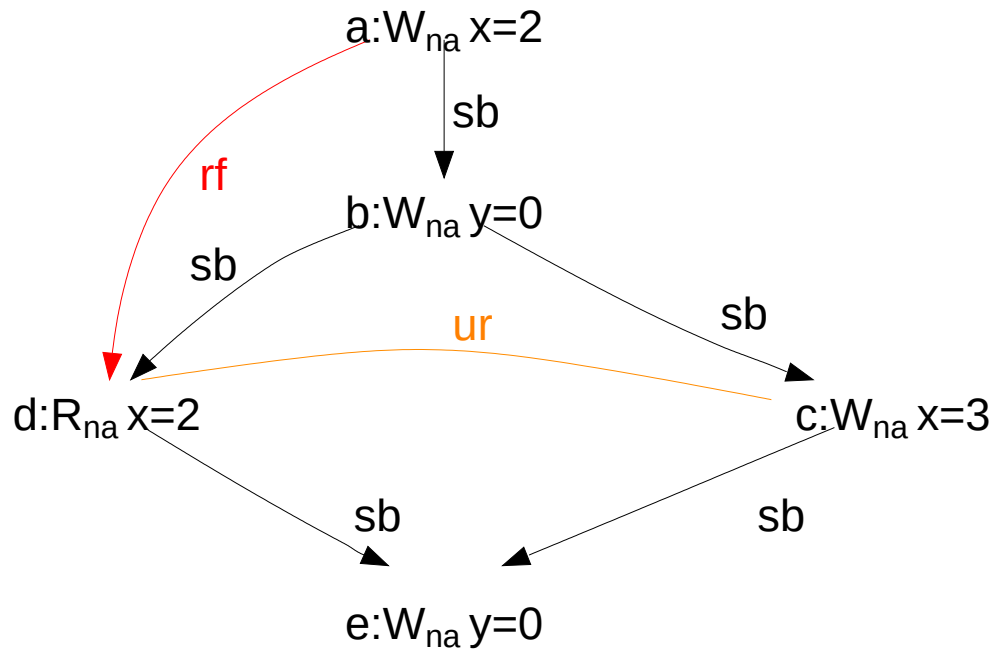
../examples/t1.c

Unsequenced race

$$\begin{aligned}&\text{unsequenced\_races} = \{(a, b).\\ &\quad \text{is\_load\_or\_store}\ \ a \wedge \text{is\_load\_or\_store}\ \ b \wedge\\ &\quad (a \neq b) \wedge \text{same\_location}\ \ a\ b \wedge (\text{is\_write}\ \ a \vee \text{is\_write}\ \ b) \wedge\\ &\quad \text{same\_thread}\ \ a\ b \wedge\\ &\quad \neg(a \xrightarrow{\textit{sequenced-before}} b \vee b \xrightarrow{\textit{sequenced-before}} a)\}\end{aligned}$$

An unsequenced race

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

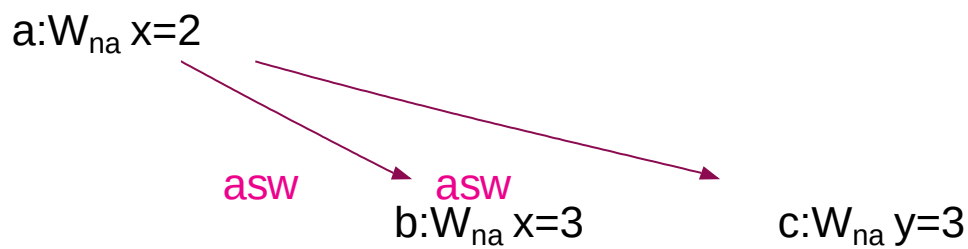


A multi-threaded program

```
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.

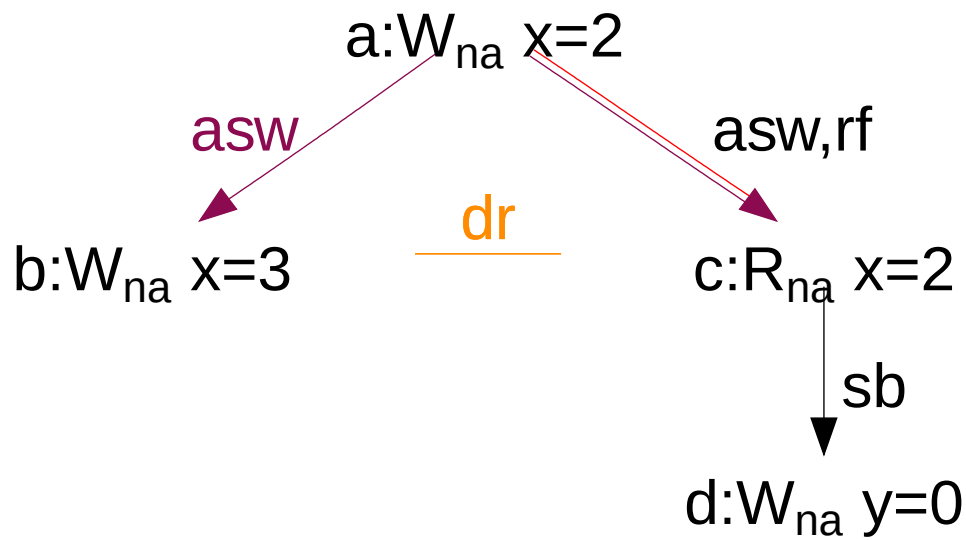
We will define the happens-before relation later. It contains the transitive closure of all synchronization edges and all sequenced before edges (amongst other things).

Data race

$$\begin{aligned}
&\text{data_races} = \{(a, b).\\
&\quad (a \neq b) \wedge \text{same_location}\ \ a\ b \wedge (\text{is_write}\ \ a \vee \text{is_write}\ \ b) \wedge\\
&\quad \neg\, \text{same_thread}\ \ a\ b \wedge\\
&\quad \neg(\text{is_atomic_action}\ \ a \wedge \text{is_atomic_action}\ \ b) \wedge\\
&\quad \neg(a \xrightarrow{\textit{happens-before}} b \vee b \xrightarrow{\textit{happens-before}} a)\}
\end{aligned}$$

A data race

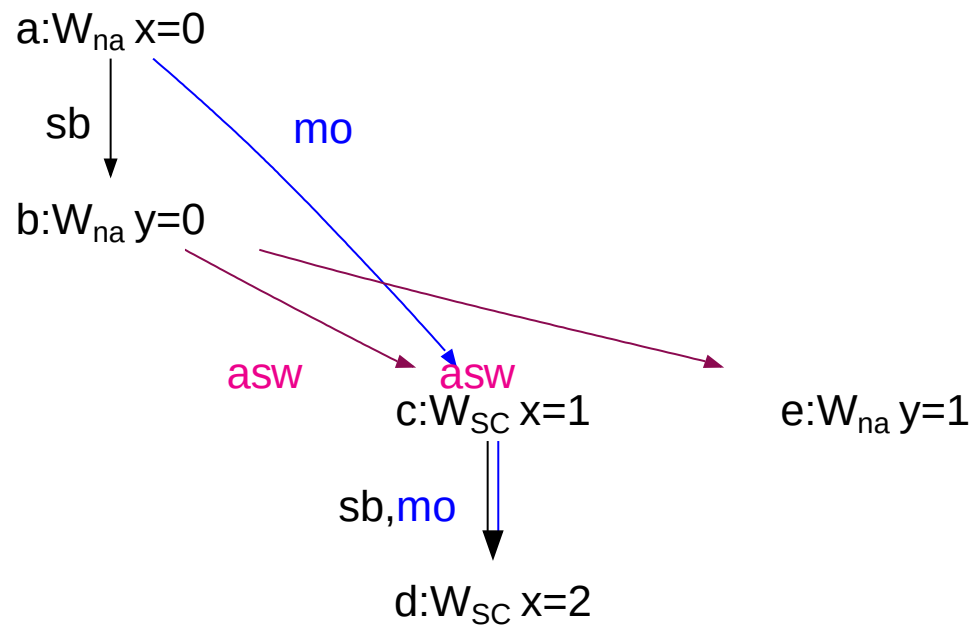
```
int main() {
  int x = 2;
  int y;
  {{{  x=3;
  |||  y=(x==3);
  }}};
  return 0; }
```



Modification order

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

```
int main() {
  atomic_int x = 0;
  int y = 0;
  {{{ { x.store(1);
        x.store(2); }
  ||| { y = 1; }
  }}}
  return 0; }
```



../examples/t70-na-mo.c

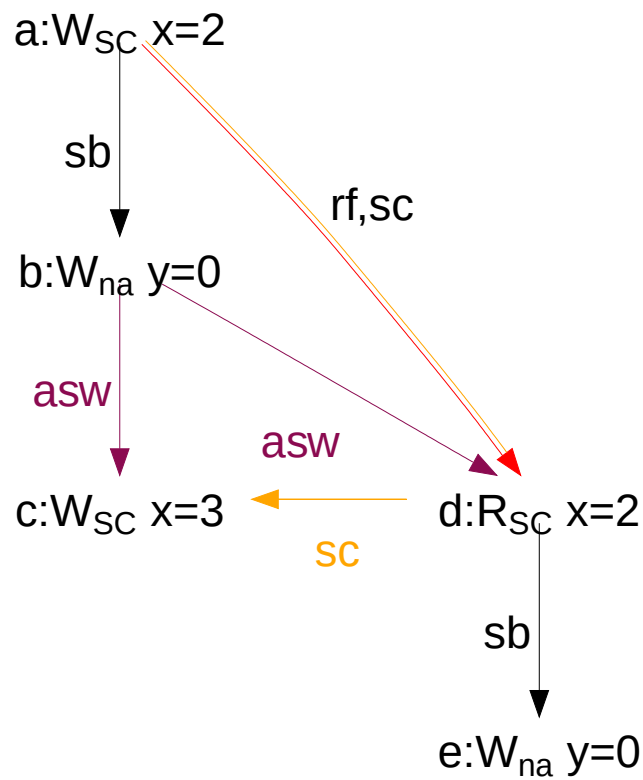
SC order

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).

$$
\begin{aligned}
&\text{consistent\_sc\_order} = \\
&\quad \textbf{let} \;\; \text{sc\_happens\_before} = \xrightarrow{\textit{happens-before}}\big|_{\text{all\_sc\_actions}} \;\textbf{in} \\
&\quad \textbf{let} \;\; \text{sc\_mod\_order} = \xrightarrow{\textit{modification-order}}\big|_{\text{all\_sc\_actions}} \;\textbf{in} \\
&\quad \text{strict\_total\_order\_over} \;\; \text{all\_sc\_actions} \; (\xrightarrow{sc}) \wedge \\
&\quad \xrightarrow{\text{sc\_happens\_before}} \subseteq \xrightarrow{sc} \wedge \\
&\quad \xrightarrow{\text{sc\_mod\_order}} \subseteq \xrightarrow{sc}
\end{aligned}
$$

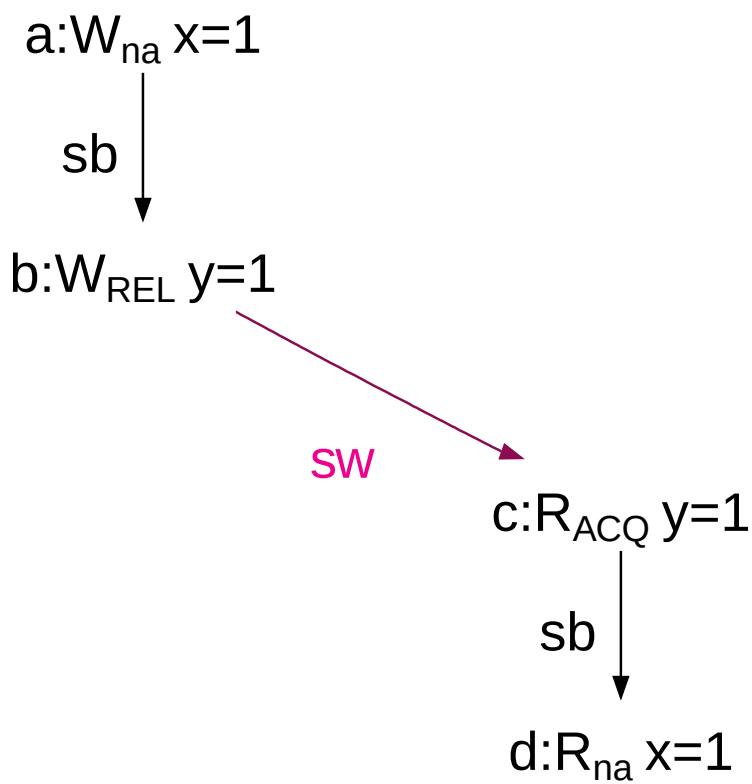
Atomic actions do not race

```
int main() {
  atomic_int x;
  x.store(2, mo_seq_cst);
  int y = 0;
  {{{ x.store(3);
  ||| y = ((x.load()) == 3);
  }}};
  return 0; }
```



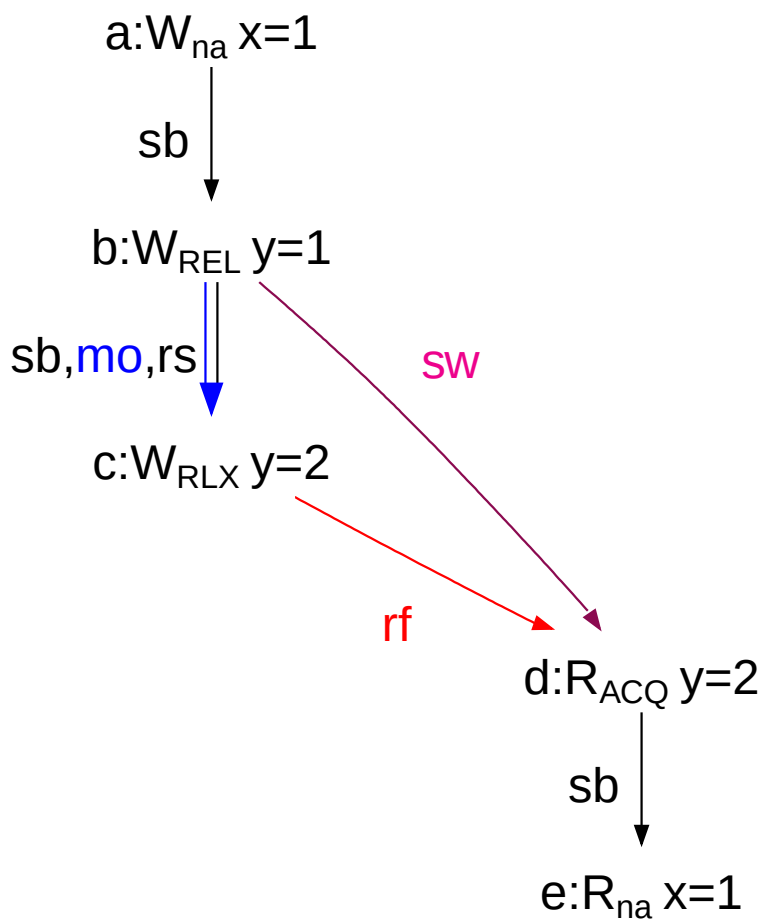
The release-acquire idiom

```
// sender          | // receiver
x = ...            | while (0 == y);
y = 1;             | r = x;
```



../examples/t15.c

Release-acquire synchronization



../examples/t8a.c

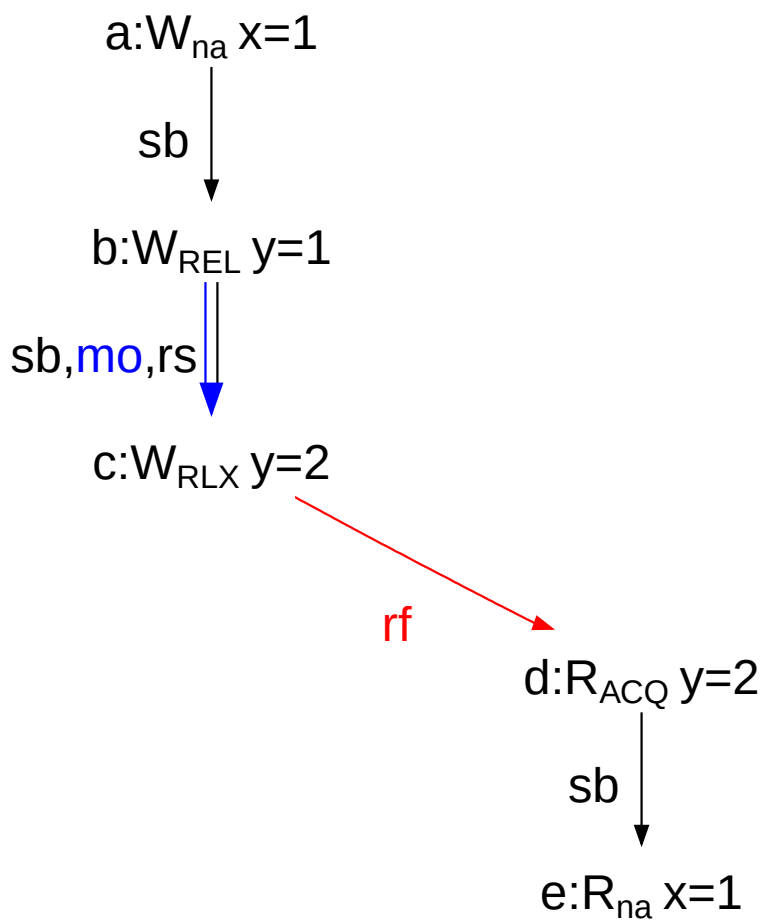
The release sequence

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

$$\text{rs\_element}\ \ rs\_head\ \ a =$$
$$\text{same\_thread}\ \ a\ \ rs\_head \vee \text{is\_atomic\_rmw}\ \ a$$

$$a_{rel} \xrightarrow{\textit{release-sequence}} b =$$
$$\text{is\_at\_atomic\_location}\ \ b \wedge$$
$$\text{is\_release}\ \ a_{rel} \wedge ($$
$$(b = a_{rel}) \vee$$
$$(\text{rs\_element}\ \ a_{rel}\ \ b \wedge a_{rel} \xrightarrow{\textit{modification-order}} b \wedge$$
$$(\forall c.\ a_{rel} \xrightarrow{\textit{modification-order}} c \xrightarrow{\textit{modification-order}} b \implies$$
$$\text{rs\_element}\ \ a_{rel}\ \ c)))$$

An execution with a release sequence



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

Synchronizes-with

$a \xrightarrow{\textit{synchronizes-with}} b =$

(* – additional synchronization, from thread create etc. – *)

$a \xrightarrow{\textit{additional-synchronized-with}} b \vee$

$(\text{same\_location}\ \ a\ b \wedge a \in \mathit{actions} \wedge b \in \mathit{actions} \wedge ($

(* – mutex synchronization – *)

$(\text{is\_unlock}\ \ a \wedge \text{is\_lock}\ \ b \wedge a \xrightarrow{\textit{sc}} b) \vee$

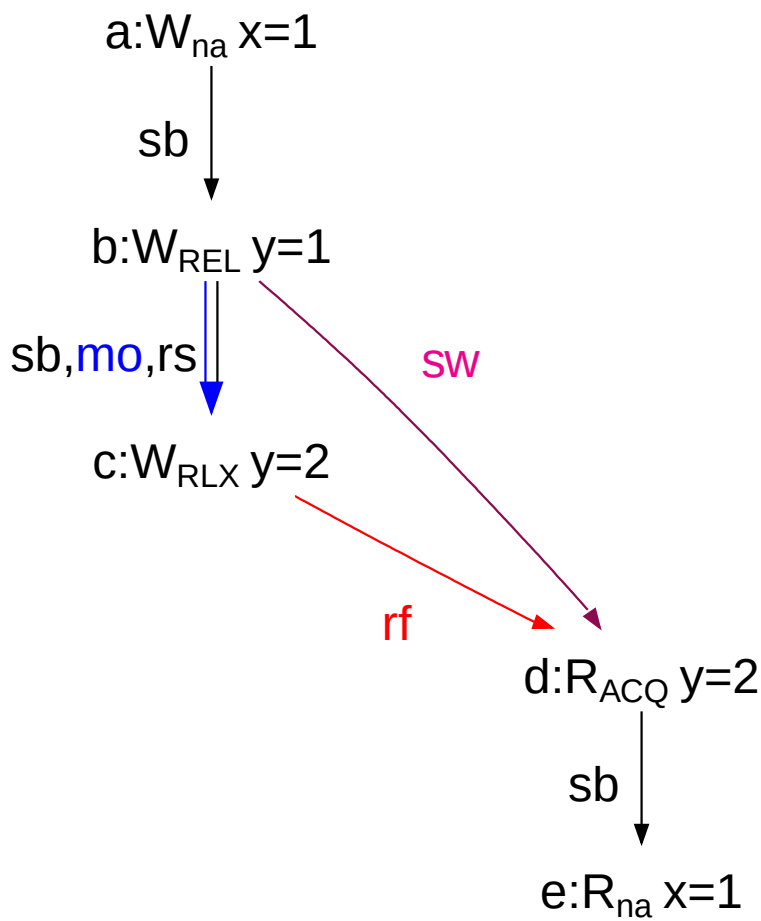
(* – release/acquire synchronization – *)

$(\text{is\_release}\ \ a \wedge \text{is\_acquire}\ \ b \wedge \neg\,\text{same\_thread}\ \ a\ b \wedge$

$(\exists c.\ a \xrightarrow{\textit{release-sequence}} c \xrightarrow{\textit{rf}} b)) \vee$

$[\ldots]))$

Release-acquire synchronization



../examples/t8a.c

Happens-before (without consume)

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

$$consistent\_simple\_happens\_before = \text{irreflexive}\left(\xrightarrow{simple\_happens\_before}\right)$$

Happens-before

$$\xrightarrow{\textit{inter-thread-happens-before}} =$$

$$\textbf{let}\ \ r = \xrightarrow{\textit{synchronizes-with}} \cup \xrightarrow{\textit{dependency-ordered-before}} \cup \left(\xrightarrow{\textit{synchronizes-with}} \circ \xrightarrow{\textit{sequenced-before}}\right)\ \textbf{in}$$

$$\left(\xrightarrow{r} \cup \left(\xrightarrow{\textit{sequenced-before}} \circ \xrightarrow{r}\right)\right)^{+}$$

$$\text{consistent\_inter\_thread\_happens\_before} = \text{irreflexive}\left(\xrightarrow{\textit{inter-thread-happens-before}}\right)$$

$$\xrightarrow{\textit{happens-before}} = \xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{inter-thread-happens-before}}$$

Visible side effect

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

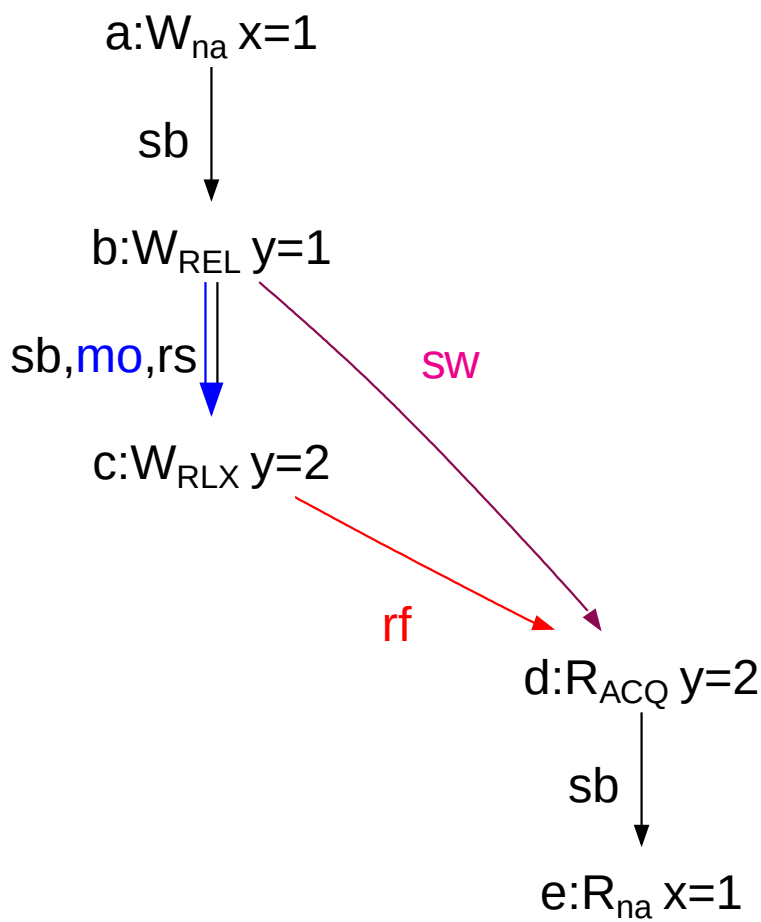
$$
\begin{aligned}
& a \xrightarrow{\textit{visible-side-effect}} b = \\
& \quad a \xrightarrow{\textit{happens-before}} b \wedge \\
& \quad \text{is\_write } a \wedge \text{is\_read } b \wedge \text{same\_location } a\ b \wedge \\
& \quad \neg(\exists c.\ (c \neq a) \wedge (c \neq b) \wedge \\
& \qquad \text{is\_write } c \wedge \text{same\_location } c\ b \wedge \\
& \qquad a \xrightarrow{\textit{happens-before}} c \xrightarrow{\textit{happens-before}} b)
\end{aligned}
$$

Visible sequence of side effects

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

$$\begin{aligned}
&\text{visible_sequence_of_side_effects_tail}\ \ vsse_head\ b = \\
&\quad \{c.\ vsse_head \xrightarrow{\textit{modification-order}} c \wedge \\
&\qquad \neg(b \xrightarrow{\textit{happens-before}} c) \wedge \\
&\qquad (\forall a.\ vsse_head \xrightarrow{\textit{modification-order}} a \xrightarrow{\textit{modification-order}} c \\
&\qquad\qquad \Longrightarrow \neg(b \xrightarrow{\textit{happens-before}} a))\}
\end{aligned}$$

An atomic read



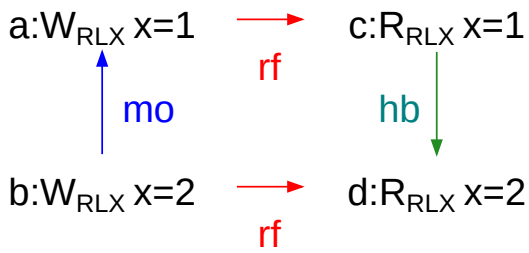
../examples/t8a.c

Consistent reads-from mapping

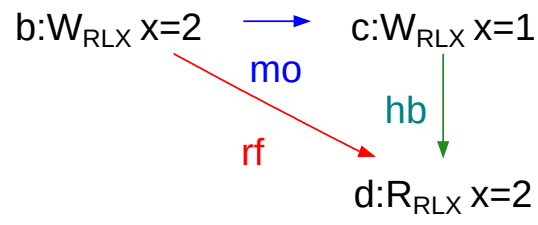
$$
\begin{aligned}
&\text{consistent\_reads\_from\_mapping} = \\
&\quad (\forall b.\ (\text{is\_read}\ b \wedge \text{is\_at\_non\_atomic\_location}\ b) \implies \\
&\qquad (\textbf{if}\ (\exists a_{vse}.\ a_{vse} \xrightarrow{\textit{visible-side-effect}} b) \\
&\qquad\ \textbf{then}\ (\exists a_{vse}.\ a_{vse} \xrightarrow{\textit{visible-side-effect}} b \wedge a_{vse} \xrightarrow{rf} b) \\
&\qquad\ \textbf{else}\ \neg(\exists a.\ a \xrightarrow{rf} b))) \wedge \\
&\quad (\forall b.\ (\text{is\_read}\ b \wedge \text{is\_at\_atomic\_location}\ b) \implies \\
&\qquad (\textbf{if}\ (\exists (b', vsse) \in \textit{visible-sequences-of-side-effects}.\ (b' = b)) \\
&\qquad\ \textbf{then}\ (\exists (b', vsse) \in \textit{visible-sequences-of-side-effects}. \\
&\qquad\qquad (b' = b) \wedge (\exists c\ \in\ vsse.\ c \xrightarrow{rf} b)) \\
&\qquad\ \textbf{else}\ \neg(\exists a.\ a \xrightarrow{rf} b))) \wedge \\
&\quad (\forall (x, a) \in \xrightarrow{rf}. \\
&\qquad \forall (y, b) \in \xrightarrow{rf}. \\
&\qquad\quad a \xrightarrow{\textit{happens-before}} b \wedge \\
&\qquad\qquad \text{same\_location}\ a\ b \wedge \text{is\_at\_atomic\_location}\ b \\
&\qquad\qquad\quad \implies (x = y) \vee x \xrightarrow{\textit{modification-order}} y) \wedge \\
&\quad (\forall (a, b) \in \xrightarrow{rf}.\ \text{is\_atomic\_rmw}\ b \\
&\qquad \implies a \xrightarrow{\textit{modification-order}} b) \wedge \\
&\quad (\forall (a, b) \in \xrightarrow{rf}.\ \text{is\_seq\_cst}\ b \\
&\qquad \implies \neg\,\text{is\_seq\_cst}\ a \vee \\
&\qquad\quad a \xrightarrow{sc}_{\lambda c.\ \text{is\_write}\ c \wedge \text{same\_location}\ b\ c} b) \wedge \\
&\quad [\ldots]
\end{aligned}
$$

Coherence

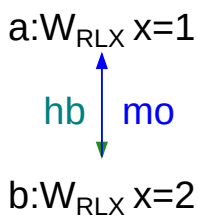
Coherence is defined an absence of four execution fragments:



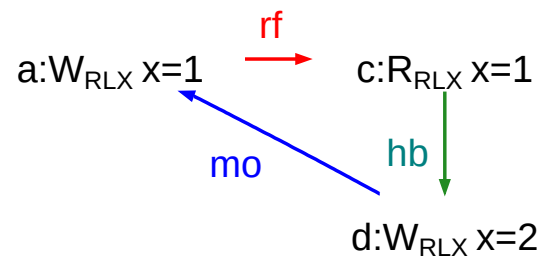
../examples/coherence-axiom-1.exc



../examples/coherence-axiom-2.exc



../examples/coherence-axiom-4.exc



../examples/coherence-axiom-3.exc

Concurrency examples that can be observed

The model allows the following non-SC behaviour:

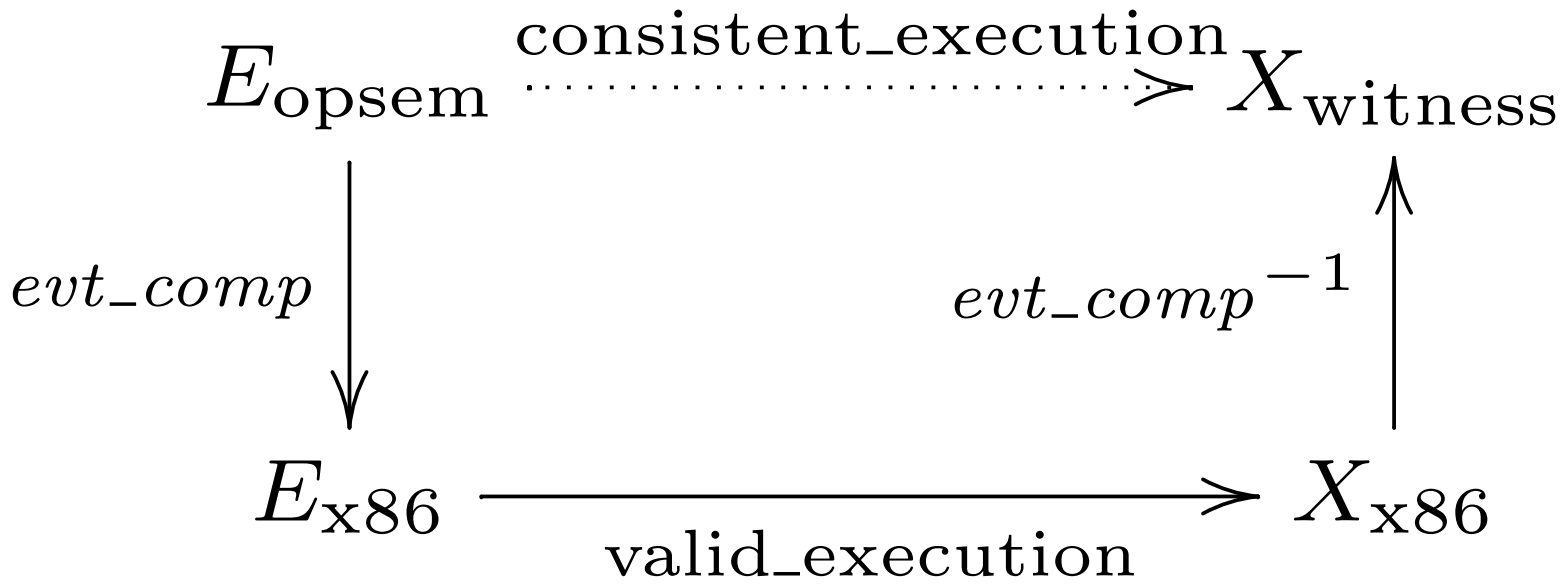
- message passing (RLX, REL-CON)
- store buffering (REL-ACQ, RLX, REL-CON)
- load buffering (RLX, CON)
- write-to-read causality (RLX, CON)
- IRIW (REL-ACQ, RLX, REL-CON)

...but DRF programs that use only the `memory_order_seq_cst` atomics should be sequentially consistent

An execution compiler

Operation	x86 Implementation	
Load non-SC	mov	
Load Seq_cst	lock xadd(0)	OR: mfence, mov
Store non-SC	mov	
Store Seq_cst	lock xchg	OR: mov , mfence
Fence non-SC	no-op	
Fence Seq_cst	mfence	

Theorem


E_{opsem}
consistent_execution
X_{witness}
evt_comp
evt_comp^{-1}
E_{x86}
X_{x86}
valid_execution

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