Multicore Programming

Java Memory Model

Peter Sewell   Jaroslav Ševčík   Tim Harris

University of Cambridge   MSR

with thanks to

Francesco Zappa Nardelli, Susmit Sarkar, Tom Ridge, Scott Owens, Magnus O. Myreen, Luc Maranget, Mark Batty, Jade Alglave

October – November, 2010

Overview

- Introduction to the Java Memory Model (JMM)
  - Motivating examples.
- Overview of transformation legality in the JMM.
- Definition of the JMM:
  - overview of the formal definition,
  - operational view of the JMM,
  - examples.
- Flaws in the JMM:
  - several standard optimisations not legal. . .
  - . . . including some that are implemented in HotSpot.

Java Memory Model

The Java Memory Model (JMM)

- is a contract between hardware, compiler and programmers.
- describes legal behaviours in a multi-threaded Java code with respect to the shared memory.
- implies:
  - Promises for programmers to enable implementation-independent reasoning about programs (DRF principle with a twist).
  - Security guarantees (no out-of-thin-air values, final fields immutable etc.).
  - Legal optimisations for compiler/JVM implementors.

Data Race Freedom

What is data race freedom in the JMM?

Program is data race free if there is no interleaving with a write immediately followed by a memory access to the same (non-volatile) memory location from a different thread.

Note: This is slightly different from the definition in the JMM, but it is equivalent to the JMM definition.

Java guarantees an illusion of sequential consistency if your program is data race free!

DRF Guarantee Example

First, consider the program

```
     x = y = 0
------------------------
y := 1      | x := 1
r1 := x     | r2 := y
print r1    | print r2
```

(Notation in examples (following the JMM): x, y, z are shared variables, rn are thread-local.)

Observe that the program has an interleaving with a data race:

$$\mathsf{W}_{t_i}\ x{=}0, \mathsf{W}_{t_i}\ y{=}0, \mathsf{W}_{t_1}\ y{=}1, \underline{\mathsf{W}_{t_2}\ x{=}1, \mathsf{R}_{t_1}\ x{=}1}, \mathsf{R}_{t_2}\ y{=}1, \mathsf{P}_{t_1}\ 1, \mathsf{P}_{t_2}\ 1$$

DRF Guarantee Example

Keep considering the program

```
       x = y = 0
-------------------------
y := 1      | x := 1
r1 := x     | r2 := y
print r1    | print r2
```

To make the program DRF, protect shared memory `x`, `y` with locks . . .

DRF Guarantee Example

Shared memory `x` protected with `m1`, `y` with `m2`:

```
        x = y = 0
-------------------------
lock m2     | lock m1
y := 1      | x := 1
unlock m2   | unlock m1
lock m1     | lock m2
r1 := x     | r2 := y
unlock m1   | unlock m2
print r1    | print r2
```

This is DRF because between any two accesses to the same memory there must be an unlock and a lock of the protecting monitor . . .

DRF Guarantee Example

Shared memory `x` protected with `m1`, `y` with `m2`:

`x = y = 0`

`lock m2`	`lock m1`
`y := 1`	`x := 1`
`unlock m2`	`unlock m1`
`lock m1`	`lock m2`
`r1 := x`	`r2 := y`
`unlock m1`	`unlock m2`
`print r1`	`print r2`

…so reasonable languages guarantee sequentially consistent behaviours, i.e., it is guaranteed that the program prints 11 or 01 or 10 (but never 00).

DRF Guarantee Example

Still keep considering the program

```
        x = y = 0
-----------------------
y := 1      | x := 1
r1 := x     | r2 := y
print r1    | print r2
```

Java offers another way of synchronization: if you explicitly mark the “racy” locations as volatile, the program is still considered data race free.

Hence, declaring `x` and `y` as volatile makes the program data race free in the JMM.

DRF Guarantee Example

Java offers another way of synchronization: if you explicitly mark the “racy” locations as volatile, the program is still considered data race free.

For example, the program

```
volatile int x = 0
volatile int y = 0
---------------------------
y := 1     | x := 1
r1 := x    | r2 := y
print r1   | print r2
```

is data race free in the JMM, and behaviour 00 is forbidden.

DRF Guarantee Example

Note that only the racy memory locations must be declared volatile.

For example, consider the program:

```
             x = y = 0
-----------------------------------
y := 1 | r := x
x := 1 | if (r == 1) print y
```

. . . and note that `y` is not racy because between the two accesses of `y` there must be an access to `x`.

So declaring `x` as volatile makes the program data race free.

Out-of-thin-air

Programs should never read values that cannot be written by the program(!?).

For example, in

initially `x = y = 0`

`r1 := x`	`r2 := y`
`y := r1`	`x := r2`
`print r1`	`print r2`

the only possible result should be printing two zeros because no other value appears in or can be created by the program.

Out-of-thin-air on references

The previous example might seem benign (program can always leak numeric values through non-determinism and arithmetic, in any case).

However, this is not so benign for references:

initially `x = y = null`	
`r1 := x`	`r2 := y`
`y := r1`	`x := r2`
`r1.run()`	

What should `r1.run()` call? If we allow out-of-thin-air, then it could do anything.

Out-of-thin-air and Optimisations

Out-of-thin-air excludes some program transformations that are correct under the DRF guarantee.

For example, under the DRF guarantee it is correct to speculate on values of writes:

```
                    y := 42
r1 := x             r1 := x
y := r1      ⇒      if (r1 != 42) y := r1;
print r1            print r1
```

Using this, our out-of-thin-air example could output 42!

Out-of-thin-air and Optimisations

Consider our out-of-thin-air example:

initially `x = y = 0`

`r1 := x`	`r2 := y`
`y := r1`	`x := r2`
`print r1`	`print r2`

which should never print 42.

However, if we use the value speculation and rewrite the first thread…

Out-of-thin-air and Optimisations

The transformed program

initially `x = y = 0`

`y := 42`	`r2 := y`
`// Interleave here`	`x := r2`
`r1 := x`	`print r2`
`if (r1 != 42) y := r1`	
`print r1`	

can suddenly print 42!

This will be theoretically possible in the upcoming revision of C++ (C++0x), but not in Java!

Final Fields

One related issue in Java are final fields and immutable objects.

For instance, programmers assume that instances of `String` never change.

This might be tricky in the presence of optimisations.

Consider the program

Initially, `s = s1 = null`

`s = "ab"`	`print s1`
`s1 = s.substring(1, 1)`	
`print s1`	

Final Fields

In reality, strings are often implemented as objects containing character buffer (b), start index (s) and length (l). So our program becomes

Initially, s = s1 = null

`r=alloc(...); r.b="ab"`	`printn s1.b+s1.s,`
`r.l=2; r.s=0; s=r`	`s1.l`
`r1=alloc(...); r1.b=s.b`	
`r1.l=1;r1.s=s.s+1;s1=r1`	
`printn s1.b+s1.s, s1.l`	

(printn p,n prints n characters, starting from pointer p.)

This can still only print b (possibly twice), but if the compiler/hardware reorders the statement s1=r1 earlier . . .

Final Fields

…then we get the program

Initially, `s = s1 = null`

`r=alloc(...); r.b="ab"`	`printn s1.b+s1.s,`
`r.l=2; r.s=0; s=r`	`s1.l`
`r1=alloc(...); s1=r1`	
`r1.b=s.b; r1.l=1;`	
`// Interleave here`	
`r1.s=s.s+1;`	
`printn s1.b+s1.s, s1.l`	

…which can print a and b. So printing the same string could yield two different values. Compilers must prevent such optimisations!

Brief History of JMM

The Java Memory Model (Manson, Pugh and Adve, POPL 2005) was introduced after the original memory model was found to be “fatally flawed” (Pugh, 2000). The main flaws were:

- Many optimisations illegal (including CSE),
- Final fields could be observed to change,
- Unclear semantics of finalisation.

The JMM aims to fix these problems with 3 different fixes.

The core of the JMM only deals with the first problem. This lecture is about the core.

Brief History of JMM

The new JMM:

- part of the Java Language Specification,
- accompanied by a POPL paper with two theorems:
  - Data race free programs have only sequentially consistent behaviours (Theorem 3 of the POPL paper, DRF guarantee). This allows using standard reasoning for DRF programs.
  - Reordering of independent statements is legal. (Theorem 1.) This was falsified by Cenciarelli et al. (2007). Can be partially fixed.
- claims several properties informally:
  - Out-of-thin-air behaviours are prevented (security).
  - Adding synchronisation is a legal transformation.

Optimisation Correctness Overview

Transformation	SC	JMM	DRF
Trace-preserving transformations	$\checkmark$	$\checkmark$	$\checkmark$
Reordering normal memory accesses	$\times$	$\times^*$	$\checkmark$
Redundant read after read elimination	$\checkmark^*$	$\times$	$\checkmark$
Redundant read after write elimination	$\checkmark^*$	$\checkmark$	$\checkmark$
Irrelevant read elimination	$\checkmark$	$\checkmark$	$\checkmark$
Irrelevant read introduction	$\checkmark$	$\times$	?
Redundant write before write elimination	$\checkmark^*$	$\checkmark$	$\checkmark$
Redundant write after read elimination	$\checkmark^*$	$\times$	$\checkmark$
Roach-motel reordering	$\checkmark^*$	$\times$	$\checkmark$
External action reordering	$\times$	$\times$	$\checkmark$

$\checkmark$– correct, $\times$– incorrect,

$\checkmark^*$ – correct only for adjacent memory accesses, $\times^*$ – easily fixable.

Optimisations and the JMM

The situation with the JMM is not settled:

Some standard optimisations, including CSE, are not valid, but compilers still perform them (Sun HotSpot). One can even observe behaviours forbidden by the JMM.

It is not likely that JVMs will sacrifice these optimisations. The JMM will have to be changed.

In addition, Java 7 will introduce explicit memory fences in the JDK. These do not have a clear meaning in the JMM.

Optimisations and the JMM

Correct compiler optimisation:

- If an optimisation does not change any sequence of shared memory accesses, it is legal. This includes:
  - Loop unrolling,
  - Final/static method inlining,
  - Redundant conditional elimination (e.g., if both branches are the same).
- Removing reads based on previous writes is legal
  - . . . even if the read and the write are not adjacent.
- Removing overwritten writes is legal.
- Reordering of independent reads/writes almost legal:
  - Loop rearrangements, code motion.

Optimisations and the JMM

Compiler optimisations that should be avoided:

- Reusing older reads/writes across synchronisation.
- Optimisations that introduce writes with new values or to memory locations that would not be otherwise written.
  - These are often illegal even in the DRF guarantee model.
  - Introducing a write with the same value and to the same location as an existing write in the same basic block is safe (but probably not profitable).
- Introducing reads (even if their value is thrown away).

Optimisations and the JMM

Optimisations to avoid in the current JMM:

- Reusing values of previous reads (including CSE).
- Reordering with I/O operations
- Reordering with synchronisation
  - Treating synchronisation and I/O as opaque is a good idea.

Reorder independent memory accesses is also illegal in the JMM, but it would be legal with a small change in the JMM.

Optimisations and the JMM

Legality of hardware optimisations is a slightly different issue because we must relate two different models–the processor model and the JMM:

- For each execution of the processor model there must be a JMM-execution with the same behaviour.

The difficulty of showing validity depends on the model:

- simple for write buffering model (Sun TSO, x86) or location consistency because these models are simple.
- straightforward for Intel Itanium because there is a total order that can be used to construct the JMM execution.
- nearly impossible for Power and ARM MMs because they are not well-understood.

Basic Definitions (Action)

Java does not have a global store or global time. These are approximated by actions, orders and a visibility function.

An action $\langle t, k, v, u\rangle$ is described by:

1. thread t performing the action,
2. kind k of the action:
   - volatile read or write, non-volatile read or write, lock, unlock, external, synthetic actions (first and last action of thread etc.),
   - volatile reads, writes, locks and unlocks are synchronization actions,
3. runtime variable or monitor v associated with the action,
4. unique identifier u.

Basic Definitions (Execution)

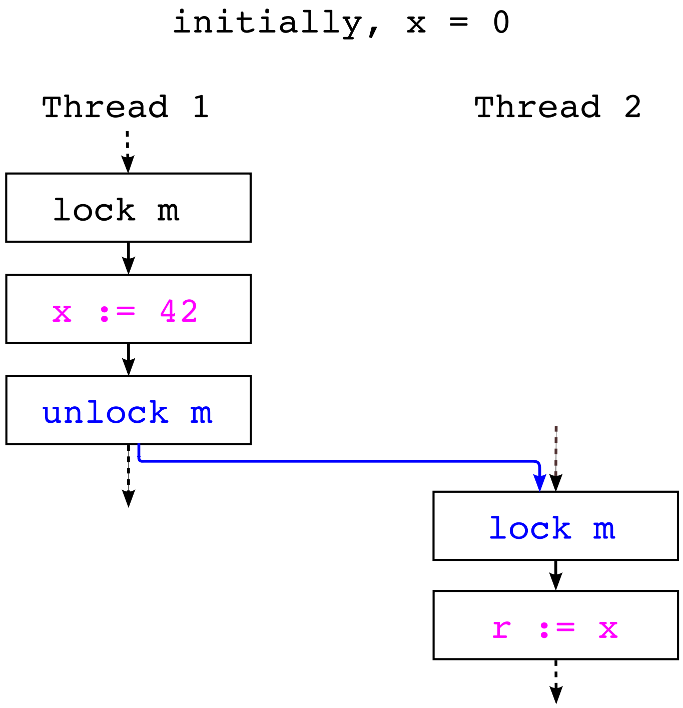
Program order $\leq_{po}$ is a union of total orders on actions of each thread.

Synchronisation order $\leq_{so}$ is a total order on synchronisation actions.

Happens-before order: $a \leq_{hb} b$ is the least order such that

1. If $a \leq_{so} b$ and a, b is a release-acquire pair then $a \leq_{hb} b$.
   a, b is a release-acquire pair if:
   - a is an unlock, b is a lock on the same monitor, or
   - a is a volatile write to v, b is a volatile read from v,
2. if $a \leq_{po} b$ then $a \leq_{hb} b$,
3. if $a \leq_{hb} c$ and $c \leq_{hb} b$ then $a \leq_{hb} b$,
4. initialisation happens-before everything else.

Happens-before Example I



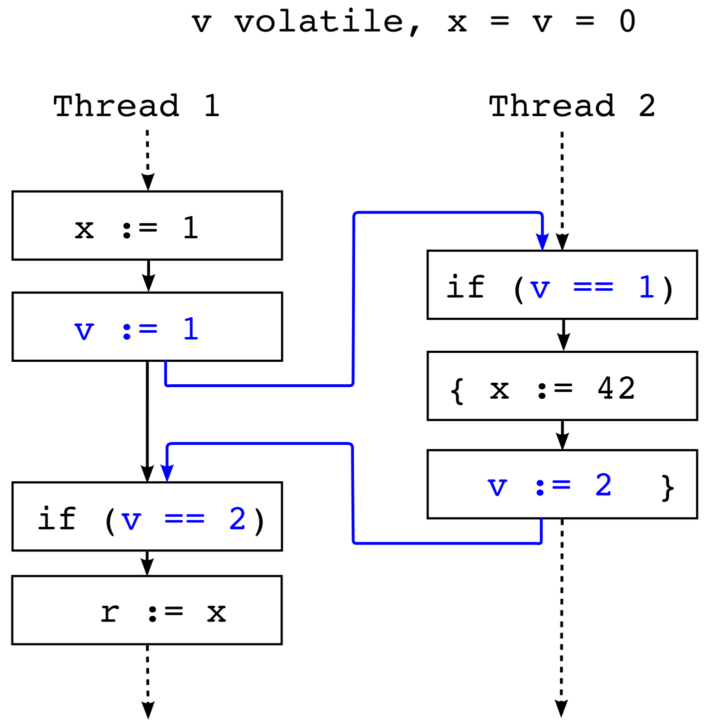
Assuming that `unlock(m)` $\leq_{so}$ `lock(m)`, we have

`x:=42` $\leq_{hb}$ `r:=x`,

because `x:=42` $\leq_{po}$ `unlock(m)`$\leq_{so}$ `lock(m)` $\leq_{po}$ `r:=x`, and `unlock(m)` and `lock(m)` are release-acquire pair.

Happens-before Example II

In general, reads cannot see writes that happen after them or are overwritten.



If r:=x gets executed, then it must see the write x:=42. The other writes to x are overwritten.

Execution Formally

Execution $\langle P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb} \rangle$ consists of:

1. program P
2. set of actions A
3. program order $\leq_{po}$ – union of total orders on actions of each thread
4. synchronization order $\leq_{so}$ – total order over all synchronization actions
5. write-seen function W assigns a write to each read
6. value-written function V assigns a value to each write
7. synchronizes-with order $<_{sw}$ are the release-acquire pairs from $\leq_{so}$
8. happens-before order $\leq_{hb}$

Well-formed executions

Execution is well-formed if:

- each read of x sees a write to x, i.e. $r.loc = W(r).loc$,
- $\{x \in A \,:\, x \leq_{so} y\}$ is finite for each $y \in A$,
- $\leq_{so}$ is consistent with $\leq_{po}$,
- $\leq_{so}$ is consistent with mutual exclusion of locks,
- the execution is intra-thread consistent,
- volatile reads are consistent with $\leq_{so}$, i.e. for all volatile reads r we have $\neg(r \leq_{so} W(r))$ and there is no w s.t. $w.loc = r.loc \wedge W(r) <_{so} w \leq_{so} r$ (volatile reads see the most recent write in $\leq_{so}$),
- all reads are consistent with $\leq_{hb}$ (reads see a most recent write in $\leq_{hb}$).

Legal Execution I

An execution $E = \langle P, A, \leq_{po}, \leq_{so}, W, V, <_{sw}, \leq_{hb} \rangle$ satisfies the causality requirement if there is a sequence of sets of actions $\{C_i\}$ satisfying

- $C_0 = \emptyset$
- $C_i \subset C_{i+1}$
- $A = \bigcup C_i$

and a sequence of well formed executions $E_i = \langle P, A_i, \leq_{po_i}, \leq_{so_i}, W_i, V_i, <_{sw_i}, \leq_{hb_i} \rangle$ such that the following holds:

Legal Execution II

1. $C_i \subseteq A_i$,
2. $\leq_{hb_i} |_{C_i} = \leq_{hb} |_{C_i}$,
3. $\leq_{so_i} |_{C_i} = \leq_{so} |_{C_i}$,
4. $V_i|_{C_i} = V|_{C_i}$,
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$,
6. for any read $r \in A_i - C_{i-1}$ we have $W_i(r) \leq_{hb_i} r$,
7. for any read $r \in C_i - C_{i-1}$ we have $W_i(r) \in C_{i-1}$ and $W(r) \in C_{i-1}$,
8. If $x <_{ssw_i} y \leq_{hb_i} z$ and $z \in C_i - C_{i-1}$ then $x <_{sw_j} y$ for all $j \geq i$ ($<_{ssw_i}$ is transitive reduction of $<_{sw_i}$ without edges from $\leq_{po_i}$)

Legal Execution II

. . . can be weakened to

1. $C_i \subseteq A_i$,
2. For all reads $r \in C_i$ we have $W(r) \leq_{hb} r \iff W(r) \leq_{hb_i} r$, and $r \not\leq_{hb_i} W(r)$,
3. $V_i|_{C_i} = V|_{C_i}$,
4. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$,
5. for any read $r \in A_i - C_{i-1}$ we have $W_i(r) \leq_{hb_i} r$,
6. for any read $r \in C_i - C_{i-1}$ we have $W(r) \in C_{i-1}$.

without invalidating the DRF guarantee.

Sequential Consistency (SC)

We say that an execution is sequentially consistent if there is a total order on actions consistent with the happens-before order such that each read sees the most recent write in that order.

In other words, sequential consistency simulates interleaved semantics.

Note:
Sequential consistency and well-formedness imply legality.

Data Race Free Program (using hb)

Two accesses to the same non-volatile variable, of which at least one is write, are a data race if they are not ordered by happens-before.

Program P is data race free (DRF) if no sequentially consistent execution of P contains a data race.

The definition of a DRF program is equivalent to the DRF definition in terms of interleavings and adjacent actions.

Committing Sequence

- Start with a “well-behaved” execution—all reads see writes from the same thread or through synchronisation,
  - i.e., reads see writes that happen-before them.
- The JMM commits one or more read-write data races.
- Then it restarts the execution, but it must keep the commitment:
  - It must perform all the committed actions.
  - The reads must see the value that they were committed with.
  - Happens-before relationships of actions in the commitment must be preserved.
- The JMM may commit more actions and restart again.

Well-behaved Executions

In a well-behaved execution, all reads see writes that happen-before them.

For DRF programs, execution is well-behaved iff SC.

Otherwise, the program can result in $\mathtt{r1} \neq \mathtt{r2}$ in a well-behaved execution, which is not possible in SC.

```
          x = 0
-----------------------------
lock m      | x:=2
x:=1        | lock m
unlock m    | r1:=x
            | r2:=x
            | unlock m
```

Well-behaved execution example

```
          x = 0
-------------------------
lock m    | x:=2
x:=1      | lock m
unlock m  | r1:=x
          | r2:=x
          | unlock m
          | print r1
          | print r2
```

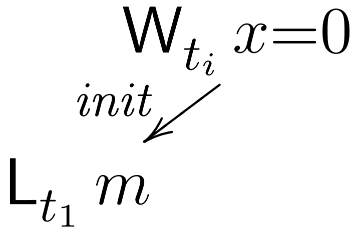
Well-behaved execution example

```
          x = 0
---------------------
lock m    | x:=2
x:=1      | lock m
unlock m  | r1:=x
          | r2:=x
          | unlock m
          | print r1
          | print r2
```

$\mathsf{W}_{t_i}\ x{=}0$

Well-behaved execution example

```
         x = 0
--------------------------
 lock m    | x:=2
 x:=1      | lock m
 unlock m  | r1:=x
           | r2:=x
           | unlock m
           | print r1
           | print r2
```



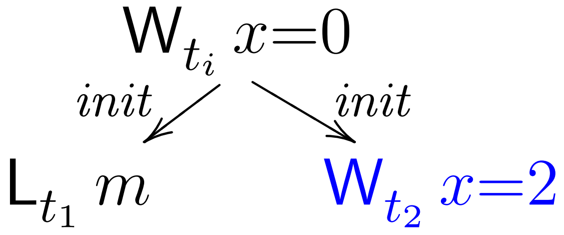
$\mathsf{W}_{t_i}\ x{=}0$

$init$

$\mathsf{L}_{t_1}\ m$

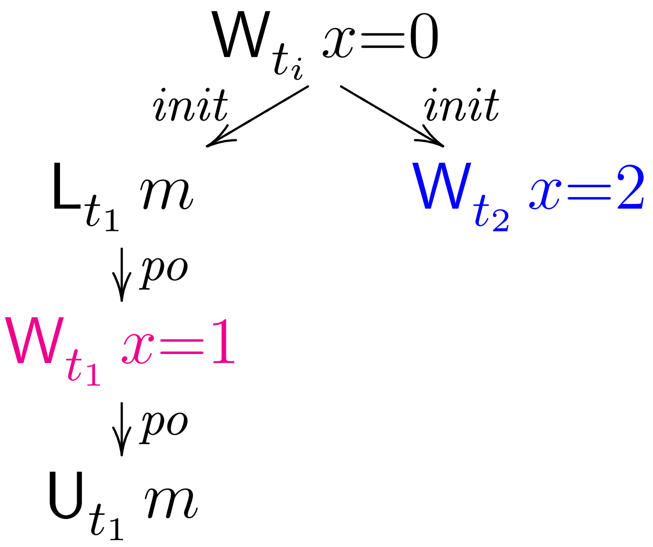
Well-behaved execution example

x = 0	
lock m	x:=2
x:=1	lock m
unlock m	r1:=x
	r2:=x
	unlock m
	print r1
	print r2



Well-behaved execution example

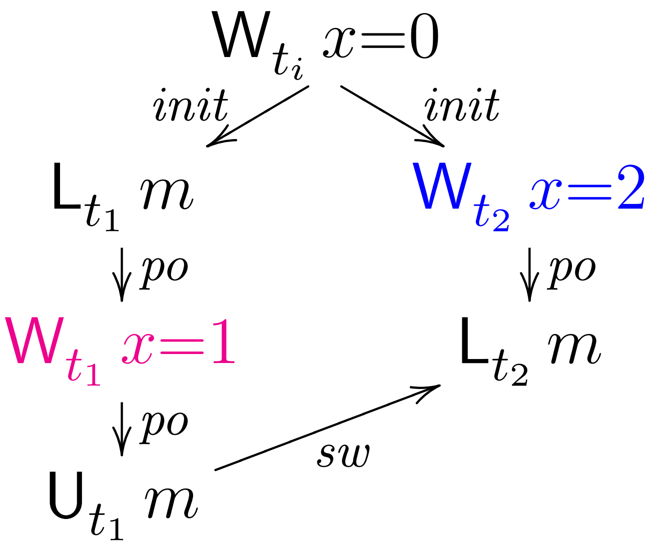
```
         x = 0
-----------------------------
 lock m      | x:=2
 x:=1        | lock m
 unlock m    | r1:=x
             | r2:=x
             | unlock m
             | print r1
             | print r2
```



$\mathsf{W}_{t_i}\ x{=}0$
$init$
$init$
$\mathsf{L}_{t_1}\ m$
$\mathsf{W}_{t_2}\ x{=}2$
po
$\mathsf{W}_{t_1}\ x{=}1$
po
$\mathsf{U}_{t_1}\ m$

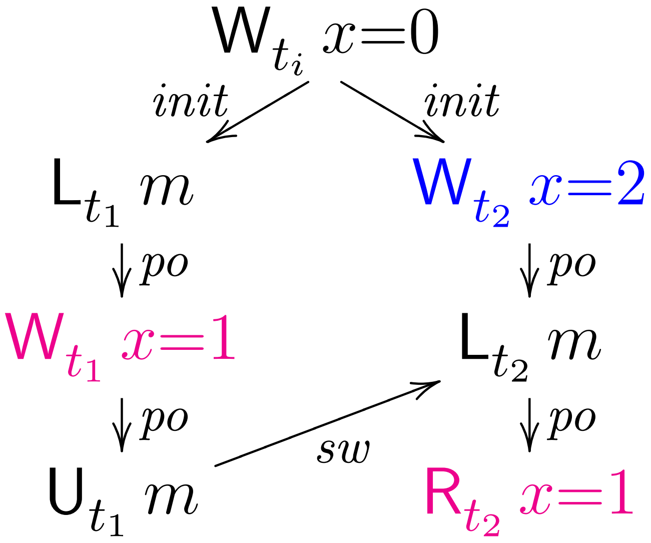
Well-behaved execution example

```
           x = 0
-----------------------------
 lock m     | x:=2
 x:=1       | lock m
 unlock m   | r1:=x
            | r2:=x
            | unlock m
            | print r1
            | print r2
```



Well-behaved execution example

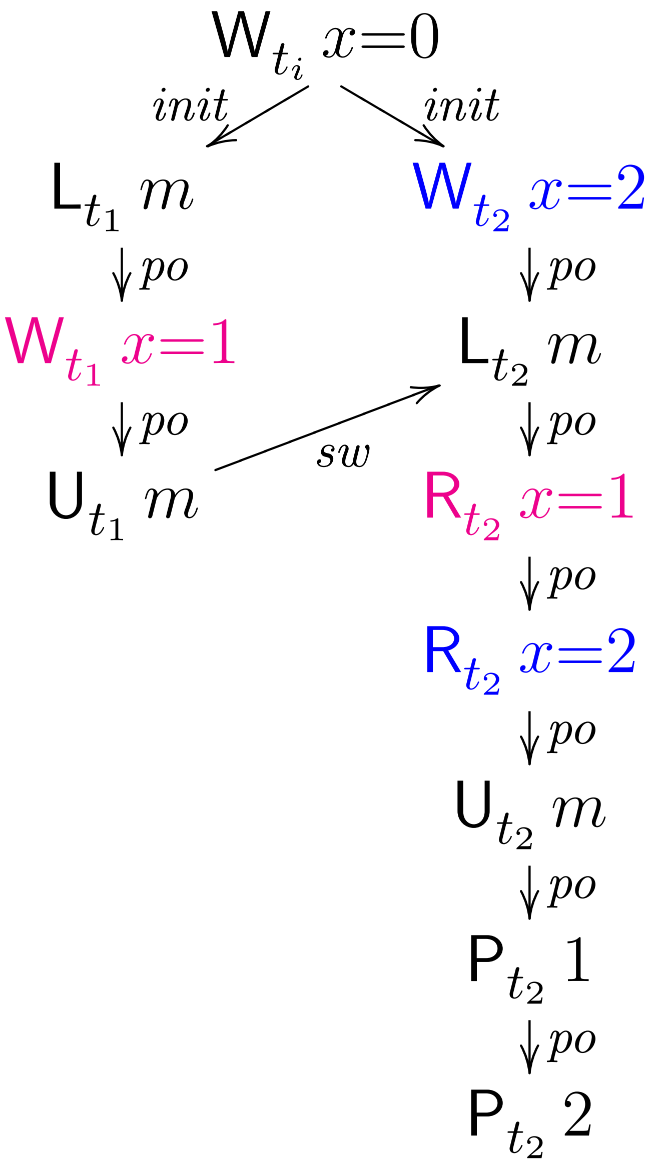
```
          x = 0
----------------------
lock m    | x:=2
x:=1      | lock m
unlock m  | r1:=x
          | r2:=x
          | unlock m
          | print r1
          | print r2
```



$\mathsf{W}_{t_i}\ x{=}0$
$init$
$init$
$\mathsf{L}_{t_1}\ m$
$\mathsf{W}_{t_2}\ x{=}2$
po
po
$\mathsf{W}_{t_1}\ x{=}1$
$\mathsf{L}_{t_2}\ m$
po
po
sw
$\mathsf{U}_{t_1}\ m$
$\mathsf{R}_{t_2}\ x{=}1$

Well-behaved execution example

```
            x = 0
---------------------------
 lock m      | x:=2
 x:=1        | lock m
 unlock m    | r1:=x
             | r2:=x
             | unlock m
             | print r1
             | print r2
```



Justification Example

```
      x = y = 0
r1 := x  | r2 := y
y  := r1 | x  := 1
```

We want to justify the result $\mathtt{r1} = \mathtt{r2} = 1$.

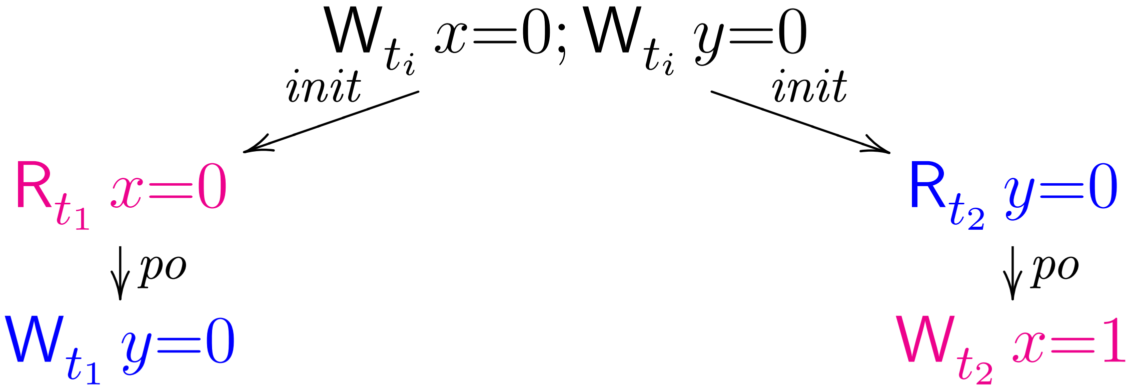
We will have to find a well-behaved execution, where each read sees a write that happens before it.

Then we commit a data race from this execution, and restart. After restarting we must use the committed races, and preserve their ordering by happens-before. The reads that are not committed will see writes that happen-before them.

Justification Example

```
        x = y = 0
---------------------------
r1 := x   |  r2 := y
y  := r1  |  x  := 1
```

The only well-behaved execution:



$\mathsf{W}_{t_i}\ x{=}0; \mathsf{W}_{t_i}\ y{=}0$

init

init

$\mathsf{R}_{t_1}\ x{=}0$

po

$\mathsf{W}_{t_1}\ y{=}0$

$\mathsf{R}_{t_2}\ y{=}0$

po

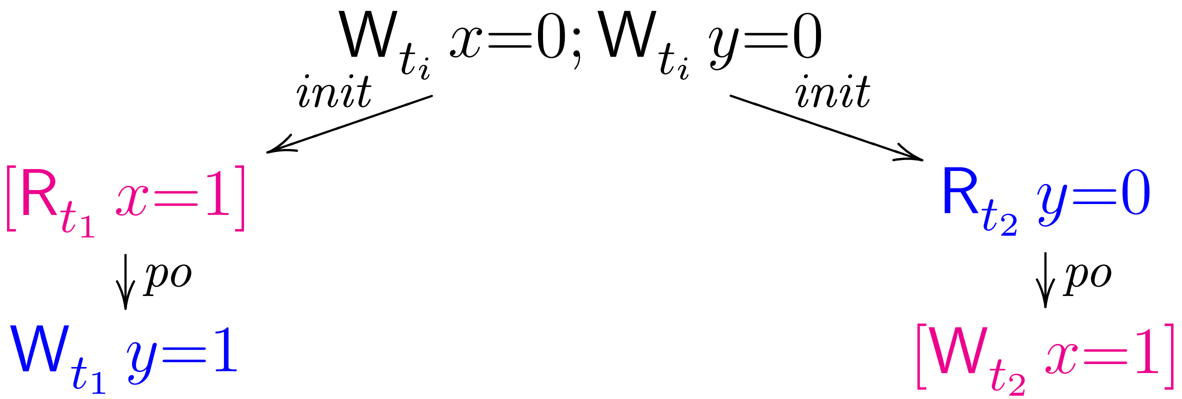
$\mathsf{W}_{t_2}\ x{=}1$

This has two data races: $\langle \mathsf{R}_{t_1}\ x{=}0, \mathsf{W}_{t_2}\ x{=}1 \rangle$ and $\langle \mathsf{R}_{t_2}\ y{=}0, \mathsf{W}_{t_1}\ y{=}0 \rangle$.

Justification Example

```
      x = y = 0
---------+---------
r1 := x  | r2 := y
y := r1  | x := 1
```

Let us commit $\langle \mathsf{R}_{t_1}\ x{=}\mathbf{1}, \mathsf{W}_{t_2}\ x{=}1 \rangle$. Now we must use the race, leaving just one possibility for execution:



$\mathsf{W}_{t_i}\ x{=}0; \mathsf{W}_{t_i}\ y{=}0$

init

init

$[\mathsf{R}_{t_1}\ x{=}1]$

po

$\mathsf{W}_{t_1}\ y{=}1$

$\mathsf{R}_{t_2}\ y{=}0$

po

$[\mathsf{W}_{t_2}\ x{=}1]$

The only available race is $\langle \mathsf{W}_{t_1}\ y{=}1, \mathsf{R}_{t_2}\ y{=}0 \rangle$.

Justification Example

```
      x = y = 0
---------------------
r1 := x  | r2 := y
y := r1  | x := 1
```

Now our commit set is:

$$\begin{tikzcd}
\mathsf{R}_{t_1}\ x{=}1 \arrow[d] & \mathsf{R}_{t_2}\ y{=}1 \arrow[d] \\
\mathsf{W}_{t_1}\ y{=}1 & \mathsf{W}_{t_2}\ x{=}1
\end{tikzcd}$$

. . . and the only execution:

$$\begin{tikzcd}
& \mathsf{W}_{t_i}\ x{=}0;\ \mathsf{W}_{t_i}\ y{=}0 \arrow[dl, "init"'] \arrow[dr, "init"] & \\
{[\mathsf{R}_{t_1}\ x{=}1]} \arrow[d, "po"] & & {[\mathsf{R}_{t_2}\ y{=}1]} \arrow[d, "po"] \\
{[\mathsf{W}_{t_1}\ y{=}1]} & & {[\mathsf{W}_{t_2}\ x{=}1]}
\end{tikzcd}$$

Justification Example

x = y = 0	
r1 := x	r2 := y
y := r1	x := 1

$$\begin{tikzcd}
 & \mathsf{W}_{t_i}\, x{=}0; \mathsf{W}_{t_i}\, y{=}0 \arrow[dl, "init"'] \arrow[dr, "init"] & \\
\mathsf{R}_{t_1}\, x{=}1 \arrow[d, "po"] & & \mathsf{R}_{t_2}\, y{=}1 \arrow[d, "po"] \\
\mathsf{W}_{t_1}\, y{=}1 & & \mathsf{W}_{t_2}\, x{=}1
\end{tikzcd}$$

means that we can get $\mathtt{r1} = \mathtt{r2} = 1$.

Bug I—reordering

Let’s take the program (Cenciarelli et al., 2007)

```
                       x = y = z = 0
--------------------------------------------------------------
 r1:=z                             | r2:=x
 if (r1==1) {x:=1; y:=1}           | r3:=y
        else {y:=1; x:=1}          | if (r2==r3==1)
                                   |   z:=1
```

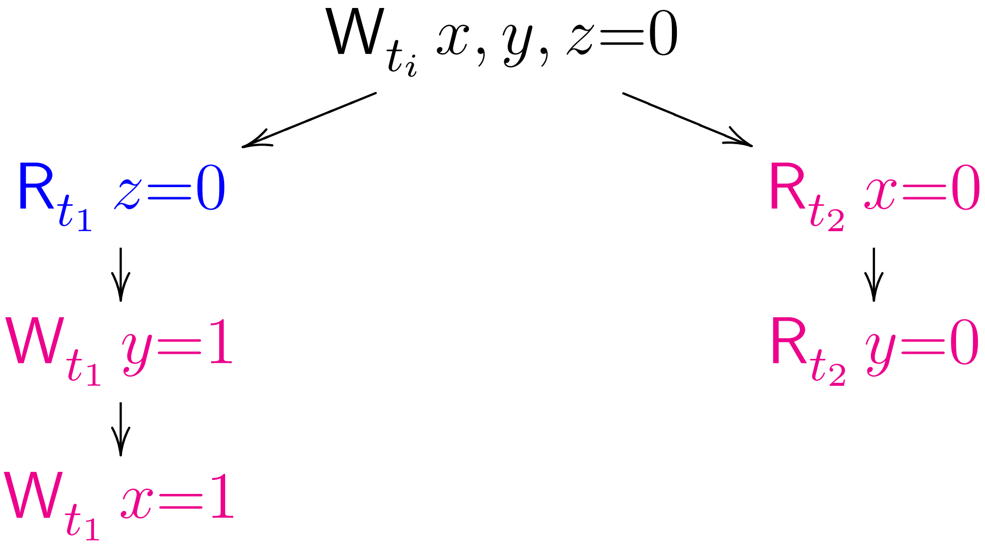
Can we get $\mathtt{r1} = \mathtt{r2} = \mathtt{r3} = 1$?

No! When we commit the races on x and y, $(\mathsf{W}_{t_1}\ y{=}1) \leq_{hb} (\mathsf{W}_{t_1}\ x{=}1)$. However, when we commit the read of 1 from z, we cannot keep the ordering of the writes.

Bug I—reordering

```
                          x = y = z = 0
--------------------------------------------------------------
r1:=z                                   | r2:=x
if (r1==1) {x:=1; y:=1}                 | r3:=y
         else {y:=1; x:=1}              | if (r2==r3==1)
                                        |   z:=1
```

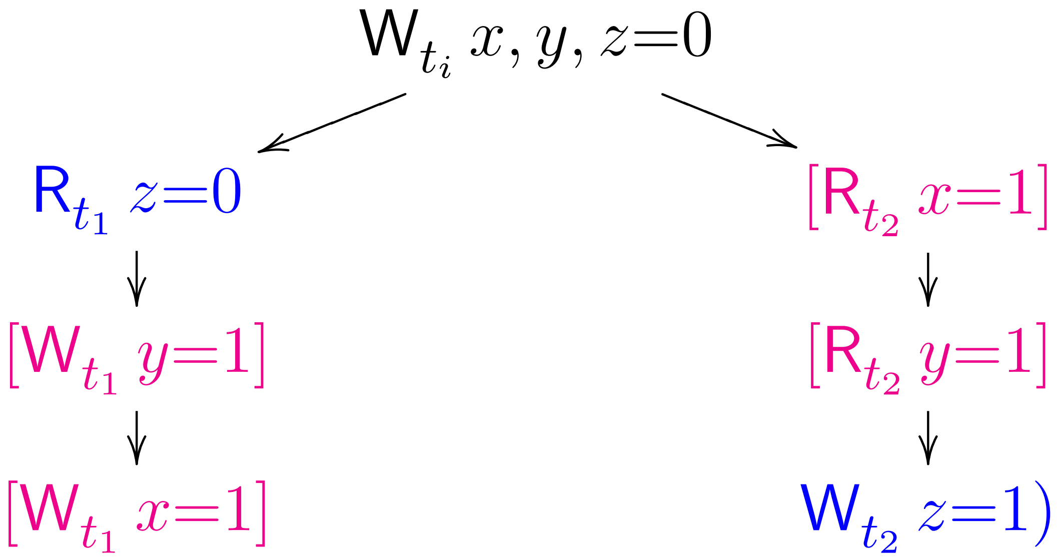
Start with a well-behaved execution:



Bug I—reordering

```
                         x = y = z = 0
--------------------------------------------------------------
r1:=z                                  | r2:=x
if (r1==1) {x:=1; y:=1}                | r3:=y
           else {y:=1; x:=1}           | if (r2==r3==1)
                                       |   z:=1
```

After committing races on x and y:



$\mathsf{W}_{t_i}\ x, y, z{=}0$
$\mathsf{R}_{t_1}\ z{=}0$
$[\mathsf{W}_{t_1}\ y{=}1]$
$[\mathsf{W}_{t_1}\ x{=}1]$
$[\mathsf{R}_{t_2}\ x{=}1]$
$[\mathsf{R}_{t_2}\ y{=}1]$
$\mathsf{W}_{t_2}\ z{=}1)$

Bug I—reordering

```
                  x = y = z = 0
------------------------------------------------------------
r1:=z                             | r2:=x
if (r1==1) {x:=1; y:=1}           | r3:=y
       else {y:=1; x:=1}          | if (r2==r3==1)
                                  |   z:=1
```

After the commits we get the commit set

$$\begin{tikzcd}
\mathsf{R}_{t_1}\ z{=}1 \arrow[d] & \mathsf{R}_{t_2}\ x{=}1 \arrow[d] \\
\mathsf{W}_{t_1}\ y{=}1 \arrow[d] & \mathsf{R}_{t_2}\ y{=}1 \arrow[d] \\
\mathsf{W}_{t_1}\ x{=}1 & \mathsf{W}_{t_2}\ z{=}1
\end{tikzcd}$$

which is impossible to honor.

Bug I—reordering

```
                      x = y = z = 0
----------------------------------------------------------
r1:=z                                 | r2:=x
if (r1==1) {x:=1; y:=1}               | r3:=y
           else {x:=1; y:=1}          | if (r2==r3==1)
                                      |   z:=1
```

However, the result is possible if we swap the assignments to x and y in one of the branches. Bug in the memory model, reordering of independent normal memory accesses should not introduce a new behaviour!

Can be fixed by relaxing the requirement on the preservation of the structure of commitments – only preserve happens-before between a read and the write it sees.

Bug II – Read After Read Elim

Reusing values from previous reads is illegal in the JMM in general:

```
          x = y = 0
-------------------------------
r1 := x | r2 := y
y := r1 | if (r2 == 1)
        | {r3 := y
        |    x := r3}
        | else x := 1
```

cannot result in $r2 = 1$.

Bug II – Read After Read Elim

Reusing values from previous reads is illegal in the JMM in general:

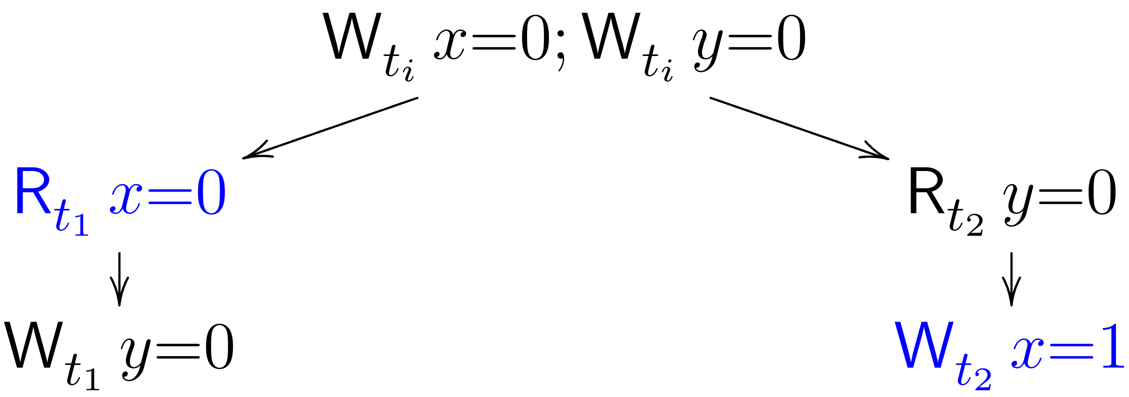
x = y = 0	
`r1 := x`	`r2 := y`
`y := r1`	`if (r2 == 1)`
	`{r3 := r2`
	`  x := r3}`
	`else x := 1`

But after replacing reusing the value of y, r2 can be 1!

Bug II – Read After Read Elim

```
                x = y = 0
─────────────────────────────────────────
r1 := x │ r2 := y
y := r1 │ if (r2 == 1)
        │    {r3 := r2; x := r3}
        │ else x := 1
```

Start with:

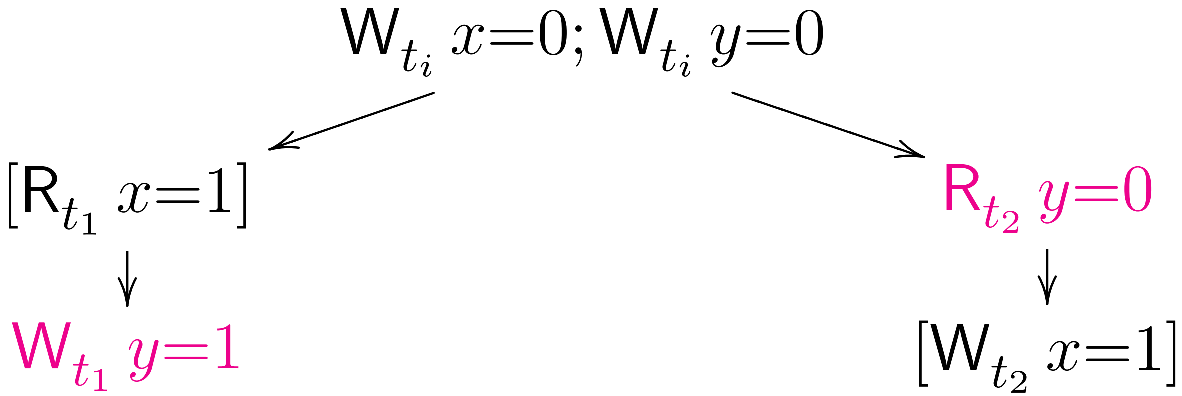

$\mathsf{W}_{t_i}\ x{=}0; \mathsf{W}_{t_i}\ y{=}0$
$\mathsf{R}_{t_1}\ x{=}0$
$\mathsf{R}_{t_2}\ y{=}0$
$\mathsf{W}_{t_1}\ y{=}0$
$\mathsf{W}_{t_2}\ x{=}1$

and then commit the data race on `x`.

Bug II – Read After Read Elim

```
              x = y = 0
--------------------------------------------
r1 := x | r2 := y
y := r1 | if (r2 == 1)
        |   {r3 := r2; x := r3}
        | else x := 1
```

After committing the race on x:



W$_{t_i}$ $x{=}0$; W$_{t_i}$ $y{=}0$

[R$_{t_1}$ $x{=}1$]

W$_{t_1}$ $y{=}1$

R$_{t_2}$ $y{=}0$

[W$_{t_2}$ $x{=}1$]

commit the data race on y.

Bug II – Read After Read Elim

```
              x = y = 0
-----------------------------------------
r1 := x  | r2 := y
y := r1  | if (r2 == 1)
         |   {r3 := r2; x := r3}
         | else x := 1
```

Finally we obtain the result:

$$\begin{tikzcd}
 & \mathsf{W}_{t_i}\, x{=}0; \mathsf{W}_{t_i}\, y{=}0 \arrow[dl] \arrow[dr] & \\
{[\mathsf{R}_{t_1}\, x{=}1]} \arrow[d] & & {[\mathsf{R}_{t_2}\, y{=}1]} \arrow[d] \\
{[\mathsf{W}_{t_1}\, y{=}1]} & & {[\mathsf{W}_{t_2}\, x{=}1]}
\end{tikzcd}$$

where $\mathtt{r2} = 1$!

Bug II – HotSpot JVM

Sun’s HotSpot JVM actually performs such an optimisation:

```
          x = y = 0                               x = y = 0
-------------------------------            --------------------------
r1=x  |  r2=y                              r1=x  |  x=1
y=r1  |  x=(r2==1)?y:1          --->       y=r1  |  r2=y
      |  print r2                                |  print r2
```

The original program cannot print “1” by the JLS.

But the optimised program can print “1” even on a sequentially consistent processor!

Bug III – write-after-read elimination

Note that the program does not have a well-behaved execution where $\mathtt{r1} \neq \mathtt{r2}$ because reads must see a most recent write in $\leq_{hb}$.

`x = 0`

`lock m`	`x:=2`
`x:=1`	`lock m`
`unlock m`	`r1:=x`
	`x:=r1`
	`r2:=x`
	`unlock m`

So it is illegal to remove the write!

Bug IV – Adding Synchronisation

One of the design goals was that increasing synchronisation should not introduce new behaviour.

By increasing synchronisation we mean:

- Moving normal accesses into synchronised blocks (roach motel).
- Making variables volatile.

However, none of these transformations are legal in the JMM in general!

Bug IV – Adding Synchronisation

initially `x = y = z = 0`

`lock m`	`lock m`	`r1:=x`	`r3:=y`
`x:=2`	`x:=1`	`lock m`	`z:=r3`
`unlock m`	`unlock m`	`r2:=z`	
		`if(r1==2)`	
		`  y:=1`	
		`else`	
		`  y:=r2`	
		`unlock m`	

Behaviour $\mathtt{r1} = \mathtt{r2} = \mathtt{r3} = 1$ not possible.

Bug IV – Adding Synchronisation

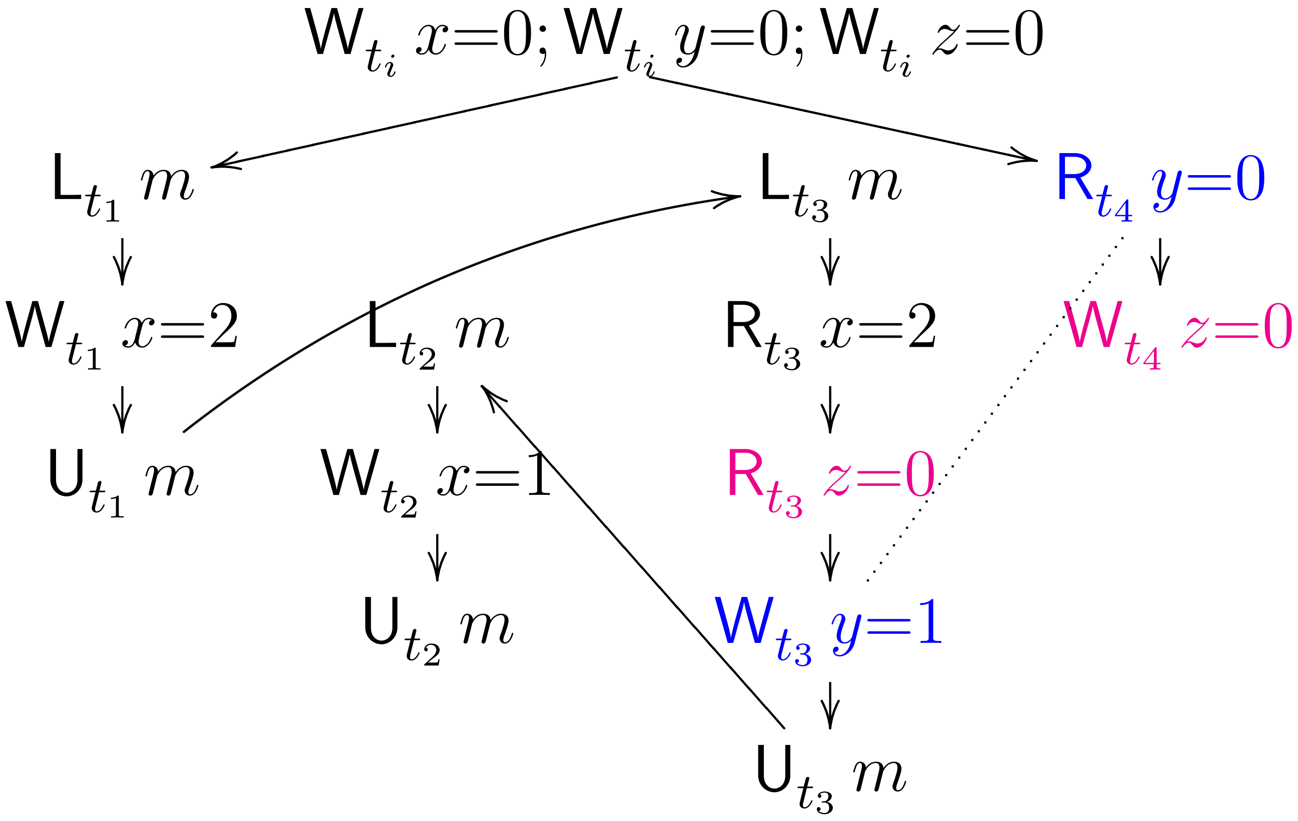
initially `x = y = z = 0`

`lock m`	`lock m`	`lock m`	`r3:=y`
`x:=2`	`x:=1`	`r1:=x`	`z:=r3`
`unlock m`	`unlock m`	`r2:=z`	
		`if(r1==2)`	
		`   y:=1`	
		`else`	
		`   y:=r2`	
		`unlock m`	

. . . but becomes possible after moving `r1:=x` inside the synchronised block. Let’s start by committing the data race on `y`, and then on `z` with value 1.

Bug IV – Adding Synchronisation

Why is $\texttt{r1} = \texttt{r2} = \texttt{r3} = 1$ possible?



$\mathsf{W}_{t_i}\ x{=}0; \mathsf{W}_{t_i}\ y{=}0; \mathsf{W}_{t_i}\ z{=}0$

$\mathsf{L}_{t_1}\ m$ $\mathsf{L}_{t_3}\ m$ $\mathsf{R}_{t_4}\ y{=}0$

$\mathsf{W}_{t_1}\ x{=}2$ $\mathsf{L}_{t_2}\ m$ $\mathsf{R}_{t_3}\ x{=}2$ $\mathsf{W}_{t_4}\ z{=}0$

$\mathsf{U}_{t_1}\ m$ $\mathsf{W}_{t_2}\ x{=}1$ $\mathsf{R}_{t_3}\ z{=}0$

$\mathsf{U}_{t_2}\ m$ $\mathsf{W}_{t_3}\ y{=}1$

$\mathsf{U}_{t_3}\ m$

Committing the data race on $\texttt{y}$ and then on $\texttt{z}$ (with value 1).

Bug IV – Adding Synchronisation

initially `x = y = z = 0`

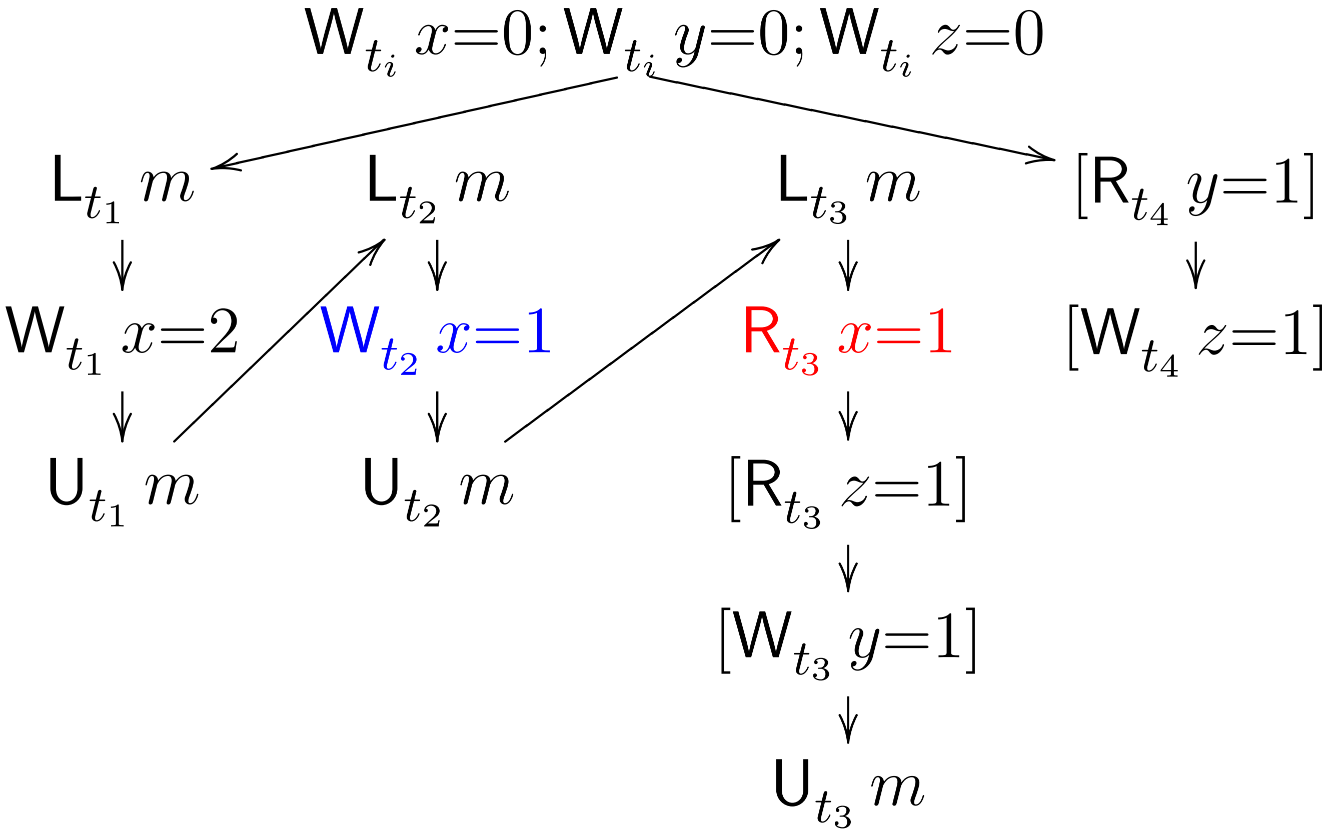
`lock m`
`x:=2`
`unlock m`
	`lock m`
	`x:=1`
	`unlock m`
		`lock m`
		`r1:=x`
		`r2:=z`
		`if(r1==2)`
		`  y:=1`
		`else`
		`  y:=r2`
		`unlock m`
			`r3:=y`
			`z:=r3`

Finally switch to the other branch of the `if` statement. . .

Note: this switch is impossible if `r1:=x` is before the lock, because `x` would have to be committed with 2.

Bug IV – Adding Synchronisation

Why is $\texttt{r1} = \texttt{r2} = \texttt{r3} = 1$ possible?



$\mathsf{W}_{t_i}\ x{=}0;\ \mathsf{W}_{t_i}\ y{=}0;\ \mathsf{W}_{t_i}\ z{=}0$

$\mathsf{L}_{t_1}\ m$

$\mathsf{W}_{t_1}\ x{=}2$

$\mathsf{U}_{t_1}\ m$

$\mathsf{L}_{t_2}\ m$

$\mathsf{W}_{t_2}\ x{=}1$

$\mathsf{U}_{t_2}\ m$

$\mathsf{L}_{t_3}\ m$

$\mathsf{R}_{t_3}\ x{=}1$

$[\mathsf{R}_{t_3}\ z{=}1]$

$[\mathsf{W}_{t_3}\ y{=}1]$

$\mathsf{U}_{t_3}\ m$

$[\mathsf{R}_{t_4}\ y{=}1]$

$[\mathsf{W}_{t_4}\ z{=}1]$

…by changing the synchronisation order, so that the read of $\texttt{x}$ sees the write of 1.

Proving Legality

Proving legality of a compiler optimisation is relatively straightforward:

- Take a justifying sequence of the transformed program. . .
- . . . and massage it into a justifying sequence of the original program.

Legality of hardware optimisations is straightforward if there is an order that we can use to commit the actions:

- For Sun TSO and Intel Itanium, the order is given directly by the processor specification.
- For Power and ARM: ???

Summary

The Java Memory Model:

- is the semantics of multi-threaded Java.
- satisfies most of its design goals. . .
- . . . but not the most important one:
  - it does not allow several standard optimisations, and it is not implemented by the reference JVM. (that does not mean that it is not implementable)
- many questions are still open.

We are looking for a new Java memory model (or a fix for the current one).

Questions?