An introduction to weak memory consistency and the out-of-thin-air problem

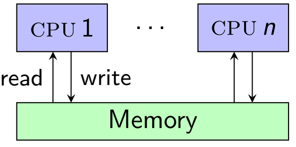
Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

CONCUR, 7 September 2017

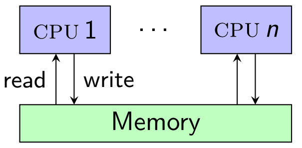
Sequential consistency (SC)

- The standard simplistic concurrency model.
- Threads access shared memory in an interleaved fashion.



Sequential consistency (SC)

- The standard simplistic concurrency model.
- Threads access shared memory in an interleaved fashion.



But...

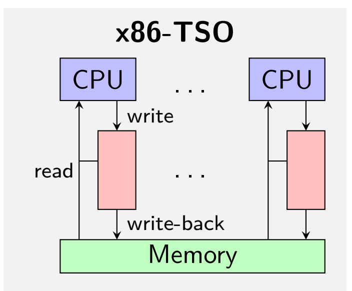
- No multicore processor implements SC.
- Compiler optimizations invalidate SC.
- In most cases, SC is not really necessary.

Weak memory consistency

Store buffering (SB)

Initially, $x = y = 0$

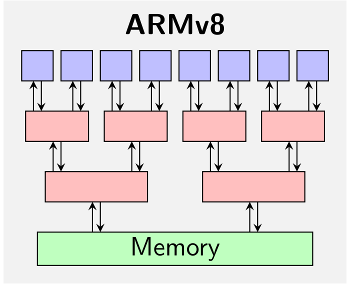
$x := 1;$ $a := y$ *// 0*	$y := 1;$ $b := x$ *// 0*



Load buffering (LB)

Initially, $x = y = 0$

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



Weak consistency in "real life"

- Messages may be delayed.

$MsgX := 1;$ $a := MsgY;$ // 0	‖	$MsgY := 1;$ $b := MsgX;$ // 0

- Messages may be sent/received out of order.

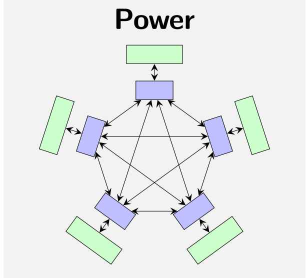
$Email := 1;$ $Sms := 1;$	‖	$a := Sms;$ // 1 $b := Email;$ // 0

Independent reads of independent writes (IRIW)

Initially, $x = y = 0$

$$x := 1 \;\Bigg\|\; \begin{array}{l} a := x; \quad /\!/\, 1 \\ \text{lwsync}; \\ b := y \quad /\!/\, 0 \end{array} \;\Bigg\|\; \begin{array}{l} c := y; \quad /\!/\, 1 \\ \text{lwsync}; \\ d := x \quad /\!/\, 0 \end{array} \;\Bigg\|\; y := 1$$

- Thread II and III can observe the $x := 1$ and $y := 1$ writes happen in different orders.
- Because of the lwsync fences, no reorderings are possible!



Embracing weak consistency

Weak consistency is not a threat, but an opportunity.

- Can lead to more scalable concurrent algorithms.
- Several open research problems.
  - What is a good memory model?

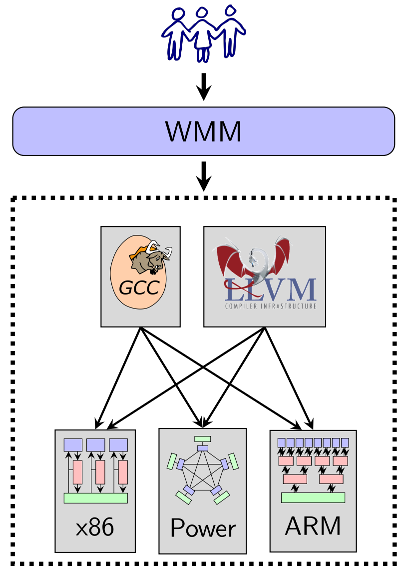
Reasoning under WMC is often easier than under SC.

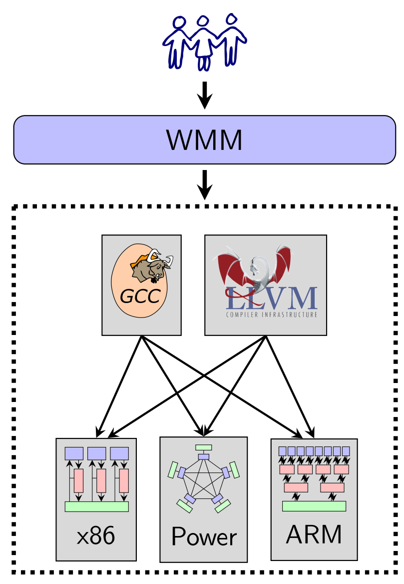
- Avoid thinking about thread interleavings.
- Many/most concurrent algorithms do not need SC!
- Positive *vs* negative knowledge.

What is the right semantics for

a concurrent programming language?

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*

Quiz. Should these transformations be allowed?

1. **CSE over acquiring a lock:**

$$\begin{array}{l} a = x; \\ lock(); \\ b = x; \end{array} \quad \leadsto \quad \begin{array}{l} a = x; \\ lock(); \\ b = a; \end{array}$$

2. **Load hoisting:**

$$\begin{array}{l} \text{if } (c) \\ \quad a = x; \end{array} \quad \leadsto \quad \begin{array}{l} t = x; \\ a = c \,?\, t : a; \end{array}$$

[x is a global variable; a, b, c are local; t is a fresh temporary.]

Allowing both is clearly wrong! [CGO'16,CGO'17]

Consider the transformation sequence:

if (c) $\quad a = x;$ $lock();$ $b = x;$	hoist $\rightsquigarrow$	$t = x;$ $a = c\ ?\ t : a;$ $lock();$ $b = x;$	CSE $\rightsquigarrow$	$t = x;$ $a = c\ ?\ t : a;$ $lock();$ $b = t;$

When c is false, x is moved out of the critical region!

So we have to forbid one transfomation.

- C11 forbids load hoisting, allows CSE over lock().
- LLVM allows load hoisting, forbids CSE over lock().

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

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

Load-buffering

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

This behavior must be allowed:
Power/ARM allow it

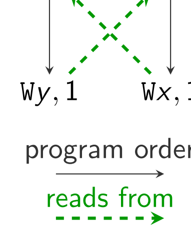
The *out-of-thin-air* problem in C11

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

Load-buffering

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

This behavior must be allowed:
Power/ARM allow it



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

Load-buffering + data dependency

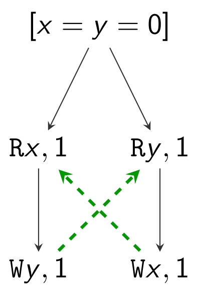
$a := x; \quad /\!/ 1$	$\|$	$b := y;$
$y := a;$	$\|$	$x := b$

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 $y := a;$	‖	$b := y;$ $x := b$



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

Same execution as before!
C11 allows these behaviors

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

Load-buffering + data dependency

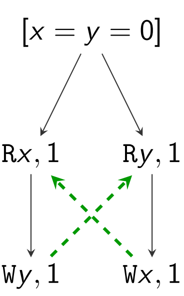
$a := x;$ // 1 $y := a;$	$b := y;$ $x := b$

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

Load-buffering + control dependencies

$a := x;$ // 1 **if** $a = 1$ **then** $y := 1$	$b := y;$ // 1 **if** $b = 1$ **then** $x := 1$

The behavior should be forbidden:
DRF guarantee is broken!



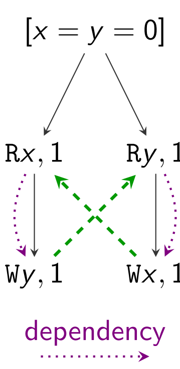
Same execution as before!
C11 allows these behaviors

The hardware solution

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

Load-buffering + data dependency

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



The hardware solution

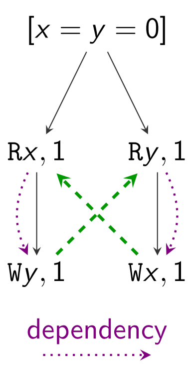
Keep track of syntactic dependencies, and forbid "dependency cycles".

Load-buffering + data dependency

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

Load-buffering + fake dependency

$a := x;$ *// 1* $y := a + 1 - a;$	$b := y;$ *// 1* $x := b;$



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

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

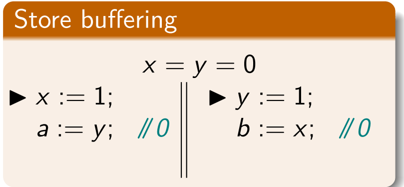
Key idea: Start with an operational interleaving semantics, but allow threads to **promise** to write in the future

Store buffering

$$x = y = 0$$

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

Simple operational semantics for C11's relaxed accesses



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

Store buffering

$$x = y = 0$$

Thread 1		Thread 2
$x := 1;$	∥	▶ $y := 1;$
▶ $a := y;$ // 0	∥	$b := x;$ // 0

Memory

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

T_1's view

x	y
~~0~~ 1	0

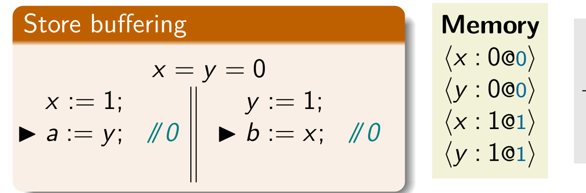
T_2's view

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

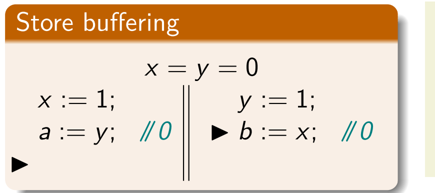


- Global memory is a pool of messages of the form

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

- Each thread maintains a *thread-local view* recording the last observed timestamp for every location

Simple operational semantics for C11's relaxed accesses



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	

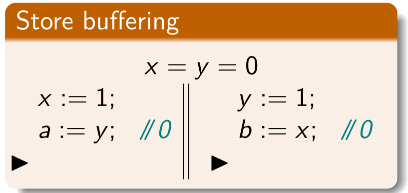
T_2's view	
x	y
0	~~0~~
	1

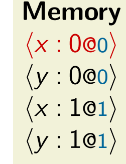
- Global memory is a pool of messages of the form

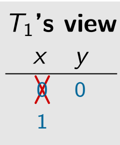
$$\langle \textit{location} \ : \ \textit{value} \ @ \ \textit{timestamp} \rangle$$

- Each thread maintains a *thread-local view* recording the last observed timestamp for every location

Simple operational semantics for C11's relaxed accesses







T_2's view

x	y
0	~~0~~
	1

- Global memory is a pool of messages of the form

$$\langle location \ : \ value \ @ \ timestamp \rangle$$

- 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

$$x = y = 0$$

Thread 1		Thread 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

$$x = 0$$

Thread 1		Thread 2
$x := 1;$	∥	$x := 2;$
$a := x;$ // 2	∥	$b := x;$ // 1

Simple operational semantics for C11's relaxed accesses

Store buffering

$$x = y = 0$$

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

T_2's view

x	y
0	~~0~~
	1

Coherence test

$$x = 0$$

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

$$x = y = 0$$

Thread 1		Thread 2
$x := 1;$	$\parallel$	$y := 1;$
$a := y;$ // 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

Coherence test

$$x = 0$$

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

Memory

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

T_1's view

x
~~0~~
1

T_2's view

x
0

Store buffering

$$x = y = 0$$

Thread 1		Thread 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

$$x = 0$$

Thread 1		Thread 2
$x := 1;$	‖	$x := 2;$
▶ $a := x;$ // 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
~~0~~
1

T_2's view

x
~~0~~
2

Simple operational semantics for C11's relaxed accesses

Store buffering

$$x = y = 0$$

Thread 1		Thread 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

$$x = 0$$

Thread 1		Thread 2	
$x := 1;$		$x := 2;$	
$a := x;$	// 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
~~0~~
~~1~~
2

T_2's view

x
~~0~~
2

Simple operational semantics for C11's relaxed accesses

Store buffering

$$x = y = 0$$

Thread 1		Thread 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

$$x = 0$$

Thread 1		Thread 2
$x := 1;$	∥	$x := 2;$
$a := x;$ // 2	∥	$b := x;$ // 1

Memory

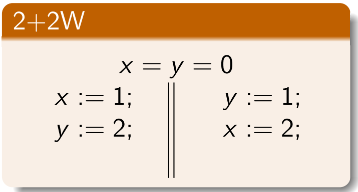
$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle x : 2@2 \rangle$

T_1**'s view**

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

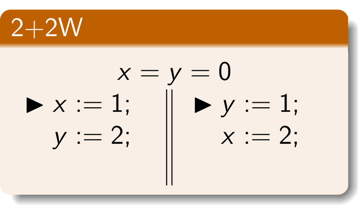
T_2**'s view**

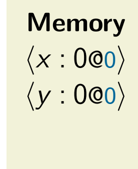
x
~~0~~
2



- We want to allow the final outcome $x = y = 1$.

Supporting write-write reordering





T_1's view

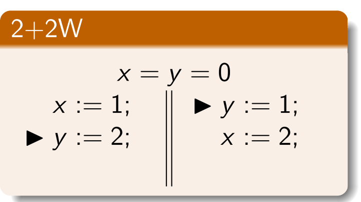
x	y
0	0

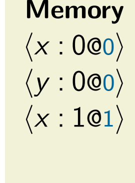
T_2's view

x	y
0	0

- We want to allow the final outcome $x = y = 1$.

Supporting write-write reordering





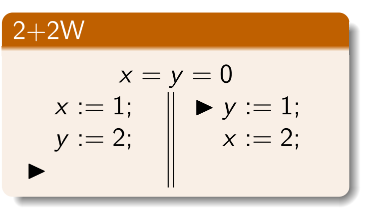
T_1's view

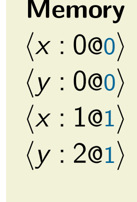
x	y
~~0~~	0
1	

T_2's view

x	y
0	0

- We want to allow the final outcome $x = y = 1$.





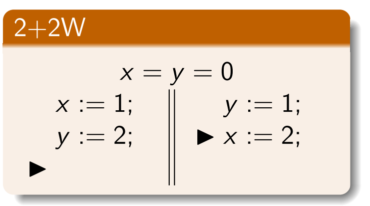
T_1's view

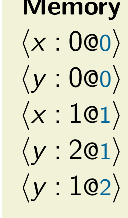
x	y
~~0~~	~~0~~
1	1

T_2's view

x	y
0	0

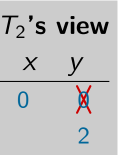
- We want to allow the final outcome $x = y = 1$.





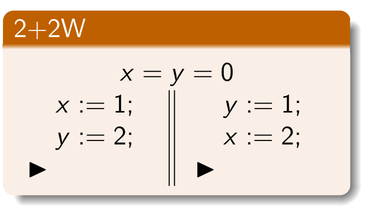
T_1's view

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



- We want to allow the final outcome $x = y = 1$.

Supporting write-write reordering

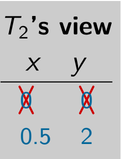


Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 2@1 \rangle$
$\langle y : 1@2 \rangle$
$\langle x : 2@0.5 \rangle$

T_1's view

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



- We want to allow the final outcome $x = y = 1$.
- Writes choose timestamp *greater than the thread's view*, not necessarily the globally greatest one.

Promises

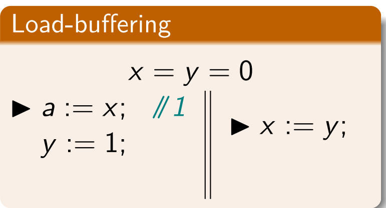
Load-buffering

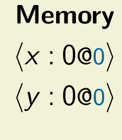
$$x = y = 0$$

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

- 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





T_1's view

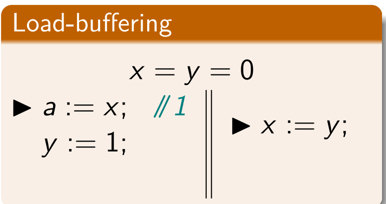
x	y
0	0

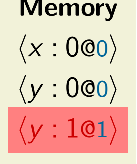
T_2's view

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





T_1's view

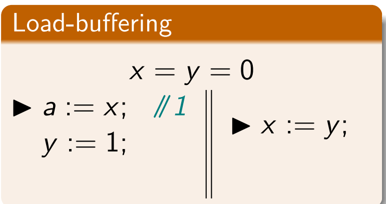
x	y
0	0

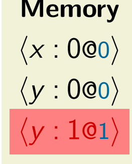
T_2's view

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





T_1's view

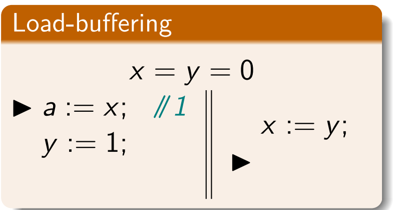
x	y
0	0

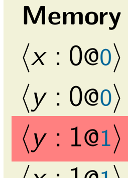
T_2's view

x	y
0	~~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





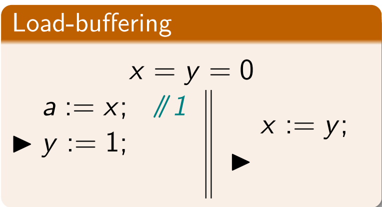
T_1's view

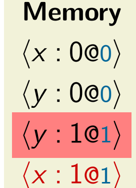
x	y
0	0

T_2's view

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

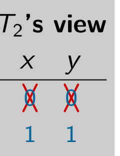
- To model load-store reordering, we allow **“promises”**.
- At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.





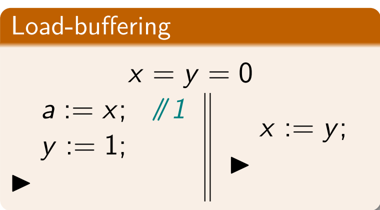
T_1's view

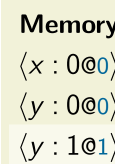
x	y
~~0~~	0
1	



- 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.

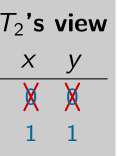
Promises





T_1'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.

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

Load-buffering + dependency

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

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 promise *only 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.

Load-buffering

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

Load buff. + fake dependency

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

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

Load buffering + dependency

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

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

Is this behavior possible?

$$a := x; \quad /\!/ 1$$
$$x := 1;$$

Quick quiz #1

Is this behavior possible?

$$a := x; \quad /\!/ 1$$
$$x := 1;$$

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?

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

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!

Quick quiz #2

Is this behavior possible?

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

Yes. And the ARM-Flowing model allows it!

This behavior can be also explained by sequentialization:

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

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

$$\begin{array}{l} a := x; \;\; /\!/ 1 \\ \textbf{if } a = 0 \textbf{ then} \\ \quad x := 1; \end{array} \;\Bigg\|\; y := x; \;\Bigg\|\; x := y; \quad \rightsquigarrow \quad \begin{array}{l} a := x; \;\; /\!/ 1 \\ \textbf{if } a = 0 \textbf{ then} \\ \quad x := 1; \\ y := x; \end{array} \;\Bigg\|\; x := y;$$

The full model

In the paper, we extend this semantics to handle:

- 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.

- A promising semantics for relaxed-memory concurrency. J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, D. Dreyer. POPL'17

Atomic updates (RMW instructions)

Ensuring atomicity:

- The timestamp order keeps track of immediate adjacency.
  (Technically, we use ranges of timestamps.)

Parallel atomic increment

$a := x{+}{+};\ /\!/\, 0 \rightarrow 1 \ \Big\| \ b := x{+}{+};\ /\!/\, 0 \rightarrow 1$

How are promises affected?

- To allow reorderings, updates can be promised.
- Performing an update may invalidate existing already-certified promises of other threads.

Main challenge

- Threads performing updates may invalidate the already-certified promises of other threads.

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

Conservative 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.

Release/acquire accesses

Message-passing

$$x = y = 0$$

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

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

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle x : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	0

Release/acquire accesses

Message-passing

$$x = y = 0$$

$x := 1;$	$\parallel$	▶ $a := y_{\mathbf{acq}};$ // 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
~~0~~	~~0~~
1	1

T_2's view

x	y
0	0

Message-passing

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$

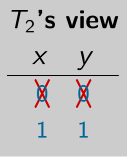
$\langle y : 0@0 \rangle$

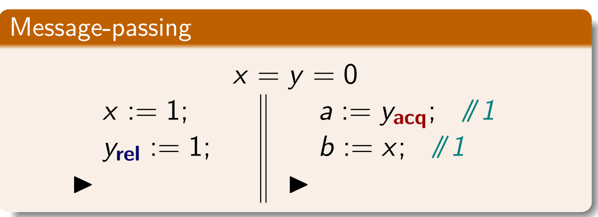
$\langle x : 1@1 \rangle$

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

T_1's view

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





Memory

$\langle x : 0@0 \rangle$

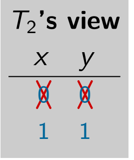
$\langle y : 0@0 \rangle$

$\langle x : 1@1 \rangle$

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

T_1's view

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



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:*

$R^x_{\sqsubseteq \mathsf{rlx}} ; R^y$	$W^x ; W^y_{\sqsubseteq \mathsf{rlx}}$	$W^x_{o_1} ; R^y_{o_2}$
$R^x_{\mathsf{pln}} ; R^x_{\mathsf{pln}}$	$R^x_{\sqsubseteq \mathsf{rlx}} ; W^y_{\sqsubseteq \mathsf{rlx}}$	$R_{\neq \mathsf{rlx}} ; F_{\mathsf{acq}}$
$W ; F_{\mathsf{acq}}$	$F_{\mathsf{rel}} ; W_{\neq \mathsf{rlx}}$	$F_{\mathsf{rel}} ; R$

- *Merges:*

$R_o ; R_o \rightsquigarrow R_o$	$W_o ; W_o \rightsquigarrow W_o$	$W ; R_{\mathsf{acq}} \rightsquigarrow W$

Results

- ☑ Compiler optimizations
- ☑ Efficient implementation on modern hardware
- ☐ DRF guarantees
- ☐ No "out-of-thin-air" values
- ☑ Avoid "undefined behavior"

Theorem (Compilation to TSO/Power/ARM)

- *Standard compilation to TSO is correct*
  - *TSO can be fully explained by transformations over SC*
- *Compilation to Power is correct*
  - *Using a declarative presentation of the promise-free machine*
- *Compilation to ARMv8 is correct*
  - *(For a subset of the features)*

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 under promise-free semantic*
$\Rightarrow$ *only promise-free behaviors*

DRF-RA *Races only on RA under release/acquire semantics*
$\Rightarrow$ *only release/acquire behaviors*

DRF-locks *Races only on lock variables under SC semantics*
$\Rightarrow$ *only SC behaviors*

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

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

Certification is needed at every step

$w_{\mathsf{rel}} := 1;$	**if** $w_{\mathsf{acq}} = 1$ **then**
	  $z := 1;$
	else
	  $y_{\mathsf{rel}} := 1;$
	$a := x$ *// 1*
	if $a = 1$ **then**
	  $z := 1;$
		if $y_{\mathsf{acq}} = 1$ **then**
		  if $z = 1$ **then**
		    $x := 1;$

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

$$x = y = 0$$

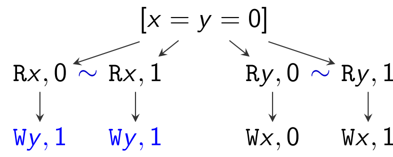
Thread 1	Thread 2
$\{J\}$	$\{J\}$
$a := x;$	$b := y;$
$\{J \wedge a = 0\}$	$\{J \wedge b = 0\}$
$y := a;$	$x := b;$
$\{J \wedge a = 0\}$	$\{J \wedge b = 0\}$

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

Distinguishing programs by event structures

Load-buffering

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



Distinguishing programs by event structures

Load-buffering

$$a := x; \quad /\!/ 1 \qquad \Big\| \qquad b := y;$$
$$y := 1; \qquad\qquad \Big\| \qquad x := b;$$

$[x = y = 0]$

$\mathtt{R}x, 0 \sim \mathtt{R}x, 1 \qquad \mathtt{R}y, 0 \sim \mathtt{R}y, 1$

$\mathtt{W}y, 1 \quad \mathtt{W}y, 1 \qquad \mathtt{W}x, 0 \quad \mathtt{W}x, 1$

LB + data dependency

$$a := x; \quad /\!/ 1 \qquad \Big\| \qquad b := y;$$
$$y := a; \qquad\qquad \Big\| \qquad x := b;$$

$[x = y = 0]$

$\mathtt{R}x, 0 \sim \mathtt{R}x, 1 \qquad \mathtt{R}y, 0 \sim \mathtt{R}y, 1$

$\mathtt{W}y, 0 \quad \mathtt{W}y, 1 \qquad \mathtt{W}x, 0 \quad \mathtt{W}x, 1$

LB + control dependency

$a := x; \quad /\!/ 1$
if $a \neq 0$ **then**
$\quad y := a;$

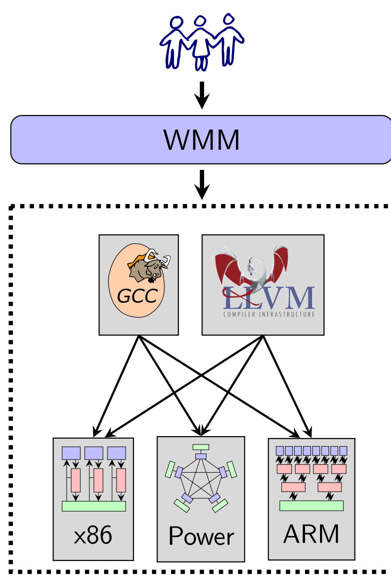
$\Big\|$

$b := y;$
$x := b;$

$[x = y = 0]$

$\mathtt{R}x, 0 \sim \mathtt{R}x, 1 \qquad \mathtt{R}y, 0 \sim \mathtt{R}y, 1$

$\mathtt{W}y, 1 \qquad \mathtt{W}x, 0 \quad \mathtt{W}x, 1$



Summary

- Weak memory consistency
- The **OOTA** problem
- The **promising model**
- An event structure model

Challenges

- Handling global optimizations
- Verification under the promising semantics
- Relating the models
- Liveness under WMC