Concurrent Library Correctness for Relaxed C/C++

Mark Batty, Mike Dodds, Alexey Gotsman
Talk structure:

1. Relaxed memory overview

2. C11 / C++11 relaxed constructs

3. Abstraction for C11 library code
Part 1

Relaxed memory
Relaxed memory

\[ x = y = 0; \]

\[ x = 1; \quad y = 1; \]

\[ a = y; \quad b = x; \]
Relaxed memory

\[ x = y = 0; \]

\[ x = 1; \quad \mid \quad y = 1; \]

\[ a = y; \quad \mid \quad b = x; \]

\{ a = b = 0 \}
Relaxed memory

\[ x = y = 0; \]

\[ x = 1; \quad y = 1; \]

\[ a = y; \quad b = x; \]

\[ \{ a = b = 0 \} \]
Relaxed memory

\[ x = y = 0; \]

\[ x = 1; \quad y = 1; \]

\[ a = y; \quad b = x; \]

\{ a = b = 0 \}
Relaxed memory

\[
x = y = 0;
\]

\[
x = 1; \quad a = y; \quad y = 1; \quad b = x;
\]

\[
\{ a = b = 0 \}
\]
Relaxed memory

\[ x = y = 0; \]

\[ x = 1; \quad y = 1; \]

\[ a = y; \quad b = x; \]

\[ \{ a = b = 0 \} \]

Writes can be delayed
Sequentially consistent memory

f() {
    *x = a;
    *y = b;
}

RAM
Sequentially consistent memory

```c
f() {
    *x = a;
    *y = b;
}
```

RAM

x : a
Sequentially consistent memory

```c
f() {
    *x = a;
    *y = b;
}
```
Relaxed memory: x86

f() {
  *x = a;
  *y = b;
}

RAM
Relaxed memory: x86

\[
f() \{
    *x = a;
    *y = b;
\}
\]
Relaxed memory: x86

```c
f() {
    *x = a;
    *y = b;
}
```

Write buffer

RAM

Tuesday, 13 November 12
Relaxed memory: x86

Write buffer

\[
\begin{align*}
  y &: b \\
  x &: a \\
\end{align*}
\]

\[
f() \{ \\
  *x = a; \\
  *y = b; \\
\}
\]
Relaxed memory: x86

Write buffer

\[
\text{RAM} \quad x : a \\
\text{y} : b
\]

\[
f() \{
    \*x = a; \\
    \*y = b;
\}
\]

Tuesday, 13 November 12
Relaxed memory: x86

f() {
    *x = a;
    *y = b;
}

RAM

Write buffer

x : a    y : b
Restoring sanity

```plaintext
x = y = 0;

x = 1;  y = 1;

a = y;  b = x;

{ a = b = 0 }
```
Restoring sanity

\[ x = y = 0; \]

\[ x = 1; \]
\[ \text{fence;} \]
\[ a = y; \]

\[ y = 1; \]
\[ \text{fence;} \]
\[ b = x; \]

\[ \{ a = b = 0 \} \] ✗
Restoring sanity

Fences are platform-specific

\[
\begin{align*}
x &= y = 0; \\
x &= 1; & y &= 1; \\
fence; & & fence; \\
a &= y; & b &= x; \\
\{a = b = 0\} & \text{✘}
\end{align*}
\]
Restoring sanity

```
x = y = 0;

x = 1;
fence;
a = y;

y = 1;
fence;
b = x;

{ a = b = 0 } ✗
```
IBM Power and ARM

data = ready = 0;

data = 1; while (!ready); 

ready = 1; a = data;
IBM Power and ARM

data = ready = 0;
data = 1; while (!ready) ;
ready = 1; a = data;
{ a = 0 }
IBM Power and ARM

data = ready = 0;

data = 1;  
ready = 1;  
while (!ready) ;  
a = data;

{ a = 0 }

Message passing
IBM Power and ARM

data = ready = 0;

while (!ready) ;

a = data;

data = ready = 0;

ready = 1;       while (!ready) ;

{ a = 0 }

Message passing

Tuesday, 13 November 12
IBM Power and ARM

data = ready = 0;

data = 1; while (!ready) ;
ready = 1; a = data;

{ a = 0 }

Message passing
IBM Power and ARM

data = ready = 0;

data = 1; while (!ready) ;

ready = 1; a = data;

{ a = 0 }
IBM Power and ARM

```c
data = ready = 0;

data = 1;               while (!ready) ;

ready = 1;             a = data;

{ a = 0 }
```

Writes can be reordered
data = ready = 0;

data = 1;        while (!ready); 

ready = 1;       a = data;

{ a = 0 }

Writes can be reordered
IBM Power and ARM

data = ready = 0;

data = 1;
fence;
ready = 1;
while (!ready);
fence;
a = data;

{ a = 0 } ✗

Writes can be reordered

Message passing
Compiler optimisations

All vars = 0;

data = 1;
requestReady = 1;
...
while (!responseReady) ;
a = data;

while (!requestReady) ;
data = 2;
responseReady = 1;
Compiler optimisations

All vars = 0;

data = 1;
requestReady = 1;
...
while (!responseReady) ;
a = data;

while (!requestReady) ;
data = 2;
responseReady = 1;
Compiler optimisations

All vars = 0;

data = 1;
requestReady = 1;
...
while (!responseReady) ;
data = 2;
while (!requestReady) ;
responseReady = 1;
a = data;
Compiler optimisations

All vars = 0;

data = 1;
requestReady = 1; while (!requestReady) ;
...
while (!responseReady) ; responseReady = 1;
a = data;

{ a = 1 }
Compiler optimisations

Constant propagation

All vars = 0;

data = 1;
requestReady = 1;
...
while (!responseReady) ;
a = data;

while (!requestReady) ;
data = 2;
responseReady = 1;

{ a = 1 }
Compiler optimisations

Constant propagation

data = 1;
requestReady = 1;
...
while (!responseReady) ;
a = data;

while (!requestReady) ;
data = 2;
responseReady = 1;

{ a = 1 }

All vars = 0;

But can also strengthen the model by inserting fences automatically
High-level language with relaxed primitives

compiler

ARM / x86 / Power assembly + fences
Java memory model

• First memory model for a mainstream language [Manson+ 2005]

• Allow useful compiler optimisation, yet stay programmable

• Sequential consistency for data-race free programs
Data race

A pair of concurrent accesses, at least one of which is a write

\[
\begin{align*}
\text{lock}(m); & \quad \text{lock}(m); \\
x &= x + 1; & \quad x &= x + 1; \\
\text{unlock}(m); & \quad \text{unlock}(m); \\
\ldots & \quad \ldots \\
x &= x + 1; & \quad x &= x + 1;
\end{align*}
\]
Java memory model

- First memory model for a mainstream language [Manson+ 2005]
- Allow useful compiler optimisation, yet stay programmable
- Sequential consistency for data-race free programs
- Fiasco: the intended compiler optimisations are unsound and no-one knows how to fix this
Part 2

The C11 memory model
C11 / C++11

• A recent effort [Boehm\(^+\) 2008, Batty\(^+\) 2011]
• Part of the C & C++ 2011 standards
• Racy programs have no semantics
• Java couldn’t cheat because of security considerations
• Ways of relaxing sequential consistency to gain performance.
• portable across x86, Power, ARM, Itanium.
Concurrency for normal people

- By default, memory accesses have to be data-race free:

  ```
  lock(m);
  x = x+1;
  unlock(m);
  
  lock(m);
  x = x+1;
  unlock(m);
  ...
  
  x = x+1;  \textcolor{red}{\times}
  ```

- Sequential consistency guaranteed
Concurrency for heroes

- Sometimes we do want programs to be 
racy: non-blocking algorithms, 
synchronisation primitives

- Potentially racy accesses declared as such: 
  atomic operations

- Programmer specifies the desired 
  consistency
Types of atomics

SC atomics

release/acquire atomics

relaxed atomics

performance

...and some other things
Types of atomics

- strong atomics
- weak atomics

...and some other things

SC atomics
release/acquire atomics
relaxed atomics

performance

consistency
SC atomics

Compiler will insert enough fences to ensure sequential consistency:

\[ x = y = 0; \]

\[ x_{sc} = 1; \quad y_{sc} = 1; \]

\[ a = y_{sc}; \quad b = x_{sc}; \]

\[ \{ a = b = 0 \} \]
Relaxed atomics: write delay

Have a very relaxed semantics, but do not require fences in compilation:

\[ x = y = 0; \]

\[ x_{\text{RLX}} = 1; \quad y_{\text{RLX}} = 1; \]

\[ a = y_{\text{RLX}}; \quad b = x_{\text{RLX}}; \]

\{ a = b = 0 \}
Using relaxed atomics

```
inc(x) {
  do {
    a = x_{RLX};
    b = a+1;
  } while CAS_{SC}(x, a, b);
}
```

The value of $x$ is double-checked with an SC atomic
Using relaxed atomics

```c
inc(x) {
    do {
        a = x_{RLX};
        b = a + 1;
    } while CAS_{SC}(x, a, b);
}
```

The value of $x$ is double-checked with an SC atomic

---

No fence

Fence

Tuesday, 13 November 12
Relaxed atomics: write reordering

\[
\text{data} = \text{ready} = 0; \\
\text{data}_{\text{RLX}} = 1; \quad \text{while} \ (\neg \text{ready}_{\text{RLX}}); \\
\text{ready}_{\text{RLX}} = 1; \quad a = \text{data}_{\text{RLX}}; \\
\{ a = 0 \}
\]
Relaxed atomics: write reordering

data = ready = 0;

data_{sc} = 1; while (!\text{ready}_{sc});

ready_{sc} = 1; a = data_{sc};

\{ a = 0 \} ❌
Relaxed atomics: write reordering

\[
data = \text{ready} = 0;
\]

\[
data_{sc} = 1; \quad \text{while} \ (\neg \text{ready}_{sc}) \ ;
\]

\[
\text{ready}_{sc} = 1; \quad a = \text{data}_{sc};
\]

\[
\{ a = 0 \} \times
\]

On Power and ARM will be compiled to fences that are too heavyweight

Message passing
Release/acquire atomics

\[
\text{data} = \text{ready} = 0;
\]

\[
\text{data}_{RLX} = 1; \quad \text{while} \ (\neg \text{ready}_{ACQ});
\]

\[
\text{ready}_{REL} = 1; \quad a = \text{data}_{RLX};
\]

\[
\{
\text{a} = 0
\}
\]
Release/acquire atomics

\[
\text{data} = \text{ready} = 0;
\]

\[
data_{\text{RLX}} = 1; \quad \text{while} \ (\neg \text{ready}_{\text{ACQ}}) ;
\]

\[
\text{ready}_{\text{REL}} = 1; \quad a = \text{data}_{\text{RLX}};
\]

\[
\{ a = 0 \} \times
\]

Anything written before the release will be seen after the acquire.
Release/acquire atomics

\[ x = y = 0; \]

\[ x_{\text{REL}} = 1; \quad \text{||} \quad y_{\text{REL}} = 1; \]

\[ a = y_{\text{ACQ}}; \quad \text{||} \quad b = x_{\text{ACQ}}; \]

\[ \{ a = b = 0 \} \]

Don’t prevent writes from being delayed
Types of atomics

- SC atomics
- release/acquire atomics
- relaxed atomics

- sequential consistency
- writes delayed
- writes delayed and reordered

Tuesday, 13 November 12
Executions

- Sequential consistency: linear sequence

- C++: partial-order happens-before relation
Executions

- Sequential consistency: linear sequence

- C++: partial-order happens-before relation

- Race: two conflicting actions unordered by hb
Execution: (A, po, rf, sc, ..., hb)

\[
\begin{align*}
\text{data} &= 0; \quad \text{ready} = 0; \\
\text{data}_{\text{RLX}} &= 1; \quad \text{ready}_{\text{ACQ}} == 1; \\
\text{ready}_{\text{REL}} &= 1; \quad \text{data}_{\text{RLX}} == 1;
\end{align*}
\]
Execution: \((A, \text{po, rf, sc, ...}, \text{hb})\)

\[
\begin{align*}
\text{data} &= 0; \quad \text{ready} = 0; \\
\text{data}_\text{RLX} &= 1; \quad \text{ready}_\text{ACQ} = 1; \\
\text{ready}_\text{REL} &= 1; \quad \text{data}_\text{RLX} = 1;
\end{align*}
\]
Execution: (A, po, rf, sc, ..., hb)

\[ data_{RLX} = 0; \quad ready_{RLX} = 0; \]

\[ data_{ACQ} = 1; \quad ready_{ACQ} = 1; \]

\[ ready_{REL} = 1; \quad data_{REL} = 1; \]

Message passing

Tuesday, 13 November 12
Execution: \((A, \text{po, rf, sc, ...}, \text{hb})\)

\[
\begin{align*}
data &= 0; \quad \text{ready} = 0; \\
data_{\text{RLX}} &= 1; \quad \text{ready}_{\text{ACQ}} = 1; \\
\text{ready}_{\text{REL}} &= 1; \quad \text{data}_{\text{RLX}} = 1; \\
\end{align*}
\]
Execution: (A, po, rf, sc, ..., hb)

Program order

Message passing

\[
\begin{align*}
\text{data} &= 0; \quad \text{ready} = 0; \\
\text{data}_{RLX} &= 1; \quad \text{ready}_{ACQ} = 1; \\
\text{ready}_{REL} &= 1; \quad \text{data}_{RLX} = 1;
\end{align*}
\]
Execution: (A, po, rf, sc, ..., hb)

\[ \text{data} = 0; \quad \text{ready} = 0; \]

\[ \text{data}_{RLX} = 1; \quad \text{ready}_{ACQ} = 1; \]

\[ \text{ready}_{REL} = 1; \quad \text{data}_{RLX} = 1; \]
Execution: (A, po, rf, sc, ..., hb)

data = 0; ready = 0;

data_{RLX} = 1; ready_{ACQ} = 1;

ready_{REL} = 1; data_{RLX} = 1;

Message passing
Execution: \((A, \text{ po, rf, sc, ...}, \text{ hb})\)

\[
\begin{align*}
\text{data} &= 0; \quad \text{ready} = 0; \\
\text{data}_{\text{RLX}} &= 1; \quad \text{ready}_{\text{ACQ}} = 1; \\
\text{ready}_{\text{REL}} &= 1; \quad \text{data}_{\text{RLX}} = 1;
\end{align*}
\]
Execution: \((A, \text{po, rf, sc, ...}, \text{hb})\)

\[ \text{hb} = (\text{program order U release/acquire or SC reads-from})^+ \]

\[
\begin{align*}
\text{data} &= 0; \quad \text{ready} = 0; \\
\text{data}_{RLX} &= 1; \quad \text{ready}_{ACQ} = 1; \\
\text{ready}_{REL} &= 1; \quad \text{data}_{RLX} = 1; \\
\end{align*}
\]
Choice of the relations has to satisfy certain axioms.
Axioms

rf cannot go against hb: you can’t read from the future

\[
\begin{align*}
  x_{RLX} &= 2; \\
  x_{RLX} &= 1;
\end{align*}
\]
Axioms

rf cannot go against hb: you can’t read from the future

$\forall a, b \in \text{RLX}$

$\text{hlX} = \bot \quad \text{rf}$

$x_{\text{RLX}} \equiv \bot$

$\text{rf} \Rightarrow x_{\text{RLX}} = 2$

$\text{hlX} \Rightarrow x_{\text{RLX}} = 1$
Axioms

rf can’t skip hb: you can’t read too far in the past

\[
\begin{align*}
\mathcal{w}_1 & \xrightarrow{\text{hb}} \mathcal{w}_2 & \xrightarrow{\text{hb}} & r \\
& \xrightarrow{\text{rf}} \\
\end{align*}
\]

\[
\begin{align*}
x_{_\text{RLX}} &= 1; \\
x_{_\text{RLX}} &= 2; \\
x_{_\text{RLX}} &= 0; \\
\end{align*}
\]
Axioms

rf can’t skip hb: you can’t read too far in the past

\[ w_1 \xrightarrow{\text{hb}} w_2 \xrightarrow{\text{hb}} r \]

\[ x_{\text{RLX}} = 1; \]

\[ x_{\text{RLX}} = 2; \]

\[ x_{\text{RLX}} == ?; \]
Axioms

rf can’t skip hb: you can’t read too far in the past

data = 0;  ready = 0;

data_{RLX} = 1;  ready_{ACQ} == 1;

ready_{REL} = 1;  data_{RLX} == 0;
The zoo of ‘negative’ axioms

\[ a \xleftarrow{\text{sc}} b \]
\[ a \xrightarrow{\text{hb}} b \]
\[ a \xleftarrow{\text{mo}} b \]
\[ a \xrightarrow{\text{sc}} b \]

\[ a \xrightarrow{\text{rf}} d \]
\[ b \rightarrow c \]
\[ a \xrightarrow{\text{rf}} b \]
\[ a \xrightarrow{\text{mo}} b \]

Tuesday, 13 November 12
Mathematizing C++ concurrency. 

- Full formalisation of the model.
- Properties proved in HOL.
- Fixed bugs in the standard.
Part 3

Library abstraction in the C++ memory model
Our goal now: abstraction

• Is abstraction on relaxed memory possible?
• Don’t want to consider the internals of L while verifying C:

• Need a notion of library specification
Our goal now: abstraction

• Is abstraction on relaxed memory possible?
• Don’t want to consider the internals of L while verifying C:

![Diagram]

• Need a notion of library specification
Challenge

Write buffer

\[ \begin{align*}
  y &: b \\
  x &: a
\end{align*} \]

\[
f() \{ \\
  *x = a; \\
  *y = b; \\
\}
\]

- Writes can be flushed after \( f \) returns
- Library behavior depends on the time of the flush
- Just parameters and return values not enough
Write buffer

\[
\begin{align*}
  x : a \\
  y : b
\end{align*}
\]

\[
f() \{ \\
  *x = a; \\
  *y = b;
}\]

- Writes can be flushed after \( f \) returns
- Library behavior depends on the time of the flush
- Just parameters and return values not enough

Not only for verification: testing, documentation, communication...
push(v) {
    ...
}

push(v) {
    ...
}

pop() {
    ...
    return v;
}
\begin{align*}
\text{push}(v) \{ & \quad \text{pop}() \{ \\
& \quad \quad \ldots & \quad \quad \ldots \\
& \quad \} & \quad \} \\
& \text{return } v; \\
& \}
\end{align*}
data_{RLX} = 0;

\[
\begin{align*}
data_{RLX} &= 1; \\
call push; &
\end{align*}
\]

... 

ready_{REL} = 1; 

... 

return push;

\[
\begin{align*}
call pop; &
\end{align*}
\]

... 

ready_{ACQ} == 1; 

... 

return pop;

val = data_{RLX};

{ val = 0 }
data_{RLX} = 1;
call push;
... 
ready_{REL} = 1;
...
return push;

\{ \text{val} = 0 \} \text{?}

val = data_{RLX};
call pop;
... 
ready_{ACQ} == 1;
...
return pop;

data_{RLX} = 0;

\text{push};
... 
\text{ready}_{REL} = 1;
...
\text{return push};

\text{pop};
... 
\text{ready}_{ACQ} == 1;
...
\text{return pop};

\text{pop};
... 
\text{ready}_{ACQ} == 1;
...
\text{return pop};
\begin{align*}
data_{RLX} &= 1; \\
call \text{push}; \\
&\quad \ldots \\
\text{ready}_{REL} &= 1; \\
&\quad \ldots \\
\text{return push;}
\end{align*}

\begin{align*}
data_{RLX} &= 0; \\
call \text{pop}; \\
&\quad \ldots \\
\text{ready}_{ACQ} &= 1; \\
&\quad \ldots \\
\text{return pop;}
\end{align*}

\text{val} = \text{data}_{RLX};

\{ \text{val} = 0 \}
Data races between non-atomic memory accesses are considered data races. However, the SC memory order is more expensive. We have already seen the first two in the stack example above. The release-acquire memory order only synchronises between threads. Only release-acquire allows writes to be delayed, the second thread could read the read. Hence, by (RD), the read from the non-atomic write of \( x \) happens before the atomic read of \( x \). This is the client-visible behaviour \([1, 3]\). However, the SC memory order is more expensive. The release-acquire memory order allows more relaxed behaviour \([1, 3]\).
data_{RLX} = 0;

data_{RLX} = 1;

call push;
    ...
    ready_{RLX} = 1;
    ...
    return push;

call pop;
    ...
    ready_{RLX} == 1;
    ...
    return pop;

val = data_{RLX};

{ val = 0 }
data_{RLX} = 1;
... ready_{RLX} = 1;
... return push;

call push;
... 

data_{RLX} = 0;

call pop;
... ready_{RLX} == 1;
... return pop;

val = data_{RLX};

{ val = 0 }
Abstraction Theorem

$L'$ specifies $L$:

$L \subseteq L'$

\[ C(L') \models P \Rightarrow C(L) \models P \]
Abstraction Theorem

L' specifies L: \[ L \subseteq L' \]

\[ C(L') \models P \Rightarrow C(L) \models P \]
Non-blocking stack

```c
struct Node {
    Node *next; int val;
} *Top;

void push(int v) {
    Node *t, *x;
    x = new Node;
    x->val = v;
    do {
        t = Top;
        x->next = t;
    } while(!CAS(&Top,t,x));
}
```

Atomic stack ADT

```c
Sequence S;

void push(int v) {
    atomic { S = v · S; }
}
```
Comparing libraries

- Take the **most general client**:

  ```
  \[
  \begin{aligned}
  n & \parallel k=1 \\
  \text{while (true) } & \\
  & \text{if (nondet()) } m_1(\text{nondet()}); \\
  & \text{else if (nondet()) } m_2(\text{nondet()}); \\
  & \text{...} \\
  & \text{else } m_l(\text{nondet()});
  \end{aligned}
  \]
  ```

- Get all possible library **histories** \([L]\): describe library behaviour relevant to the client

- \(L \subseteq L' \iff \forall H \in [L]. \exists H' \in [L']. H \subseteq H'\)
Comparing libraries

- Take the **most general client**: 

  \[
  \begin{align*}
  \text{while (true) } \{ \\
  & \quad \text{if (nondet()) } m_1(\text{nondet()}) ; \\
  & \quad \text{else if (nondet()) } m_2(\text{nondet()}) ; \\
  & \quad \ldots \\
  & \quad \text{else } m_l(\text{nondet()}) ; \\
  \}
  \end{align*}
  \]

- Get all possible library **histories** \([L]\): describe library behaviour relevant to the client

- \( L \subseteq L' \iff \forall H \in [L]. \exists H' \in [L']. H \subseteq H' \)
Comparing libraries

• Take the most general client:

\[
\begin{align*}
\text{while (true) } & \{ \\
\quad & \text{if (nondet()) } m_1(\text{nondet()}); \\
\quad & \text{else if (nondet()) } m_2(\text{nondet()}); \\
\quad & \ldots \\
\quad & \text{else } m_l(\text{nondet()}); \\
\} \\
\end{align*}
\]

• Get all possible library histories \([L]\): describe library behaviour relevant to the client

• \(L \sqsubseteq L' \iff \forall H \in [L]. \exists H' \in [L']. H \sqsubseteq H'\)

Any methods, in any order, with any parameters

Any number of threads

Tuesday, 13 November 12
Comparing libraries

• Take the most general client:

\[
\begin{align*}
\text{while (true) } \{ \\
\quad \text{if (nondet()) } & \quad m_1(\text{nondet()}); \\
\quad \text{else if (nondet()) } & \quad m_2(\text{nondet()}); \\
\quad \text{... } & \quad m_l(\text{nondet()}); \\
\quad \text{else } & \\
\} \\
\end{align*}
\]

Any number of threads

• Get all possible library histories \([L]\): describe library behaviour relevant to the client

• \(L \subseteq L' \iff \forall H \in [L]. \exists H' \in [L']. H \subseteq H'\)

Tuesday, 13 November 12
Subgraph replacement
Subgraph replacement
Subgraph replacement
Guarantee relations

Histories include the projection of happens-before to calls and returns:

```
call push;
  ...
  ready_{\text{REL}} = 1;
  ...
  return push;

call pop;
  ...
  ready_{\text{ACQ}} == 1;
  ...
  return pop;
```
Guarantee relations

Histories include the projection of happens-before to calls and returns:

call push;
...
ready_{REL} = 1;
...
return push;

call pop;
...
ready_{ACQ} == 1;
...
return pop;
Guarantee relations

Histories include the projection of happens-before to calls and returns:

\[
\begin{align*}
\text{call push;} \\
\ldots & \downarrow_{\text{po}} \\
\text{ready}_{\text{REL}} = 1; \\
\ldots & \quad \rightarrow_{\text{rf}} \\
\text{return push;} \\
\end{align*}
\]

\[
\begin{align*}
\text{call pop;} \\
\ldots & \downarrow_{\text{po}} \\
\text{ready}_{\text{ACQ}} == 1; \\
\ldots & \quad \rightarrow_{\text{rf}} \\
\text{return pop;} \\
\end{align*}
\]
Guarantee relations

Histories include the projection of happens-before to calls and returns:

prepare data

↓ po

call push;
... ↓ po

readyREL = 1; ↓ hb
... readyACQ == 1;
... rf

return push; return pop;
... po

access data
Require that hb cannot contain certain edges:
Deny relations

\[ a \xrightarrow{\text{hb}} b \]

\[ \xleftarrow{\text{SC}} \]

call lib;

\[ \text{val}_{\text{SC}} = 1; \]

\[ \xrightarrow{\text{po}} \]

\[ \text{...} \]

\[ \xrightarrow{\text{sc}} \]

\[ \text{val}_{\text{SC}} = 2; \]

\[ \xrightarrow{\text{po}} \]

\[ \text{...} \]

\[ \xrightarrow{\text{po}} \]

\[ \text{return lib}; \]
Client can invalidate the axiom by establishing an \textcolor{red}{hb} edge from return to call.
Deny relations

\[\begin{align*}
\text{\textit{a} \xrightarrow{\text{hb}} \text{\textit{b}}} \\
\text{\textsc{sc} \xleftarrow{\text{\textit{a}}} \text{\textit{b}}} \\
\text{v} &= \text{data}_{\text{ACQ}} \\
\text{call lib;} \\
\text{val}_{\text{SC}} &= 1; \\
\text{return lib;} \\
\text{data}_{\text{REL}} &= 0;
\end{align*}\]
Deny relations

\[ a \xrightarrow{\text{hb}} b \xleftarrow{\text{SC}} \]

call lib;
\[
\begin{align*}
\text{val}_{\text{SC}} &= 1; \\
\rightarrow^\text{po} \\
\rightarrow^\text{po} \\
\text{...}
\end{align*}
\]
\[
\begin{align*}
\text{val}_{\text{SC}} &= 2; \\
\rightarrow^\text{po} \\
\rightarrow^\text{po} \\
\text{...}
\end{align*}
\]
return lib;
Deny relations

\[ a \xrightarrow{hb} b \xleftarrow{sc} \]

```
call lib;
val_{sc} = 1;
\]
```

```
\]
return lib;
```
Abstraction

• History: (Guarantee, Deny)

• \((G, D) \subseteq (G', D') \iff G' \subseteq G \land D' \subseteq D\)

• Specification can guarantee less to the client and require more

• So client has more behaviors when using specification instead of implementation

• \(L \subseteq L' \iff \forall (G, D) \in \llbracket L \rrbracket. \exists (G', D') \in \llbracket L' \rrbracket. (G, D) \subseteq (G', D')\)
Abstraction Theorem

Assume:

- \( L \subseteq L' \)
- \( C(L'), L \) and \( L' \) are safe

Then \( \text{client}(\llbracket C(L) \rrbracket) \subseteq \text{client}(\llbracket C(L') \rrbracket) \)
Abstraction Theorem

Assume:

- \( L \subseteq L' \)
- \( C(L'), L \) and \( L' \) are safe

Then \( \text{client}([C(L)]) \subseteq \text{client}([C(L')]) \)
Abstraction Theorem

Assume:

- \( L \subseteq L' \)
- \( C(L'), L \) and \( L' \) are safe

Then \( \text{client}(\llbracket C(L) \rrbracket) \subseteq \text{client}(\llbracket C(L') \rrbracket) \)
Abstraction Theorem

Assume:

- \( L \subseteq L' \)
- \( C(L'), L \) and \( L' \) are safe

Then \( \text{client}(\llbracket C(L) \rrbracket) \subseteq \text{client}(\llbracket C(L') \rrbracket) \)

valid only without relaxed atomics
Culprit:

\[ x = y = 0; \]

\[ a = x_{RLX}; \quad b = y_{RLX}; \]

\[ y_{RLX} = a; \quad x_{RLX} = b; \]
Culprit: satisfaction cycles

\[ x = y = 0; \]

\[ a = x_{RLX}; \quad b = y_{RLX}; \]

\[ y_{RLX} = a; \quad x_{RLX} = b; \]

\{ a = b = 42 \}
Culprit: satisfaction cycles

\[ x = y = 0; \]

\[ a = x_{RLX}; \quad \quad b = y_{RLX}; \]

\[ y_{RLX} = a; \quad \quad x_{RLX} = b; \]

\{ a = b = 42 \}
Culprit: satisfaction cycles

\[ x = y = 0; \]

\[ a = x_{RLX}; \quad b = y_{RLX}; \]

\[ y_{RLX} = a; \quad x_{RLX} = b; \]

\{ a = b = 42 \}
Culprit: satisfaction cycles

\[ x = y = 0; \]

\[ a = x_{RLX}; \]
\[ b = y_{RLX}; \]

\[ y_{RLX} = a; \]
\[ x_{RLX} = b; \]

\{ \ a = b = 42 \ \}

Load buffering
Culprit: satisfaction cycles

\[ x = y = 0; \]

\[ a = x_{RLX}; \]
\[ b = y_{RLX}; \]

\[ y_{RLX} = a; \]
\[ x_{RLX} = b; \]

\{ a = b = 42 \}

Out-of-thin-air value

Load buffering
Culprit: satisfaction cycles

\[ a = x_{RLX}; \]
\[ b = y_{RLX}; \]
\[ x_{RLX} = a; \]
\[ y_{RLX} = b; \]

\{ a = b = 42 \}

Out-of-thin-air value

Load buffering

Tuesday, 13 November 12
Culprit: satisfaction cycles

\[ a = x_{RLX}; \]
\[ y_{RLX} = a; \]
\[ b = y_{RLX}; \]
\[ x_{RLX} = b; \]

\{ a = b = 42 \}

Out-of-thin-air value
Culprit: satisfaction cycles

\[ x = y = 0; \]

\[ a = x_{RLX}; \quad b = y_{RLX}; \]

\[ y_{RLX} = a; \quad x_{RLX} = b; \]

\{ a = b = 42 \}

Out-of-thin-air value
Culprit: satisfaction cycles

\[
\begin{align*}
    a &= x_{RLX} \\
    b &= y_{RLX} \\
    x &= y = 0 \\
    \{ a = b = 42 \}
\end{align*}
\]

Out-of-thin-air value
Culprit: satisfaction cycles

\[ a = x_{RLX}; \]
\[ b = y_{RLX}; \]
\[ y_{RLX} = a; \]
\[ x_{RLX} = b; \]
\[ \{ a = b = 42 \} \]

Out-of-thin-air value

Reproducible in C(L), but not in C(L')

Client screws up library

Library returns a crazy value

Client goes crazy
Fix summary

- $C(L)$ is safe, not $C(L')$

- $L \subseteq L' \iff \forall R. \forall H \in [L, R]. \exists H' \in [L', R]. H \subseteq H'$

- $H \subseteq H'$ requires equal projections of happens-before to calls and returns
Fix summary

- $C(L)$ is safe, not $C(L')$

- $L \subseteq L' \iff \forall R. \forall H \in [L, R]. \exists H' \in [L', R]. H \subseteq H'$

- $H \subseteq H'$ requires equal projections of happens-before to calls and returns
Can satisfaction cycles happen?

\[ \begin{align*}
    x &= y = 0; \\
    a &= x_{\text{RLX}}; \\
    b &= y_{\text{RLX}}; \\
    y_{\text{RLX}} &= a; \\
    x_{\text{RLX}} &= b; \\
    \{a = b = 42\}
\end{align*} \]

Processors don’t do this
Can satisfaction cycles happen?

\[ x = y = 0; \]

\[ a = x_{RLX}; \quad b = y_{RLX}; \]

\[ y_{RLX} = 1; \quad x_{RLX} = b; \]

\{ a = b = 1 \}

But Power & ARM do this due to speculation.
Can satisfaction cycles happen?

\[ x = y = 0; \]

\[ a = x_{RLX}; \]
\[ b = y_{RLX}; \]
\[ y_{RLX} = 1; \]
\[ x_{RLX} = b; \]

\{ a = b = 1 \}

But Power & ARM do this due to speculation.

(No dependency)
Can satisfaction cycles happen?

\[ x = y = 0; \]

\[ a = x_{RLX}; \quad b = y_{RLX}; \]

\[ y_{RLX} = 1; \quad x_{RLX} = b; \]

\{ a = b = 1 \}

Compilers do break dependencies.

Too expensive to preserve all dependencies.

Too expensive to preserve all dependencies.
Can satisfaction cycles happen?

\[ x = y = 0; \]

\[ a = x_{RLX}; \quad b = y_{RLX}; \]

\[ y_{RLX} = 1; \quad x_{RLX} = b; \]

\{ a = b = 1 \}

Compilers do break dependencies.

Too expensive to preserve all dependencies.

Upshot: we don’t know whether satisfaction cycles can be observed (or will be in future).
Conclusions
Conclusion

• C11 atomic constructs are very subtle!

• Modular specifications are possible on C11.

• The model is arguably broken in important ways (eg. satisfaction cycles).

• Jury out on whether we can find a better compromise between the ease of reasoning and performance.