On thin air reads

Towards an event structures model of relaxed memory

Alan Jeffrey (Bell Labs and Mozilla Research)
James Riely (DePaul University)

Logic in Computer Science 2016

2 years of conversation in 20 minutes

Relaxed memory models

An example:

> *thread 1:* `y=1; r1=x;`
> *thread 2:* `r2=y; x=r2;`

Assume variables initialized to 0.
Is it possible for both `r1` and `r2` to read `1`?

Relaxed memory models

An example:

> *thread 1:* `y=1; r1=x;`
> *thread 2:* `r2=y; x=r2;`

Assume variables initialized to 0.
Is it possible for both `r1` and `r2` to read `1`?
Yes! There is a sequential schedule that respects thread order:

```
y=1; r2=y; x=r2; r1=x;
```

This is called *sequential consistency (SC)*

Relaxed memory models

An example:

> *thread 1:* `y=1; r1=x;`
> *thread 2:* `r2=y; x=r2;`

Assume variables initialized to 0.
Is it possible for both `r1` and `r2` to read `1`?
Yes! There is a sequential schedule that respects thread order:

```
y=1; r2=y; x=r2; r1=x;
```

This is called *sequential consistency (SC)*

The operations of thread 1 act on different variables.
What if we swap them?

> *thread 1:* `r1=x; y=1;`
> *thread 2:* `r2=y; x=r2;`

Is it possible for both `r1` and `r2` to read `1`?

Relaxed memory models

An example:

> *thread 1:* `y=1; r1=x;`
> *thread 2:* `r2=y; x=r2;`

Assume variables initialized to 0.
Is it possible for both `r1` and `r2` to read `1`?
Yes! There is a sequential schedule that respects thread order:

```
y=1; r2=y; x=r2; r1=x;
```

This is called *sequential consistency (SC)*

The operations of thread 1 act on different variables.
What if we swap them?

> *thread 1:* `r1=x; y=1;`
> *thread 2:* `r2=y; x=r2;`

Is it possible for both `r1` and `r2` to read `1`?
Yes!

Relaxed memory models

An example:

> *thread 1:* `y=1; r1=x;`
> *thread 2:* `r2=y; x=r2;`

Assume variables initialized to 0.
Is it possible for both `r1` and `r2` to read `1`?
Yes! There is a sequential schedule that respects thread order:

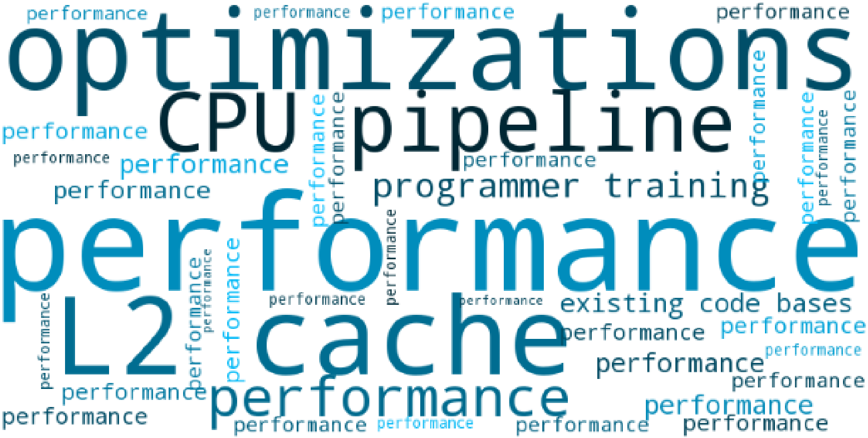
```
y=1; r2=y; x=r2; r1=x;
```

This is called *sequential consistency (SC)*

The operations of thread 1 act on different variables.
What if we swap them?

> *thread 1:* `r1=x; y=1;`
> *thread 2:* `r2=y; x=r2;`

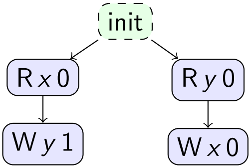
Is it possible for both `r1` and `r2` to read `1`?
Yes! Possible by reordering non-conflicting operations!


optimizations
CPU
pipeline
programmer training
performance
L2
cache
existing code bases

Java Memory Model (Mason, Pugh and Adve 2005)

- Guess the future
  without making things up

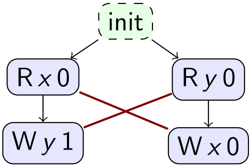
Java Memory Model (Mason, Pugh and Adve 2005)



thread 1: `r1=x; y=1;`
thread 2: `r2=y; x=r2;`

- Guess the future
  without making things up
  - Run the program
    reading only committed values

Java Memory Model (Mason, Pugh and Adve 2005)

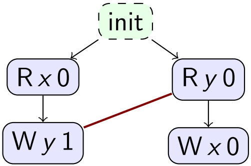


thread 1: `r1=x; y=1;`
thread 2: `r2=y; x=r2;`

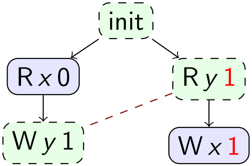
- Guess the future
  without making things up
  - Run the program
    reading only committed values
  - Spot read/write races

Java Memory Model (Mason, Pugh and Adve 2005)

thread 1: `r1=x; y=1;`
thread 2: `r2=y; x=r2;`

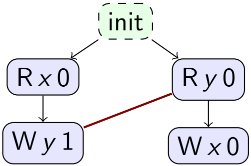


- Guess the future
  without making things up
  - Run the program
    reading only committed values
  - Spot read/write races
  - Commit a race and re-run

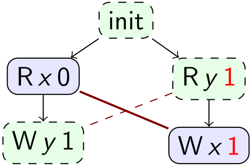


Java Memory Model (Mason, Pugh and Adve 2005)

thread 1: `r1=x; y=1;`
thread 2: `r2=y; x=r2;`



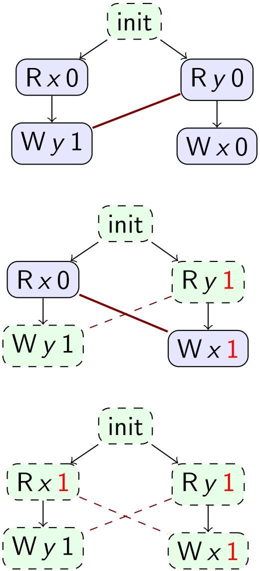
- Guess the future
  without making things up
  - Run the program
    reading only committed values
  - Spot read/write races
  - Commit a race and re-run
  - Repeat



Java Memory Model (Mason, Pugh and Adve 2005)

thread 1: `r1=x; y=1;`
thread 2: `r2=y; x=r2;`

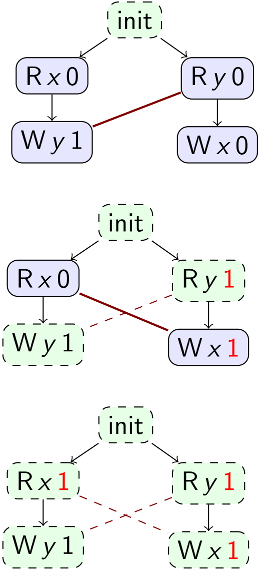
- Guess the future
  without making things up
  - Run the program
    reading only committed values
  - Spot read/write races
  - Commit a race and re-run
  - Repeat
- (Some fiddly bits concerning
  conditionals and identity of actions)



Java Memory Model (Mason, Pugh and Adve 2005)

thread 1: `r1=x; y=1;`
thread 2: `r2=y; x=r2;`

- Guess the future
  without making things up
  - Run the program
    reading only committed values
  - Spot read/write races
  - Commit a race and re-run
  - Repeat
- (Some fiddly bits concerning conditionals and identity of actions)
- Simple enough?



Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y

```
r1 = x     r2 = y
y = 1      x = r2
```

Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y
- Split execution, generating writes

```
r1 = x      r2 = y
y = 1       x = r2
```

~~?y=1~~

?y=1

```
r1 = x      r2 = y
y = 1       x = r2
```

initial

```
r1 = x      r2 = y
y = 1       x = r2
```

final

Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y
- Split execution, generating writes
  - Final branch may see speculation
  - Initial branch may not

```
r1 = x     r2 = y
y = 1      x = r2
```

~~?y=1~~

?y=1

```
r1 = x     0 = y
y = 1      x = 0
```

initial

```
r1 = x     1 = y
y = 1      x = 1
```

final

Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y
- Split execution, generating writes
  - Final branch may see speculation
  - Initial branch may not

```
r1 = x      r2 = y
y = 1       x = r2
```

~~?y=1~~ ?y=1

Wx=0

```
r1 = x      0 = y
y = 1       x = 0
```

initial

Wx=1

```
r1 = x      1 = y
y = 1       x = 1
```

final

Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y
- Split execution, generating writes
  - Final branch may see speculation
  - Initial branch may not

```
r1 = x     r2 = y
y = 1      x = r2
```

~~?y=1~~

?y=1

Wx=0

```
0 = x     0 = y
y = 1     x = 0
```

initial

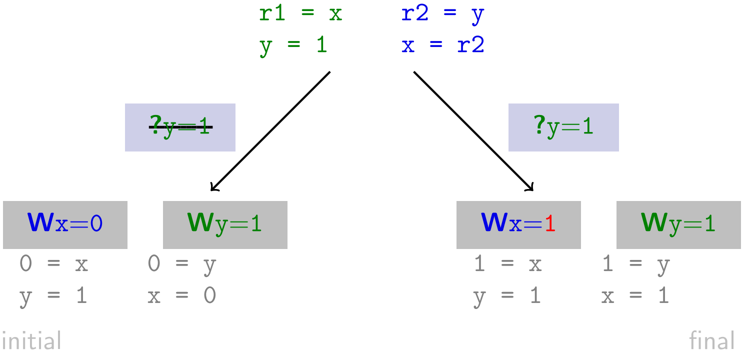
Wx=1

```
1 = x     1 = y
y = 1     x = 1
```

final

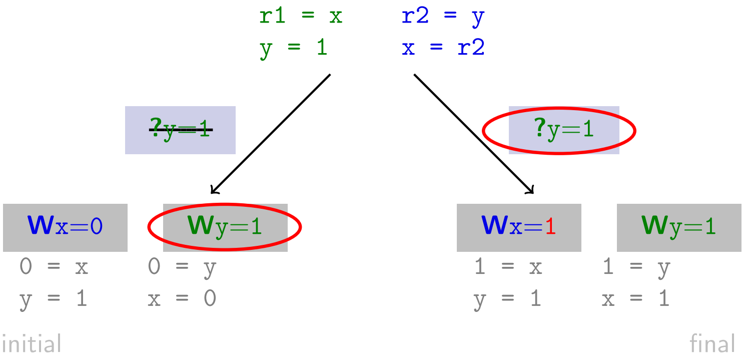
Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y
- Split execution, generating writes
  - Final branch may see speculation
  - Initial branch may not



Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y
- Split execution, generating writes
  - Final branch may see speculation
  - Initial branch may not
- Initial branch must justify speculation



Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y
- Split execution, generating writes
  - Final branch may see speculation
  - Initial branch may not
- Initial branch must justify speculation
- Afterwards, only final copy remains

Wx=1	**W**y=1
`1 = x`	`1 = y`
`y = 1`	`x = 1`

Speculation (Jagadeesan, Pitcher, Riely 2010)

- Speculate that there will be write of 1 to y
- Split execution, generating writes
  - Final branch may see speculation
  - Initial branch may not
- Initial branch must justify speculation
- Afterwards, only final copy remains
- Simple enough?

Wx=1	**W**y=1
`1 = x`	`1 = y`
`y = 1`	`x = 1`

C/C++ (Boehm, et al 2010s)

$$\begin{aligned}
\mathsf{isread}_{\ell,v}(a) &\overset{\text{def}}{=} \exists X, v'.\ lab(a) \in \{\mathsf{R}_X(\ell,v), \mathsf{C}_X(\ell,v,v')\} \\
\mathsf{iswrite}_{\ell,v}(a) &\overset{\text{def}}{=} \exists X, v'.\ lab(a) \in \{\mathsf{W}_X(\ell,v), \mathsf{C}_X(\ell,v',v)\} \\
\mathsf{isfence}(a) &\overset{\text{def}}{=} lab(a) \in \{\mathsf{F}_{\text{ACQ}}, \mathsf{F}_{\text{REL}}\} \\
\mathsf{sameThread}(a,b) &\overset{\text{def}}{=} \mathsf{tid}(a) = \mathsf{tid}(b) \\
\mathsf{rsElem}(a,b) &\overset{\text{def}}{=} \mathsf{sameThread}(a,b) \lor \mathsf{isrmw}(b)
\end{aligned}$$

$$\begin{aligned}
\mathsf{isread}_{\ell}(a) &\overset{\text{def}}{=} \exists v.\ \mathsf{isread}_{\ell,v}(a) \\
\mathsf{iswrite}_{\ell}(a) &\overset{\text{def}}{=} \exists v.\ \mathsf{iswrite}_{\ell,v}(a) \\
\mathsf{isaccess}(a) &\overset{\text{def}}{=} \mathsf{isread}(a) \lor \mathsf{iswrite}(a) \\
\mathsf{isrmw}(a) &\overset{\text{def}}{=} \mathsf{isread}(a) \land \mathsf{iswrite}(a) \\
\mathsf{isAcq}(a) &\overset{\text{def}}{=} \mathsf{mode}(a) \sqsupseteq \text{ACQ}
\end{aligned}$$

$$\begin{aligned}
\mathsf{isread}(a) &\overset{\text{def}}{=} \exists \ell.\ \mathsf{isread}_{\ell}(a) \\
\mathsf{iswrite}(a) &\overset{\text{def}}{=} \exists \ell.\ \mathsf{iswrite}_{\ell}(a) \\
\mathsf{isNA}(a) &\overset{\text{def}}{=} \mathsf{mode}(a) = \text{NA} \\
\mathsf{isSC}(a) &\overset{\text{def}}{=} \mathsf{mode}(a) = \text{SC} \\
\mathsf{isRel}(a) &\overset{\text{def}}{=} \mathsf{mode}(a) \sqsupseteq \text{REL}
\end{aligned}$$

$$\begin{aligned}
\mathsf{rseq}(a,b) &\overset{\text{def}}{=} a = b \lor \mathsf{rsElem}(a,b) \land mo(a,b) \land (\forall c.\ mo(a,c) \land mo(c,b) \Rightarrow \mathsf{rsElem}(a,c)) \\
\mathsf{sw}(a,b) &\overset{\text{def}}{=} \exists c, d.\ \begin{array}{l} \neg\mathsf{sameThread}(a,b) \land \mathsf{isRel}(a) \land \mathsf{isAcq}(b) \land \mathsf{rseq}(c, rf(d)) \\ \land\ (a = c \lor \mathsf{isfence}(a) \land sb^+(a,c)) \land (d = b \lor \mathsf{isfence}(b) \land sb^+(d,b)) \end{array} \\
\mathsf{hb} &\overset{\text{def}}{=} (sb \cup \mathsf{sw} \cup asw)^+ \\
\mathsf{Racy} &\overset{\text{def}}{=} \exists a, b.\ \begin{array}{l} \mathsf{isaccess}(a) \land \mathsf{isaccess}(b) \land \mathsf{loc}(a) = \mathsf{loc}(b) \land a \neq b \\ \land(\mathsf{iswrite}(a) \lor \mathsf{iswrite}(b)) \land (\mathsf{isNA}(a) \lor \mathsf{isNA}(b)) \land \neg(\mathsf{hb}(a,b) \lor \mathsf{hb}(b,a)) \end{array} \\
\mathsf{Observation} &\overset{\text{def}}{=} \{(a,b) \mid mo(a,b) \land \mathsf{loc}(a) = \mathsf{loc}(b) = \mathtt{world}\}
\end{aligned}$$

Figure 2. Auxiliary definitions for a C11 execution $(lab, sb, asw, rf, mo, sc)$.

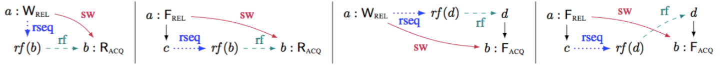
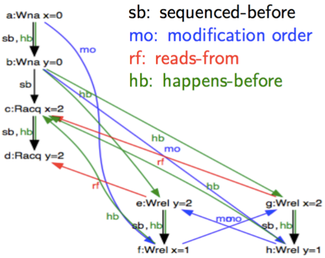


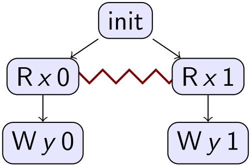
Figure 3. Illustration of the "synchronizes-with" definition: the four cases inducing an sw edge.

thread 1: `r1=x; r2=y;`
thread 2: `y=2; x=1;`
thread 3: `x=2; y=1;`



Event structures (Winskel 1980s)

The event structure for `r=x; y=r;`



Visualizes conflicting executions in a single structure

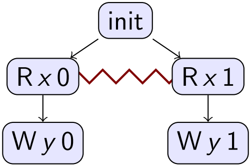
Fix an alphabet of actions Σ (e.g. init, R x v, W x v, ...).

A labelled prime event structure $(E, \leq, \#, \lambda)$ consists of:

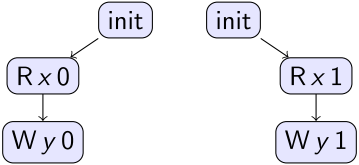
- A partial order $(E, \leq)$ (events with program order)
- A function $\lambda : E \to \Sigma$ (labelling)
- A binary relation $\#$ on E (conflict)
- If $d \# e$ then $d \neq e$, and if $c \# d \leq e$ then $c \# e$

Configurations model executions

The event structure for `r=x; y=r;`



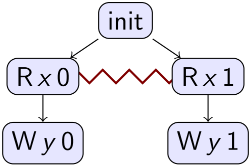
Has configurations:



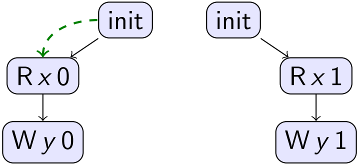
A **configuration** is a $\leq$-downclosed, $\#$-free set of events.

Configurations model executions

The event structure for `r=x; y=r;`



Has configurations:



A **configuration** is a $\leq$-downclosed, $\#$-free set of events.

Justified configurations

We need more structure on Σ. A **memory alphabet** has:

- $R \subseteq \Sigma$ (**read** actions, e.g. $(\mathsf{R}\,x\,1) \in R$)
- $W \subseteq \Sigma$ (**write** actions, e.g. $(\mathsf{W}\,x\,1) \in W$)
- $J \subseteq (W \times R)$ (**justification** relation, e.g. $(\mathsf{W}\,x\,1, \mathsf{R}\,x\,1) \in J$)

Justified configurations

We need more structure on Σ. A **memory alphabet** has:

- $R \subseteq \Sigma$ (**read** actions, e.g. $(\mathsf{R}\,x\,1) \in R$)
- $W \subseteq \Sigma$ (**write** actions, e.g. $(\mathsf{W}\,x\,1) \in W$)
- $J \subseteq (W \times R)$ (**justification** relation, e.g. $(\mathsf{W}\,x\,1, \mathsf{R}\,x\,1) \in J$)

On events, d **justifies** e (e.g. $\boxed{\mathsf{init}} \dashrightarrow \boxed{\mathsf{R}\,x\,0}$) if:

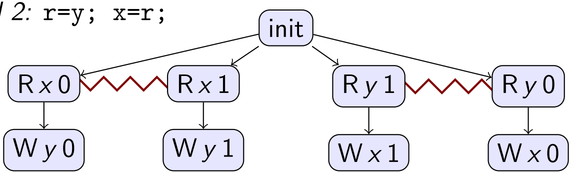
- $(\lambda(d), \lambda(e)) \in J$,
- $(d, e) \notin \#$, and
- $d \not\geq e$.

On configurations:

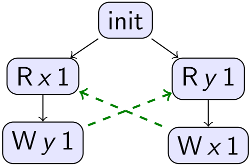
- C **justifies** D when every read event in D has a justifier in C.
- C **is justified** when C justifies itself.

The TAR* pit

thread 1: `r=x; y=r;`
thread 2: `r=y; x=r;`



Has justified configuration:



A TAR caused by a cycle in (justification + program-order).

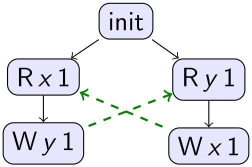
*Thin Air Read

Sequentially justified configurations

Ban such cycles! On configurations:

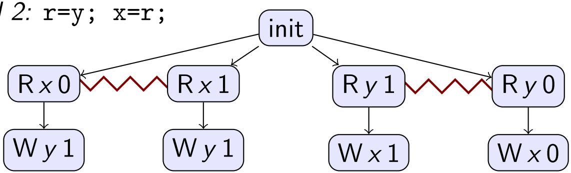
- C **sequentially justifies** D when $C = C_0 \subseteq \cdots \subseteq C_n = D$, where each C_i justifies C_{i+1}.
- C **is sequentially justified** when $\emptyset$ sequentially justifies C.

For example, the TAR pit is justified, but not sequentially justified:

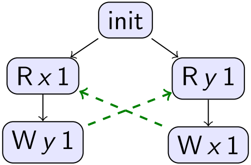


Instruction reordering example

thread 1: `r=x; y=1;` *// These may be reordered*
thread 2: `r=y; x=r;`



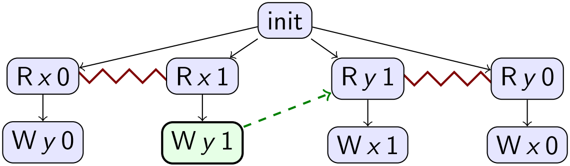
should allow:



TAR pit versus Instruction reordering

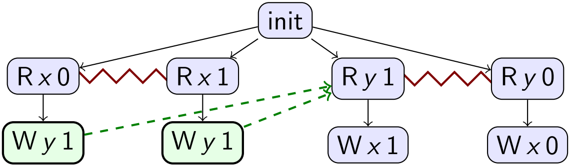
TAR pit

thread 1: `r=x; y=r;`
thread 2: `r=y; x=r;`



Instruction reordering

thread 1: `r=x; y=1;`
thread 2: `r=y; x=r;`



Sewell *et al.* 2015: there is no per-candidate-execution model of relaxed memory which supports instruction reordering

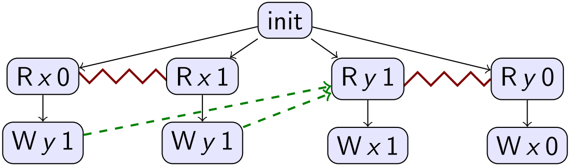
Sewell *et al.* 2015: there is no per-candidate-execution model of relaxed memory which supports instruction reordering

Good thing event structures aren't per-candidate-execution

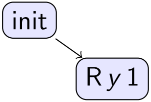
AE justification (Always Eventual Justification)

On configurations, C **AE-justifies** D when
for all C' sequentially justified by C,
there exists C'' sequentially justified by C',
where C'' justifies D.

Instruction reordering example:



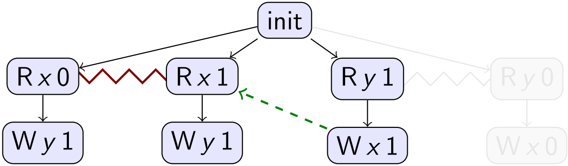
$\emptyset$ AE-justifies



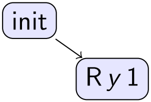
AE justification (Always Eventual Justification)

On configurations, C **AE-justifies** D when
for all C' sequentially justified by C,
there exists C'' sequentially justified by C',
where C'' justifies D.

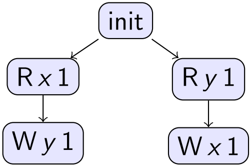
Instruction reordering example:



$\emptyset$ AE-justifies



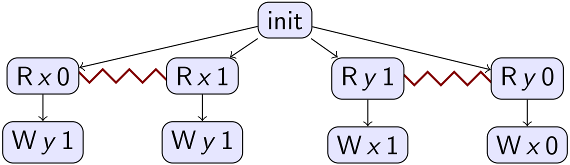
AE-justifies

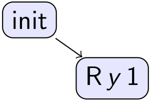


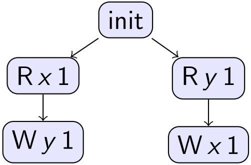
Well justification

- C **well-justifies** D when $C = C_0 \subseteq \cdots \subseteq C_n = D$, where each C_i AE-justifies C_{i+1}.
- C **is well-justified** when $\emptyset$ well-justifies C and C is justified.

Instruction reordering example:



$\emptyset$ AE-justifies 

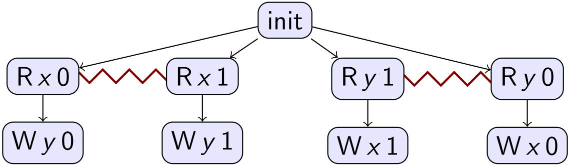
AE-justifies 

That allows one to show existance of an execution.

What about non-existence? E.g., safety?

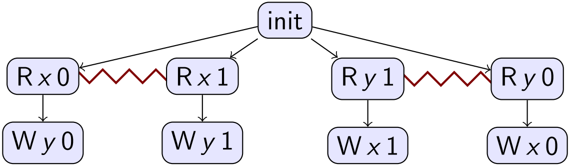
Invariant reasoning

TAR pit has an invariant $x{\neq}1 \wedge y{\neq}1$



Invariant reasoning

TAR pit has an invariant $x{\neq}1 \wedge y{\neq}1$



Define a simple program logic with judgements $A \vDash \phi$ (for $A \subseteq \Sigma$).

A formula ϕ is a *tauology* of an ES whenever
$\lambda(C) \vDash \phi$ for every well-justified C.

A formula ϕ is a *invariant* of an ES whenever
$\lambda(C) \cap R \vDash \phi$ implies $\lambda(C) \vDash \phi$ for every well-justified C.

Theorem.[1] If ϕ is an invariant then ϕ is a tautology.

Proof. Mechanized in Agda.

Examples. TAR pit; type soundness.

[1] Under mild technical conditions

Thesis: No TAR $\stackrel{\text{def}}{=}$ invariant reasoning is sound

Thesis: No TAR $\stackrel{\text{def}}{=}$ invariant reasoning is sound

Are we done?

DRF

Theorem.[2] If every SC configuration is DRF
then every well-justified configuration is SC.

SC: Sequentially Consistent.
DRF: Data Race Free.

Proof. Mechanized in Agda.

[2] Under mild technical conditions

DRF

Theorem.[2] If every SC configuration is DRF
then every well-justified configuration is SC.

SC: Sequentially Consistent.
DRF: Data Race Free.

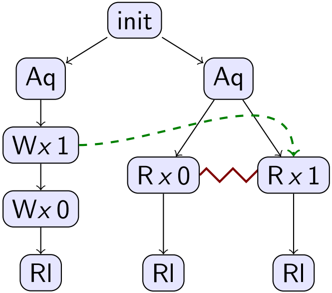
Proof. Mechanized in Agda.

Are we done now?

[2]Under mild technical conditions

What about synchronization?

Thread 1: `acq; x=1; x=0; rel;`
Thread 2: `acq; r=x; rel;`

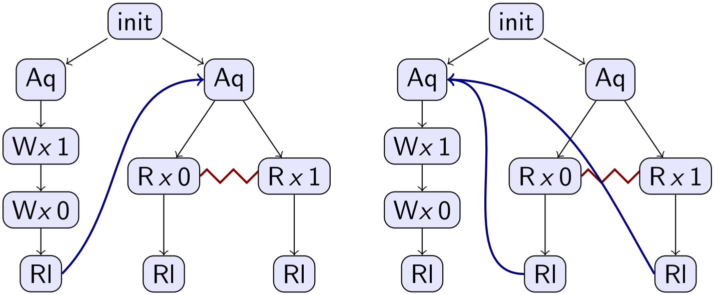


With one lock, it should be impossible for second thread to read 1

[3] Details in the paper

What about synchronization?

Thread 1: `acq; x=1; x=0; rel;`
Thread 2: `acq; r=x; rel;`



With one lock, it should be impossible for second thread to read 1
Two possible fencings[3]
Neither allows read of 1

[3] Details in the paper

Goals for relaxed memory

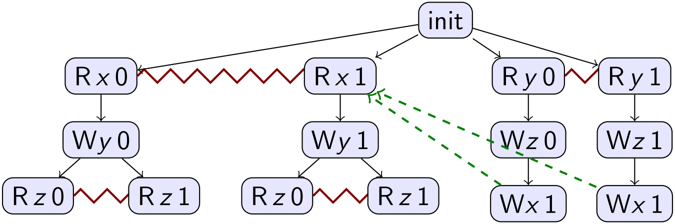
Contributions of this paper

Contributions of this paper

Reordering independent reads

Possible for all reads to resolve to 1.

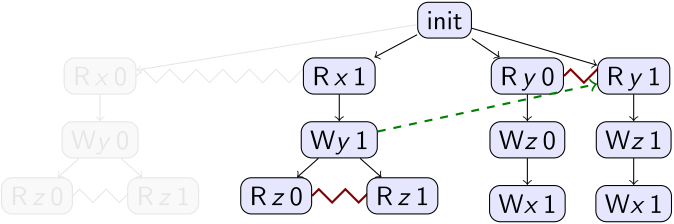
```
(y=x; r=z;) || (z=y; x=1;)
```



Reordering independent reads

Possible for all reads to resolve to 1.

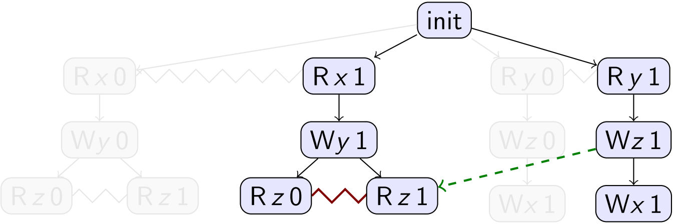
```
(y=x; r=z;) || (z=y; x=1;)
```



Reordering independent reads

Possible for all reads to resolve to 1.

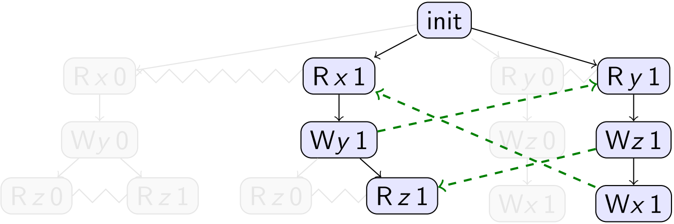
```
(y=x; r=z;) || (z=y; x=1;)
```



Reordering independent reads

Possible for all reads to resolve to 1.

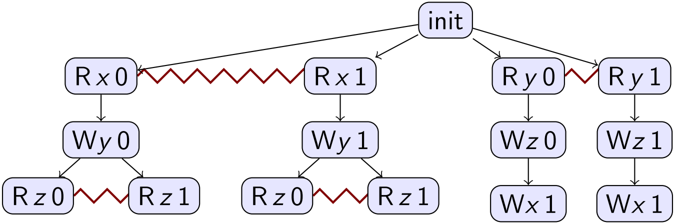
```
(y=x; r=z;) || (z=y; x=1;)
```



Reordering independent reads

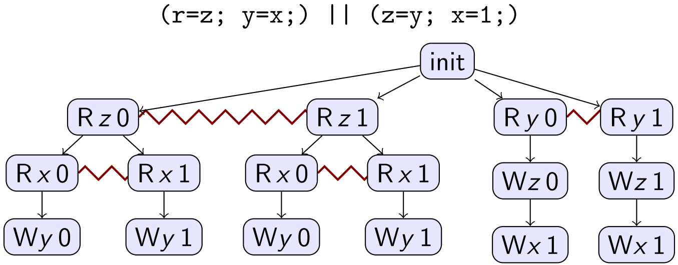
Possible for all reads to resolve to 1

```
(y=x; r=z;) || (z=y; x=1;)
```



Reordering independent reads

Impossible for all reads to resolve to 1

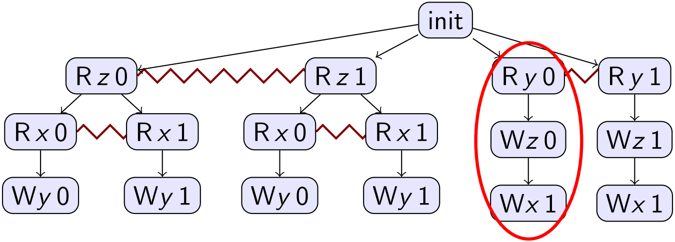


Should be possible
Reordering of independent reads

Reordering independent reads

Impossible for all reads to resolve to 1

`(r=z; y=x;) || (z=y; x=1;)`

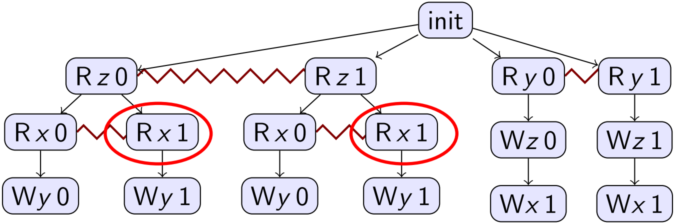


Should be possible
Reordering of independent reads

Reordering independent reads

Impossible for all reads to resolve to 1

`(r=z; y=x;) || (z=y; x=1;)`

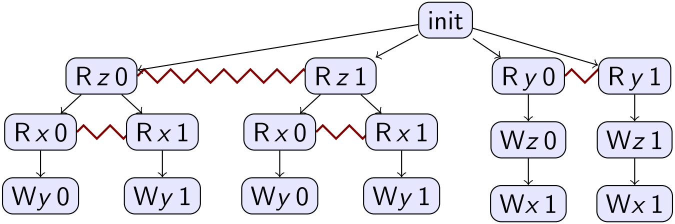


Should be possible
Reordering of independent reads

Reordering independent reads

Impossible for all reads to resolve to 1

```
(r=z; y=x;) || (z=y; x=1;)
```



Should be possible
Reordering of independent reads
Possible fix: use conlict-free sets instead of configurations

Contributions of this paper

Contributions of this paper

Contributions of this paper

Contributions of this paper

Contributions of this paper

Contributions of this paper

Contributions of this paper

Questions?