The out-of-thin-air problem and a promising solution

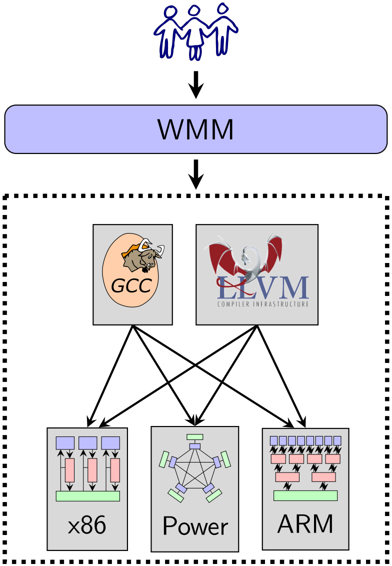
Ori Lahav Viktor Vafeiadis

31 August 2017

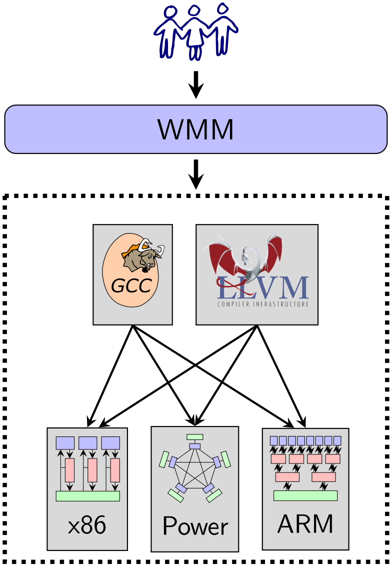
What is the right semantics for

a concurrent programming language?

Programming language concurrency semantics



Programming language concurrency semantics



WMM desiderata

1. Mathematically sane
   (e.g., monotone)
2. Not too strong
   (good for hardware)
3. Not too weak
   (allows reasoning)
4. Admits optimizations
   (good for compilers)
5. No undefined behavior

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

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

Load-buffering

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

This behavior must be allowed:
Power/ARM allow it

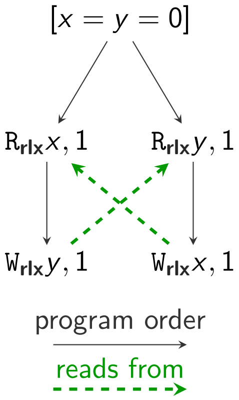
The *out-of-thin-air* problem in C11

- 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



$[x = y = 0]$
$\mathtt{R}_{\mathsf{rlx}}\, x, 1$
$\mathtt{R}_{\mathsf{rlx}}\, y, 1$
$\mathtt{W}_{\mathsf{rlx}}\, y, 1$
$\mathtt{W}_{\mathsf{rlx}}\, x, 1$
program order
reads from

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!

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

Load-buffering + data dependency

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

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

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

Same execution as before!
C11 allows these behaviors

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

Load-buffering + data dependency

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

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

Load-buffering + control dependencies

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

The behavior should be forbidden:
DRF guarantee is broken!

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

Same execution as before!
C11 allows these behaviors

The hardware solution

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

Load-buffering + data dependency

$a := x;$ *// 1*	$\Vert$	$x := y;$
$y := a;$		

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

dependency
$\xrightarrow{\quad\cdots\quad}$

The hardware solution

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

Load-buffering + data dependency

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

Load-buffering + fake dependency

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

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

dependency $\xrightarrow{\quad}$ (dotted)

This approach is not suitable for a programming language:
Compilers do not preserve syntactic dependencies.

A “promising” semantics for relaxed-memory concurrency

We will now describe 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 interleaving semantics, but allow threads to **promise** to write in the future

Simple operational semantics for C11's relaxed accesses

Store buffering

$$x = y = 0$$

```
x := 1;          ||   y := 1;
a := y;   // 0   ||   b := x;   // 0
```

Simple operational semantics for C11's relaxed accesses

Store buffering

$$x = y = 0$$

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

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

$$x = y = 0$$

$x := 1;$	$\parallel$	$\blacktriangleright\ y := 1;$
$\blacktriangleright\ a := y; \quad /\!/ 0$	$\parallel$	$b := x; \quad /\!/ 0$

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

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

- 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

$$x = y = 0$$

$x := 1;$	$\|\|$	$y := 1;$
$\blacktriangleright\ a := y; \quad /\!/\,0$		$\blacktriangleright\ b := x; \quad /\!/\,0$

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

- 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
  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

- Global memory is a pool of messages of the form

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

- 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

$$x = y = 0$$

$x := 1;$	$\parallel$	$y := 1;$
$a := y;$ *// 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
~~0~~	0
1	

T_2's view

x	y
0	~~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

$$x = y = 0$$

T_1	T_2
$x := 1;$	$y := 1;$
$a := y;$ // 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
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

Coherence test

$$x = 0$$

T_1	T_2
$x := 1;$	$x := 2;$
$a := x;$ // 2	$b := x;$ // 1

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
$\langle x : 0@0 \rangle$

T_1's view

x
0

T_2's view

x
0

Simple operational semantics for C11's relaxed accesses

Store buffering

$x = y = 0$

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

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

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

$$x = y = 0$$

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

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

$$x = 0$$

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

Memory

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

T_1's view

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

T_2's view

x
$\cancel{0}$
2

Simple operational semantics for C11's relaxed accesses

Store buffering

$$x = y = 0$$

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

$$x = 0$$

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

Memory

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

T_1's view

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

T_2's view

x
$\cancel{0}$
2

Promises

Load-buffering

$$x = y = 0$$

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

- 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

$$x = y = 0$$

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

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

$$x = y = 0$$

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

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
~~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	$x := y;$
▶ $y := 1;$	▶

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
$\cancel{0}$	$\cancel{0}$
1	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

$$x = y = 0$$

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

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

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

Must not admit
the same execution!

Promises

Load-buffering

$$x = y = 0$$

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

Load-buffering + dependency

```
a := x;  // 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 *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

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

Load buff. + fake dependency

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

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

Load buffering + dependency

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

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

Quick quiz #1

Is this behavior possible?

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

Quick quiz #1

Is this behavior possible?

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

No.

Suppose the thread promises $x = 1$. Then, once $a := x$ reads 1, the thread view is increased and so the promise cannot be fulfilled.

Quick quiz #2

Is this behavior possible?

$$\begin{array}{l} a := x; \quad \textit{// 1} \\ x := 1; \end{array} \;\left\|\; y := x; \;\right\|\; x := y;$$

Quick quiz #2

Is this behavior possible?

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

Yes. And the ARM-Flowing model allows it!

Quick quiz #2

Is this behavior possible?

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

Yes. And the ARM-Flowing model allows it!

This behavior can be also explained by sequentialization:

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

Quick quiz #2

But, note that sequentialization is generally unsound in our model:

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

The full model

- Atomic updates (e.g., CAS, fetch-and-add)
- Release/acquire fences and accesses
- Release sequences
- SC fences (no SC accesses)
- Plain accesses (C11's non-atomics & Java's normal accesses)

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

Release/acquire accesses

Message-passing

$$x = y = 0$$

$$\begin{array}{l|l} x := 1; & a := y_{\mathbf{acq}}; \quad \text{// 1} \\ y :=_{\mathbf{rel}} 1; & b := x; \quad \text{// 1} \end{array}$$

Release/acquire accesses

Message-passing

$$x = y = 0$$

▶ $x := 1;$	▶ $a := y_{\mathbf{acq}};$ // 1
$y :=_{\mathbf{rel}} 1;$	$b := x;$ // 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

Release/acquire accesses

Message-passing

$$x = y = 0$$

$$\begin{array}{r||l}
x := 1; & \blacktriangleright\ a := y_{\textbf{acq}}; \quad /\!/\ 1 \\
\blacktriangleright\ y :=_{\textbf{rel}} 1; & b := x; \quad /\!/\ 1
\end{array}$$

Memory

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

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	0

Release/acquire accesses

Message-passing

$$x = y = 0$$

$x := 1;$	$\|\|$	▶ $a := y_{\mathbf{acq}};$ *// 1*
$y :=_{\mathbf{rel}} 1;$		$b := x;$ *// 1*
▶		

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

T_1's view

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

T_2's view

x	y
0	0

Release/acquire accesses

Message-passing

$$x = y = 0$$

$$\begin{array}{rl||rl}
 & x := 1; & & a := y_{\mathbf{acq}}; \quad /\!/ \mathit{1} \\
 & y :=_{\mathbf{rel}} 1; & \blacktriangleright & b := x; \quad /\!/ \mathit{1} \\
\blacktriangleright & & &
\end{array}$$

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

T_1's view	
x	y
$\cancel{0}$	$\cancel{0}$
1	1

T_2's view	
x	y
$\cancel{0}$	$\cancel{0}$
1	1

Release/acquire accesses

Message-passing

$$x = y = 0$$

$$\begin{array}{l||l} x := 1; & a := y_{\mathbf{acq}}; \quad /\!/ \, 1 \\ y :=_{\mathbf{rel}} 1; & b := x; \quad /\!/ \, 1 \\ \blacktriangleright & \blacktriangleright \end{array}$$

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

T_1's view

x	y
~~0~~	~~0~~
1	1

T_2's view

x	y
~~0~~	~~0~~
1	1

Certification is needed at every step

Key lemma for DRF

Races only on RA under promise-free semantics
$\Rightarrow$ only promise-free behaviors

```
                      ‖ if $w_{\mathbf{acq}} = 1$ then   ‖
                      ‖    $z := 1$;                    ‖
                      ‖ else                            ‖ if $y_{\mathbf{acq}} = 1$ then
$w :=_{\mathbf{rel}} 1$;  ‖    $y :=_{\mathbf{rel}} 1$;      ‖    if $z = 1$ then
                      ‖ $a := x$;   // 1                ‖       $x := 1$;
                      ‖ if $a = 1$ then                 ‖
                      ‖    $z := 1$;                    ‖
```

Invariant-based program logic

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.

- Useful for proving absence of OOTA.

Load-buffering + data dependency

$$x = y = 0$$

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

$$J \triangleq (x = 0) \wedge (y = 0)$$