Programming language memory models:
Problems, Solutions, and Directions

Anton Podkopaev
anton@podkopaev.net

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

JET BRAINS RESEARCH

HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Anton Podkopaev

Researcher @ JetBrains Research
Postdoc @ MPI-SWS
Docent @ HSE

Programming languages
Weak memory concurrency
Compilation correctness
Functional programming

~~Software~~ Proof Engineer (Coq)

Programming language memory models:
Problems, Solutions, and Directions

Programming language **memory models**:
Problems, Solutions, and Directions

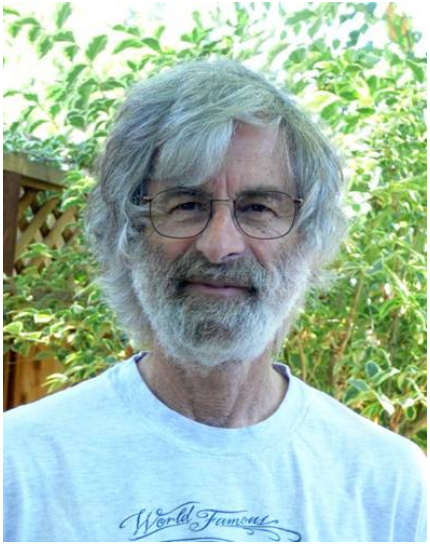
Programming language **memory models**:
Problems, Solutions, and Directions

Memory model defines behaviors
of concurrent system

Programming language **memory models**:
Problems, Solutions, and Directions

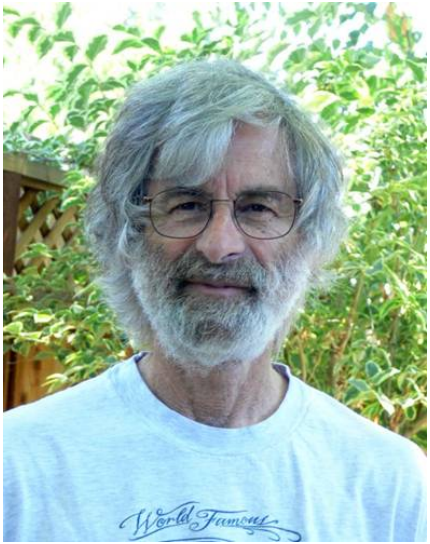
Memory model defines behaviors
of concurrent system

Doesn't there exist **The Memory Model**?



How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT



Sequential **C**onsistency:
system's behavior —
interleaving of threads

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

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

$[x] := 1;$

$a := [y];$

$[y] := 1;$

$b := [x];$

$$[x] := 0; [y] := 0;$$

$$\begin{array}{l} [x] := 1; \\ \\ a := [y]; \end{array} \quad \Bigg\| \quad \begin{array}{l} [y] := 1; \\ \\ b := [x]; \end{array}$$

$$\frac{\begin{array}{l} [x] := 1; \\ a := [y]; \\ [y] := 1; \\ b := [x]; \end{array}}{a = 0; b = 1}$$

$$[x] := 0; [y] := 0;$$

$$\begin{array}{l} [x] := 1; \\ \\ a := [y]; \end{array} \quad \Bigg\| \quad \begin{array}{l} [y] := 1; \\ \\ b := [x]; \end{array}$$

$$\begin{array}{l|l} [x] := 1; & [y] := 1; \\ a := [y]; & b := [x]; \\ [y] := 1; & [x] := 1; \\ b := [x]; & a := [y]; \end{array}$$

$a = 0; b = 1$

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

$$
\begin{array}{c|c}
[x] := 1; & [y] := 1; \\
a := [y]; & b := [x]; \\
[y] := 1; & [x] := 1; \\
b := [x]; & a := [y]; \\
\hline
a = 0; b = 1 & a = 1; b = 0
\end{array}
$$

$$[x] := 0; [y] := 0;$$

$$\begin{array}{l || l}
[x] := 1; & [y] := 1; \\
a := [y]; & b := [x];
\end{array}$$

$[x] := 1;$	$[y] := 1;$	$[x] := 1;$	$[x] := 1;$	$[y] := 1;$	$[y] := 1;$
$a := [y];$	$b := [x];$	$[y] := 1;$	$[y] := 1;$	$[x] := 1;$	$[x] := 1;$
$[y] := 1;$	$[x] := 1;$	$b := [x];$	$a := [y];$	$b := [x];$	$a := [y];$
$b := [x];$	$a := [y];$	$a := [y];$	$b := [x];$	$a := [y];$	$b := [x];$
$a = 0; b = 1$	$a = 1; b = 0$	$a = 1; b = 1$			

$$[x] := 0; [y] := 0;$$

$$\begin{array}{l} [x] := 1; \\ \\ a := [y]; \end{array} \quad \Big\| \quad \begin{array}{l} [y] := 1; \\ \\ b := [x]; \end{array}$$

$[x] := 1;$	$[y] := 1;$	$[x] := 1;$	$[x] := 1;$	$[y] := 1;$	$[y] := 1;$
$a := [y];$	$b := [x];$	$[y] := 1;$	$[y] := 1;$	$[x] := 1;$	$[x] := 1;$
$[y] := 1;$	$[x] := 1;$	$b := [x];$	$a := [y];$	$b := [x];$	$a := [y];$
$b := [x];$	$a := [y];$	$a := [y];$	$b := [x];$	$a := [y];$	$b := [x];$
$a = 0; b = 1$	$a = 1; b = 0$	$a = 1; b = 1$			

SC disallows $a = 0; b = 0$

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

$a = 0; b = 1$

$a = 1; b = 0$

$a = 1; b = 1$

$a = 0; b = 0$

$$[x] := 0; [y] := 0;$$

$[x] := 1;$	$[y] := 1;$
$a := [y];$	$b := [x];$
if $a = 0$	**if** $b = 0$
critical section	*critical section*

$a = 0; b = 1$

$a = 1; b = 0$

$a = 1; b = 1$

$a = 0; b = 0$

Dekker's lock

$[x] := 0; [y] := 0;$

$[x] := 1;$	$[y] := 1;$
$a := [y];$	$b := [x];$
if $a = 0$	**if** $b = 0$
critical section	*critical section*

$a = 0; b = 1$

$a = 1; b = 0$

$a = 1; b = 1$

$a = 0; b = 0$

Dekker's lock

$[x] := 0; [y] := 0;$

$[x] := 1;$	$[y] := 1;$
$a := [y];$	$b := [x];$
if $a = 0$	**if** $b = 0$
critical section	*critical section*

$a = 0; b = 1$

$a = 1; b = 0$

$a = 1; b = 1$

$a = 0; b = 0$

Does **not** work on GCC+x86!

Dekker's lock

$[x] := 0; [y] := 0;$

$[x] := 1;$	$[y] := 1;$
$a := [y];$	$b := [x];$
if $a = 0$	**if** $b = 0$
critical section	*critical section*

$a = 0; b = 1$

$a = 1; b = 0$

$a = 1; b = 1$

$a = 0; b = 0$

Does **not** work on GCC+x86!

1. GCC may reorder instructions
2. x86 buffers writes

Dekker's lock

$[x] := 0; [y] := 0;$

$[x] := 1;$	$[y] := 1;$
mfence;	**mfence**;
$a := [y];$	$b := [x];$
if $a = 0$	**if** $b = 0$
critical section	*critical section*

$a = 0; b = 1$

$a = 1; b = 0$

$a = 1; b = 1$

$a = 0; b = 0$

Does **not** work on GCC+x86!

1. GCC may reorder instructions
2. x86 buffers writes

Dekker's lock

$[x] := 0; [y] := 0;$

$[x] := 1;$	$[y] := 1;$
mfence;	**mfence**;
$a := [y];$	$b := [x];$
if $a = 0$	**if** $b = 0$
critical section	*critical section*

$a = 0; b = 1$

$a = 1; b = 0$

$a = 1; b = 1$

$a = 0; b = 0$

Works on GCC+x86!

Non-SC behaviors called **weak**

Weak Memory **M**odels allow weak behaviors

Real systems have weak MMs

(x86, Power, ARM, RISC-V, C/C++, Java)

Requirements to (Weak) Memory Models

Hardware MMs should [x86, Power, ARM, RISC-V]

Programming languages' MMs should [C/C++, Java, JS, Wasm, OCaml]

Requirements to (Weak) Memory Models

Hardware MMs should [x86, Power, ARM, RISC-V]

1. describe real CPUs
2. save room for future optimizations
3. provide reasonable guarantees for PLs

Programming languages' MMs should [C/C++, Java, JS, Wasm, OCaml]

Requirements to (Weak) Memory Models

Hardware MMs should [x86, Power, ARM, RISC-V]

1. describe real CPUs
2. save room for future optimizations
3. provide reasonable guarantees for PLs

Programming languages' MMs should [C/C++, Java, JS, Wasm, OCaml]

1. support compiler optimizations
2. provide efficient compilation to hardware
3. have easy non-expert mode

Requirements to (Weak) Memory Models

Hardware MMs should [x86, Power, ARM, RISC-V]

1. describe real CPUs
2. save room for future optimizations
3. provide reasonable guarantees for PLs

Programming languages' MMs should [C/C++, Java, JS, Wasm, OCaml]

1. support compiler optimizations
2. provide efficient compilation to hardware
3. have easy non-expert mode

1. Compiler optimizations

Source

$$\begin{array}{rcl} [x] & := & 1; \\ a & := & [y]; \end{array} \quad \Bigg\| \quad \begin{array}{rcl} [y] & := & 1; \\ b & := & [x]; \end{array}$$

1. Compiler optimizations

Source

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \quad \Bigg\| \quad \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$

1. Compiler optimizations

Source

$$\begin{array}{l||l} [x] := 1; & [y] := 1; \\ a := [y]; & b := [x]; \end{array}$$

Optimized

$$\begin{array}{l||l} a := [y]; & [y] := 1; \\ [x] := 1; & b := [x]; \end{array}$$

1. Compiler optimizations

Source

$$\left[\!\left[\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \;\middle\|\; \begin{array}{l} [y] := 1; \\ b := [x]; \end{array} \right]\!\right]$$

$$\sqsupseteq$$

Optimized

$$\left[\!\left[\begin{array}{l} a := [y]; \\ [x] := 1; \end{array} \;\middle\|\; \begin{array}{l} [y] := 1; \\ b := [x]; \end{array} \right]\!\right]$$

2. Efficient compilation to hardware

Source MM (SC)

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \;\Bigg\|\; \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$

$$\downarrow$$

Target MM (x86)

$$\begin{array}{l} [x] := 1; \\ \texttt{mfence}; \\ a := [y]; \end{array} \;\Bigg\|\; \begin{array}{l} [y] := 1; \\ \texttt{mfence}; \\ b := [x]; \end{array}$$

2. Efficient compilation to hardware

Source MM (SC)

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \quad \Bigg\| \quad \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$

No compilation scheme w/o fences

$$\downarrow$$

Target MM (x86)

$$\begin{array}{l} [x] := 1; \\ \texttt{mfence}; \\ a := [y]; \end{array} \quad \Bigg\| \quad \begin{array}{l} [y] := 1; \\ \texttt{mfence}; \\ b := [x]; \end{array}$$

3. Easy non-expert mode

Nice program $\Rightarrow$ nice behaviors

3. Easy non-expert mode

No data races $\Rightarrow$ only SC behaviors

3. Easy non-expert mode

No data races in SC executions $\Rightarrow$ only SC behaviors

3. Easy non-expert mode

Data-**R**ace-**F**reedom guarantee:

No data races in SC executions $\Rightarrow$ only SC behaviors

3. Easy non-expert mode

Data-**R**ace-**F**reedom guarantee:

No data races in SC executions $\Rightarrow$ only SC behaviors

$$
\begin{array}{l}
a := [x]; \\
\texttt{if } a \texttt{ then} \\
\quad [y] := 1
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
b := [y]; \\
\texttt{if } b \texttt{ then} \\
\quad [x] := 1
\end{array}
$$

3. Easy non-expert mode

Data-**R**ace-**F**reedom guarantee:

No data races in SC executions $\Rightarrow$ only SC behaviors

$$
\begin{array}{l}
a := [x]; \\
\texttt{if } a \texttt{ then} \\
\quad [y] := 1
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
b := [y]; \\
\texttt{if } b \texttt{ then} \\
\quad [x] := 1
\end{array}
$$

C/C++ MM allows to get $a = b = 1$

3. Easy non-expert mode

Data-**R**ace-**F**reedom guarantee:

No data races in SC executions $\Rightarrow$ only SC behaviors

$$
\begin{array}{l}
a := [x]; \\
\texttt{if } a \texttt{ then} \\
\quad [y] := 1
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
b := [y]; \\
\texttt{if } b \texttt{ then} \\
\quad [x] := 1
\end{array}
$$

C/C++ MM allows to get $a = b = 1$

$a = b = 1$ is **O**ut-**O**f-**T**hin-**A**ir outcome

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]			
Java MM	[Manson et al., 2005]			
C/C++ MM	[Batty et al., 2011]			

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	:(		
Java MM	[Manson et al., 2005]			
C/C++ MM	[Batty et al., 2011]			

	SC
Trace-preserving transformations	✓
Reordering normal memory accesses	✗
Redundant read after read elimination	✓
Redundant read after write elimination	✓
Irrelevant read elimination	✓
Irrelevant read introduction	✓
Redundant write before write elimination	✓
Redundant write after read elimination	✓
External action reordering	✗

SC-preserving optimizations in LLVM [Marino et al., 2011]

Average slowdown:

- **34%** w/ only SC preserving optimizations
- **5.5%** w/ optimizations modified to preserve SC

SC-preserving optimizations in LLVM [Marino et al., 2011]

Average slowdown:

- **34%** w/ only SC preserving optimizations
- **5.5%** w/ optimizations modified to preserve SC

Drawbacks:

- Hardware still allows weak behaviors, i.e., <u>no end-to-end SC</u>
- Requires modifying existing compilers

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐		
Java MM	[Manson et al., 2005]	☹		
C/C++ MM	[Batty et al., 2011]			

Validity of transformations [Ševčík and Aspinall, 2008]

	SC	JMM*
Trace-preserving transformations	✓	✓
Reordering normal memory accesses	✗	✓*
Redundant read after read elimination	✓	✗
Redundant read after write elimination	✓	✓
Irrelevant read elimination	✓	✓
Irrelevant read introduction	✓	✗
Redundant write before write elimination	✓	✓
Redundant write after read elimination	✓	✗
External action reordering	✗	✗

Validity of transformations [Ševčík and Aspinall, 2008]

	SC	JMM*
Trace-preserving transformations	✓	✓
Reordering normal memory accesses	✗	✓*
Redundant read after read elimination	✓	✗
Redundant read after write elimination	✓	✓
Irrelevant read elimination	✓	✓
Irrelevant read introduction	✓	✗
Redundant write before write elimination	✓	✓
Redundant write after read elimination	✓	✗
External action reordering	✗	✗

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐		
Java MM	[Manson et al., 2005]	☹		
C/C++ MM	[Batty et al., 2011]			

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]			
Java MM	[Manson et al., 2005]			
C/C++ MM	[Batty et al., 2011]			

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	☹	
Java MM	[Manson et al., 2005]	☹		
C/C++ MM	[Batty et al., 2011]	🙂		

End-to-end SC via Volatile JVM [Liu et al., 2017, Liu et al., 2019]

Java MM guarantees **D**ata-**R**ace-**F**reedom:

Shared locations are volatile (no data races) $\Rightarrow$ SC semantics

End-to-end SC via Volatile JVM [Liu et al., 2017, Liu et al., 2019]

		Benchmarks	
	Slowdown, in %	DaCapo	spark-perf
x86	Average		
	Max		
ARM (1)	Average		
	Max		
ARM (2)	Average		
	Max		

End-to-end SC via Volatile JVM [Liu et al., 2017, Liu et al., 2019]

		Benchmarks	
	Slowdown, in %	DaCapo	spark-perf
x86	Average	28	79
	Max	81	164
ARM (1)	Average		
	Max		
ARM (2)	Average		
	Max		

End-to-end SC via Volatile JVM [Liu et al., 2017, Liu et al., 2019]

		Benchmarks	
	Slowdown, in %	DaCapo	spark-perf
x86	Average	28	79
	Max	81	164
ARM (1)	Average	57	85
	Max	157	∞
ARM (2)	Average		
	Max		

End-to-end SC via Volatile JVM [Liu et al., 2017, Liu et al., 2019]

		Benchmarks	
	Slowdown, in %	DaCapo	spark-perf
x86	Average	28	79
	Max	81	164
ARM (1)	Average	57	85
	Max	157	∞
ARM (2)	Average	73	125
	Max	103	∞

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	☹	
Java MM	[Manson et al., 2005]	☹		
C/C++ MM	[Batty et al., 2011]	🙂		

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	😡	
Java MM	[Manson et al., 2005]	☹️	😐	
C/C++ MM	[Batty et al., 2011]	😊	😊	

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	😡	
Java MM	[Manson et al., 2005]	☹	😐	
C/C++ MM	[Batty et al., 2011]	😊	😊	

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	😡	😊
Java MM	[Manson et al., 2005]	☹️	😐	😊
C/C++ MM	[Batty et al., 2011]	😊	😊	

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	😡	🙂
Java MM	[Manson et al., 2005]	☹️	😐	🙂
C/C++ MM	[Batty et al., 2011]	🙂	🙂	$\boxed{☹️}$

C/C++ MM allows to get $a = b = 1$, OOTA

$$
\begin{array}{l}
a := [x]; \\
\textbf{if } a \textbf{ then} \\
\quad [y] := 1
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
b := [y]; \\
\textbf{if } b \textbf{ then} \\
\quad [x] := 1
\end{array}
$$

Executions in C/C++ MM

$$
\begin{array}{l}
a := [x]; \\
[y] := 1
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
b := [y]; \\
\texttt{if } b \texttt{ then} \\
\quad [x] := 1
\end{array}
$$

Executions in C/C++ MM

$$
\begin{array}{rcl}
a & := & [x]; \\
[y] & := & 1
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
b := [y]; \\
\textbf{if } b \textbf{ then} \\
\quad [x] := 1
\end{array}
$$

$$
\left\{ \begin{array}{ccc}
\mathtt{R}x0 & & \mathtt{R}y0 \\
\mathtt{po} \downarrow & & \\
\mathtt{W}y1 & &
\end{array} \right\}
$$

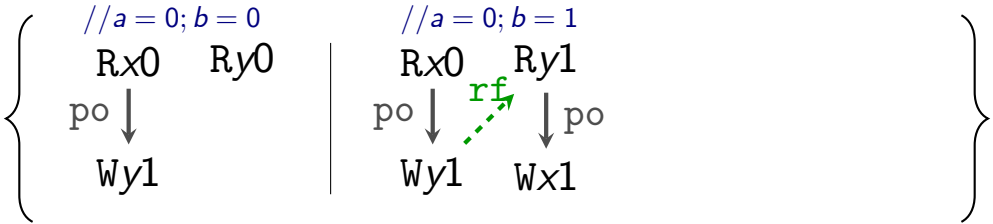
Executions in C/C++ MM

$$
\begin{array}{rcl||l}
a & := & [x]; & b := [y]; \\
[y] & := & 1 & \textbf{if } b \textbf{ then} \\
 & & & \quad [x] := 1
\end{array}
$$

$$
\left\{
\begin{array}{l}
//a = 0; b = 0 \\
\mathtt{R}x0 \quad \mathtt{R}y0 \\
\mathtt{po} \downarrow \\
\mathtt{W}y1
\end{array}
\right\}
$$

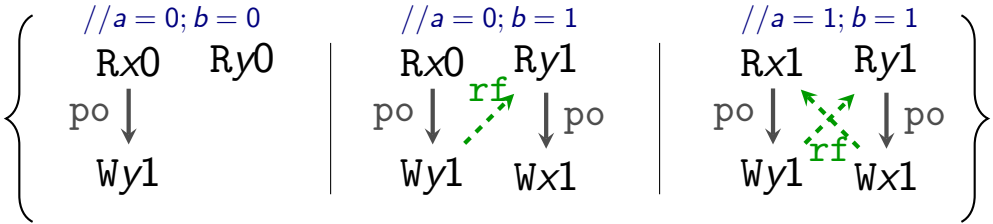
Executions in C/C++ MM

$$
\begin{array}{l}
a := [x]; \\
[y] := 1
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
b := [y]; \\
\texttt{if } b \texttt{ then} \\
\quad [x] := 1
\end{array}
$$



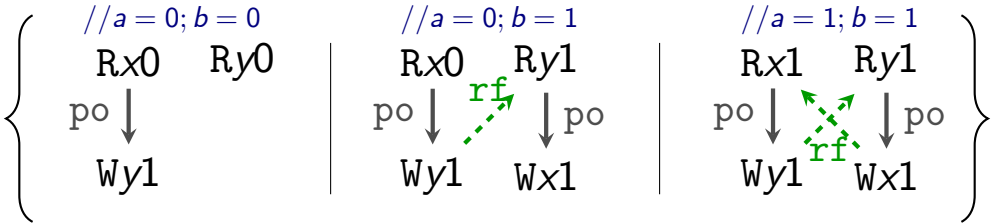
Executions in C/C++ MM

$$
\begin{array}{l}
a := [x]; \\
[y] := 1
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
b := [y]; \\
\texttt{if } b \texttt{ then} \\
\quad [x] := 1
\end{array}
$$



Executions in C/C++ MM

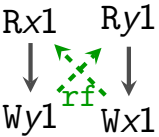
$$
\begin{array}{l||l}
a := [x]; & b := [y]; \\
[y] := 1 & \texttt{if } b \texttt{ then} \\
 & \quad [x] := 1
\end{array}
$$



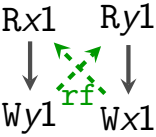
Axioms:
1. $\mathtt{po} \cup \mathtt{rf}_{\text{preserved}}$ is acyclic ($\mathtt{rf}_{\text{preserved}} \subseteq \mathtt{rf}$)
2. ...

Out-Of-Thin-Air in C/C++ MM

$a := [x];$ ‖ $b := [y];$
$[y] := 1$ ‖ **if** b **then**
‖ $[x] := 1$

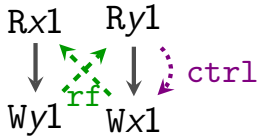


$a := [x];$ ‖ $b := [y];$
if a **then** ‖ **if** b **then**
$[y] := 1$ ‖ $[x] := 1$

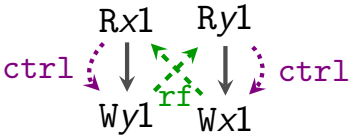


Out-Of-Thin-Air in C/C++ MM

```
a := [x];          b := [y];
[y] := 1           if b then
                      [x] := 1
```

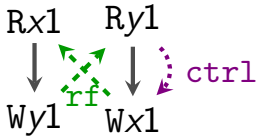



```
a := [x];          b := [y];
if a then          if b then
   [y] := 1           [x] := 1
```

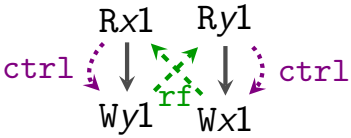


Out-Of-Thin-Air in C/C++ MM

```
a := [x];          ‖  b := [y];
[y] := 1           ‖  if b then
                   ‖     [x] := 1
```



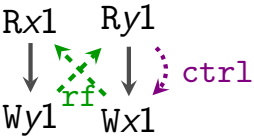

```
a := [x];          ‖  b := [y];
if a then          ‖  if b then
   [y] := 1        ‖     [x] := 1
```



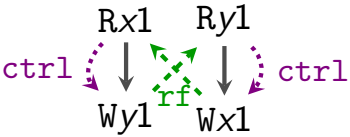

```
a := [x];          ‖  b := [y];
if a then          ‖  if b then
   [y] := 1        ‖     [x] := 1
else               ‖
   [y] := 1        ‖
```

Out-Of-Thin-Air in C/C++ MM

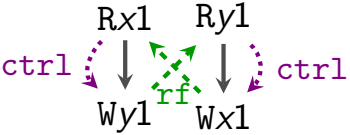
```
a := [x];          ‖  b := [y];
[y] := 1           ‖  if b then
                   ‖     [x] := 1
```




```
a := [x];          ‖  b := [y];
if a then          ‖  if b then
   [y] := 1        ‖     [x] := 1
```

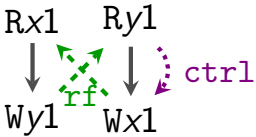



```
a := [x];          ‖  b := [y];
if a then          ‖  if b then
   [y] := 1        ‖     [x] := 1
else               ‖
   [y] := 1        ‖
```

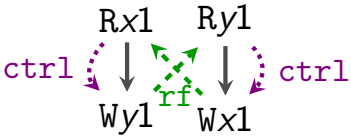


Out-Of-Thin-Air in C/C++ MM

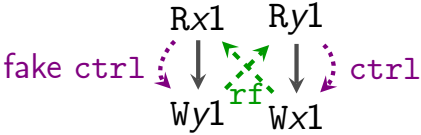
```
a := [x];          ‖  b := [y];
[y] := 1           ‖  if b then
                   ‖     [x] := 1
```




```
a := [x];          ‖  b := [y];
if a then          ‖  if b then
   [y] := 1        ‖     [x] := 1
```




```
a := [x];          ‖  b := [y];
if a then          ‖  if b then
   [y] := 1        ‖     [x] := 1
else               ‖
   [y] := 1        ‖
```



Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	😡	😊
Java MM	[Manson et al., 2005]	☹️	😐	😊
C/C++ MM	[Batty et al., 2011]	😊	😊	☹️

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	☹	🙂
Java MM	[Manson et al., 2005]	☹	😐	🙂
C/C++ MM	[Batty et al., 2011]	🙂	🙂	☹
RC11	[Lahav et al., 2017]	😐	😐	🙂

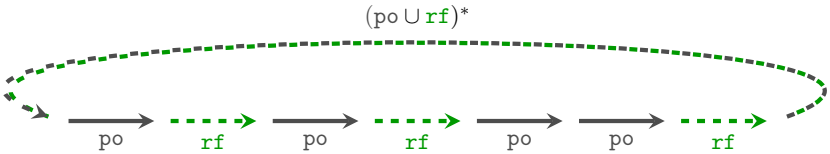
Forbids all $\mathtt{po} \cup \mathtt{rf}$ cycles

Forbidding $\mathtt{po} \cup \mathtt{rf}$ cycles

Enough to respect $[\mathtt{R}] \,;\, \mathtt{po} \,;\, [\mathtt{W}]$

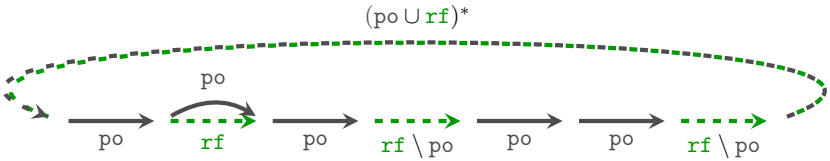
Forbidding po ∪ rf cycles

Enough to respect [R] ; po ; [W]



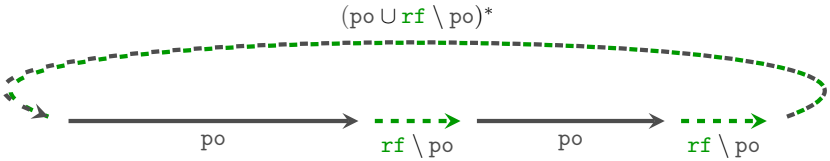
Forbidding $\mathtt{po} \cup \mathtt{rf}$ cycles

Enough to respect $[\mathtt{R}] ; \mathtt{po} ; [\mathtt{W}]$



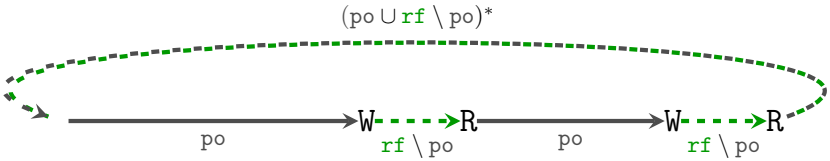
Forbidding $\mathtt{po} \cup \mathtt{rf}$ cycles

Enough to respect $[\mathtt{R}] ; \mathtt{po} ; [\mathtt{W}]$



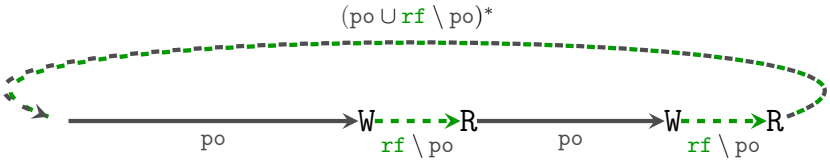
Forbidding po ∪ rf cycles

Enough to respect [R] ; po ; [W]



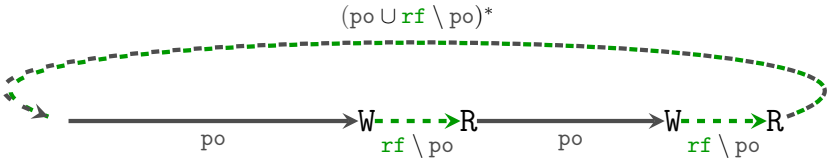
Forbidding $\mathtt{po} \cup \mathtt{rf}$ cycles

Enough to respect $[\mathtt{R}] ; \mathtt{po} ; [\mathtt{W}]$ since hardware respects $\mathtt{rf} \setminus \mathtt{po}$



Forbidding po ∪ rf cycles

Enough to respect [R] ; po ; [W] since hardware respects rf \ po

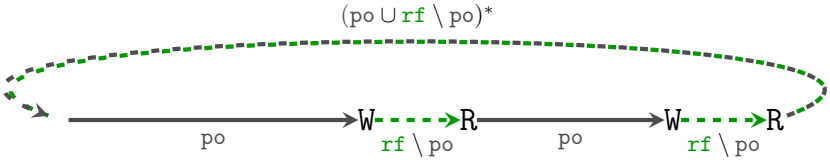


How?

1. Restrict compiler optimizations
2. Put a fence between R and W

Forbidding po ∪ rf cycles

Enough to respect [R] ; po ; [W] since hardware respects rf \ po



How?

1. Restrict compiler optimizations
2. Put a fence between R and W

Cheaper for C/C++ than for Java!

C/C++ has *undefined behavior*

Undefined Behavior and Memory Models

$$
\begin{array}{l}
[data] := 42; \\
[f] := 1;
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
\texttt{while}\ ([f] == 0)\ \{\}; \\
\texttt{print}([data]);
\end{array}
$$

Undefined Behavior and Memory Models

$$\texttt{int}\ data = 0; \qquad\qquad \texttt{int}\ \ f = 0;$$

$$\begin{array}{l} [data] := 42; \\ [f] \quad := 1; \end{array} \;\Bigg\|\; \begin{array}{l} \texttt{while}\ ([f] \quad == 0)\ \{\}; \\ \texttt{print}([data]); \end{array}$$

Undefined Behavior and Memory Models

$$\texttt{int}\ data = 0; \qquad\qquad \texttt{int}\ \ f = 0;$$

$$\begin{array}{l|l} [data] := 42; & \texttt{while}\ ([f] \quad == 0)\ \{\}; \\ [f] \quad := 1; & \texttt{print}([data]); \end{array}$$

Java: Fine, but may print 0

C/C++: Undefined Behavior! Race on normal location!

Undefined Behavior and Memory Models

$$\texttt{int}\ data = 0; \quad \texttt{atomic<}\ \texttt{int}\ \texttt{>}\ f = 0;$$

$$\begin{array}{l} [data] := 42; \\ [f] := 1; \end{array} \;\Bigg\|\; \begin{array}{l} \texttt{while}\ ([f] == 0)\ \{\}; \\ \texttt{print}([data]); \end{array}$$

Undefined Behavior and Memory Models

$$\texttt{int}\ \mathit{data} = 0; \quad \texttt{atomic<}\ \texttt{int}\ \texttt{>}\ f = 0;$$

$$\begin{array}{l} [\mathit{data}] := 42; \\ [f]^{\texttt{rel}} := 1; \end{array} \quad \Bigg\| \quad \begin{array}{l} \texttt{while}\ ([f]^{\texttt{acq}} == 0)\ \{\}; \\ \texttt{print}([\mathit{data}]); \end{array}$$

Undefined Behavior and Memory Models

$$\texttt{int}\ \mathit{data} = 0; \quad \texttt{atomic<}\ \texttt{int}\ \texttt{>}\ f = 0;$$

$$\begin{array}{l} [\mathit{data}] := 42; \\ [f]^{\texttt{rel}} := 1; \end{array} \quad \Bigg\| \quad \begin{array}{l} \texttt{while}\ ([f]^{\texttt{acq}} == 0)\ \{\}; \\ \texttt{print}([\mathit{data}]); \end{array}$$

	Java MM	C/C++ MM
special locations	`volatile int`	`atomic<int>`
data race on `int`	weak guarantees	undefined behavior
	access to `int`	relaxed (`rlx`) access to `atomic<int>`

Undefined Behavior and Memory Models

$$\texttt{int}\ \mathit{data} = 0; \quad \texttt{atomic<}\ \texttt{int}\ \texttt{>}\ f = 0;$$

$$\begin{array}{l} [\mathit{data}] := 42; \\ [f]^{\texttt{rel}} := 1; \end{array} \quad \Bigg\| \quad \begin{array}{l} \texttt{while}\ ([f]^{\texttt{acq}} == 0)\ \{\}; \\ \texttt{print}([\mathit{data}]); \end{array}$$

	Java MM	C/C++ MM
special locations	`volatile int`	`atomic<int>`
data race on `int`	weak guarantees	undefined behavior
subject to OOTA	access to `int`	relaxed (`rlx`) access to `atomic<int>`

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)
SC	[Lamport, 1979]	😐	☹	🙂
Java MM	[Manson et al., 2005]	☹	😐	🙂
C/C++ MM	[Batty et al., 2011]	🙂	🙂	☹
RC11	[Lahav et al., 2017]	😐	😐	🙂

Forbids all $\mathtt{po} \cup \mathtt{rf}$ cycles

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)	No UB
SC	[Lamport, 1979]	😐	☹	🙂	🙂
Java MM	[Manson et al., 2005]	☹	😐	🙂	🙂
C/C++ MM	[Batty et al., 2011]	🙂	🙂	☹	☹
RC11	[Lahav et al., 2017]	😐	😐	🙂	☹

Forbids all $\mathtt{po} \cup \mathtt{rf}$ cycles

To forbid $\mathtt{po} \cup \mathtt{rf}$ cycles in C/C++
enough to respect $[\mathtt{R}] ; \mathtt{po} ; [\mathtt{W}]$ on **atomics**

Preserving $[\mathtt{R}]$; $\mathtt{po}$; $[\mathtt{W}]$ for **atomics** in LLVM [Ou and Demsky, 2018]

Preserving [R] ; po ; [W] for **atomics** in LLVM [Ou and Demsky, 2018]

1. *Restrict compiler optimizations:*
2. *Put a fence between* R *and* W

Preserving [R] ; po ; [W] for **atomics** in LLVM [Ou and Demsky, 2018]

1. *Restrict compiler optimizations:* No changes for LLVM
2. *Put a fence between* R *and* W

Preserving [R] ; po ; [W] for **atomics** in LLVM [Ou and Demsky, 2018]

1. *Restrict compiler optimizations:* No changes for LLVM
2. *Put a fence between* R *and* W
   - x86: no fences

Preserving [R] ; po ; [W] for **atomics** in LLVM [Ou and Demsky, 2018]

1. *Restrict compiler optimizations:* No changes for LLVM
2. *Put a fence between* R *and* W
   - x86: no fences
   - ARMv8: bogus conditional branch for relaxed **atomic** reads

Preserving [R] ; po ; [W] for **atomics** in LLVM [Ou and Demsky, 2018]

1. *Restrict compiler optimizations:* No changes for LLVM
2. *Put a fence between* R *and* W
   - x86: no fences
   - ARMv8: bogus conditional branch for relaxed **atomic** reads

Slowdown on ARMv8 is **0%** on average and **6.3%** max

CDS from CDS C++, Folly, Junction, Rigtorp libs and 6 bechmarks from CDSSpec

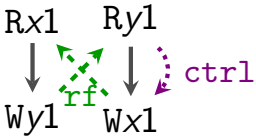
Preserving $[\mathtt{R}] ; \mathtt{po} ; [\mathtt{W}]$ is good if done only for **atomics**

Preserving $[\mathtt{R}] ; \mathtt{po} ; [\mathtt{W}]$ is good if done only for **atomics**

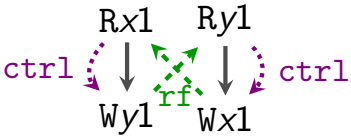
Anything suitable for ‘No UB‘ case (i.e., Java)?

Out-Of-Thin-Air in C/C++ MM

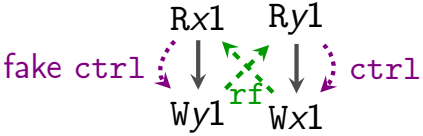
```
a := [x];          ‖  b := [y];
[y] := 1           ‖  if b then
                   ‖     [x] := 1
```




```
a := [x];          ‖  b := [y];
if a then          ‖  if b then
   [y] := 1        ‖     [x] := 1
```




```
a := [x];          ‖  b := [y];
if a then          ‖  if b then
   [y] := 1        ‖     [x] := 1
else               ‖
   [y] := 1        ‖
```



Preserving dependencies in LLVM [Ou and Demsky, 2018]

Modified 35/46 optimization passes, others turned off

Slowdown on ARMv8 is **3.1%** on average and **17.6%** max

SPEC CINT2006 benchmark

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)	No UB
SC	[Lamport, 1979]	😐	☹	🙂	🙂
Java MM	[Manson et al., 2005]	☹	😐	🙂	🙂
C/C++ MM	[Batty et al., 2011]	🙂	🙂	☹	☹
RC11	[Lahav et al., 2017]	😐	😐	🙂	☹

Forbids all $\mathtt{po} \cup \mathtt{rf}$ cycles

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)	No UB
SC	[Lamport, 1979]	😐	😡	🙂	🙂
Java MM	[Manson et al., 2005]	☹️	😐	🙂	🙂
C/C++ MM	[Batty et al., 2011]	🙂	🙂	☹️	☹️
RC11	[Lahav et al., 2017]	😐	😐	🙂	😡
Promising	[Kang et al., 2017, Lee et al., 2020]	🙂	🙂	🙂	🙂
Weakestmo	[Chakraborty and Vafeiadis, 2019]	🙂	🙂	🙂	☹️
Modular Relaxed Dep.	[Paviotti et al., 2020]	😐	🙂	🙂	☹️

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)	No UB
SC	[Lamport, 1979]	😐	😡	🙂	🙂
Java MM	[Manson et al., 2005]	☹️	😐	🙂	🙂
C/C++ MM	[Batty et al., 2011]	🙂	🙂	☹️	☹️
RC11	[Lahav et al., 2017]	😐	😐	🙂	☹️
Promising	[Kang et al., 2017, Lee et al., 2020]	🙂	🙂	🙂	🙂
Weakestmo	[Chakraborty and Vafeiadis, 2019]	🙂	🙂	🙂	☹️
Modular Relaxed Dep.	[Paviotti et al., 2020]	😐	🙂	🙂	☹️
OCaml MM	[Dolan et al., 2018]	😐	☹️	🙂	🙂

Usual Data-Race-Freedom:

$$\textit{No data races} \Rightarrow \textit{only SC behaviors}$$

Usual Data-Race-Freedom:

$$\textit{No data races} \Rightarrow \textit{only SC behaviors}$$

No guarantees in case of *irrelevant* races!

Usual Data-Race-Freedom:

$$\textit{No data races} \Rightarrow \textit{only SC behaviors}$$

No guarantees in case of *irrelevant* races!

$$\begin{array}{l} [x] := a + 10; \\ \ldots \\ [y] := a + 10; \end{array} \quad \Bigg\| \quad [x] := 1;$$

Usual Data-Race-Freedom:

No data races $\Rightarrow$ *only SC behaviors*

No guarantees in case of *irrelevant* races!

$$\begin{array}{l} [x] := a + 10; \\ \ldots \\ [y] := a + 10; \end{array} \;\Bigg\|\; [x] := 1; \qquad \begin{array}{l} t := a + 10; \\ [x] := t \\ \ldots \\ [y] := t; \end{array} \;\Bigg\|\; [x] := 1;$$

OCaml MM provides **Local** DRF

Usual Data-Race-Freedom:

No data races ⇒ only SC behaviors

No guarantees in case of *irrelevant* races!

$$\begin{array}{l} [x] := a + 10; \\ \ldots \\ [y] := a + 10; \end{array} \quad \Bigg\| \quad [x] := 1;$$

$$\begin{array}{l} t := a + 10; \\ [x] := t \\ \ldots \\ [y] := t; \end{array} \quad \Bigg\| \quad [x] := 1;$$

$$\begin{array}{l} t := a + 10; \\ [x] := t \\ \ldots \\ [y] := [x]; \end{array} \quad \Bigg\| \quad [x] := 1;$$

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)	No UB
SC	[Lamport, 1979]	😐	😡	🙂	🙂
Java MM	[Manson et al., 2005]	☹️	😐	🙂	🙂
C/C++ MM	[Batty et al., 2011]	🙂	🙂	☹️	☹️
RC11	[Lahav et al., 2017]	😐	😐	🙂	☹️
Promising	[Kang et al., 2017, Lee et al., 2020]	🙂	🙂	🙂	🙂
Weakestmo	[Chakraborty and Vafeiadis, 2019]	🙂	🙂	🙂	☹️
Modular Relaxed Dep.	[Paviotti et al., 2020]	😐	🙂	🙂	☹️
OCaml MM	[Dolan et al., 2018]	😐	☹️	🙂	🙂

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)	No UB	Simplicity
SC	[Lamport, 1979]	😐	😡	🙂	🙂	
Java MM	[Manson et al., 2005]	☹️	😐	🙂	🙂	
C/C++ MM	[Batty et al., 2011]	🙂	🙂	☹️	☹️	
RC11	[Lahav et al., 2017]	😐	😐	🙂	😡	
Promising	[Kang et al., 2017, Lee et al., 2020]	🙂	🙂	🙂	🙂	
Weakestmo	[Chakraborty and Vafeiadis, 2019]	🙂	🙂	🙂	☹️	
Modular Relaxed Dep.	[Paviotti et al., 2020]	😐	🙂	🙂	☹️	
OCaml MM	[Dolan et al., 2018]	😐	☹️	🙂	🙂	

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)	No UB	Simplicity
SC	[Lamport, 1979]	😐	😡	🙂	🙂	🙂
Java MM	[Manson et al., 2005]	☹️	😐	🙂	🙂	☹️
C/C++ MM	[Batty et al., 2011]	🙂	🙂	😡	😡	😐
RC11	[Lahav et al., 2017]	😐	😐	🙂	😡	🙂
Promising	[Kang et al., 2017, Lee et al., 2020]	🙂	🙂	🙂	🙂	☹️
Weakestmo	[Chakraborty and Vafeiadis, 2019]	🙂	🙂	🙂	😡	☹️
Modular Relaxed Dep.	[Paviotti et al., 2020]	😐	🙂	🙂	☹️	☹️
OCaml MM	[Dolan et al., 2018]	😐	☹️	🙂	🙂	🙂

To take away

Mainstream MM (SC, C/C++ MM and JMM) have major issues

Existing solutions make different compromises

- How much performance can you sacrifice?
- How complicated and new can your MM be?
- Can you have UB?
- What guarantees do you want to provide?

Programming languages' MM

		Comp. Opt.	Eff. Comp. to Hardware	DRF (No OOTA)	No UB	Simplicity
SC	[Lamport, 1979]	😐	😡	🙂	🙂	🙂
Java MM	[Manson et al., 2005]	☹️	😐	🙂	🙂	☹️
C/C++ MM	[Batty et al., 2011]	🙂	🙂	😡	😡	😐
RC11	[Lahav et al., 2017]	😐	😐	🙂	😡	🙂
Promising	[Kang et al., 2017, Lee et al., 2020]	🙂	🙂	🙂	🙂	☹️
Weakestmo	[Chakraborty and Vafeiadis, 2019]	🙂	🙂	🙂	😡	☹️
Modular Relaxed Dep.	[Paviotti et al., 2020]	😐	🙂	🙂	☹️	☹️
OCaml MM	[Dolan et al., 2018]	😐	☹️	🙂	🙂	🙂

http://podkopaev.net

Thank you!

Links I

Batty, M., Owens, S., Sarkar, S., Sewell, P., and Weber, T. (2011).
Mathematizing C++ concurrency.
In *POPL 2011*, pages 55–66. ACM.

Chakraborty, S. and Vafeiadis, V. (2019).
Grounding thin-air reads with event structures.
In *POPL 2019*. ACM.

Dolan, S., Sivaramakrishnan, K., and Madhavapeddy, A. (2018).
Bounding data races in space and time.
In *PLDI 2018*.

Kang, J., Hur, C.-K., Lahav, O., Vafeiadis, V., and Dreyer, D. (2017).
A promising semantics for relaxed-memory concurrency.
In *POPL 2017*. ACM.

Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., and Dreyer, D. (2017).
Repairing sequential consistency in C/C++11.
In *PLDI 2017*. ACM.

Lamport, L. (1979).
How to make a multiprocessor computer that correctly executes multiprocess programs.
IEEE Trans. Computers, 28(9):690–691.

Links II

Lee, S.-H., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C.-K., Lahav, O., and Vafeiadis, V. (2020).
Promising 2.0: Global optimizations in relaxed memory concurrency.
In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 362–376, New York, NY, USA. Association for Computing Machinery.

Liu, L., Millstein, T., and Musuvathi, M. (2017).
A volatile-by-default JVM for server applications.
In *OOPSLA 2017*.

Liu, L., Millstein, T., and Musuvathi, M. (2019).
Accelerating sequential consistency for Java with speculative compilation.
In *PLDI 2019*.

Manson, J., Pugh, W., and Adve, S. V. (2005).
The Java memory model.
In *POPL 2005*, pages 378–391. ACM.

Marino, D., Singh, A., Millstein, T., Musuvathi, M., and Narayanasamy, S. (2011).
A case for an SC-preserving compiler.
In *PLDI 2011*.

Ou, P. and Demsky, B. (2018).
Towards understanding the costs of avoiding Out-of-Thin-Air results.
In *OOPSLA 2018*.

Links III

Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., and Batty, M. (2020).
Modular relaxed dependencies in weak memory concurrency.
In *ESOP 2020*.

Ševčík, J. and Aspinall, D. (2008).
On validity of program transformations in the Java memory model.
In *ECOOP 2008*.

Backup slides

Bonus: HotSpot breaks JMM's DRF-SC for Power

```
                    volatile int x, y, z;
x = 1;          ‖ y = 1;  ‖ int b = y; // 1 ‖     z = 2;          ‖ int d = z; // 1
int a = y; // 0 ‖         ‖       z = 1;    ‖ int c = x; // 0 ‖ int e = z; // 2
```

Compilation schemes	*Alt.* 1	*Alt.* 2
volatile write	lwsync; st; sync	lwsync; st
volatile read	ld; lwsync	sync; ld; lwsync

`https://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/file/ac7b3be2fdb5/src/share/vm/opto/library_call.cpp#l2633`

Validity of transformations [Ševčík and Aspinall, 2008]

	SC	JMM*
Trace-preserving transformations	✓	✓
Reordering normal memory accesses	✗	✓*
Redundant read after read elimination	✓	✗
Redundant read after write elimination	✓	✓
Irrelevant read elimination	✓	✓
Irrelevant read introduction	✓	✗
Redundant write before write elimination	✓	✓
Redundant write after read elimination	✓	✗
External action reordering	✗	✗

Compiler optimization invalidated in JMM [Ševčík and Aspinall, 2008]

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

⟶

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

⟶

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

Fig. 5. Hotspot JVM's transformations violating the JMM.

OCaml MM to ARMv8 compilation scheme

Operation	Implementation
Nonatomic read	`ldr R, [x]; cbz R, L; L:`
Nonatomic write	`str R, [x]`
Atomic read	`dmb ld; ldar R, [x]`
Atomic write	`L: ldaxr; stlxr; cbnz L; dmb st`

(a) Compilation scheme 1

Operation	Implementation
Nonatomic read	`ldr R, [x]`
Nonatomic write	`dmb ld; str R, [x]`
Atomic read	`dmb ld; ldar R, [x]`
Atomic write	`L: ldaxr; stlxr; cbnz L; dmb st`

(b) Compilation scheme 2

Table 5. Compilation to ARMv8 (AArch64)