Verification of Concurrent Programs under Relaxed Memory Models

Moscow State University

Roland Meyer

Technische Universität Kaiserslautern

Concurrent Programs with Shared Memory

- Finite number of shared variables $\{x, y, x_1, \ldots\}$
- Finite data domain $\{d, d_0, d_1, \ldots\}$
- Finite number of finite-control threads $T_1, \ldots, T_n$ with operations:

$$w(x, d), \quad r(x, d)$$

$x = y = 0$

Thread 1	Thread 2
$a: x = 1$	$p: y = 1$
$b: if(y == 0)\{$	$q: if(x == 0)\{$
$c: \quad crit.sect.1$	$r: \quad crit.sect.2$
$d: \}$	$s: \}$

Dekker's mutual exclusion protocol.

Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical interleaving semantics
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c :$ $crit.sect.1$	$r :$ $crit.sect.2$
$d : \}$	$s : \}$

	Mem
Thread 1	x
$pc = a$	0
Thread 2	y
$pc = p$	0

Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical interleaving semantics
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c :$ $crit.sect.1$	$r :$ $crit.sect.2$
$d : \}$	$s : \}$

Thread 1
$pc = b$

Thread 2
$pc = p$

Mem

x
1
y
0

Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical interleaving semantics
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

Thread 1

$pc = c$

Thread 2

$pc = p$

Mem

x 1

y 0

Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical interleaving semantics
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

Thread 1

$pc = c$

Thread 2

$pc = q$

Mem

x 1

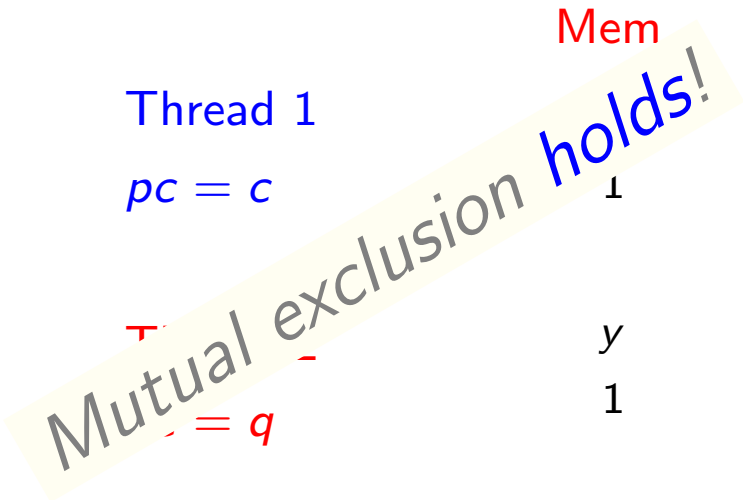
y 1

Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical interleaving semantics
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$



Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

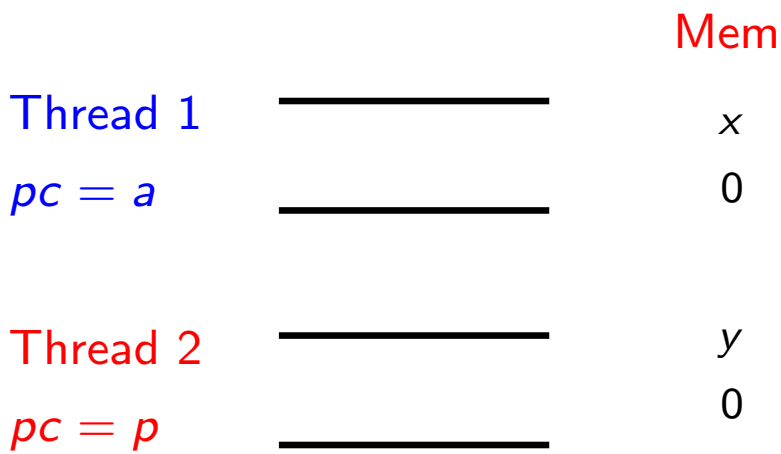
- Sequential Consistency forbids compiler and hardware optimizations
- Hence is not implemented by any processor
- Processors have various buffers to reduce latency of memory accesses
- Behavior captured by relaxed memory models
- Here: Total Store Ordering (TSO) memory model

Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write in the buffer on that variable

$x = y = 0$

Thread 1	Thread 2
$a: x = 1$	$p: y = 1$
$b: if(y == 0)\{$	$q: if(x == 0)\{$
$c: \quad crit.sect.1$	$r: \quad crit.sect.2$
$d: \}$	$s: \}$

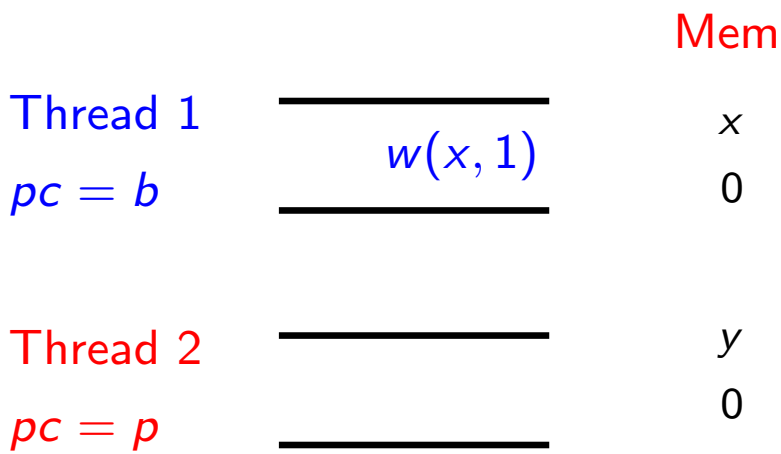


Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write in the buffer on that variable

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

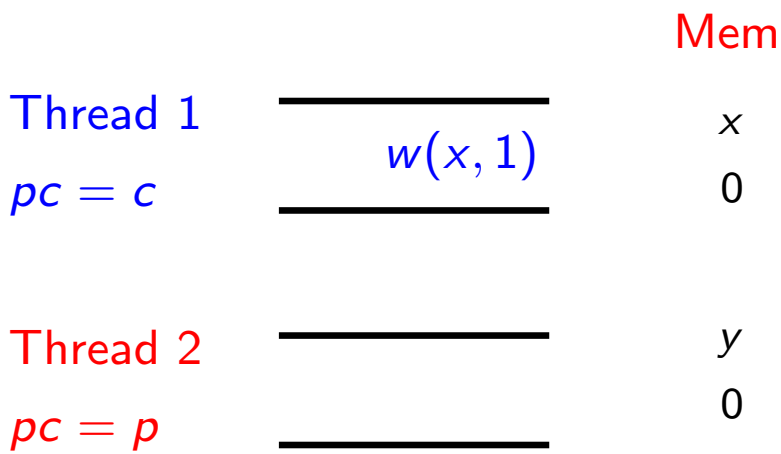


Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write in the buffer on that variable

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

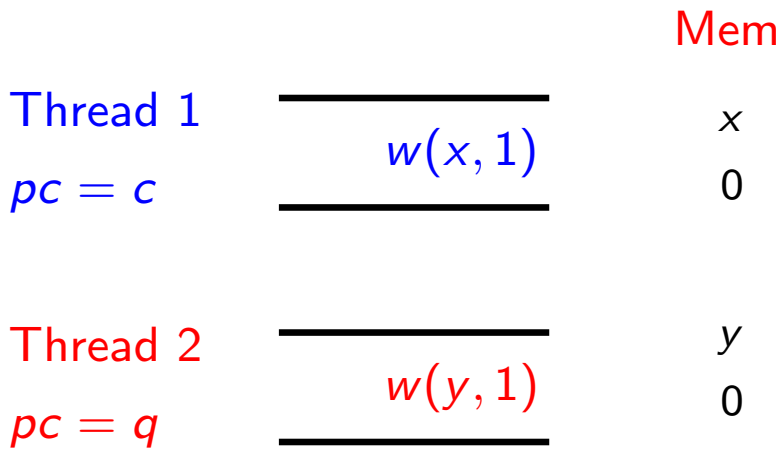


Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write in the buffer on that variable

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

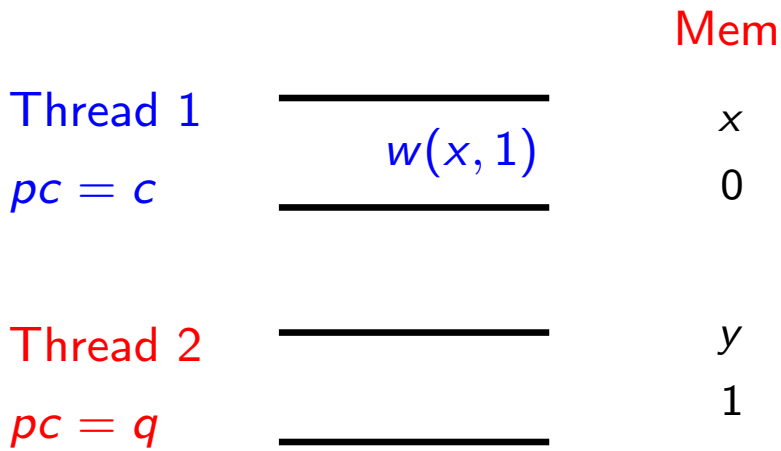


Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write in the buffer on that variable

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

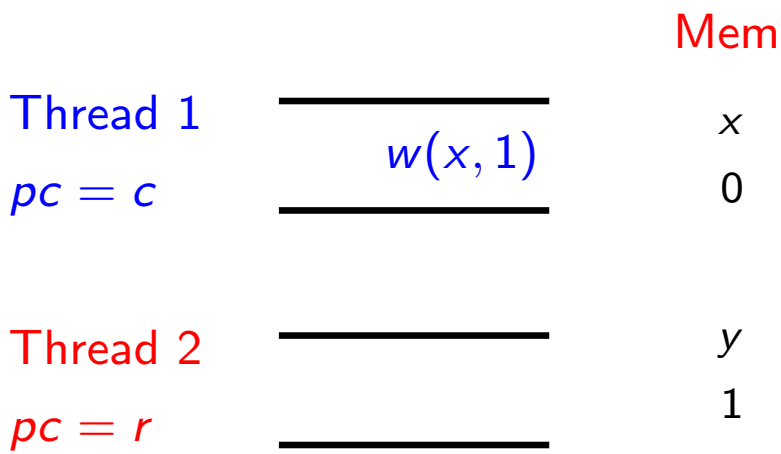


Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write in the buffer on that variable

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

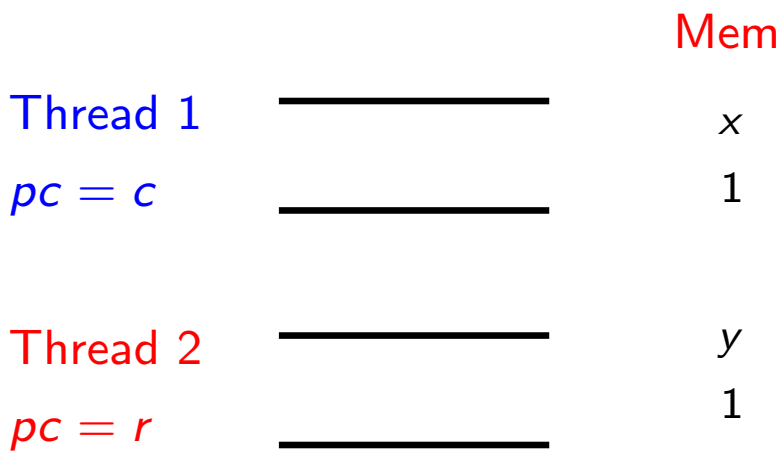


Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write in the buffer on that variable

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

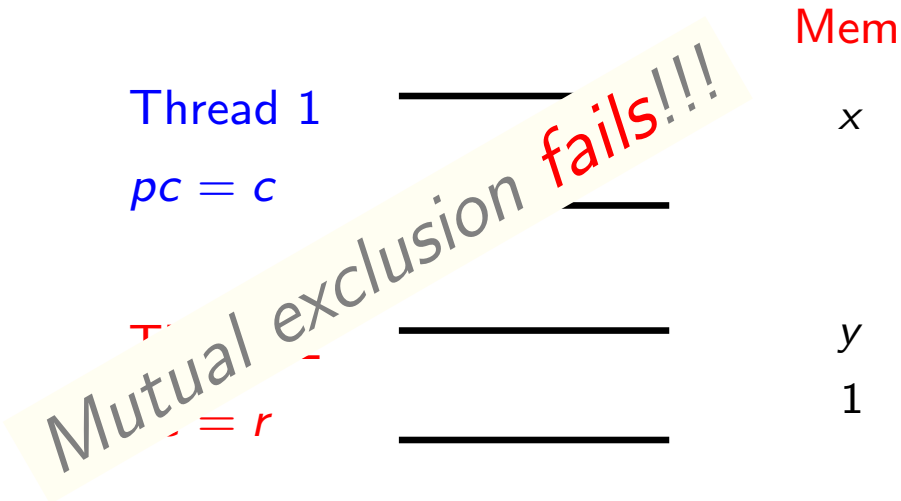


Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write in the buffer on that variable

$x = y = 0$

Thread 1	Thread 2
$a: x = 1$	$p: y = 1$
$b: if(y == 0)\{$	$q: if(x == 0)\{$
$c: \quad crit.sect.1$	$r: \quad crit.sect.2$
$d: \}$	$s: \}$



Verification Required?!

Relaxed executions may lead to bad behaviour

Verification Required?!

Relaxed executions may lead to bad behaviour

If this is the real world, why does anything work?

Verification Required?!

Relaxed executions may lead to bad behaviour

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993] If a program is data-race-free, then SC and TSO semantics coincide.

Verification Required?!

Relaxed executions may lead to bad behaviour

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993] If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Verification Required?!

Relaxed executions may lead to bad behaviour

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993] If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

Verification Required?!

Relaxed executions may lead to bad behaviour

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993] If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

Performance-critical code has data races

Verification Required?!

Relaxed executions may lead to bad behaviour

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993] If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

Performance-critical code has data races

Concurrency libraries Operating systems HPC@Fraunhofer ITWM

Verification Required?!

Relaxed executions may lead to bad behaviour

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993] If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

Performance-critical code has data races

Concurrency libraries Operating systems HPC@Fraunhofer ITWM

This is where our verification techniques apply

Outline

Reachability

[Atig, Bouajjani, Burckhardt, Musuvathi, POPL'10]

State Reachability Problem

Consider a memory model *MM*

State Reachability Problem for *MM*

Input: *Program P and a (control + memory) state s.*

Problem: *Is s reachable when P is run under MM?*

State Reachability Problem

Consider a memory model *MM*

State Reachability Problem for *MM*

Input: *Program P and a (control + memory) state s.*

Problem: *Is s reachable when P is run under MM?*

Decidability / Complexity ?

Each thread is finite-state

- For the SC memory model, this problem is PSPACE-complete

State Reachability Problem

Consider a memory model *MM*

State Reachability Problem for *MM*

Input: *Program P and a (control + memory) state s.*

Problem: *Is s reachable when P is run under MM?*

Decidability / Complexity ?

Each thread is finite-state

- For the SC memory model, this problem is PSPACE-complete
- Non-trivial for relaxed memory models:

$$Paths_{TSO}(P) = Closure_{TSO}(Paths_{SC}(P)) \text{ is non-regular}$$

Reachability

[Atig, Bouajjani, Burckhardt, Musuvathi, POPL'10]

Decidability:

Simulation of TSO semantics by Lossy Channel Systems

Decidability of State Reachability for TSO

Theorem [ABBM 2010]

The state reachability problem for TSO is reducible to the control-state reachability problem for LCS.

Decidability of State Reachability for TSO

Theorem [ABBM 2010]

The state reachability problem for TSO is reducible to the control-state reachability problem for LCS.

Theorem [Abdulla, Jonsson 1993]

The control-state reachability problem for LCS is decidable.

Decidability of State Reachability for TSO

Theorem [ABBM 2010]

The state reachability problem for TSO is reducible to the control-state reachability problem for LCS.

Theorem [Abdulla, Jonsson 1993]

The control-state reachability problem for LCS is decidable.

Corollary

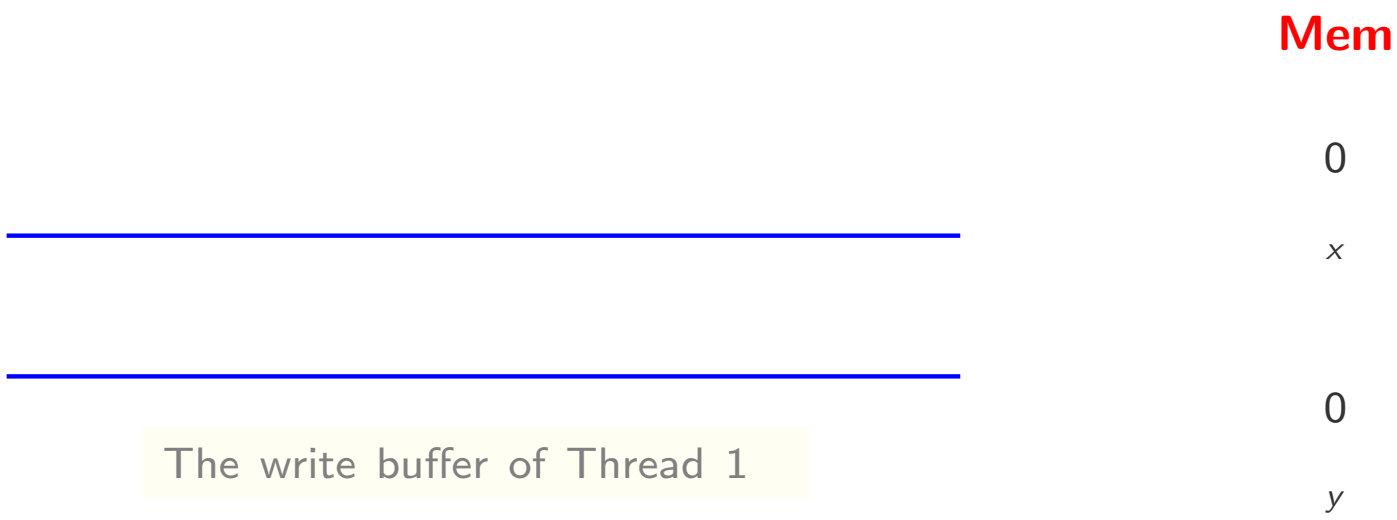
The state reachability problem for TSO is decidable.

From TSO to LCS 1/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

Write buffers are perfect FIFO channels

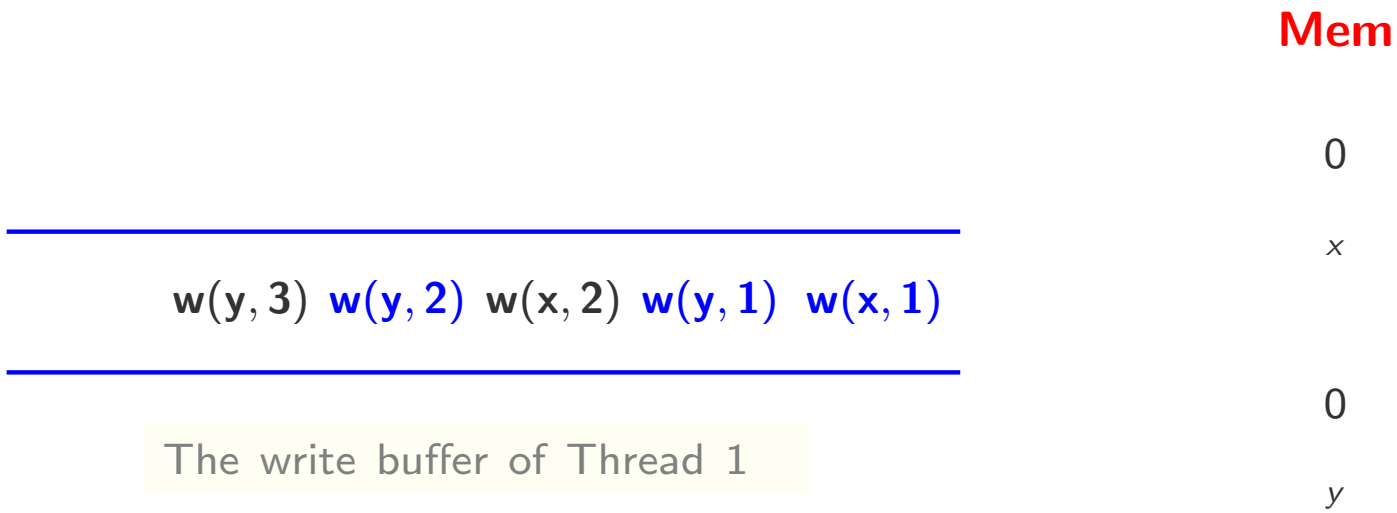


From TSO to LCS 1/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

Write buffers are perfect FIFO channels

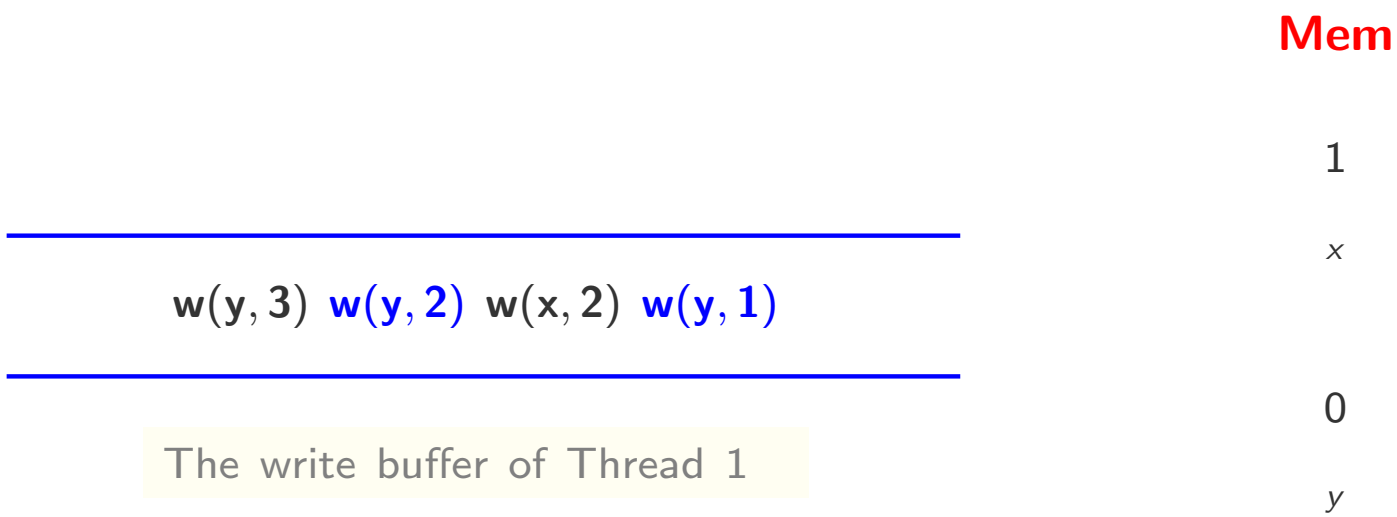


From TSO to LCS 1/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

Write buffers are perfect FIFO channels

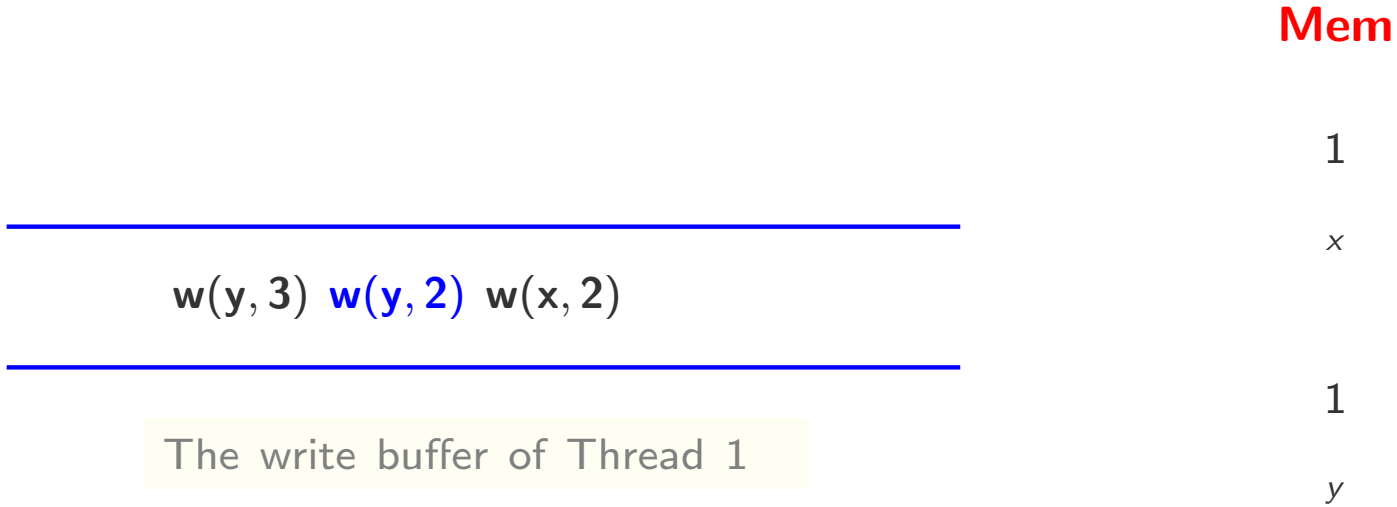


From TSO to LCS 1/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

Write buffers are perfect FIFO channels

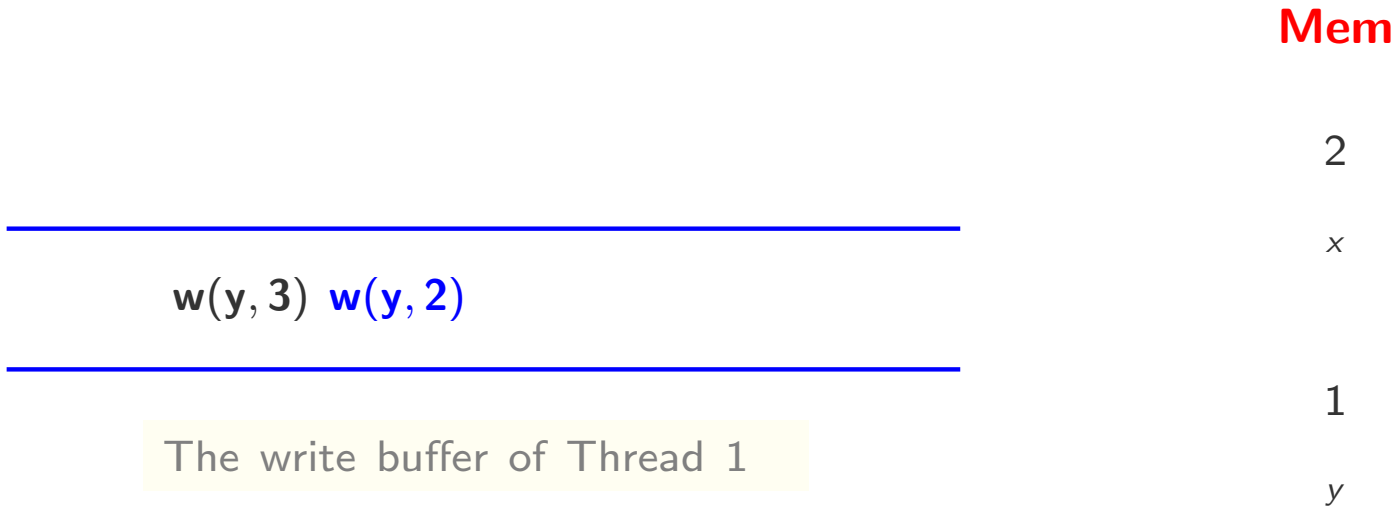


From TSO to LCS 1/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

Write buffers are perfect FIFO channels

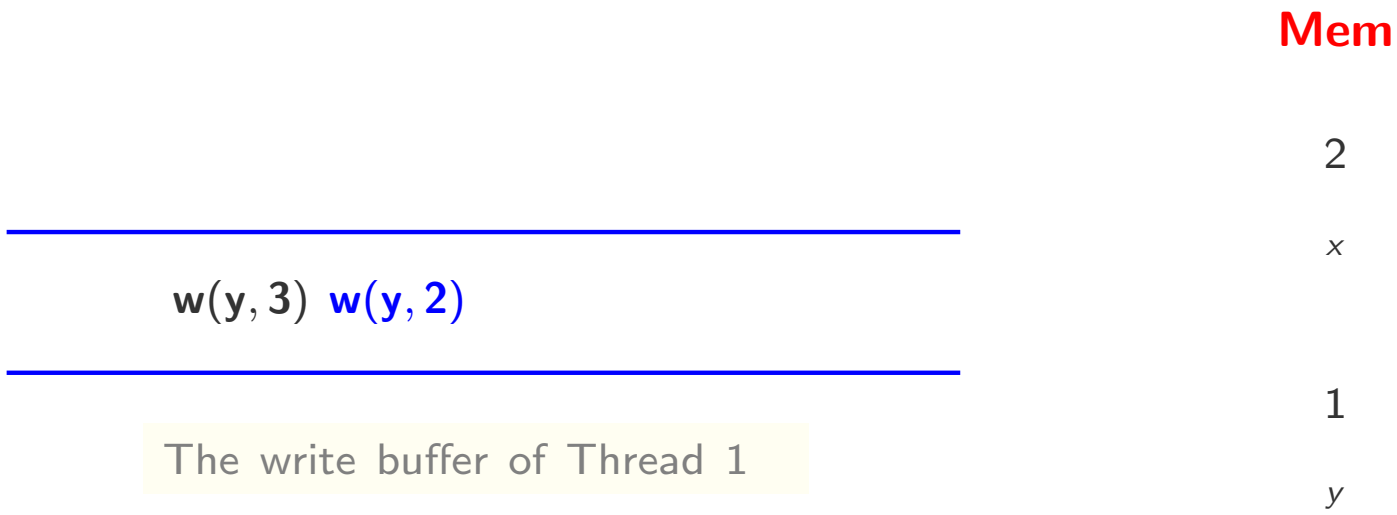


From TSO to LCS 1/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

Write buffers are perfect FIFO channels



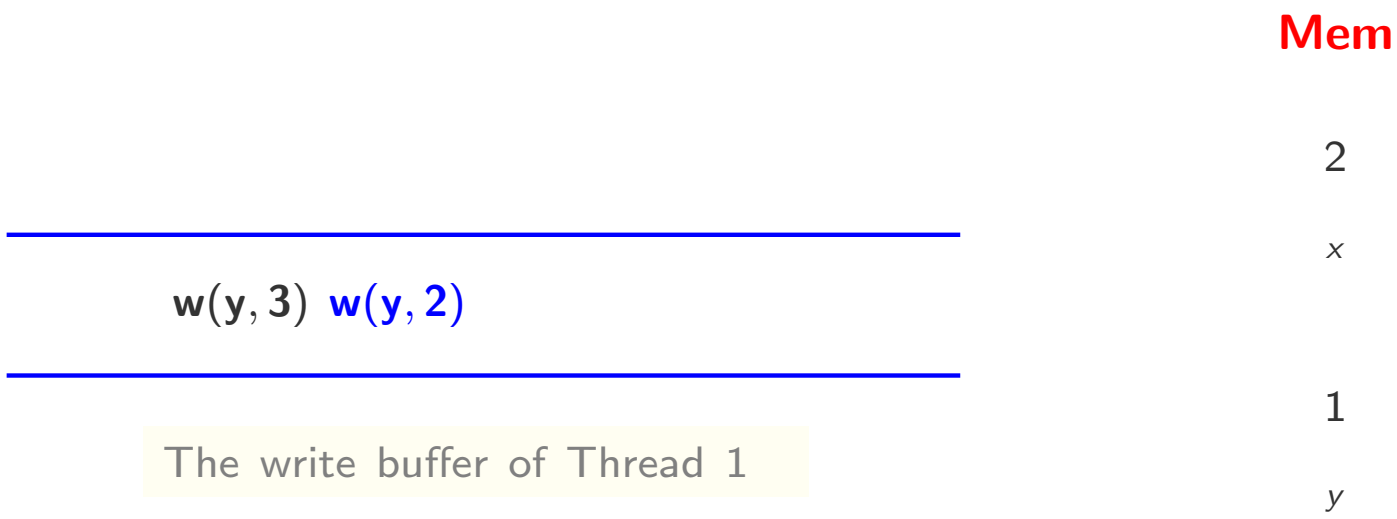
Thread 2 reads x = 2

From TSO to LCS 1/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

Write buffers are perfect FIFO channels



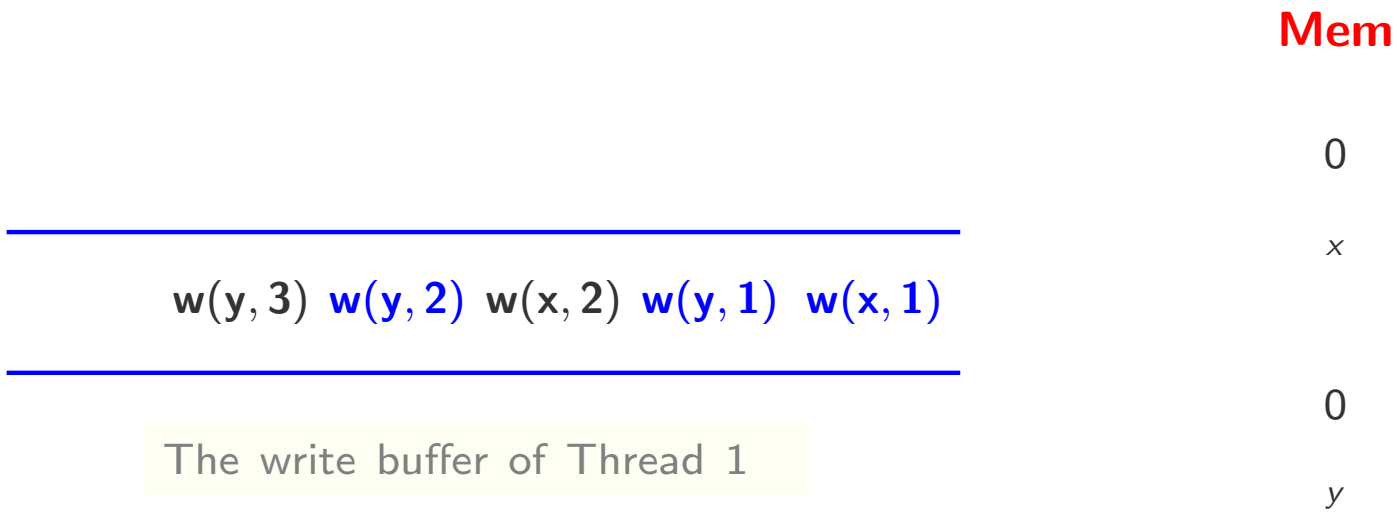
Thread 2 deadlocks as y = 1

From TSO to LCS 2/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

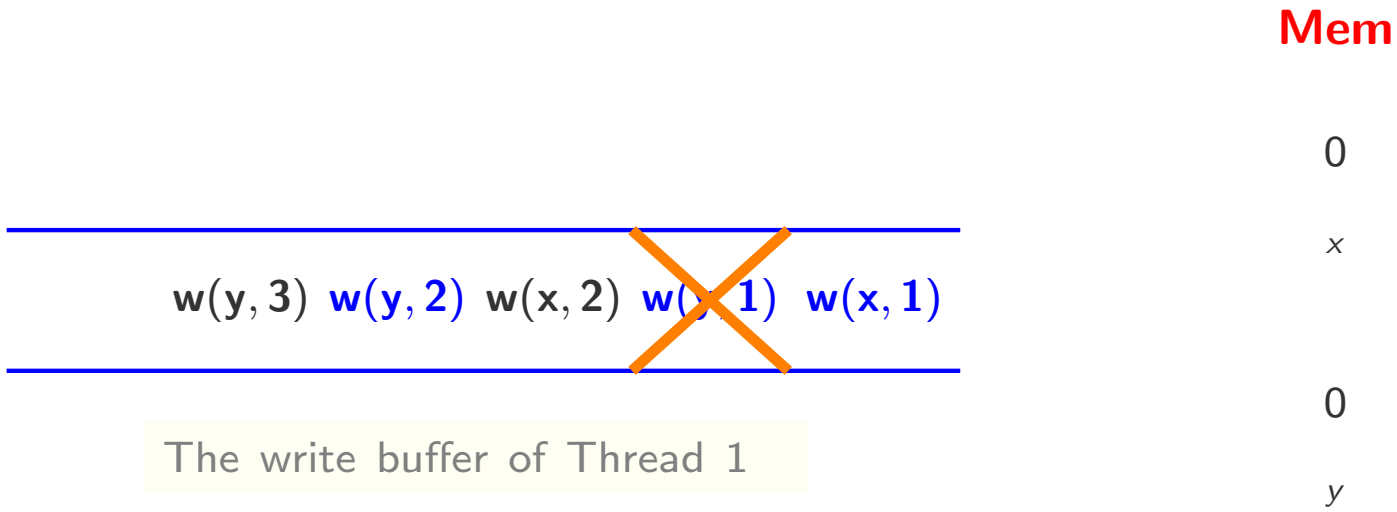


From TSO to LCS 2/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

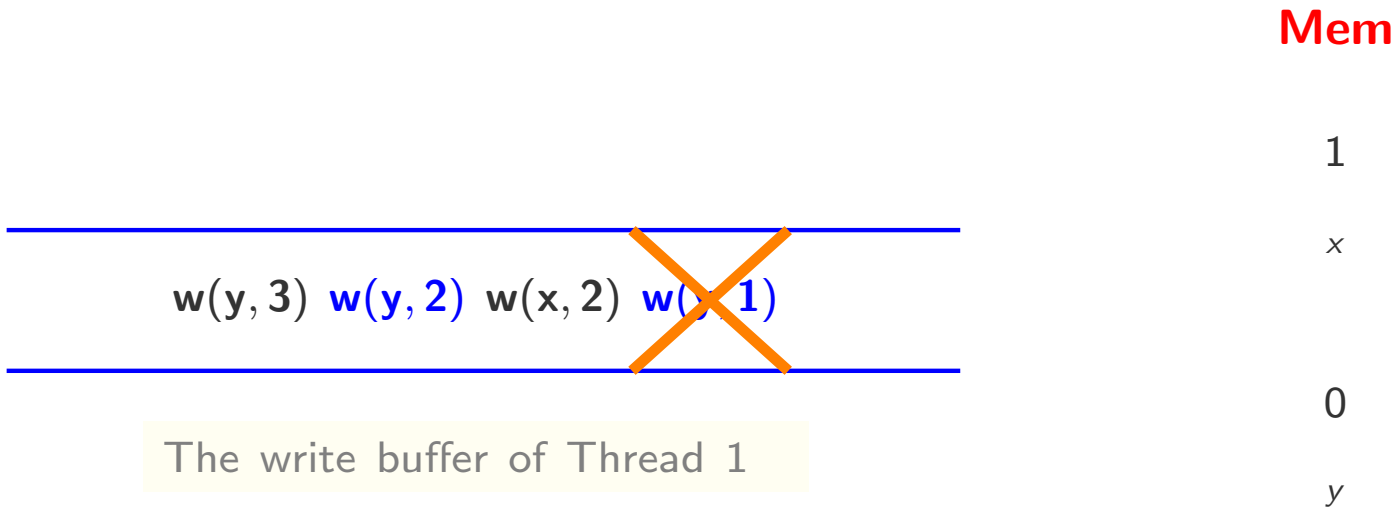


From TSO to LCS 2/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

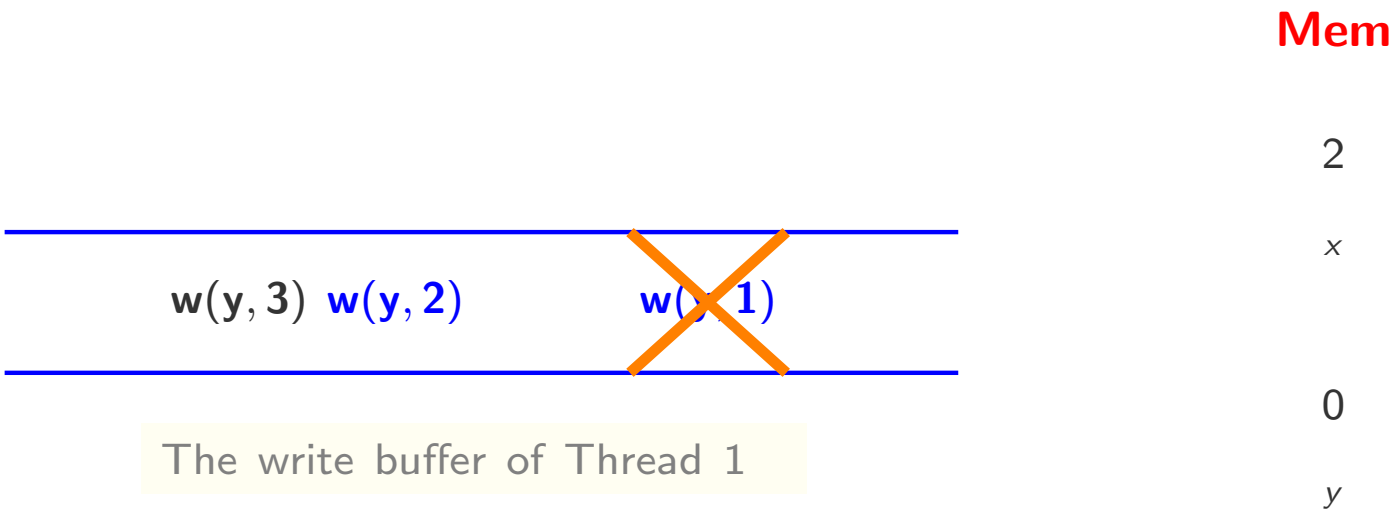


From TSO to LCS 2/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

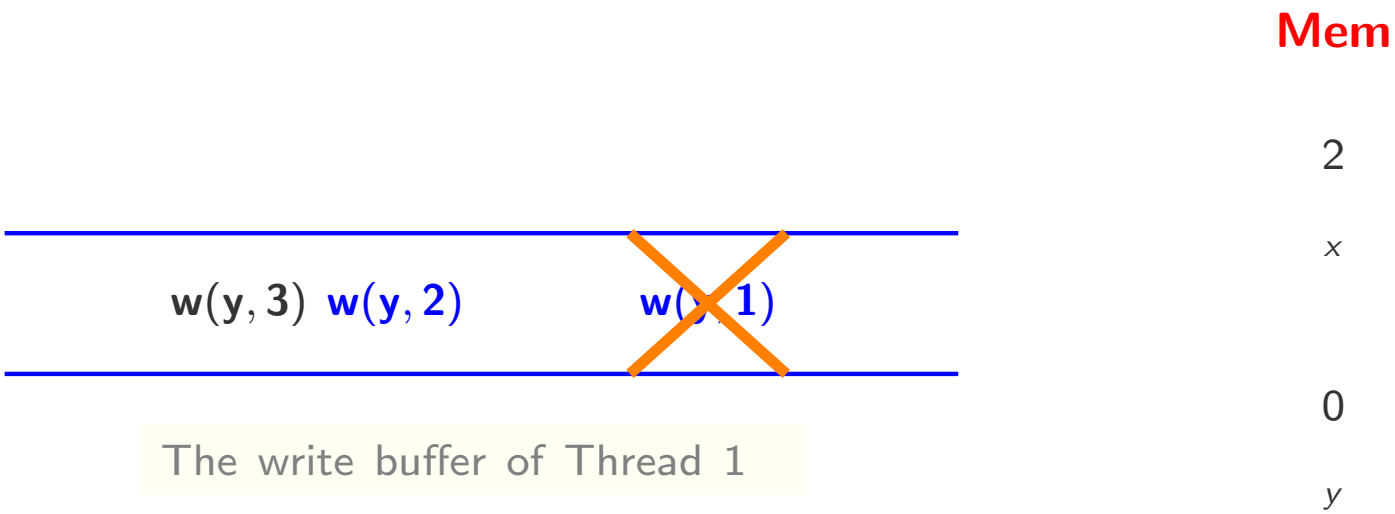


From TSO to LCS 2/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

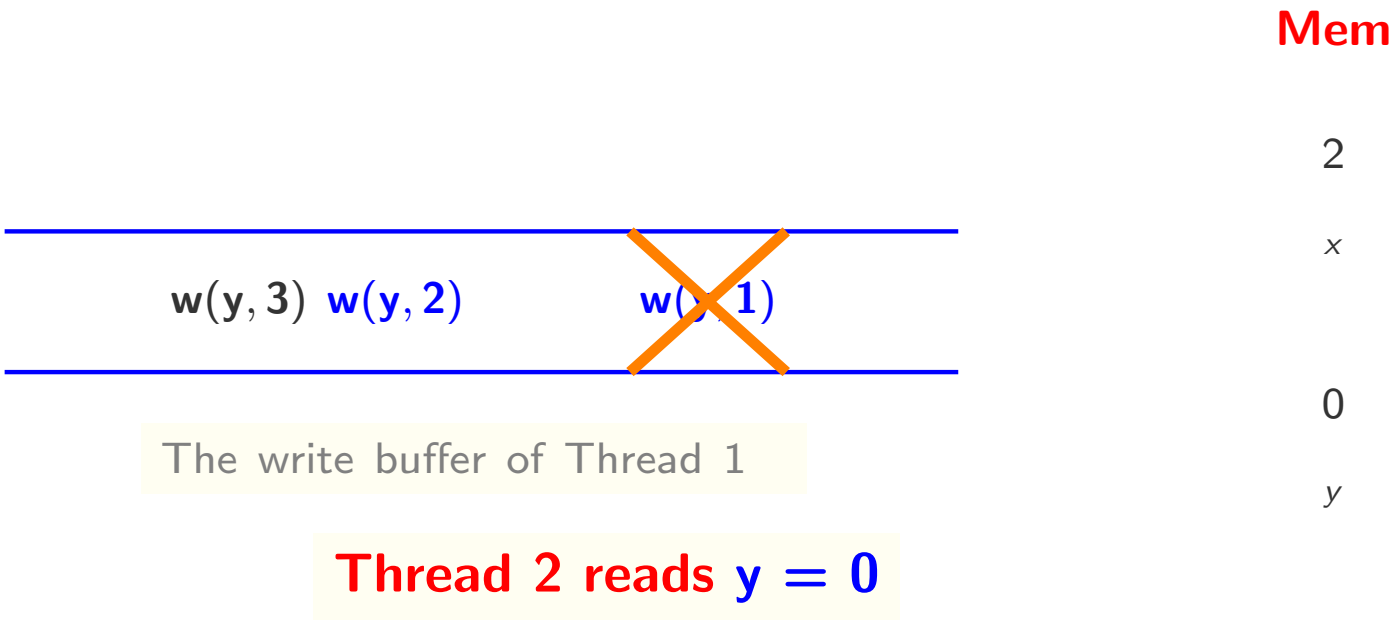


From TSO to LCS 2/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

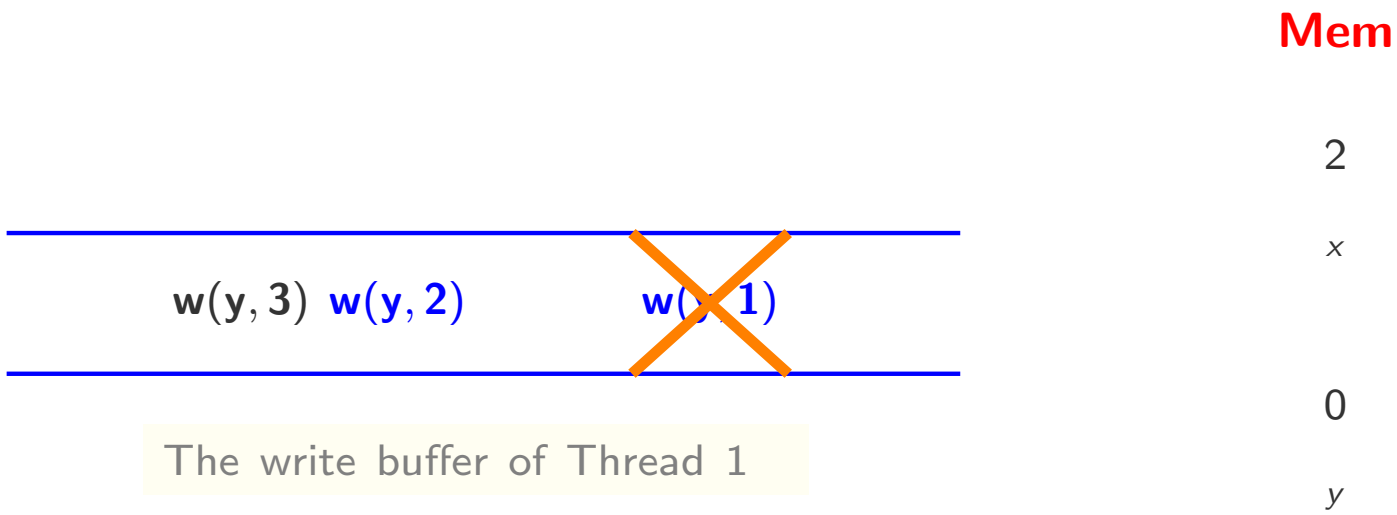


From TSO to LCS 2/5

Thread 1: $x = 1;\ y = 1;\ x = 2;\ y = 2;\ y = 3;$

Thread 2: $\text{if } (x == 2)\ \{\quad \text{if } (y == 0)\ \{\ldots\}\quad \}$

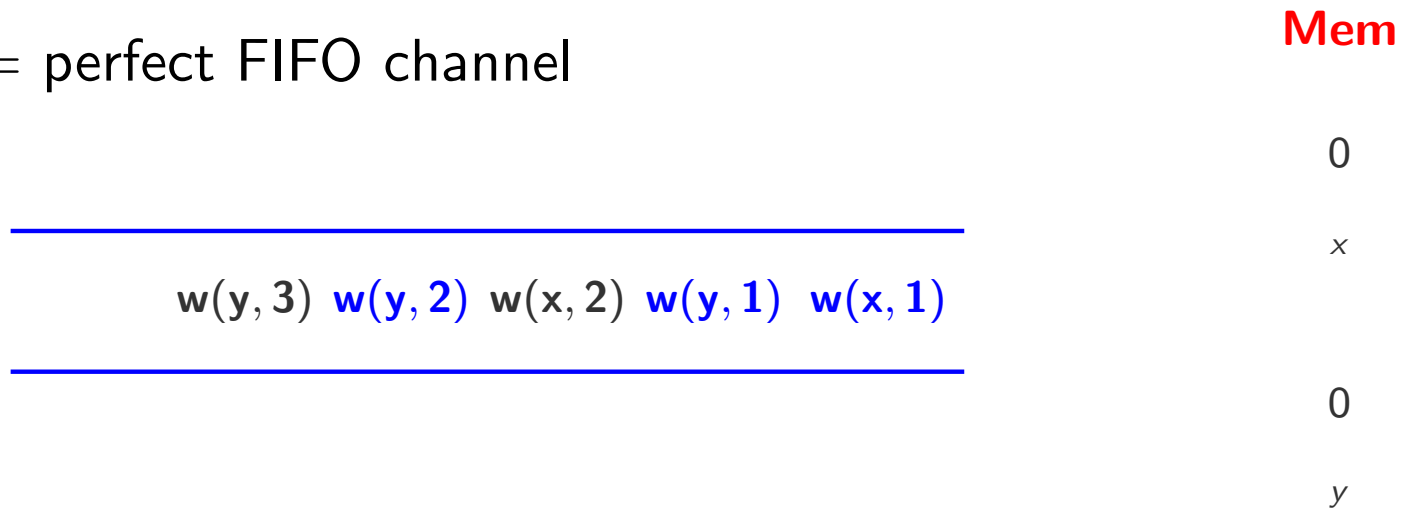
- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels



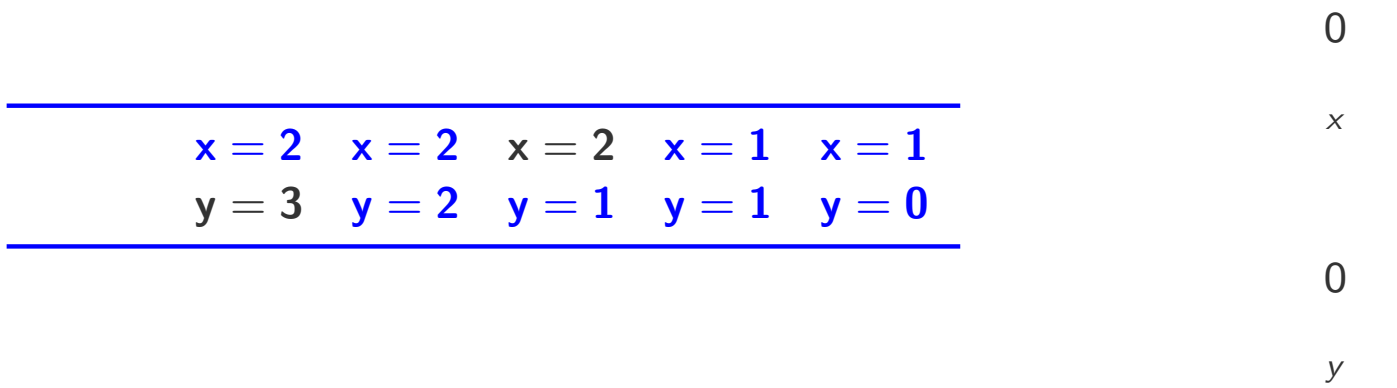
This is wrong! Lost the effect of w(y, 1).

From TSO to LCS 3/5

TSO buffer = perfect FIFO channel

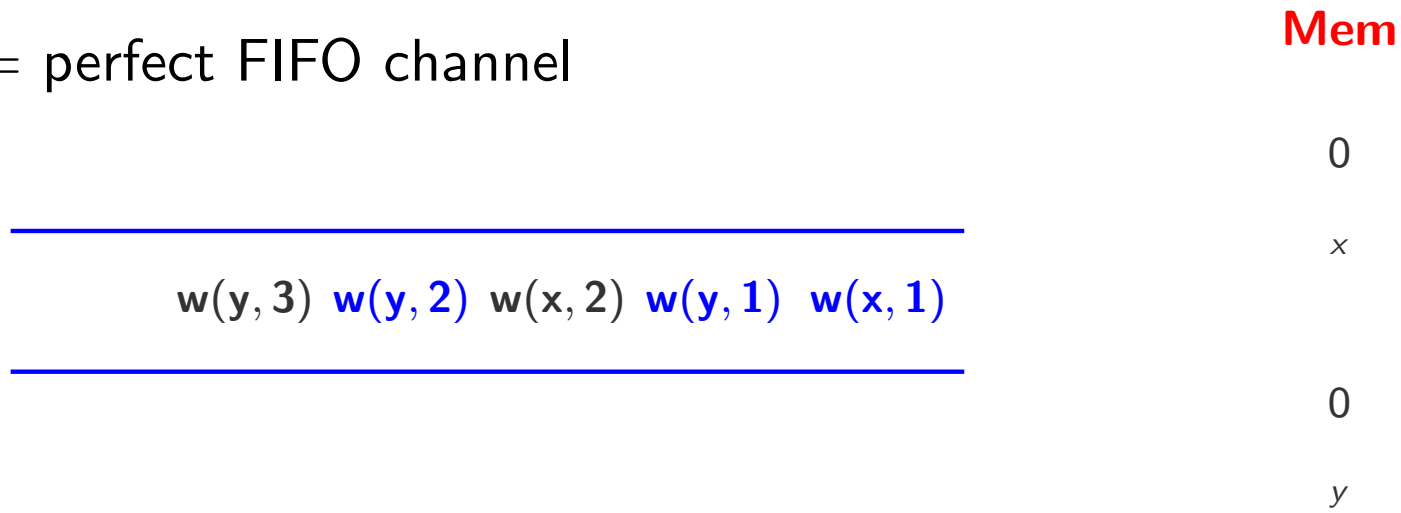


Channel = sequence of memory states + lossiness

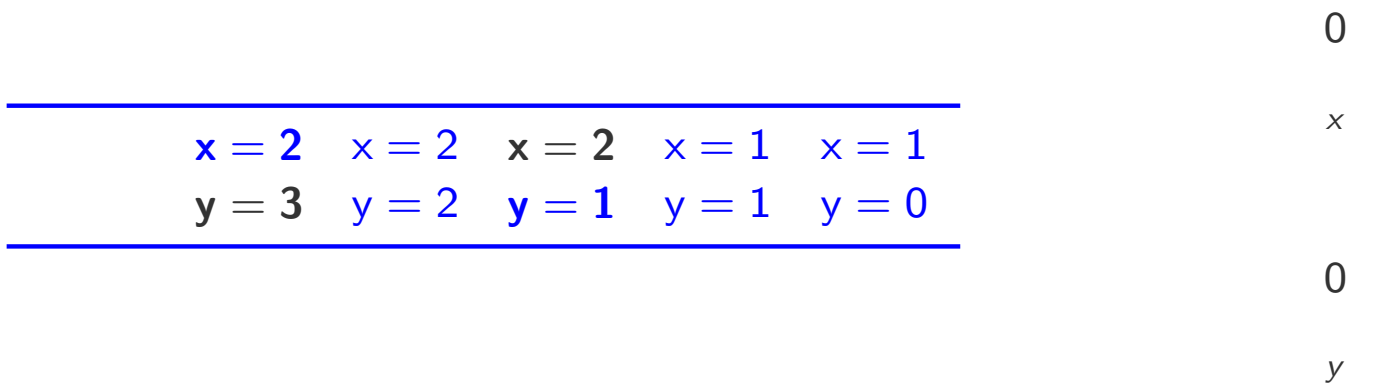


From TSO to LCS 3/5

TSO buffer = perfect FIFO channel



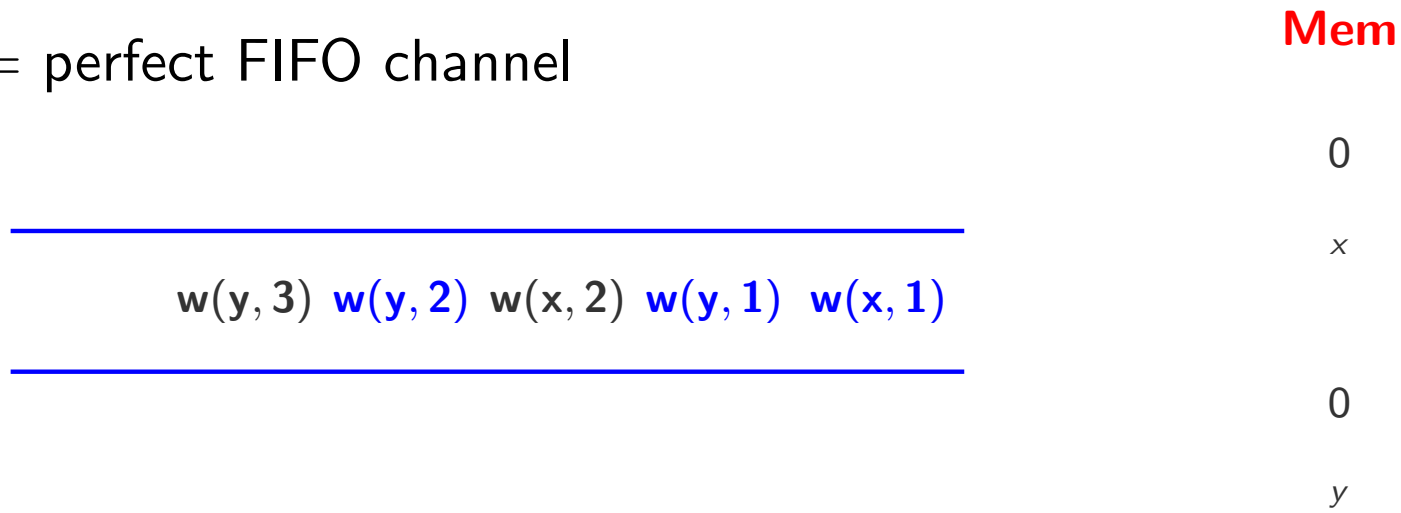
Channel = sequence of memory states + lossiness



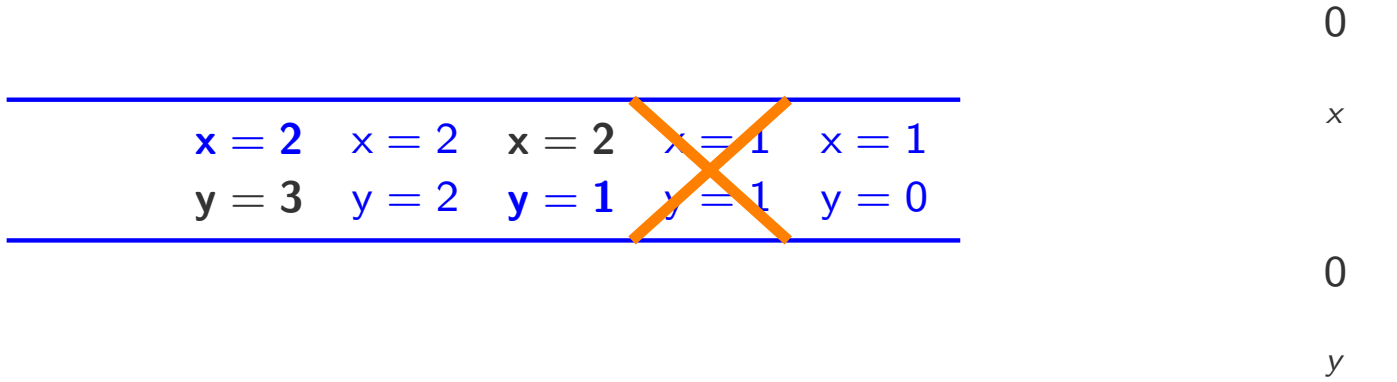
Lossiness = unobservable memory states

From TSO to LCS 3/5

TSO buffer = perfect FIFO channel



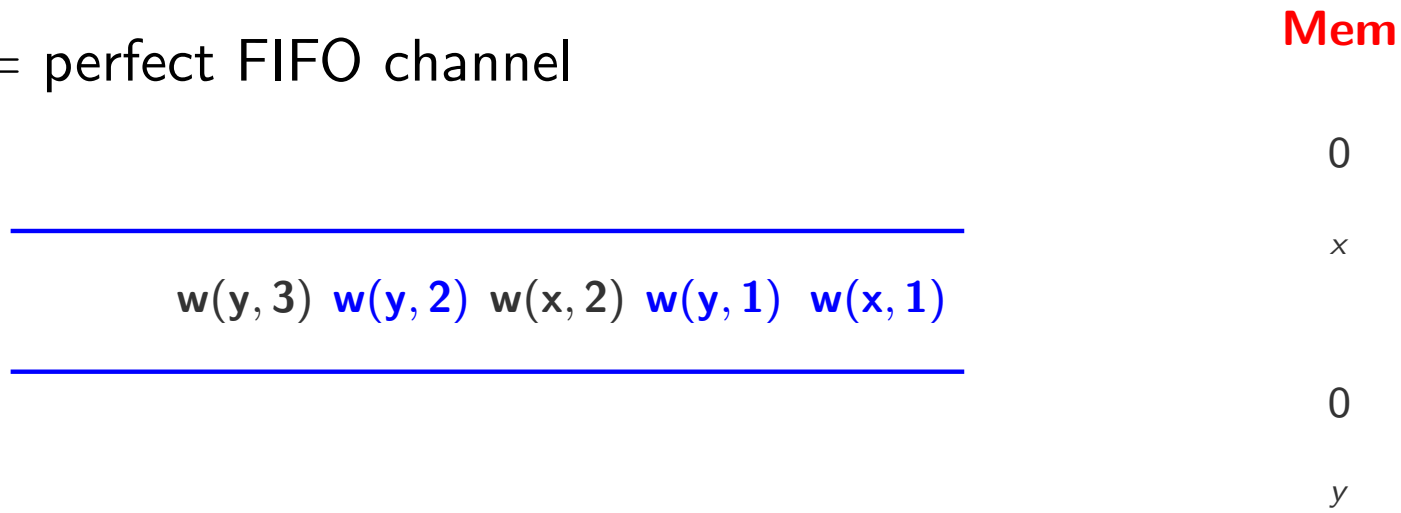
Channel = sequence of memory states + lossiness



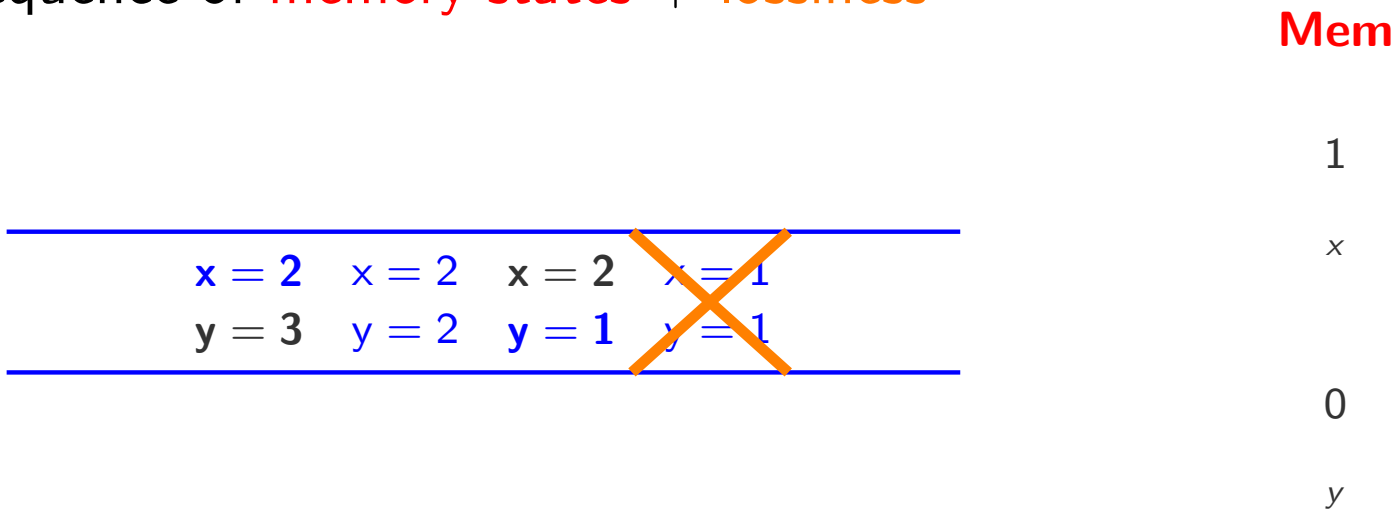
Lossiness = unobservable memory states

From TSO to LCS 3/5

TSO buffer = perfect FIFO channel



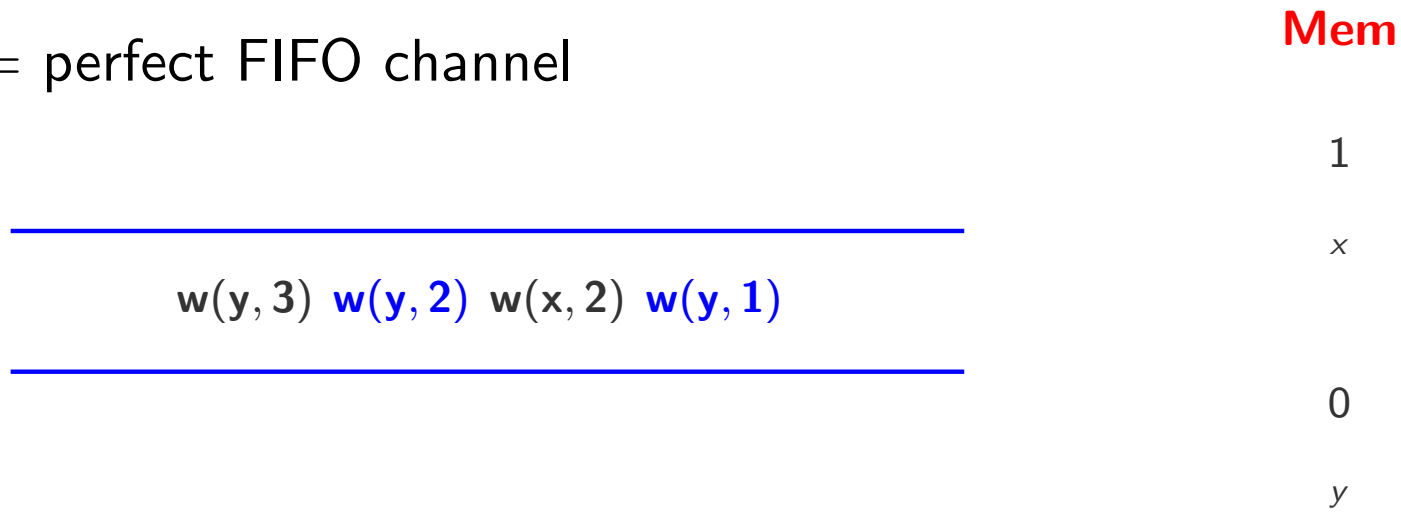
Channel = sequence of memory states + lossiness



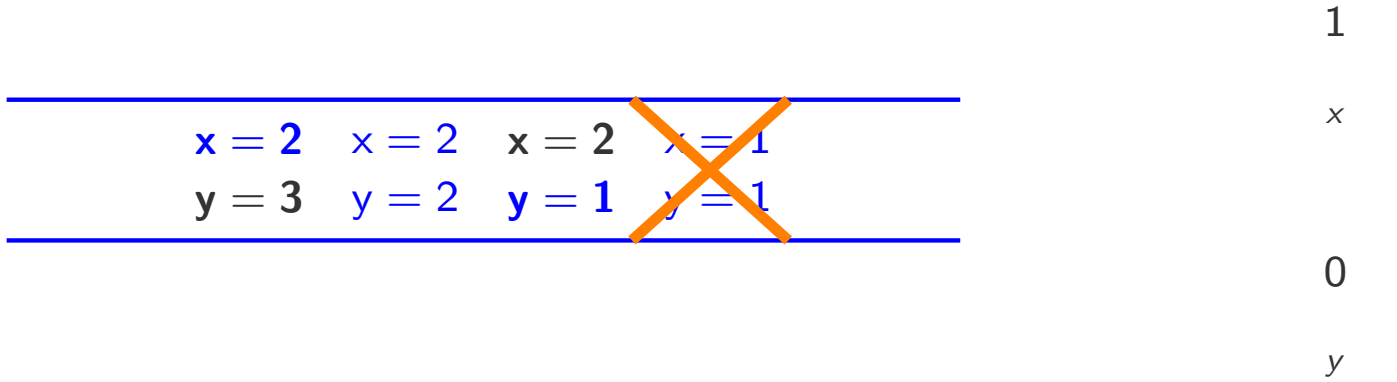
Lossiness = unobservable memory states

From TSO to LCS 3/5

TSO buffer = perfect FIFO channel



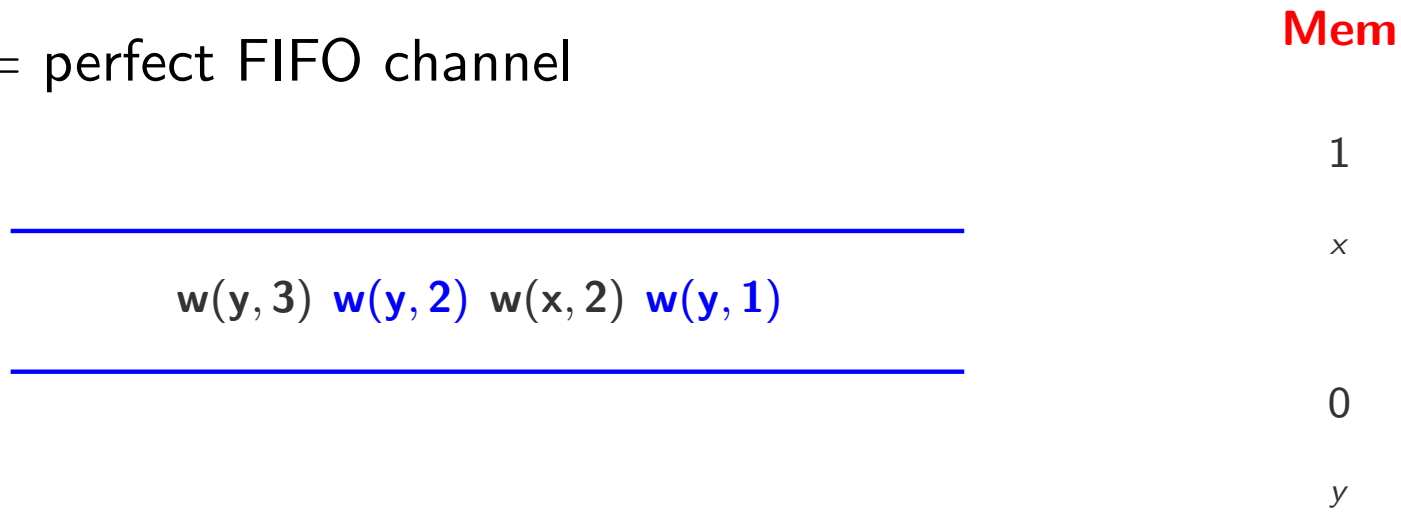
Channel = sequence of memory states + lossiness



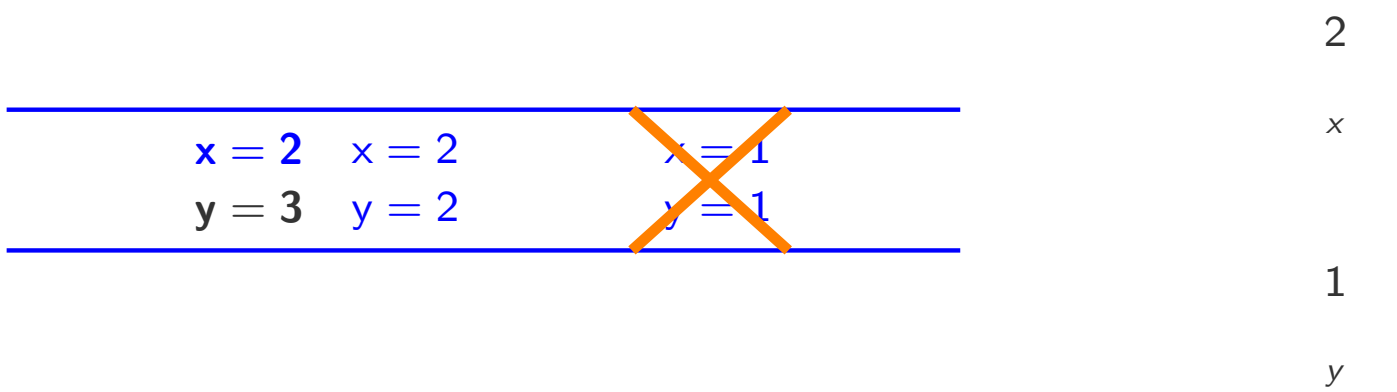
Lossiness = unobservable memory states

From TSO to LCS 3/5

TSO buffer = perfect FIFO channel



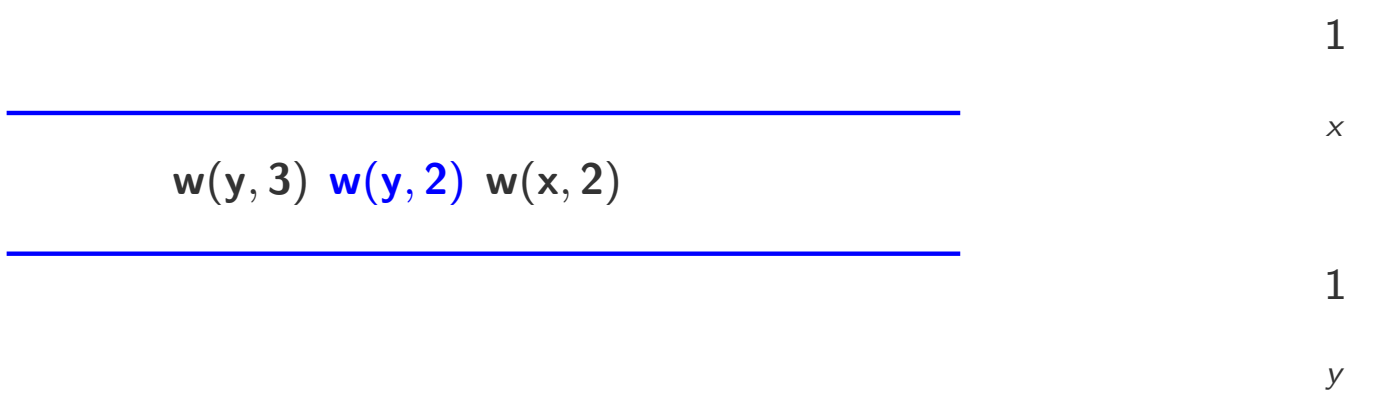
Channel = sequence of memory states + lossiness



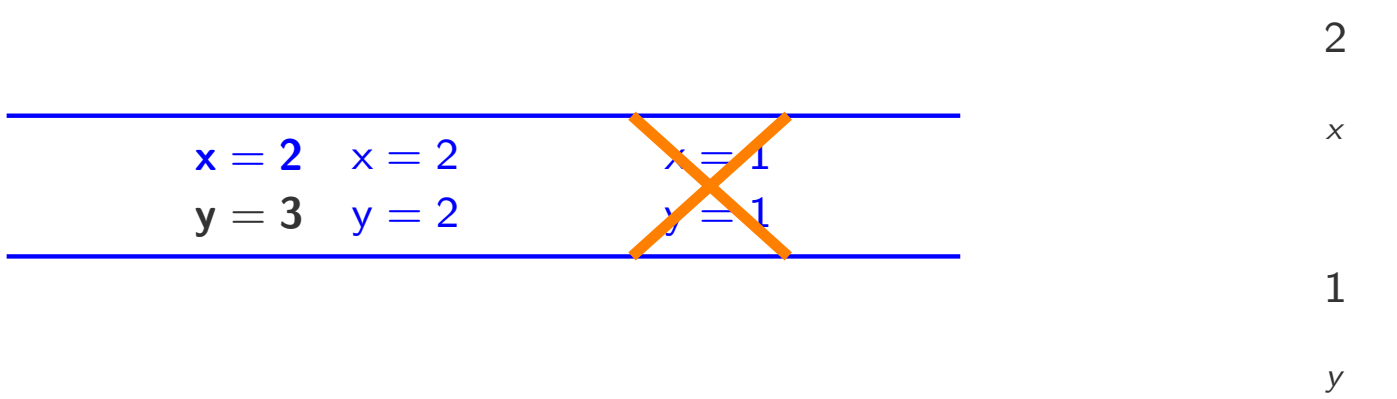
Lossiness = unobservable memory states

From TSO to LCS 3/5

TSO buffer = perfect FIFO channel



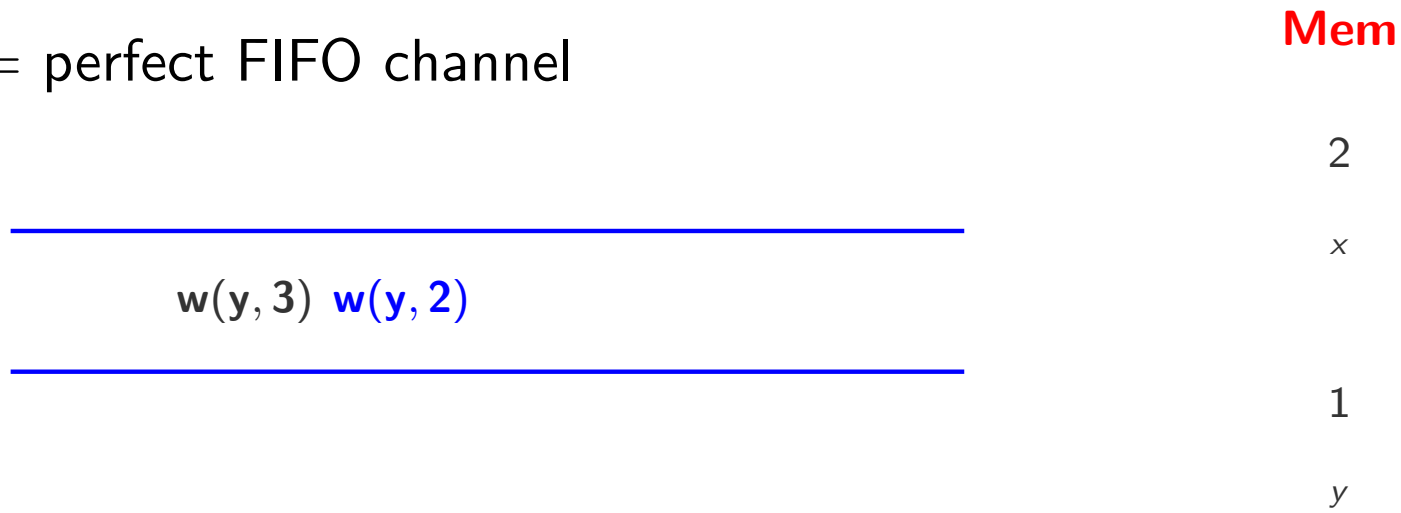
Channel = sequence of memory states + lossiness



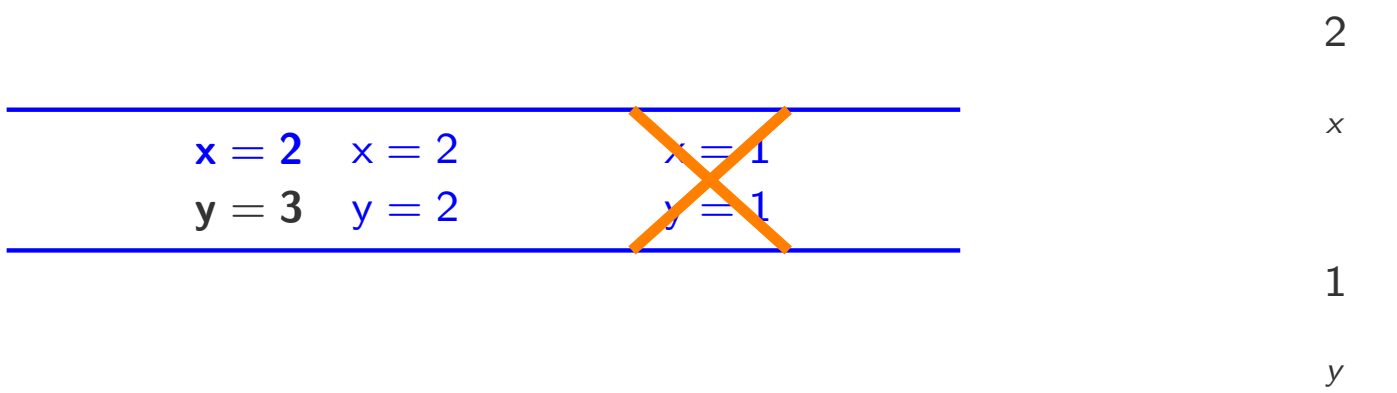
Lossiness = unobservable memory states

From TSO to LCS 3/5

TSO buffer = perfect FIFO channel



Channel = sequence of memory states + lossiness



Lossiness = unobservable memory states

From TSO to LCS 4/5

- *Write:* *Compute a new memory state; send it to the channel*
- *Read:* *Check the channel/memory*
- *Memory update:* *Receive a state; copy it to the memory*

From TSO to LCS 4/5

Problem: Interference between threads?

- *Write: Compute a new memory state; send it to the channel*
- *Read: Check the channel/memory*
- *Memory update: Receive a state; copy it to the memory*

From TSO to LCS 4/5

Problem: Interference between threads?

Each thread guesses writes of other threads

- *Write: Compute a new memory state; send it to the channel*
- *Read: Check the channel/memory*
- *Memory update: Receive a state; copy it to the memory*
- *Guessed Write: Send the guessed state to the channel*

From TSO to LCS 4/5

Problem: Interference between threads?

Each thread guesses writes of other threads

- *Write: Compute a new memory state; send it to the channel*
- *Read: Check the channel/memory*
- *Memory update: Receive a state; copy it to the memory*
- *Guessed Write: Send the guessed state to the channel*

Check that all threads agree on their guesses

From TSO to LCS 4/5

Problem: Interference between threads?

Each thread guesses writes of other threads

- *Write: Compute a new memory state; send it to the channel*
- *Read: Check the channel/memory*
- *Memory update: Receive a state; copy it to the memory*
- *Guessed Write: Send the guessed state to the channel*

Check that all threads agree on their guesses

Synchronization of the LCS over send actions

From TSO to LCS 5/5

Theorem [ABBM 2010]

The state reachability problem for TSO is reducible to the control-state reachability problem for LCS.

State Reachability: Conclusion

- Decidable for TSO (and beyond)
- But it is a hard problem — non-primitive recursive
- However, it is possible to have efficient analysis techniques
- Abstraction-based techniques:

  e.g., [Kuperstein, Vechev, Yahav, PLDI'11]

- Symbolic techniques:

  [Abdulla, Atig, Chen, Leonardson, Rezine, TACAS'12]
  [Linden, Wolper, SPIN'10'11]

Robustness

[Bouajjani, M., Möhlmann, ICALP'11]

[Bouajjani, Derevenetc, M., ESOP'13]

Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

What is the notion of behavior?

Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

What is the notion of behavior?

Trace Robustness:

TSO- and SC-traces are the same [Shasha, Snir'88]

Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

What is the notion of behavior?

Trace Robustness:

TSO- and SC-traces are the same [Shasha, Snir'88]

Good: Allows for quite relaxed behaviors

Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

What is the notion of behavior?

Trace Robustness:

TSO- and SC-traces are the same [Shasha, Snir'88]

Good: Allows for quite relaxed behaviors

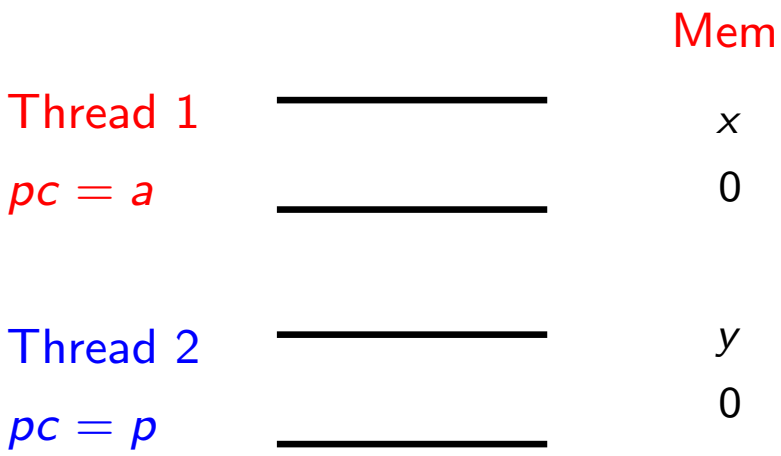
Very Good: Only PSPACE-complete

Traces 1/2

Computation = sequence of actions as seen by memory

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

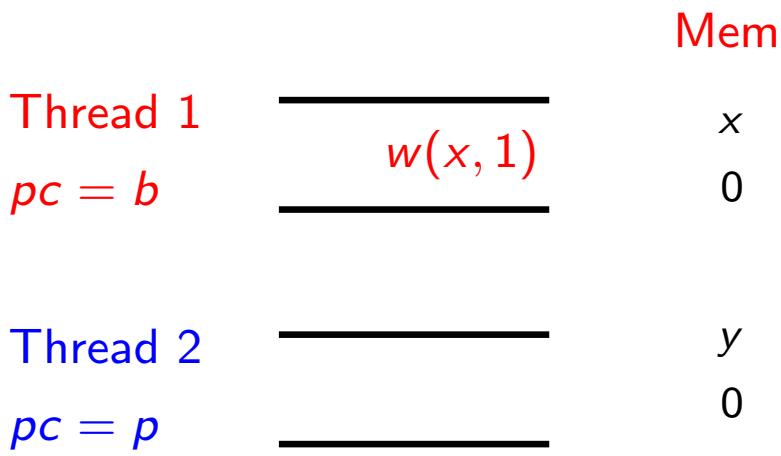


Traces 1/2

Computation = sequence of actions as seen by memory

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$

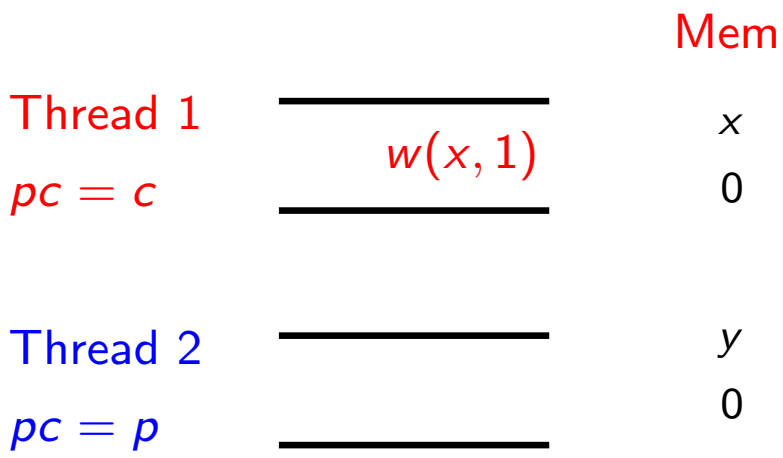


Traces 1/2

Computation = sequence of actions as seen by memory

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$



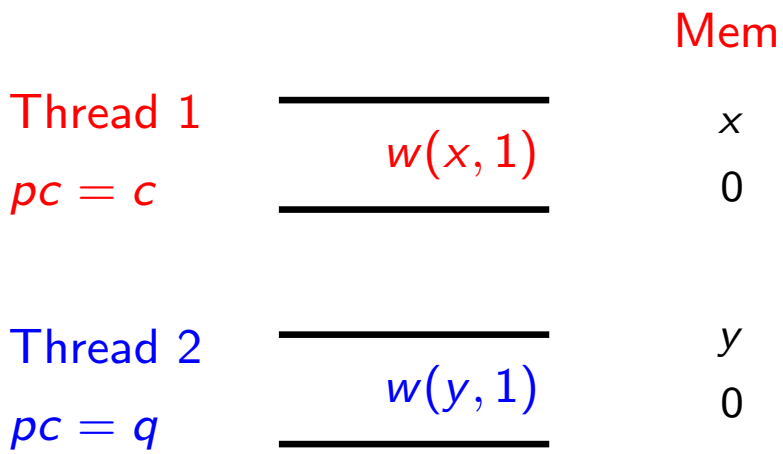
$r(y, 0)$

Traces 1/2

Computation = sequence of actions as seen by memory

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$



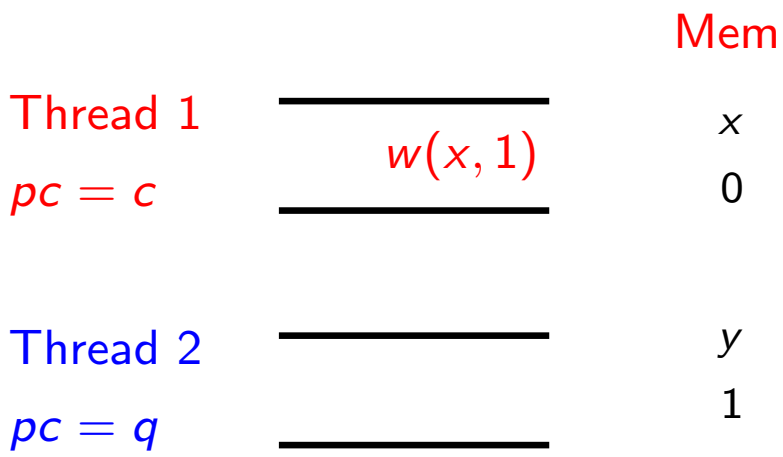
$r(y, 0)$

Traces 1/2

Computation = sequence of actions as seen by memory

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$



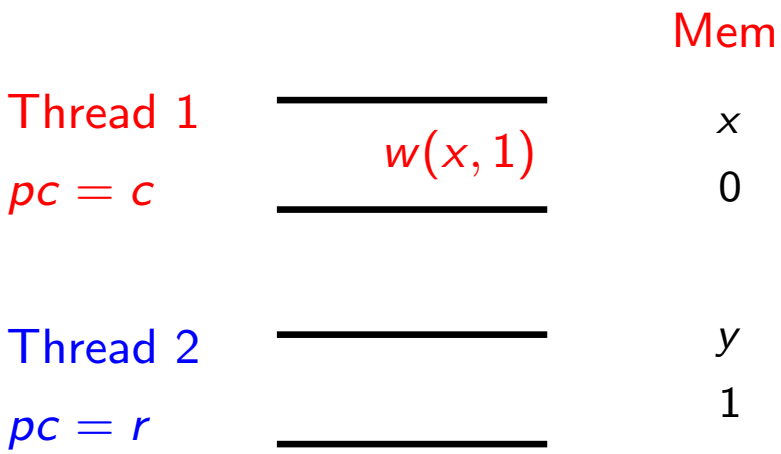
$r(y, 0) \cdot w(y, 1)$

Traces 1/2

Computation = sequence of actions as seen by memory

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$



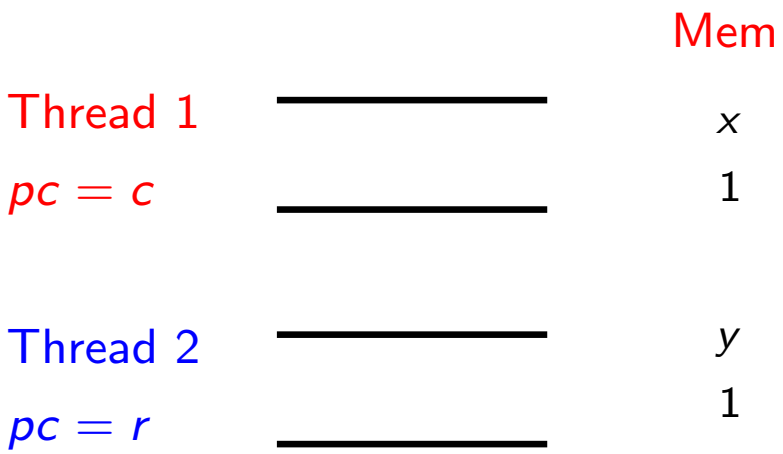
$$r(y,0) \cdot w(y,1) \cdot r(x,0)$$

Traces 1/2

Computation = sequence of actions as seen by memory

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$



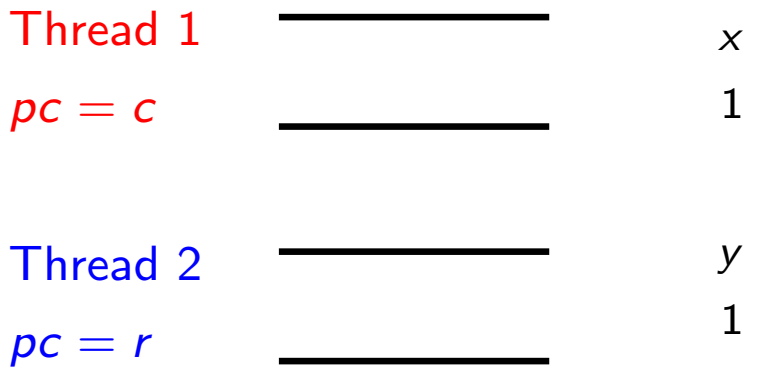
$$r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1)$$

Traces 1/2

Computation = sequence of actions as seen by memory

$x = y = 0$

Thread 1	Thread 2
$a : x = 1$	$p : y = 1$
$b : if(y == 0)\{$	$q : if(x == 0)\{$
$c : \quad crit.sect.1$	$r : \quad crit.sect.2$
$d : \}$	$s : \}$



$r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1)$

Traces 2/2

Traces abstract computations to happens before dependencies

$\mathsf{Trace}(r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1))$

Traces 2/2

Traces abstract computations to happens before dependencies

- Program order: Order of actions issued by a thread

Trace($r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1)$)

Traces 2/2

Traces abstract computations to happens before dependencies

- Program order: Order of actions issued by a thread
- Store order: Order of writes to a variable

Trace($r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1)$)

Traces 2/2

Traces abstract computations to happens before dependencies

- Program order: Order of actions issued by a thread
- Store order: Order of writes to a variable
- Source relation: *write* is source of *read*.

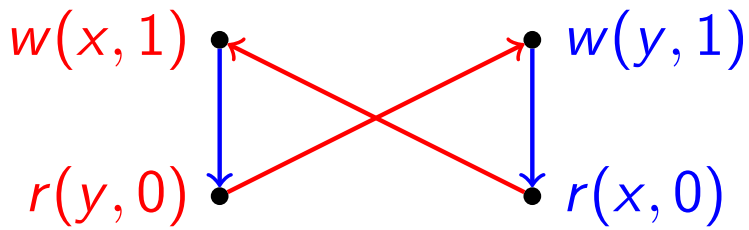
Trace($r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1)$)

Traces 2/2

Traces abstract computations to happens before dependencies

- Program order: Order of actions issued by a thread
- Store order: Order of writes to a variable
- Source relation: *write* is source of *read*.
- Conflict relation: *read* is overwritten by *write*.

Trace($r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1)$)



Trace Robustness Problem

Consider a memory model MM

Trace Robustness Problem against MM

Input: *Program* P.

Problem: *Does* $Traces_{MM}(P) \subseteq Traces_{SC}(P)$ *hold?*

Trace Robustness Problem

Consider a memory model MM

Trace Robustness Problem against MM

Input: *Program* P.

Problem: *Does* $Traces_{MM}(P) \subseteq Traces_{SC}(P)$ *hold?*

Decidability / Complexity ?

Trace Robustness Problem

Consider a memory model MM

Trace Robustness Problem against MM

> **Input**: *Program* P.
>
> **Problem**: *Does* $Traces_{MM}(P) \subseteq Traces_{SC}(P)$ *hold?*

Decidability / Complexity ?

Proof method

Theorem [Shasha, Snir 1988]

> *Program P is robust against MM iff all traces in* $Traces_{MM}(P)$ *are acyclic.*

Trace Robustness Problem

Consider a memory model *MM*

Trace Robustness Problem against *MM*

> **Input**: *Program* P.
>
> **Problem**: *Does* $Traces_{MM}(P) \subseteq Traces_{SC}(P)$ *hold?*

Decidability / Complexity ?

Proof method

Theorem [Shasha, Snir 1988]

> *Program P is robust against MM iff all traces in* $Traces_{MM}(P)$ *are acyclic.*

Shasha and Snir do not give an algorithm to find cyclic traces!

Robustness

[Bouajjani, M., Möhlmann, ICALP'11]

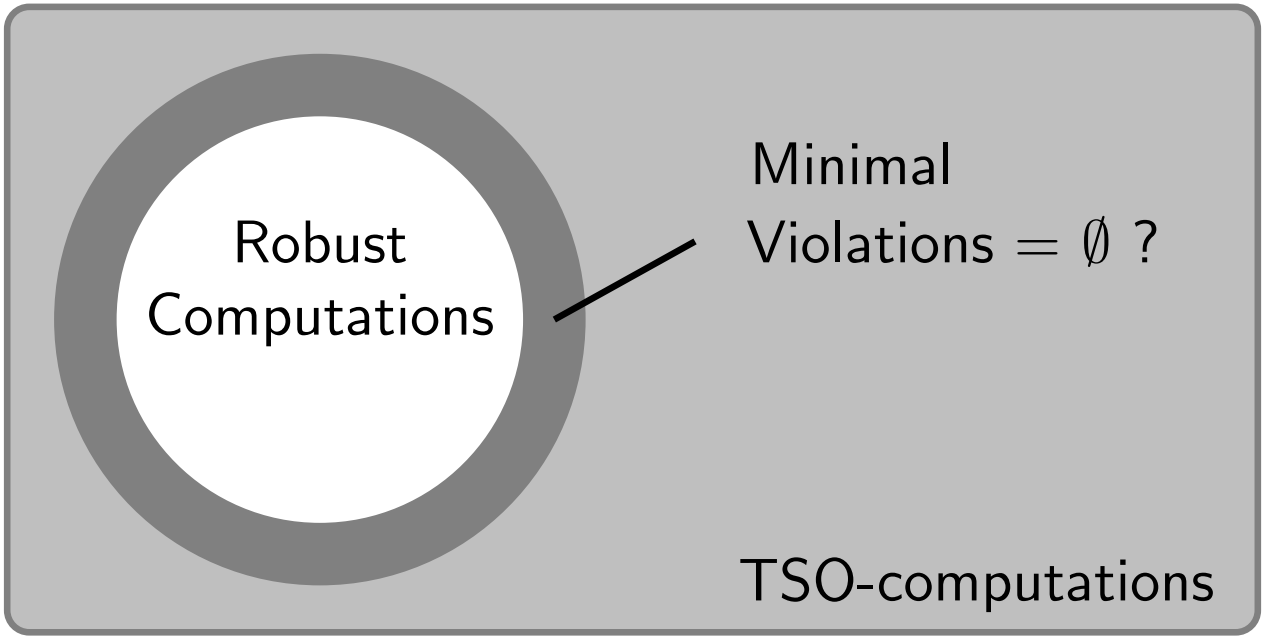
[Bouajjani, Derevenetc, M., ESOP'13]

Upper Bound:

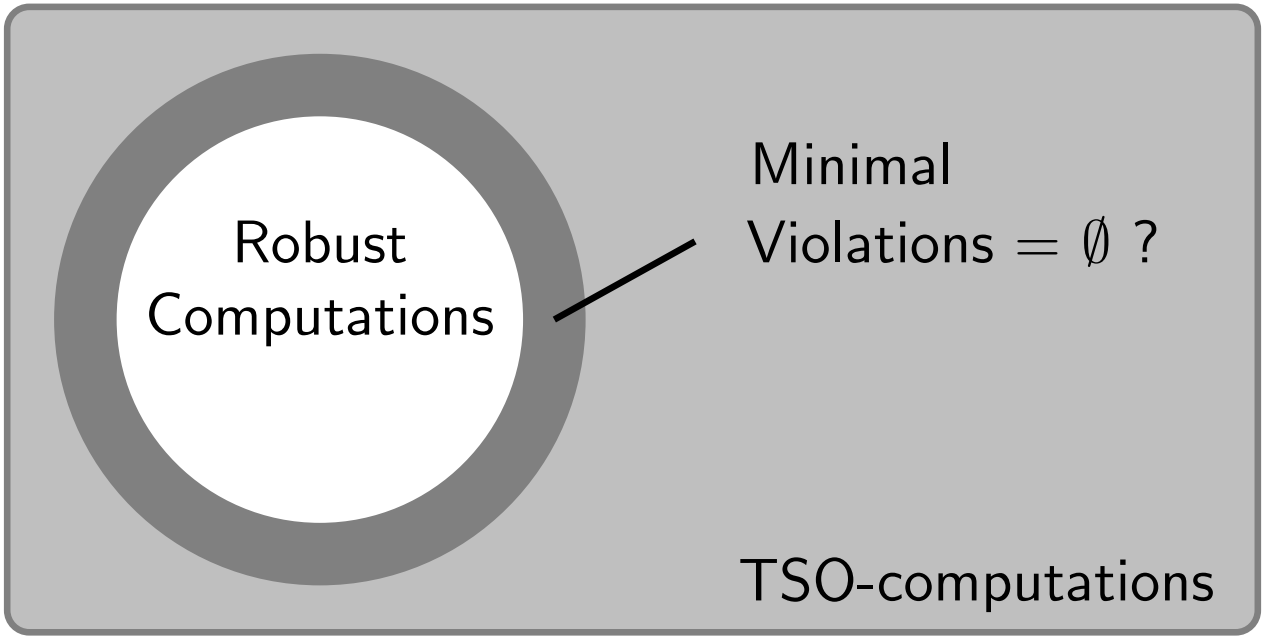
Combinatorics

From Robustness to SC Reachability

Deciding Robustness

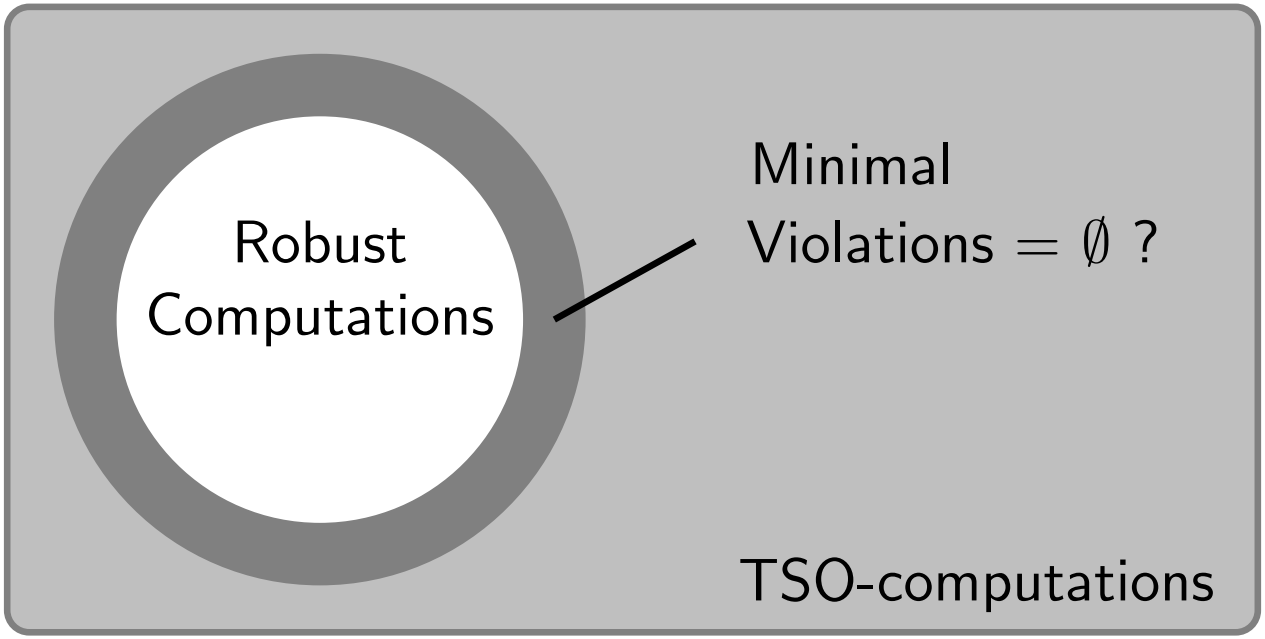


Deciding Robustness



Understand shape of minimal violations

Deciding Robustness



Understand shape of minimal violations

Check whether computation of this shape exists

Robustness

[Bouajjani, M., Möhlmann, ICALP'11]

[Bouajjani, Derevenetc, M., ESOP'13]

Upper Bound:

Combinatorics — Locality and Attacks

From Robustness to SC Reachability

Locality of Robustness 1/3

Goal: Locality

We can restrict ourselves to violations where only one thread reorders its actions.

Proof tool: Minimal violations

Number of inversions (out-of-program-order placements) minimal among all violating computations

Locality of Robustness 2/3

Consider minimal violation $\alpha \cdot b \cdot \beta \cdot a \cdot \gamma$ where b has overtaken a

Locality of Robustness 2/3

Consider minimal violation $\alpha \cdot b \cdot \beta \cdot a \cdot \gamma$ where b has overtaken a

Then b and a have happens before path through β

Locality of Robustness 2/3

Consider minimal violation $\alpha \cdot b \cdot \beta \cdot a \cdot \gamma$ where b has overtaken a

Then b and a have happens before path through β

Subword $b_1 \dots b_k$ with

$$b_i \rightarrow_{src/st/cf} b_{i+1} \qquad \text{or} \qquad b_i \rightarrow_p^+ b_{i+1}$$

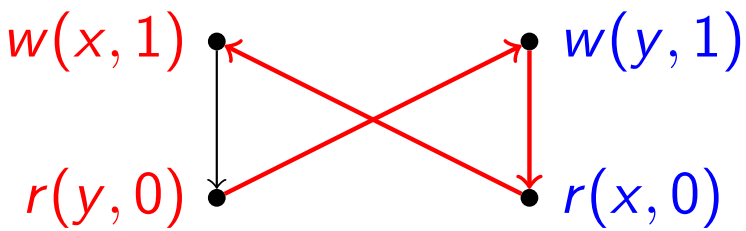
Locality of Robustness 2/3

Consider minimal violation $\alpha \cdot b \cdot \beta \cdot a \cdot \gamma$ where b has overtaken a

Then b and a have happens before path through β

Subword $b_1 \dots b_k$ with

$$b_i \rightarrow_{src/st/cf} b_{i+1} \qquad \text{or} \qquad b_i \rightarrow_p^+ b_{i+1}$$



$$\underbrace{r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1)}_{\rightarrow_{hb}}$$

Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:

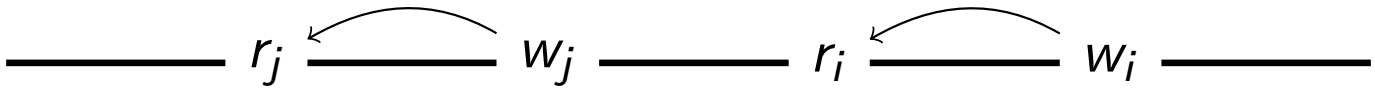
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 1: No interference



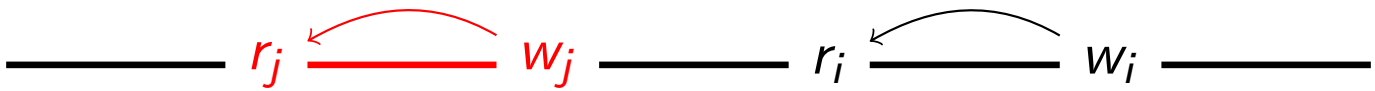
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 1: No interference



Lemma: happens before cycle $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$

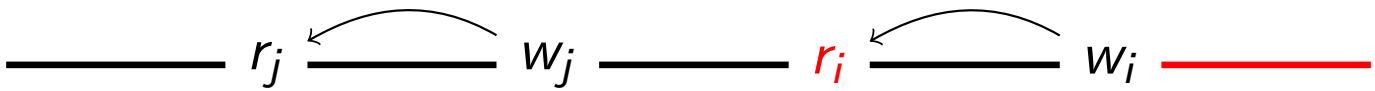
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 1: No interference



Lemma: happens before cycle $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$
Read r_i not involved, delete everything from r_i on

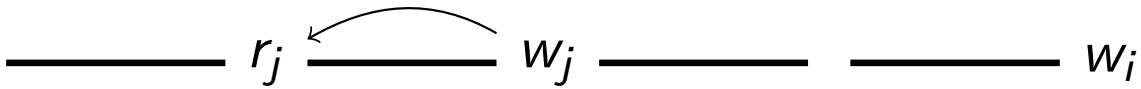
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 1: No interference



Lemma: happens before cycle $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$
Read r_i not involved, delete everything from r_i on
Saves a reordering, contradiction to minimality

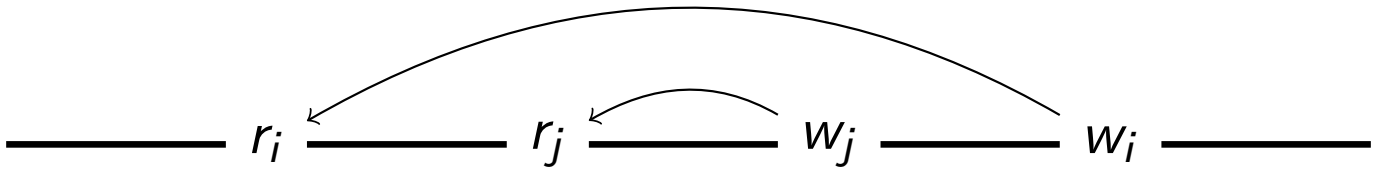
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 2: Overlap



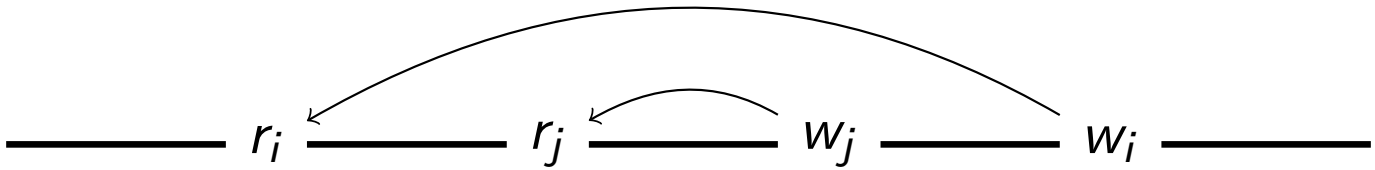
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 2: Overlap



Argumentation similar, delete again r_i

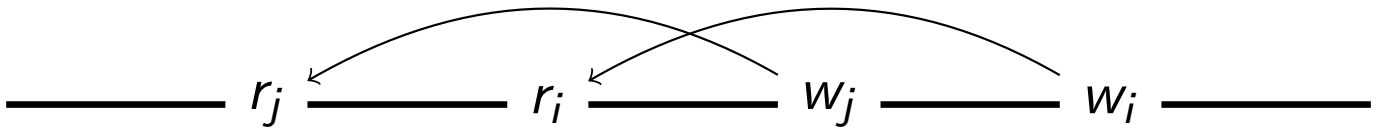
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 3: Interference



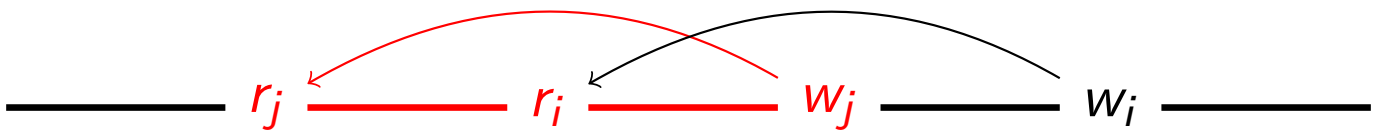
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 3: Interference



Lemma: happens before cycle $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$

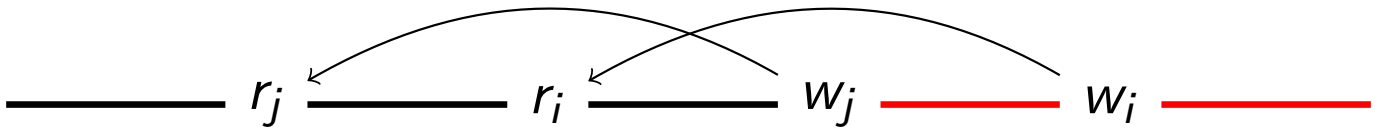
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 3: Interference



Lemma: happens before cycle $r_j \rightarrow^+_{hb} w_j \rightarrow^+_p r_j$
Only thread t_i may contribute, delete rest

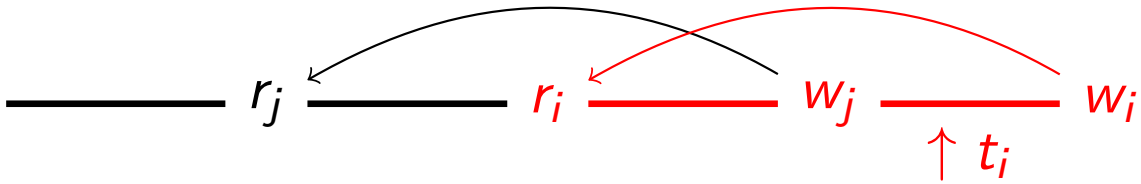
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 3: Interference



Lemma: happens before cycle $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$
Only thread t_i may contribute, delete rest
Lemma: happens before cycle $r_i \rightarrow_{hb}^{+} w_i \rightarrow_{p}^{+} r_i$

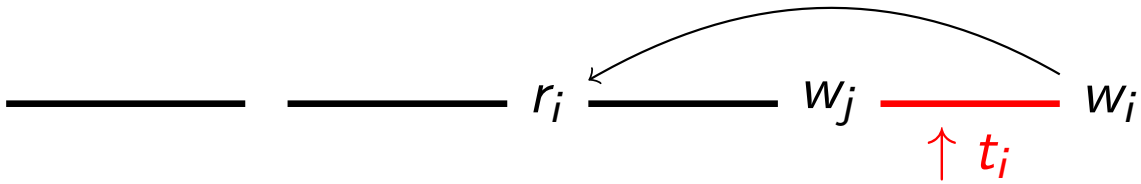
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads t_i and t_j:
Case 3: Interference



Lemma: happens before cycle $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$
Only thread t_i may contribute, delete rest
Lemma: happens before cycle $r_i \rightarrow_{hb}^{+} w_i \rightarrow_{p}^{+} r_i$
Read r_j not on this cycle, delete it, contradiction

Characterization of Robustness via Attacks 1/2

Reformulate Robustness

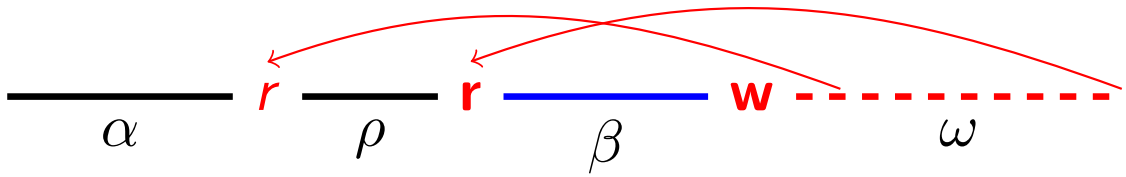
absence of feasible attacks

Characterization of Robustness via Attacks 1/2

Reformulate Robustness

absence of feasible attacks

If P is not robust, there are these violation:

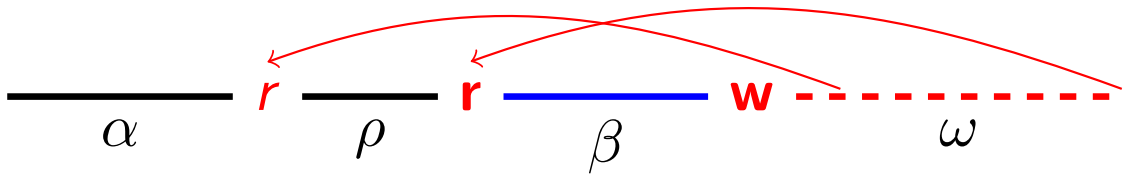


Characterization of Robustness via Attacks 1/2

Reformulate Robustness

absence of feasible attacks

If P is not robust, there are these violation:



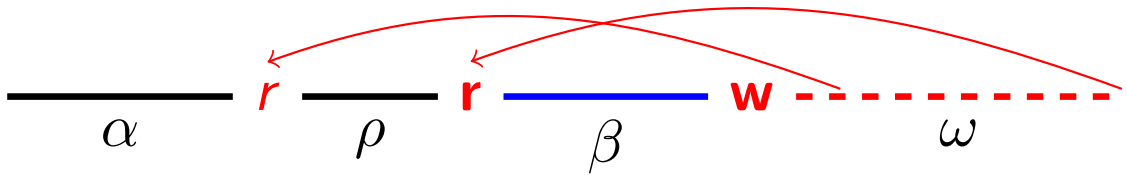
Attacker The thread that uses its buffer: only one by locality

Characterization of Robustness via Attacks 1/2

Reformulate Robustness

absence of feasible attacks

If P is not robust, there are these violation:



Attacker The thread that uses its buffer: only one by locality

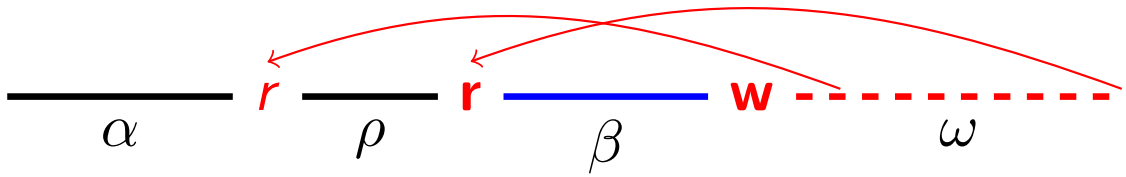
Helpers Remaining threads close cycle: $\mathbf{r} \rightarrow_{hb}^{+} \mathbf{w}$ $\mathbf{w} \rightarrow_{p}^{+} \mathbf{r}$

Characterization of Robustness via Attacks 1/2

Reformulate Robustness

absence of feasible attacks

If P is not robust, there are these violation:



Attacker The thread that uses its buffer: only one by locality

Helpers Remaining threads close cycle: $\mathbf{r} \rightarrow_{hb}^{+} \mathbf{w}$ $\mathbf{w} \rightarrow_{p}^{+} \mathbf{r}$

$$\underbrace{r(y,0) \cdot w(y,1) \cdot r(x,0) \cdot w(x,1)}_{\rightarrow_{hb}}$$

Characterization of Robustness via Attacks 2/2

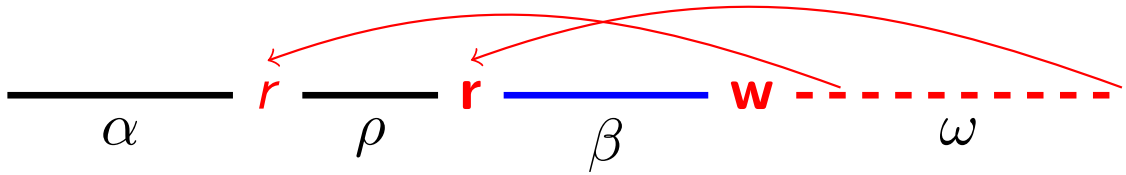
- Fix thread, write instruction, read instruction
- Given these parameters, find a violation as above

Characterization of Robustness via Attacks 2/2

- Fix thread, write instruction, read instruction
- Given these parameters, find a violation as above

Attack

- An attack is a triple $A = (thread, write, read)$
- A TSO witness for attack A is a computation as above:

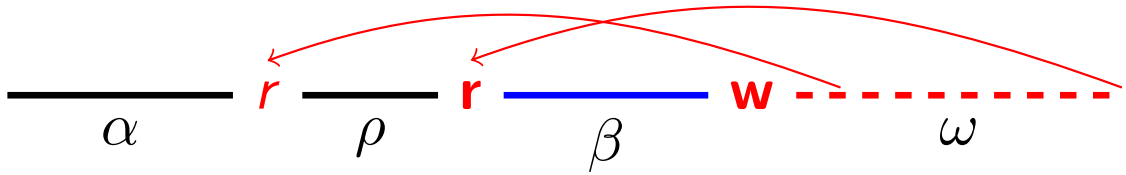


Characterization of Robustness via Attacks 2/2

- Fix thread, write instruction, read instruction
- Given these parameters, find a violation as above

Attack

- An attack is a triple $A = (thread, write, read)$
- A TSO witness for attack A is a computation as above:



Theorem [BDM'13]

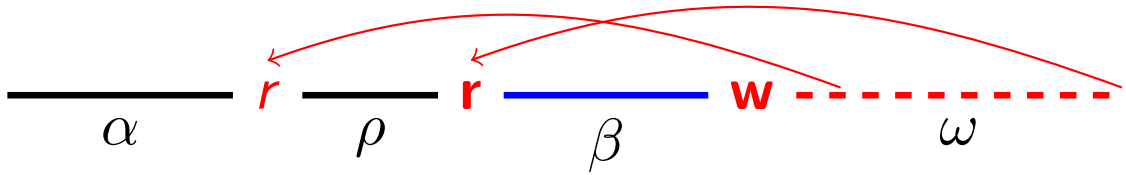
Program P is robust if and only if no attack has a TSO witness.

Characterization of Robustness via Attacks 2/2

- Fix thread, write instruction, read instruction
- Given these parameters, find a violation as above

Attack

- An attack is a triple $A = (thread, write, read)$
- A TSO witness for attack A is a computation as above:



Theorem [BDM'13]

Program P is robust if and only if no attack has a TSO witness.

The number of attacks is quadratic in the size of P.

Robustness

[Bouajjani, M., Möhlmann, ICALP'11]

[Bouajjani, Derevenetc, M., ESOP'13]

Upper Bound:

Combinatorics

From Robustness to SC Reachability

Finding TSO Witnesses with SC Reachability

TSO witnesses for attack A considerably restrict TSO behavior,

Finding TSO Witnesses with SC Reachability

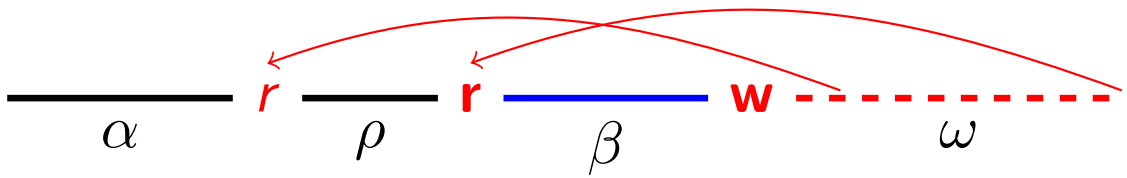
TSO witnesses for attack A considerably restrict TSO behavior,

enough to find TSO witnesses with SC reachability

Finding TSO Witnesses with SC Reachability

TSO witnesses for attack A considerably restrict TSO behavior,

enough to find TSO witnesses with SC reachability

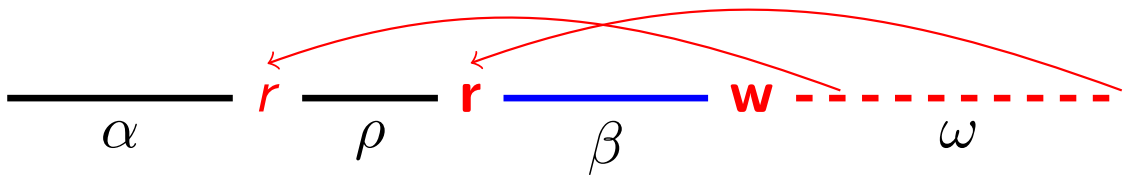


Let attacker execute under SC

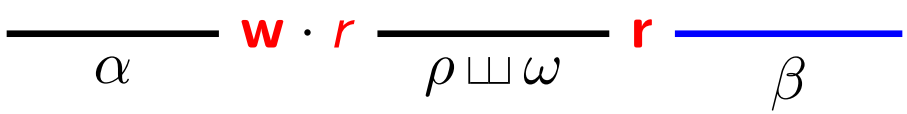
Finding TSO Witnesses with SC Reachability

TSO witnesses for attack A considerably restrict TSO behavior,

enough to find TSO witnesses with SC reachability



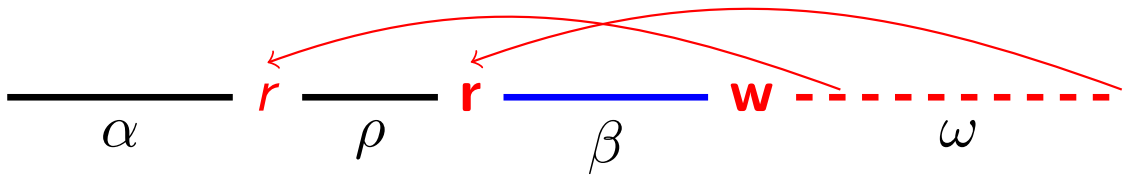
Let attacker execute under SC



Finding TSO Witnesses with SC Reachability

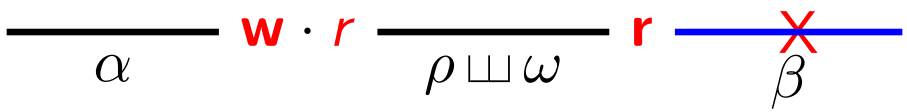
TSO witnesses for attack A considerably restrict TSO behavior,

enough to find TSO witnesses with SC reachability



Let attacker execute under SC

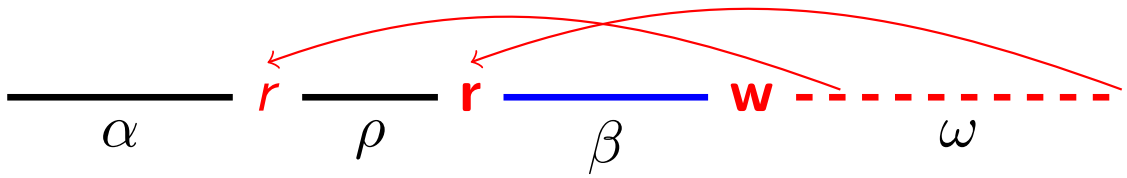
Problem Writes may conflict with helper reads



Finding TSO Witnesses with SC Reachability

TSO witnesses for attack A considerably restrict TSO behavior,

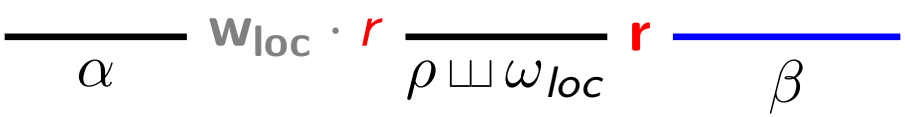
enough to find TSO witnesses with SC reachability



Let attacker execute under SC

Problem Writes may conflict with helper reads

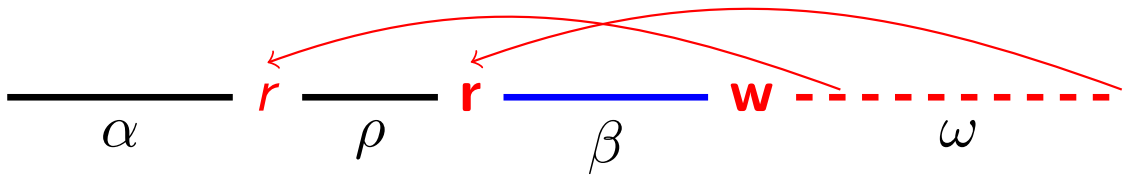
Solution Hide them from other threads



Finding TSO Witnesses with SC Reachability

TSO witnesses for attack A considerably restrict TSO behavior,

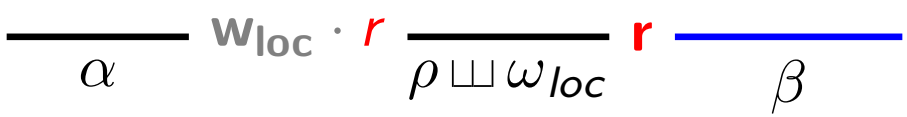
enough to find TSO witnesses with SC reachability



Let attacker execute under SC

Problem Writes may conflict with helper reads

Solution Hide them from other threads



Theorem [BDM'13]

Attack A has a TSO witness iff P_A reaches goal state under SC.

Trace Robustness: Conclusion

- Decidable for TSO (and beyond)
- Is an easy problem — PSPACE-complete
- Locality: only one thread uses the buffer
- Analysis parallelizable
- Monitoring techniques:

  e.g., [Burckhardt, Musuvathi CAV'08, Sen et al. TACAS'11]

- Static analysis:

  [Shasha Snir TOPLAS'88, Alglave, Maranget CAV'11]

- Semantics:

  [Owens ECOOP'10]