A Promising Semantics for Relaxed-Memory Concurrency

Jeehoon Kang\textsuperscript{1}  Chung-Kil Hur\textsuperscript{1}  Ori Lahav\textsuperscript{2}  Viktor Vafeiadis\textsuperscript{2}  Derek Dreyer\textsuperscript{2}

\textsuperscript{1}Seoul National University
\textsuperscript{2}Max Planck Institute for Software Systems (MPI-SWS)

Kent Concurrency Workshop, July 2016
What is the right semantics for a concurrent programming language?

- Allow efficient implementation on modern hardware
- Validate compiler optimizations
- Support high-level reasoning principles
- Avoid “undefined behavior”

Despite many years of research, no semantics was proven to admit all of the desired properties.
In particular:

- The Java model fails to validate common compiler optimizations.

- The C11 model allows out-of-thin-air behaviors, that break fundamental reasoning principles.

- Stronger semantics for C11 (preserve load-store ordering for relaxed accesses) has some performance impact, and relies on undefined behavior for non-atomic accesses.
The *out-of-thin-air* problem in C11

- Initially, $x = y = 0$.
- All accesses are “relaxed”.

Load-buffering

```markdown
\[
\begin{align*}
a & := x; \quad \text{// 1} \\
y & := 1; \\
x & := y;
\end{align*}
\]
```

This behavior must be allowed:

*Power/ARM allow it*
The \textit{out-of-thin-air} problem in C11

- Initially, $x = y = 0$.
- All accesses are “relaxed”.

This behavior must be allowed:

- Power/ARM allow it

\[
\begin{align*}
\text{Load-buffering} \\
a &:= x; \quad \text{// 1} \\
y &:= 1; \\
\text{reads from} \\
x &:= y;
\end{align*}
\]
The *out-of-thin-air* problem in C11

**Load-buffering + data dependency**

\[
\begin{align*}
\text{a} & \assign x; \quad // 1 \\
\text{y} & \assign a; \\
\text{x} & \assign y;
\end{align*}
\]

The behavior should be forbidden:

**Values appear out-of-thin-air!**
The *out-of-thin-air* problem in C11

### Load-buffering + data dependency

\[
\begin{align*}
  a & := x; \quad \text{// 1} \\
  y & := a;
\end{align*}
\]

The behavior should be forbidden: **Values appear out-of-thin-air!**

\[
\begin{align*}
  x & := y;
\end{align*}
\]

\[
[x = y = 0]
\]

\[
\begin{align*}
  R_{rlx} x, 1 & \quad R_{rlx} y, 1 \\
  W_{rlx} y, 1 & \quad W_{rlx} x, 1
\end{align*}
\]

Same execution as before! C11 allows these behaviors
The out-of-thin-air problem in C11

Load-buffering + data dependency

\[
\begin{align*}
  a &:= x; \quad \text{\texttt{// 1}} \\
y &:= a;
\end{align*}
\]

$\Rightarrow$

\[
  x := y;
\]

The behavior should be forbidden:
Values appear out-of-thin-air!

Load-buffering + control dependencies

\[
\begin{align*}
  a &:= x; \quad \text{\texttt{// 1}} \\
  \text{if } (a = 1) &\Rightarrow \\
y &:= 1 \\
  \text{if } (y = 1) &\Rightarrow \\
x &:= 1
\end{align*}
\]

$\Rightarrow$

\[
\begin{align*}
  R_{rlx}x, 1 \\
  W_{rlx}y, 1
\end{align*}
\]

Same execution as before!
C11 allows these behaviors

$\Rightarrow$

$\begin{align*}
[x = y = 0] \\
R_{rlx}y, 1 \\
W_{rlx}x, 1
\end{align*}$
The hardware solution

Keep track of syntactic dependencies, and forbid “dependency cycles”.

Load-buffering + data dependency

\[
\begin{align*}
a &:= x; \quad // 1 \\
y &:= a; \\
x &:= y;
\end{align*}
\]

Load-buffering + fake dependency

\[
\begin{align*}
a &:= x; \\
y &:= a + 1 - a; \\
x &:= y;
\end{align*}
\]

\[\{x = y = 0\}\]

This approach is not suitable for a programming language:

Compilers do not preserve syntactic dependencies.
The hardware solution

Keep track of syntactic dependencies, and forbid “dependency cycles”.

Load-buffering + data dependency

\[
\begin{align*}
a &:= x; \quad \text{// 1} \\
y &:= a;
\end{align*}
\]

Load-buffering + fake dependency

\[
\begin{align*}
a &:= x; \quad \text{// 1} \\
y &:= a + 1 - a;
\end{align*}
\]

This approach is not suitable for a programming language: Compilers do not preserve syntactic dependencies.
A “promising” semantics for relaxed-memory concurrency

We propose a model that satisfies all these goals, and covers nearly all features of C11.

- DRF guarantees
- No “out-of-thin-air” values
- Avoid “undefined behavior”
- Efficient implementation on modern hardware
- Compiler optimizations

**Key idea:** Start with an operational semantics, and allow threads to promise to write in the future
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[ x = y = 0 \]
\[ x := 1; \quad y := 1; \]
\[ a := y \# 0 \quad b := x \# 0 \]
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[ x = y = 0 \]

\[ \begin{align*}
   & \uparrow x := 1; \\
   & a := y \parallel 0 \\
   & \uparrow y := 1; \\
   & b := x \parallel 0
\end{align*} \]

Memory

\begin{align*}
   \langle x : 0 @ 0 \rangle \\
   \langle y : 0 @ 0 \rangle
\end{align*}

\[
\begin{array}{c|c|c|c|c}
   T_1’s view & x & y \\
   \hline
   0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
   T_2’s view & x & y \\
   \hline
   0 & 0 & 0 & 0
\end{array}
\]

- Global memory is a pool of messages of the form

\[ \langle \text{location} : \text{value} @ \text{timestamp} \rangle \]

- Each thread maintains a \textit{thread-local view} recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[
\begin{align*}
  x &= y = 0 \\
  x &:= 1; \\
  \triangleright a &:= y \quad \triangleright y := 1; \\
  \triangleright b &:= x
\end{align*}
\]

Memory

\[
\begin{array}{ll}
  T_1's \text{ view} & \\
  \langle x : 0@0 \rangle & \\
  \langle y : 0@0 \rangle & \\
  \langle x : 1@1 \rangle & \\
  T_2's \text{ view} & \\
  \begin{array}{cc}
    x & y \\
    0 & 0 \\
    0 & 0
  \end{array}
\end{array}
\]

- Global memory is a pool of messages of the form

\[
\langle \text{location} : \text{value} @ \text{timestamp} \rangle
\]

- Each thread maintains a \textit{thread-local view} recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[ x = y = 0 \]
\[ x := 1; \]
\[ a := y /\ 0 \]
\[ y := 1; \]
\[ b := x /\ 0 \]

**Memory**

- **\( T_1 \)'s view**
  - \( x : 0@0 \)
  - \( y : 0@0 \)
  - \( x : 1@1 \)
  - \( y : 1@1 \)

- **\( T_2 \)'s view**
  - \( x : 0@0 \)
  - \( y : 0@0 \)
  - \( x : 1@1 \)
  - \( y : 1@1 \)

- Global memory is a pool of messages of the form
  \[ \langle \text{location} : \text{value} @ \text{timestamp} \rangle \]

- Each thread maintains a *thread-local view* recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[ x = y = 0 \]

\[ x := 1; \]
\[ a := y \] /\ 0

\[ b := x \] /\ 0

Memory

\[ \langle x : 0@0 \rangle \]
\[ \langle y : 0@0 \rangle \]
\[ \langle x : 1@1 \rangle \]
\[ \langle y : 1@1 \rangle \]

\[ T_1\text{'s view} \]
\[ x \]
\[ y \]
\[ 0 \]
\[ 0 \]
\[ T_2\text{'s view} \]
\[ x \]
\[ y \]
\[ 0 \]
\[ 1 \]

- Global memory is a pool of messages of the form

\[ \langle \text{location} : \text{value} @ \text{timestamp} \rangle \]

- Each thread maintains a thread-local view recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[ x = y = 0 \]
\[ x := 1; \]
\[ a := y \ // \ 0 \]
\[ y := 1; \]
\[ b := x \ // \ 0 \]

Memory
\[
\begin{array}{c}
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\langle x : 1@1 \rangle \\
\langle y : 1@1 \rangle \\
\end{array}
\]

\[
\begin{array}{c|c}
T_1’s \ view \\
\hline
x & y \\
\hline
0 & 0 \\
1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c}
T_2’s \ view \\
\hline
x & y \\
\hline
0 & 0 \\
1 & 1 \\
\end{array}
\]

- Global memory is a pool of messages of the form
  \[ \langle \text{location} : \text{value} \ @ \ \text{timestamp} \rangle \]

- Each thread maintains a \textit{thread-local view} recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[ x = y = 0 \]
\[ x := 1; \]
\[ a := y \parallel 0 \]
\[ y := 1; \]
\[ b := x \parallel 0 \]

Memory

\[ T_1 \text{’s view} \]
\[ x \]
\[ y \]
\[ x : 0 @ 0 \]
\[ y : 0 @ 0 \]
\[ x : 1 @ 1 \]
\[ y : 1 @ 1 \]

\[ T_2 \text{’s view} \]
\[ x \]
\[ y \]
\[ x : 0 @ 0 \]
\[ y : 0 @ 0 \]
\[ x : 1 @ 1 \]
\[ y : 1 @ 1 \]

Coherence Test

\[ x = 0 \]
\[ x := 1; \]
\[ a := x \parallel 2 \]
\[ x := 2; \]
\[ b := x \parallel 1 \]
Simple operational semantics for C11’s relaxed accesses

### Store-buffering

\[
\begin{array}{c}
x = y = 0 \\
x := 1; \\
a := y \; // \; 0 \\
y := 1; \\
b := x \; // \; 0 \\
\end{array}
\]

### Coherence Test

\[
\begin{array}{c}
x = 0 \\
x := 1; \\
a := x \; // \; 2 \\
x := 2; \\
b := x \; // \; 1 \\
\end{array}
\]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[ x = y = 0 \]
\[ x := 1; \quad y := 1; \]
\[ a := y \quad b := x \]

\[ T_1 \text{’s view} \]
\[ \langle x : 0 @ 0 \rangle \]
\[ \langle y : 0 @ 0 \rangle \]
\[ \langle x : 1 @ 1 \rangle \]
\[ \langle y : 1 @ 1 \rangle \]

\[ T_2 \text{’s view} \]
\[ \langle x : 0 @ 0 \rangle \]
\[ \langle y : 0 @ 0 \rangle \]
\[ \langle x : 1 @ 1 \rangle \]
\[ \langle y : 1 @ 1 \rangle \]

**Coherence Test**

\[ x = 0 \]
\[ x := 1; \]
\[ a := x \]
\[ x := 2; \]
\[ b := x \]

\[ T_1 \text{’s view} \]
\[ \langle x : 0 @ 0 \rangle \]
\[ \langle x : 1 @ 1 \rangle \]

\[ T_2 \text{’s view} \]
\[ \langle x : 0 @ 0 \rangle \]
\[ \langle x : 1 @ 1 \rangle \]
Store-buffering

\[ x = y = 0 \]
\[ x := 1; \quad y := 1; \]
\[ a := y \quad /\quad 0 \quad b := x \quad /\quad 0 \]

Coherence Test

\[ x = 0 \]
\[ x := 1; \quad x := 2; \]
\[ a := x \quad /\quad 2 \quad b := x \quad /\quad 1 \]
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[
\begin{align*}
x &= y = 0 \\
x &:= 1; \\
a &:= y \parr 0 \\
y &:= 1; \\
b &:= x \parr 0
\end{align*}
\]

Memory

\[
\begin{array}{c|c|c}

<table>
<thead>
<tr>
<th>T_1’s view</th>
<th>T_2’s view</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y</td>
<td>x y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
\end{array}
\]

Coherence Test

\[
\begin{align*}
x &= 0 \\
x &:= 1; \\
a &:= x \parr 2 \\
x &:= 2; \\
b &:= x \parr 1
\end{align*}
\]

Memory

\[
\begin{array}{c|c|c}

<table>
<thead>
<tr>
<th>T_1’s view</th>
<th>T_2’s view</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
</tbody>
</table>
\end{array}
\]
Simple operational semantics for C11’s relaxed accesses

### Store-buffering

\[
x = y = 0
\]
\[
x := 1;
\]
\[
a := y \quad \text{// 0}
\]
\[
y := 1;
\]
\[
b := x \quad \text{// 0}
\]

---

### Memory

\[
\langle x : 0@0 \rangle
\]
\[
\langle y : 0@0 \rangle
\]
\[
\langle x : 1@1 \rangle
\]
\[
\langle y : 1@1 \rangle
\]

\[\begin{array}{cc}
T_1’s \ \text{view} \\
x & y \\
\hline
0 & 0 \\
1 & 1
\end{array}\]

\[\begin{array}{cc}
T_2’s \ \text{view} \\
x & y \\
\hline
0 & 1 \\
1 & 0
\end{array}\]

### Coherence Test

\[
x = 0
\]
\[
x := 1;
\]
\[
a := x \quad \text{// 2}
\]
\[
x := 2;
\]
\[
b := x \quad \text{// 1}
\]

---

### Memory

\[
\langle x : 0@0 \rangle
\]
\[
\langle x : 1@1 \rangle
\]
\[
\langle x : 2@2 \rangle
\]

\[\begin{array}{cc}
T_1’s \ \text{view} \\
x \\
\hline
0 \\
2
\end{array}\]

\[\begin{array}{cc}
T_2’s \ \text{view} \\
x \\
\hline
2
\end{array}\]
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
Promises

Load-buffering

\[ x = y = 0 \]

\[ a := x; \quad // 1 \]
\[ y := 1; \]
\[ x := y; \]

Memory

\[ \langle x : 0 @ 0 \rangle \]
\[ \langle y : 0 @ 0 \rangle \]
\[ \langle y : 1 @ 1 \rangle \]

\[ T_1 \text{'s view} \]
\[ \begin{array}{cc}
    x & y \\
    \hline
    0 & 0
\end{array} \]

\[ T_2 \text{'s view} \]
\[ \begin{array}{cc}
    x & y \\
    \hline
    0 & 0
\end{array} \]

- To model load-store reordering, we allow “promises”.
- At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
Promises

Load-buffering

\[ x = y = 0 \]
\[ a := x; \quad // 1 \]
\[ y := 1; \]
\[ x := y; \]

Load-buffering + dependency

\[ a := x; \quad // 1 \]
\[ y := a; \]
\[ x := y; \]

Must not admit the same execution!

Memory

\[ \langle x : 0@0 \rangle \]
\[ \langle y : 0@0 \rangle \]
\[ \langle y : 1@1 \rangle \]
\[ \langle x : 1@1 \rangle \]

\[ T_1 \text{'s view} \]
\[ \begin{array}{c|c}
  x & y \\
  \hline
  \text{X} & \text{X} \\
\end{array} \]
\[ \begin{array}{c|c}
  1 & 1 \\
\end{array} \]

\[ T_2 \text{'s view} \]
\[ \begin{array}{c|c}
  x & y \\
  \hline
  \text{X} & \text{X} \\
\end{array} \]
\[ \begin{array}{c|c}
  1 & 1 \\
\end{array} \]
Promises

Load-buffering

\[ x = y = 0 \]

\[ a := x; \quad // 1 \]
\[ y := 1; \]
\[ x := y; \]

Load-buffering + dependency

\[ a := x; \quad // 1 \]
\[ y := a; \]
\[ x := y; \]

Key Idea

A thread can only promise if it can perform the write anyway (even without having made the promise)
Certified promises

Thread-local certification

A thread can promise to write a message, if it can \textit{thread-locally certify} that its promise will be fulfilled.
Certified promises

Thread-local certification
A thread can promise to write a message, if it can *thread-locally certify* that its promise will be fulfilled.

**Load-buffering**

\[
\begin{align*}
a & := x; \quad // 1 \\
y & := 1;
\end{align*}
\]
\[
\begin{align*}
x & := y;
\end{align*}
\]

\[T_1 \text{ may promise } y = 1, \text{ since it is able to write } y = 1 \text{ by itself.}\]

**Load buffering + fake dependency**

\[
\begin{align*}
a & := x; \quad // 1 \\
y & := a + 1 - a;
\end{align*}
\]
\[
\begin{align*}
x & := y;
\end{align*}
\]

**Load buffering + dependency**

\[
\begin{align*}
a & := x; \quad // 1 \\
y & := a;
\end{align*}
\]
\[
\begin{align*}
x & := y;
\end{align*}
\]

\[T_1 \text{ may NOT promise } y = 1, \text{ since it is not able to write } y = 1 \text{ by itself.}\]
The full model

- Atomic updates
- Release/acquire fences and accesses
- Release sequences
- SC fences and accesses
- Plain accesses (C11’s non-atomics & Java’s normal accesses)

Access Modes

pln □ rlx □ ra □ sc

To achieve all of this we enrich our timestamps, messages, and thread views.
Results

- Compiler optimizations
- Efficient implementation on modern hardware
- DRF guarantees
- No “out-of-thin-air” values
  - Avoid “undefined behavior”
Results

- Compiler optimizations
- Efficient implementation on modern hardware
- DRF guarantees
- No “out-of-thin-air” values
- Avoid “undefined behavior”

Theorem (Local Program Transformations)

The following transformations are sound:

- **Trace-preserving transformations**

- **Reorderings:**
  \[ R^X_{rlx}; R^Y \]
  \[ W^X; W^Y_{rlx} \]
  \[ W_{o_1}^X; R_{o_2}^Y \] unless \( o_1 = o_2 = sc \)

- \[ R^X_{rlx}; R^X_{pln} \]
  \[ R^X_{rlx}; W^Y_{rlx} \]
  \[ R_{\neq rlx}; F_{acq} \]

- \[ W; F_{acq} \]
  \[ F_{rel}; W_{\neq rlx} \]
  \[ F_{rel}; R \]

- **Merges:**
  \[ R_o; R_o \sim R_0 \]
  \[ W_o; W_o \sim W_o \]
  \[ W; R_{ra} \sim W \]
  \[ W_{sc}; R_{sc} \sim W_{sc} \]
Results

- Compiler optimizations
- Efficient implementation on modern hardware
- DRF guarantees
- No “out-of-thin-air” values
- Avoid “undefined behavior”

Theorem (Compilation to TSO/Power)

- Standard compilation to TSO is correct
  - TSO can be fully explained by transformations over SC
- Compilation to Power is correct
  - Using an axiomatic presentation of the promise-free machine
Results

- Compiler optimizations
- Efficient implementation on modern hardware
- DRF guarantees
- No “out-of-thin-air” values
- Avoid “undefined behavior”

Theorem (DRF Theorems)

**Key Lemma**

Races only on ra/sc under promise-free semantics \[\implies\] only promise-free behaviors

**DRF-RA**

Races only on ra/sc under release/acquire semantics \[\implies\] only release/acquire behaviors

**DRF-SC**

Races only on sc under SC semantics \[\implies\] only SC behaviors
Results

- Compiler optimizations
- Efficient implementation on modern hardware
- DRF guarantees
- No “out-of-thin-air” values
- Avoid “undefined behavior”

Theorem (Invariant-Based Program Logic)

Fix a global invariant $J$. Hoare logic where all assertions are of the form $P \land J$, where $P$ mentions only local variables, is sound.
Results

- Compiler optimizations
- Efficient implementation on modern hardware
- DRF guarantees
- No “out-of-thin-air” values
- Avoid “undefined behavior”

Theorem (Invariant-Based Program Logic)

Fix a global invariant $J$. Hoare logic where all assertions are of the form $P \land J$, where $P$ mentions only local variables, is sound.

Load-buffering + data dependency

$x = y = 0$

\[
\begin{align*}
\{ J \} \\
\{ J \land (a = 0) \} \\
\{ J \}
\end{align*}
\begin{align*}
\{ J \} \\
\{ J \}
\end{align*}
\]

$x := y;$
$y := a;$
$J \overset{\text{def}}{=} (x = 0) \land (y = 0)$
Future Work

- Correct compilation to ARMv8
- Global transformations and sequentialization
- Liveness
- Program logic

Future Work

- Correct compilation to ARMv8
- Global transformations and sequentialization
- Liveness
- Program logic


Thank you!
Atomic updates

\[
a := x++; \quad // 0 \quad \| \quad b := x++; \quad // 0
\]

- To obtain \textit{atomicity}, the timestamp order keeps track of immediate adjacency.
- Main challenge: threads performing updates may invalidate the already-certified promises of other threads.

Guiding Principle of Thread Locality

The set of actions a thread can take is determined only by the current memory and its own state.
Atomic updates

To obtain *atomicity*, the timestamp order keeps track of immediate adjacency.

Main challenge: threads performing updates may invalidate the already-certified promises of other threads.

Solution: require certification for *every future memory*.

**Guiding Principle of Thread Locality**

The set of actions a thread can take is determined only by the current memory and its own state.
Certification is needed at every step

\[
\begin{align*}
a &:= x; \quad /\!\!/ 1 \\
b &:= z; \quad /\!\!/ 1 \\
\text{if } b = 0 \text{ then } y &:= 1; \\
x &:= y; \quad /\!\!/ \quad z := 1;
\end{align*}
\]
Sequentialization is unsound

\[
\begin{align*}
a &:= x; \quad \text{// 1} \\
\text{if } a = 0 \text{ then} & \quad y := x; \quad x := y; \quad \sim \quad a := x; \quad \text{// 1} \\
& \quad x := 1; \\
y &:= x; \quad x := 1; \\
y &:= x; \quad x := y;
\end{align*}
\]