The C1x and C++11 concurrency model

Mark Batty

University of Cambridge

January 16, 2013
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?

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

\[
x.\text{store}(1, \text{mo_seq_cst}); \quad \text{y.\text{store}}(1, \text{mo_seq_cst}); \\
\text{r1 = y.\text{load}}(\text{mo_seq_cst}); \quad \text{r2 = x.\text{load}}(\text{mo_seq_cst});
\]

The program above cannot end with \( r1 = r2 = 0 \).
The compiler must ensure that \texttt{mo_seq_cst} atomics have SC semantics.

\begin{verbatim}
x.store(1, mo_seq_cst);  \quad y.store(1, mo_seq_cst);
r1 = y.load(mo_seq_cst);  \quad r2 = x.load(mo_seq_cst);
\end{verbatim}

The program above cannot end with \( r1 = r2 = 0 \).

...so, MP is forbidden over \texttt{mo_seq_cst}. So are all other relaxed behaviours.
Supports fast implementation of the message passing idiom.

\[
x = 1; \\
y.\text{store}(1, \text{mo\_release}); \\
r1 = y.\text{load}(\text{mo\_acquire}); \\
r2 = x;
\]

The program above cannot end with \( r1 = 1 \) and \( r2 = 0 \).
**mo_release / mo_acquire**

Supports fast implementation of the message passing idiom.

```plaintext
x = 1;
y.store(1, mo_release);
```

```plaintext
r1 = y.load(mo_acquire);
r2 = x;
```

The program above cannot end with \( r1 = 1 \) and \( r2 = 0 \).

...so, MP is forbidden using \texttt{mo\_release} and \texttt{mo\_acquire}. SB and IRIW are allowed though.
Supports faster implementation of the message passing idiom on Power.

```
x = 1;
y.store(&x, mo_release);
r1 = y.load(mo_consume);
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 $r2 = 1$. 
Very fast access, but also lots of strange behaviour.

\[
\begin{align*}
  r1 &= x\.load(mo\_relaxed); & r2 &= y\.load(mo\_relaxed); \\
  y\.store(1, mo\_relaxed); & x\.store(1, mo\_relaxed);
\end{align*}
\]

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

A single threaded program
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

```c
int y, x = 2;
x = 3;   \[ y = (x==3); \]
```
A data race

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

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

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

```
b:W_{na} x=3
```

```
c:R_{na} x=2
```

```
\text{asw, rf}
```

\text{asw, rf, dr}

\text{sb} 

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);
```
Simple concurrency: Decker’s example and SC

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

x.store(1, seq_cst);  // x = 1
y.store(1, seq_cst);  // y = 1
y.load(seq_cst);      // y = 0
x.load(seq_cst);      // x = 0
```

```
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

```c
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
x.load(seq_cst);        // e: W Sc x = 1
y.store(1, seq_cst);    // f: R Sc y = 1
```
Read the last write in SC order.

Using only seq_cst reads and writes gives SC.

(Initialization is not seq_cst though...)

SC atomics
Expert concurrency: The release-acquire idiom

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

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

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

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

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

\[a:W_{na} \quad x=1\]

\[sb\]

\[b:W_{rel} \quad y=1\]

\[sw\]

\[c:R_{acq} \quad y=1\]

\[sb\]

\[d:R_{na} \quad x=1\]
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;

\[
\begin{align*}
\text{simple-happens-before} & \quad = \\
(\text{sequenced-before} \cup \text{synchronizes-with})^+ 
\end{align*}
\]
Locks and unlocks

Unlocks and locks synchronise too:

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

![Diagram of mutex synchronization](diagram.png)
Locks and unlocks

Unlocks and locks synchronise too:

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

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

![Diagram of mutex synchronization]
Unlocks and locks synchronise too:

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

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

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

![Diagram showing the synchronization between unlocks and locks]
Locks and unlocks

Unlocks and locks synchronise too:

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

m.lock();
x = ...  // Access x
m.unlock();

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

Diagram:

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

h:L mutex
   sb
  i:R_{na} x=1
    rf
    hb```

Legend:
- c: L - Critical section
- d: W_{na} - Write on access
- f: U - Exclusive access
- h: L - Hold
- i: R_{na} - Read on access
- sb - Shared barrier
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

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

The diagram illustrates the data race with the following operations:

- `a: W_{na} x=2`
- `b: W_{na} x=3`
- `c: R_{na} x=2`
- `d: W_{na} y=0`

The operations `asw, rf` and `dr` are indicated, showing the potential for race conditions.
let data_races actions hb =
    { (a, b) | ∀ a∈actions b∈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) ∈ hb ∨ (b, a) ∈ hb) }

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

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

c:Rrlx x=1  e:Rrlx y=1
sb               sb
rfrf

d:Wrlx 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);
y.load(relaxed);
x.load(relaxed);
c:Wrlx x=1
d:Wrlx y=1
e:Rrlx x=1
g:Rrlx y=1
rf
rf
sb
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;

c:W_{na} x=1
  ↓ sb
  ↓ rf
  ↓ sb
d:W_{rel} y=1

e:R_{rlx} y=1
  ↓ sb
  ↓ rf
  ↓ sb
f:F_{acq}
  ↓ sb
g:R_{na} x=1
// sender
x = ...
y.store(1, release);

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

c:W_{na} x=1
\downarrow sb

d:W_{rel} y=1
\downarrow rf
\downarrow sw

e:R_{rlx} y=1
\downarrow sb

f:F_{acq}
\downarrow 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;

c: W$_{na}$ x=1
  ↓ sb
  ↓ rf

d: W$_{rel}$ y=1
  ↓ hb
  ↓ sb
  ↓ sw

e: R$_{rlx}$ y=1
  ↓ sb
  ↓ rf
  ↓ sw
  ↓ g: R$_{na}$ x=1

f: F$_{acq}$
  ↓ sb

a: W$_{na}$ x=1
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);
```
**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);
```

```
<table>
<thead>
<tr>
<th>Location</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>W_{rlx}, x=1</td>
<td></td>
</tr>
<tr>
<td>sb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>W_{rlx}, x=2</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>R_{rlx}, x=1</td>
<td></td>
</tr>
<tr>
<td>sb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>R_{rlx}, x=2</td>
<td></td>
</tr>
</tbody>
</table>
```
**Modification order** is a per-location total order over atomic writes of any memory order.

\[
x.\text{store}(1, \text{relaxed}); \quad \left| \quad x.\text{load}(\text{relaxed});
\right.
\]

\[
x.\text{store}(2, \text{relaxed}); \quad \left| \quad x.\text{load}(\text{relaxed});
\right.
\]
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:

```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:

```
x.store(1, 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:

\[
\begin{align*}
x.\text{store}(1, \text{relaxed}); & \quad \text{compare}\_\text{exchange}(&x, 2, 3, \text{relaxed}, \text{relaxed}); \\
x.\text{store}(2, \text{relaxed}); & \quad \text{compare}\_\text{exchange}(&x, 2, 3, \text{relaxed}, \text{relaxed}); \\
x.\text{store}(4, \text{relaxed}); & \quad \text{compare}\_\text{exchange}(&x, 2, 3, \text{relaxed}, \text{relaxed});
\end{align*}
\]
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. $P \mapsto E_1, \ldots, E_n$
How may a program execute in the model?

1. \( P \mapsto E_1, \ldots, E_n \)
   — find memory accesses with thread local semantics
How may a program execute in the model?

1. \( P \mapsto E_1, \ldots, E_n \)
   — find memory accesses with thread local semantics

2. \( E_i \mapsto X_{i1}, \ldots, X_{im} \)
How may a program execute in the model?

1. $P \mapsto E_1, \ldots, E_n$
   — find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, \ldots, X_{im}$
   — calculate happens before, check the rules
How may a program execute in the model?

1. $P \mapsto E_1, \ldots, E_n$
   — find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, \ldots, X_{im}$
   — calculate happens before, check the rules

3. is there an $X_{ij}$ with a race?
How may a program execute in the model?

1. $P \mapsto E_1, \ldots, E_n$
   — find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, \ldots, X_{im}$
   — 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
Code in, all executions out

How may a program execute in \texttt{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?
C1x and C++11 support many modes of programming:
- sequential
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
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
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
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>mov</td>
</tr>
<tr>
<td>store(non-seq_cst)</td>
<td>mov</td>
</tr>
<tr>
<td>store(seq_cst)</td>
<td>mov; mfence</td>
</tr>
<tr>
<td>fence(non-seq_cst)</td>
<td>no-op</td>
</tr>
<tr>
<td>fence(seq_cst)</td>
<td>mfence</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 $E_{\text{thread}}$
Recall the C/C++ semantics for program \( P \):

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

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

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

1. $P \mapsto E_1, \ldots, E_n$, each an $E_{\text{thread}}$
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, $E_{x86}$ are analogous to $E_{\text{thread}}$.
Execution witnesses, $X_{x86}$ are analogous to $X_{\text{witness}}$. 
Recall the C/C++ semantics for program $P$:

1. $P \mapsto E_1, \ldots, E_n$, each an $E_{\text{thread}}$
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 $E_{\text{thread}}$.
Execution witnesses, $X_{\text{x86}}$ are analogous to $X_{\text{witness}}$.
There is not a DRF semantics.
Theorem

\[
\begin{align*}
E_{\text{thread}} \xrightarrow{\text{consistent_execution}} X_{\text{witness}} \\
E_{\text{x86}} \xrightarrow{\text{valid_execution}} X_{\text{x86}}
\end{align*}
\]
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.
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.

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.

\[
\begin{align*}
x &= 1; \\
y &= 1, \text{ mo_release}) \\
r1 &= y.\text{load(mo_acquire)}; \\
r2 &= x;
\end{align*}
\]

The program above cannot end with \( r1 = 1 \) and \( r2 = 0 \).

Accesses to the data could be reordered/optimised with \text{mo\_relaxed}.

**Sample compilation on x86:**

store: mov
load: mov

**Sample compilation on Power:**

store: lwsync; st
load: ld; cmp; bc; isync
mo_release / mo_consume

Supports faster implementation of the message passing idiom on Power.

\[
\begin{align*}
x &= 1; \\
y &.store(&x, mo\_release); \\
\end{align*}
\]

\[
\begin{align*}
r1 &= y.load(mo\_consume); \\
r2 &= *r1;
\end{align*}
\]

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).

```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...
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

\[ r1 = x.\text{load}(\text{mo\_relaxed}); \]
\[ \text{if} \ (r1 == 42) \]
\[ \text{y.\text{store}}(r1, \text{mo\_relaxed}); \]
\[ r2 = y.\text{load}(\text{mo\_relaxed}); \]
\[ \text{if} \ (r2 == 42) \]
\[ \text{x.\text{store}}(42, \text{mo\_relaxed}); \]
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);

"However, implementations **should** not allow such behavior."
Self-satisfying conditionals

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

```plaintext
r2 = y.load(mo_relaxed);
if (r2 == 42)
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!