A Promising Semantics for Relaxed-Memory Concurrency

Jeehoon Kang[1] Chung-Kil Hur[1] **Ori Lahav**[2]
Viktor Vafeiadis[2] Derek Dreyer[2]

[1]Seoul National University

[2]Max Planck Institute for Software Systems (MPI-SWS)

Kent Concurrency Workshop, July 2016

Programming language concurrency semantics

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.

Programming language concurrency semantics

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

$$
\begin{array}{l||l}
a := x; \; /\!/ \, 1 & \\
& x := y; \\
y := 1; &
\end{array}
$$

This behavior must be allowed:

Power/ARM allow it

The *out-of-thin-air* problem in C11

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

Load-buffering

$$\begin{array}{l||l} a := x;\quad /\!\!/\ 1 & \\ y := 1; & x := y; \end{array}$$

This behavior must be allowed:
Power/ARM allow it

$$\begin{tikzcd}
 & {[x = y = 0]} \arrow[dl] \arrow[dr] & \\
\mathtt{R}_{\mathtt{rlx}} x, 1 \arrow[d] & & \mathtt{R}_{\mathtt{rlx}} y, 1 \arrow[d] \\
\mathtt{W}_{\mathtt{rlx}} y, 1 \arrow[urr] & & \mathtt{W}_{\mathtt{rlx}} x, 1 \arrow[ull]
\end{tikzcd}$$

program order $\longrightarrow$

reads from $\longrightarrow$

The *out-of-thin-air* problem in C11

Load-buffering + data dependency

$$\begin{array}{l|l} a := x; \ /\!/ 1 & \\ y := a; & x := y; \end{array}$$

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

The *out-of-thin-air* problem in C11

Load-buffering + data dependency

```
a := x; // 1    ||    x := y;
y := a;         ||
```

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

$$\begin{tikzcd}
 & {[x = y = 0]} \arrow[dl] \arrow[dr] & \\
\mathtt{R}_{\mathtt{rlx}} x, 1 \arrow[d] & & \mathtt{R}_{\mathtt{rlx}} y, 1 \arrow[d] \\
\mathtt{W}_{\mathtt{rlx}} y, 1 \arrow[urr] & & \mathtt{W}_{\mathtt{rlx}} x, 1 \arrow[ull]
\end{tikzcd}$$

Same execution as before!
C11 allows these behaviors

The *out-of-thin-air* problem in C11

Load-buffering + data dependency

```
a := x;  // 1   ||   x := y;
y := a;         ||
```

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

Load-buffering + control dependencies

```
a := x;  // 1   ||   if (y = 1)
if (a = 1)      ||     x := 1
  y := 1        ||
```

The behavior should be forbidden:
DRF guarantee is broken!

$$\begin{tikzcd}
 & {[x = y = 0]} \arrow[dl] \arrow[dr] & \\
\mathtt{R}_{\mathtt{rlx}} x, 1 \arrow[d] & & \mathtt{R}_{\mathtt{rlx}} y, 1 \arrow[d] \\
\mathtt{W}_{\mathtt{rlx}} y, 1 \arrow[urr] & & \mathtt{W}_{\mathtt{rlx}} x, 1 \arrow[ull]
\end{tikzcd}$$

Same execution as before!
C11 allows these behaviors

The hardware solution

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

Load-buffering + data dependency

$$
\begin{array}{l||l}
a := x; \quad /\!/ \, 1 & \\
y := a; & x := y;
\end{array}
$$

$$\begin{tikzcd}
 & {[x = y = 0]} \arrow[dl] \arrow[dr] & \\
\mathtt{R}_{\mathtt{rlx}} x, 1 \arrow[d] \arrow[d, dashed, bend right] & & \mathtt{R}_{\mathtt{rlx}} y, 1 \arrow[d] \arrow[d, dashed, bend left] \\
\mathtt{W}_{\mathtt{rlx}} y, 1 \arrow[urr] & & \mathtt{W}_{\mathtt{rlx}} x, 1 \arrow[ull]
\end{tikzcd}$$

$$\begin{tikzcd} {} \arrow[r, dashed, "\text{dependency}"] & {} \end{tikzcd}$$

The hardware solution

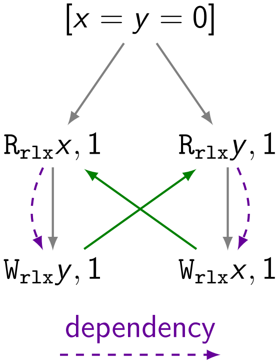
Keep track of syntactic dependencies,
and forbid "dependency cycles".

Load-buffering + data dependency

$a := x;$ // 1 $\quad\parallel\quad x := y;$
$y := a;$

Load-buffering + fake dependency

$a := x;$ // 1 $\quad\parallel\quad x := y;$
$y := a + 1 - a;$


$$\begin{tikzcd}
 & {[x = y = 0]} \arrow[dl] \arrow[dr] & \\
\mathtt{R}_{\mathtt{rlx}} x, 1 \arrow[d] \arrow[d, dashed, bend right] & & \mathtt{R}_{\mathtt{rlx}} y, 1 \arrow[d] \arrow[d, dashed, bend left] \\
\mathtt{W}_{\mathtt{rlx}} y, 1 \arrow[urr] & & \mathtt{W}_{\mathtt{rlx}} x, 1 \arrow[ull]
\end{tikzcd}$$

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

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & y := 1; \\ a := y \ /\!\!/ \ 0 & b := x \ /\!\!/ \ 0 \end{array}$$

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

▶ $x := 1;$	▶ $y := 1;$
$a := y$ // 0	$b := x$ // 0

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
0	0

- Global memory is a pool of messages of the form

$$\langle \textit{location} \; : \; \textit{value} \; @ \; \textit{timestamp} \rangle$$

- 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

$$x = y = 0$$

$x := 1;$	▶ $y := 1;$
▶ $a := y$ // 0	$b := x$ // 0

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$

T_1's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	0

- Global memory is a pool of messages of the form

$$\langle \textit{location} \ : \ \textit{value} \ @ \ \textit{timestamp} \rangle$$

- 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

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & y := 1; \\ \blacktriangleright\ a := y \ \ /\!/\ 0 & \blacktriangleright\ b := x \ \ /\!/\ 0 \end{array}$$

Memory

$\langle x : 0 @ 0 \rangle$
$\langle y : 0 @ 0 \rangle$
$\langle x : 1 @ 1 \rangle$
$\langle y : 1 @ 1 \rangle$

T_1's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	$\cancel{0}$
	1

- Global memory is a pool of messages of the form

$$\langle \textit{location} \ : \ \textit{value} \ @ \ \textit{timestamp} \rangle$$

- 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

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & \quad y := 1; \\ a := y \quad /\!/ \ 0 & \blacktriangleright\ b := x \quad /\!/ \ 0 \\ \blacktriangleright & \end{array}$$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle x : 1@1 \rangle$

$\langle y : 1@1 \rangle$

T_1's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	$\cancel{0}$
	1

- Global memory is a pool of messages of the form

$$\langle \textit{location} \ : \ \textit{value} \ @ \ \textit{timestamp} \rangle$$

- 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

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & y := 1; \\ a := y \quad /\!/ \ 0 & b := x \quad /\!/ \ 0 \\ \blacktriangleright & \blacktriangleright \end{array}$$

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	$\cancel{0}$
	1

- Global memory is a pool of messages of the form

$$\langle \mathit{location} \ : \ \mathit{value} \ @ \ \mathit{timestamp} \rangle$$

- 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

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & y := 1; \\ a := y \quad /\!/\ 0 & b := x \quad /\!/\ 0 \\ \blacktriangleright & \blacktriangleright \end{array}$$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle x : 1@1 \rangle$

$\langle y : 1@1 \rangle$

T_1's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	$\cancel{0}$
	1

Coherence Test

$$x = 0$$

$$\begin{array}{l||l} x := 1; & x := 2; \\ a := x \quad /\!/\ 2 & b := x \quad /\!/\ 1 \end{array}$$

Simple operational semantics for C11's relaxed accesses

Store-buffering

$x = y = 0$

$x := 1;$	$y := 1;$
$a := y$ // 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's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	$\cancel{0}$
	1

Coherence Test

$x = 0$

▶ $x := 1;$	▶ $x := 2;$
$a := x$ // 2	$b := x$ // 1

Memory

$\langle x : 0@0 \rangle$

T_1's view

x
0

T_2's view

x
0

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & y := 1; \\ a := y \quad /\!/\ 0 & b := x \quad /\!/\ 0 \\ \blacktriangleright & \blacktriangleright \end{array}$$

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	$\cancel{0}$
	1

Coherence Test

$$x = 0$$

$$\begin{array}{l||l} x := 1; & \blacktriangleright\ x := 2; \\ \blacktriangleright\ a := x \quad /\!/\ 2 & b := x \quad /\!/\ 1 \end{array}$$

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$

T_1's view

x
$\cancel{0}$
1

T_2's view

x
0

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

$x := 1;$	$\parallel$	$y := 1;$
$a := y$ // 0	$\parallel$	$b := x$ // 0
▶	$\parallel$	▶

Memory
$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	$\cancel{0}$
	1

Coherence Test

$$x = 0$$

$x := 1;$	$\parallel$	$x := 2;$
▶ $a := x$ // 2	$\parallel$	▶ $b := x$ // 1

Memory
$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle x : 2@2 \rangle$

T_1's view

x
$\cancel{0}$
1

T_2's view

x
$\cancel{0}$
2

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & y := 1; \\ a := y \ /\!/ \ 0 & b := x \ /\!/ \ 0 \\ \blacktriangleright & \blacktriangleright \end{array}$$

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

Coherence Test

$$x = 0$$

$$\begin{array}{l||l} x := 1; & x := 2; \\ a := x \ /\!/ \ 2 & \blacktriangleright\ b := x \ /\!/ \ 1 \\ \blacktriangleright & \end{array}$$

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle x : 2@2 \rangle$

T_1's view

x
~~0~~
~~1~~
2

T_2's view

x
~~0~~
2

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & y := 1; \\ a := y \quad /\!/\ 0 & b := x \quad /\!/\ 0 \end{array}$$

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
$\cancel{0}$	0
1	

T_2's view

x	y
0	$\cancel{0}$
	1

Coherence Test

$$x = 0$$

$$\begin{array}{l||l} x := 1; & x := 2; \\ a := x \quad /\!/\ 2 & b := x \quad /\!/\ 1 \end{array}$$

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle x : 2@2 \rangle$

T_1's view

x
$\cancel{0}$
$\cancel{1}$
2

T_2's view

x
$\cancel{0}$
2

Promises

Load-buffering

$$x = y = 0$$

$$\begin{array}{l||l} a := x;\ \text{// } 1 & \\ y := 1; & x := y; \end{array}$$

- 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.

Promises

Load-buffering

$$x = y = 0$$

```
▶ a := x;  // 1   ||   ▶ x := y;
  y := 1;         ||
```

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

T_1**'s view**

x	y
0	0

T_2**'s view**

x	y
0	0

- 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

$$x = y = 0$$

$$\begin{array}{l||l} \blacktriangleright\ a := x;\ \ /\!/\ 1 & \blacktriangleright\ x := y; \\ \quad y := 1; & \end{array}$$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
0	0

- 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

$$x = y = 0$$

$$\begin{array}{l|l} \blacktriangleright\ a := x; \ \text{// 1} & \blacktriangleright\ x := y; \\ \quad y := 1; & \end{array}$$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

T_1**'s view**

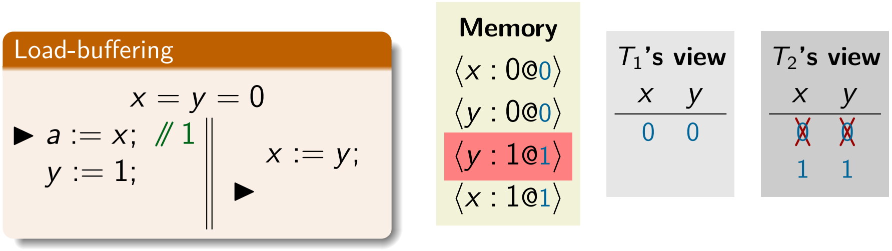
x	y
0	0

T_2**'s view**

x	y
0	$\cancel{0}$
	1

- 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

$$x = y = 0$$

▶ $a := x;$ // 1 $y := 1;$	‖	$x := y;$ ▶

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

$\langle x : 1@1 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
$\cancel{0}$	$\cancel{0}$
1	1

- 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

$$x = y = 0$$

$a := x;$ // 1	$\|$	
▶ $y := 1;$	$\|$	$x := y;$
	$\|$	▶

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

$\langle x : 1@1 \rangle$

T_1**'s view**

x	y
$\cancel{0}$	0
1	

T_2**'s view**

x	y
$\cancel{0}$	$\cancel{0}$
1	1

- 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

$$x = y = 0$$

$a := x;$ // 1 $y := 1;$ ▶	$x := y;$ ▶

Memory
$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle y : 1@1 \rangle$
$\langle x : 1@1 \rangle$

T_1's view	
x	y
~~0~~	~~0~~
1	1

T_2's view	
x	y
~~0~~	~~0~~
1	1

- 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

$$x = y = 0$$

$a := x;$ // 1 $y := 1;$ ▶	$\parallel$	▶ $x := y;$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

$\langle x : 1@1 \rangle$

T_1's view

x	y
$\cancel{0}$	$\cancel{0}$
1	1

T_2's view

x	y
$\cancel{0}$	$\cancel{0}$
1	1

Load-buffering + dependency

$a := x;$ // 1 $y := a;$	$\parallel$	$x := y;$

Must not admit the same execution!

Promises

Load-buffering

$$x = y = 0$$

$$\begin{array}{l||l}
a := x; \quad /\!/\ 1 & \\
y := 1; & x := y; \\
 & \blacktriangleright \\
\blacktriangleright &
\end{array}$$

Load-buffering + dependency

$$\begin{array}{l||l}
a := x; \quad /\!/\ 1 & x := y; \\
y := a; &
\end{array}$$

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 *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

```
a := x;  // 1     ||   x := y;
y := 1;           ||
```

Load buffering + fake dependency

```
a := x;  // 1         ||   x := y;
y := a + 1 − a;       ||
```

T_1 **may promise** $y = 1$, since it is able to write $y = 1$ by itself.

Load buffering + dependency

```
a := x;   // 1    ||   x := y;
y := a;           ||
```

T_1 **may NOT promise** $y = 1$, since it is not able to write $y = 1$ 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

$$\texttt{pln} \sqsubset \texttt{rlx} \sqsubset \texttt{ra} \sqsubset \texttt{sc}$$

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:*

$\mathtt{R}^x_{\sqsubseteq\mathtt{rlx}} ; \mathtt{R}^y$	$\mathtt{W}^x ; \mathtt{W}^y_{\sqsubseteq\mathtt{rlx}}$	$\mathtt{W}^x_{o_1} ; \mathtt{R}^y_{o_2}$ unless $o_1 = o_2 = \mathtt{sc}$
$\mathtt{R}^x_{\sqsubseteq\mathtt{rlx}} ; \mathtt{R}^x_{\mathtt{pln}}$	$\mathtt{R}^x_{\sqsubseteq\mathtt{rlx}} ; \mathtt{W}^y_{\sqsubseteq\mathtt{rlx}}$	$\mathtt{R}_{\neq\mathtt{rlx}} ; \mathtt{F}_{\mathtt{acq}}$
$\mathtt{W} ; \mathtt{F}_{\mathtt{acq}}$	$\mathtt{F}_{\mathtt{rel}} ; \mathtt{W}_{\neq\mathtt{rlx}}$	$\mathtt{F}_{\mathtt{rel}} ; \mathtt{R}$

- *Merges:*

$\mathtt{R}_o ; \mathtt{R}_o \rightsquigarrow \mathtt{R}_0$ $\quad$ $\mathtt{W}_o ; \mathtt{W}_o \rightsquigarrow \mathtt{W}_o$ $\quad$ $\mathtt{W} ; \mathtt{R}_{\mathtt{ra}} \rightsquigarrow \mathtt{W}$ $\quad$ $\mathtt{W}_{\mathtt{sc}} ; \mathtt{R}_{\mathtt{sc}} \rightsquigarrow \mathtt{W}_{\mathtt{sc}}$

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* $\Longrightarrow$ *only promise-free behaviors*
DRF-RA	*Races only on* `ra`/`sc` *under release/acquire semantics* $\Longrightarrow$ *only release/acquire behaviors*
DRF-SC	*Races only on* `sc` *under SC semantics* $\Longrightarrow$ *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 $P \wedge J$, 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 $P \wedge J$, where P mentions only local variables, is sound.

Load-buffering + data dependency

$$x = y = 0$$

$$\begin{array}{l|l} \{J\} & \\ a := x; & \{J\} \\ \{J \wedge (a = 0)\} & x := y; \\ y := a; & \{J\} \\ \{J\} & \end{array}$$

$$J \overset{\text{def}}{=} (x = 0) \wedge (y = 0)$$

Future Work

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

See `http://sf.snu.ac.kr/promise-concurrency/` for Coq proofs.

Future Work

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

See `http://sf.snu.ac.kr/promise-concurrency/` for Coq proofs.

Thank you!

Atomic updates

$$a := x{+}{+}; \quad /\!/\, 0 \;\Big\|\; b := x{+}{+}; \quad /\!/\, 0$$

- 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.

Atomic updates

$$a := x{+}{+};\quad /\!/\ 0 \;\Big\|\; b := x{+}{+};\quad /\!/\ 0$$

- 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.

$$\begin{array}{l} a := x;\ /\!/\ 1 \\ b := z{+}{+};\ /\!/\ 0 \\ y := b + 1; \end{array} \;\Bigg\|\; x := y; \;\Bigg\|\; z{+}{+};$$

- 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

$$
\begin{array}{l}
a := x; \quad /\!\!/\, 1 \\
b := z; \quad /\!\!/\, 1 \\
\texttt{if } b = 0 \texttt{ then } y := 1;
\end{array}
\;\left\|\; x := y; \;\right\|\; z := 1;
$$

Sequentialization is unsound

$$\left.\begin{array}{l} a := x; \quad /\!\!/\ 1 \\ \texttt{if } a = 0 \texttt{ then} \\ \quad x := 1; \end{array}\right\| \; y := x; \; \left\| \; x := y; \right. \quad \rightsquigarrow \quad \left.\begin{array}{l} a := x; \; /\!\!/\ 1 \\ \texttt{if } a = 0 \texttt{ then} \\ \quad x := 1; \\ y := x; \end{array}\right\| \; x := y;$$