

Replication and Consistency*

* in and hardware **general** memory consistency models in particular

Roland Meyer

Technische Universität Kaiserslautern

Replication and Consistency

Setting: Concurrent threads accessing **shared data**

Replication and Consistency

Setting: Concurrent threads accessing **shared data**



Replication and Consistency

Setting: Concurrent threads accessing **shared data**



Problem 1: Access to shared data is **slow**

Replication and Consistency

Setting: Concurrent threads accessing **shared data**



Problem 1: Access to shared data is **slow**

Replication and Consistency

Setting: Concurrent threads accessing **shared data**



Problem 1: Access to shared data is **slow**

Solution 1: Replicate data so that every thread has a **copy**

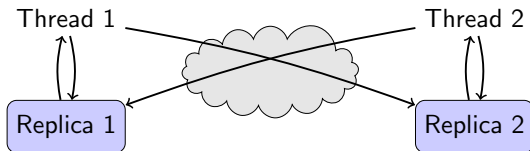
Replication and Consistency

Setting: Concurrent threads accessing **shared data**



Problem 1: Access to shared data is **slow**

Solution 1: Replicate data so that every thread has a **copy**



Replication and Consistency

Problem 2: Announce **updates** to other replicas

Replication and Consistency

Problem 2: Announce **updates** to other replicas

Solution 2: Halt the system and inform everybody

Replication and Consistency

Problem 2: Announce **updates** to other replicas

Solution 2: Halt the system and inform everybody

Ruins all performance benefits (back to **Problem 1**)

Replication and Consistency

Problem 2: Announce **updates** to other replicas

Solution 2: Halt the system and inform everybody

Ruins all performance benefits (back to **Problem 1**)

Solution 2': Inform other threads in a **delayed fashion**

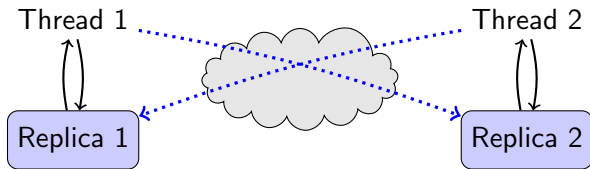
Replication and Consistency

Problem 2: Announce **updates** to other replicas

Solution 2: Halt the system and inform everybody

Ruins all performance benefits (back to **Problem 1**)

Solution 2': Inform other threads in a **delayed** fashion

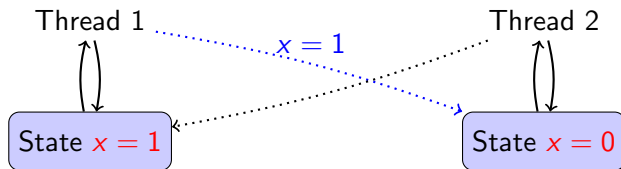


Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**

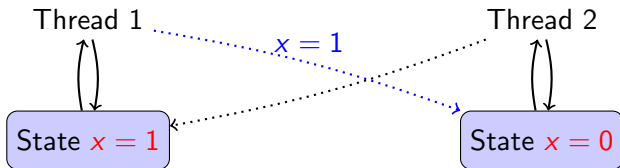
Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**



Replication and Consistency

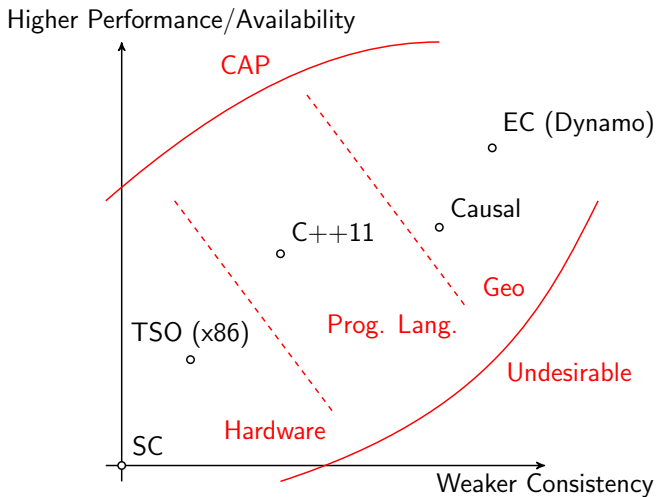
Problem 3: **Inconsistent** replicas while updates **travel**



Solution 3:

Live with it, inconsistency is **here to stay!**

Replication and Consistency



Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**

Solution 3: Live with it

Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**

Solution 3: Live with it

Solution 3': Architectures give **guarantees** about
updates

Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**

Solution 3: Live with it

Solution 3': Architectures give **guarantees** about
ordering of updates

Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**

Solution 3: Live with it

Solution 3': Architectures give **guarantees** about **ordering** and **visibility** of updates

Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**

Solution 3: Live with it

Solution 3': Architectures give **guarantees** about **ordering** and **visibility** of updates

Problem 4: But there are **so many** architectures

Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**

Solution 3: Live with it

Solution 3': Architectures give **guarantees** about **ordering** and **visibility** of updates

Problem 4: But there are **so many** architectures

Solution 4: Yes, but there are **underlying principles**

Replication and Consistency

Problem 3: **Inconsistent** replicas while updates **travel**

Solution 3: Live with it

Solution 3': Architectures give **guarantees** about **ordering** and **visibility** of updates

Problem 4: But there are **so many** architectures

Solution 4: Yes, but there are **underlying principles** ... at least in **hardware**

Replication and Consistency

Principles in hardware memory consistency models

Replication and Consistency

Principles in hardware memory consistency models

Guarantees in the update mechanism [Alglave, TOPLAS'14]:

Replication and Consistency

Principles in hardware memory consistency models

Guarantees in the update mechanism [Alglave, TOPLAS'14]:

SC per thread: For one thread running in isolation
the system looks consistent

Replication and Consistency

Principles in hardware memory consistency models

Guarantees in the update mechanism [Alglave, TOPLAS'14]:

SC per thread: For one thread running in isolation
the system looks consistent

Consequence: We can always rely on address and data dependencies

Replication and Consistency

Principles in hardware memory consistency models

Guarantees in the update mechanism [Alglave, TOPLAS'14]:

SC per thread: For one thread running in isolation
the system looks consistent

Consequence: We can always rely on address and data dependencies

Coherence: For every variable all threads will see the stores to this variable
in the same order

Replication and Consistency

Principles in hardware memory consistency models

Guarantees in the update mechanism [Alglave, TOPLAS'14]:

SC per thread: For one thread running in isolation
the system looks consistent

Consequence: We can always rely on address and data dependencies

Coherence: For every variable all threads will see the stores to this variable
in the same order

Why? Programmability

Replication and Consistency

Principles in hardware memory consistency models

Guarantees in the update mechanism [Alglave, TOPLAS'14]:

SC per thread: For one thread running in isolation the system looks consistent

Consequence: We can always rely on address and data dependencies

Coherence: For every variable all threads will see the stores to this variable in the same order

Why? Programmability + historical reasons

Replication and Consistency

Principles in hardware memory consistency models

Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order

o
TSO+W/R

Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order



Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order

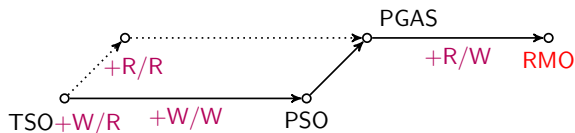


Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order



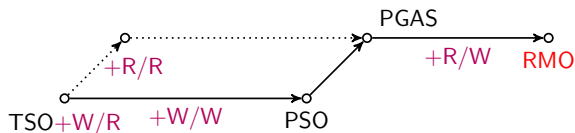
Very strange (and not in this talk):

Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order



Very strange (and not in this talk):

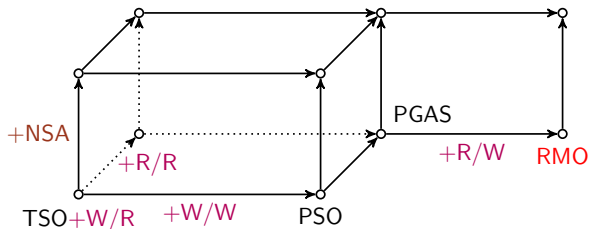
Out-of-thin-air values — arise when threads consistently lie to each other

Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order + store order



Very strange (and not in this talk):

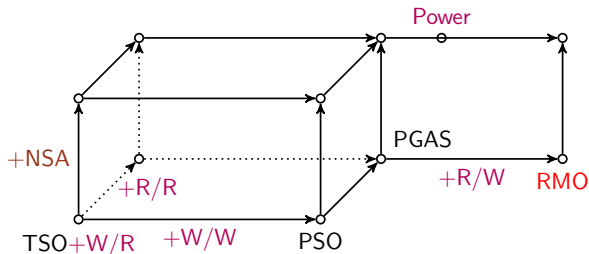
Out-of-thin-air values — arise when threads consistently lie to each other

Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order + store order



Very strange (and not in this talk):

Out-of-thin-air values — arise when threads consistently lie to each other

Lines of Research

Consistency models

Axiomatic, programming language (herd) for consistency models (Alglave)

Lines of Research

Consistency models

Axiomatic, programming language (herd) for consistency models (Alglave)

Geo-replicated consistency

Conflict-free replicated data types (Shapiro)

Lines of Research

Consistency models

Axiomatic, programming language (herd) for consistency models (Alglave)

Geo-replicated consistency

Conflict-free replicated data types (Shapiro)

C++11

Compilation (COMPCERT, ADVENT)

Lines of Research

Consistency models

Axiomatic, programming language (herd) for consistency models (Alglave)

Geo-replicated consistency

Conflict-free replicated data types (Shapiro)

C++11

Compilation (COMPCERT, ADVENT)

Linearizability

Semantics and algorithmics (Paderborn, Paris, Uppsala)

Lines of Research

Consistency models

Axiomatic, programming language (herd) for consistency models (Alglave)

Geo-replicated consistency

Conflict-free replicated data types (Shapiro)

C++11

Compilation (COMPCERT, ADVENT)

Linearizability

Semantics and algorithmics (Paderborn, Paris, Uppsala)

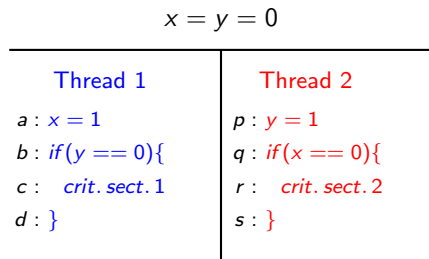
Verification under relaxed consistency models

Reachability and robustness (Paris, Uppsala, MSR, KL)

Memory Consistency Models: TSO and SC

Concurrent Programs with Shared Memory

- Finite number of **shared variables** $\{x, y, x_1, \dots\}$
- Finite **data domain** $\{d, d_0, d_1, \dots\}$
- Finite number of **finite-control threads** T_1, \dots, T_n with **operations**:
 $w(x, d), \quad r(x, d)$



Dekker's mutual exclusion protocol.

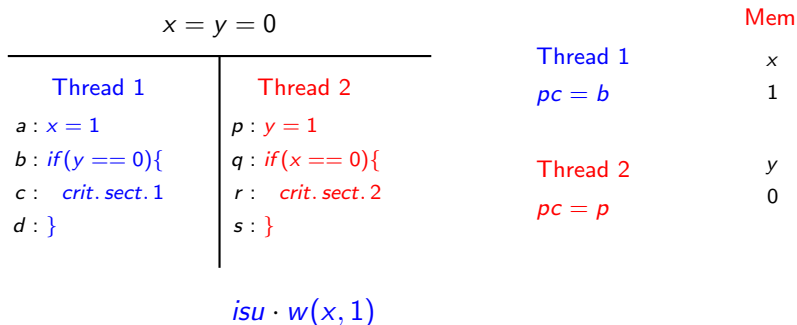
Sequential Consistency (SC) Semantics [Lamport 1979]

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

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

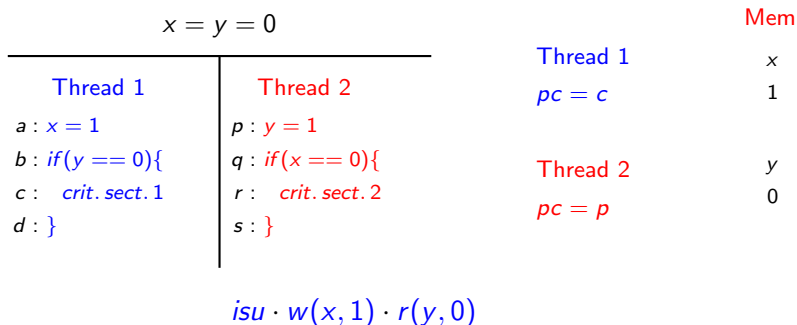
Sequential Consistency (SC) Semantics [Lamport 1979]

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



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



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

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

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

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 : <i>crit. sect. 1</i>	r : <i>crit. sect. 2</i>
d : $\}$	s : $\}$

Thread 1
 $pc = c$

Mem

Mutual exclusion holds!

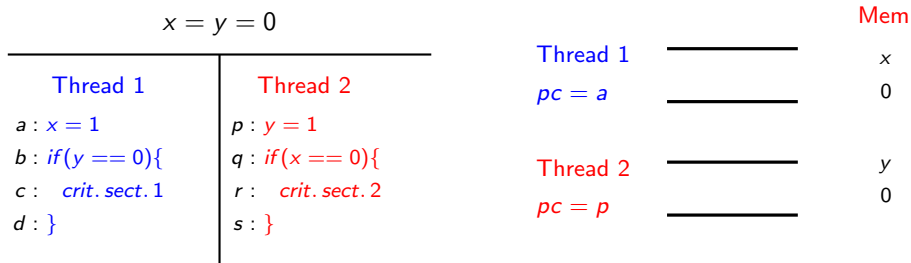
$isu \cdot w(x, 1) \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot \cancel{r(x, 0)}$

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

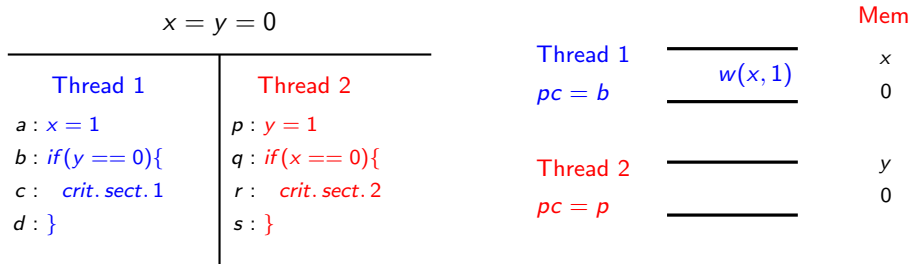
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



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

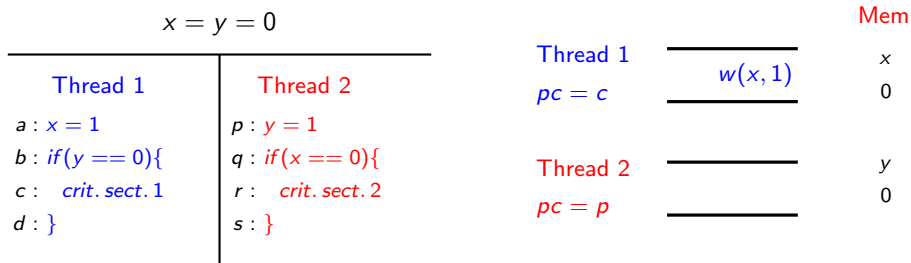
- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



isu

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

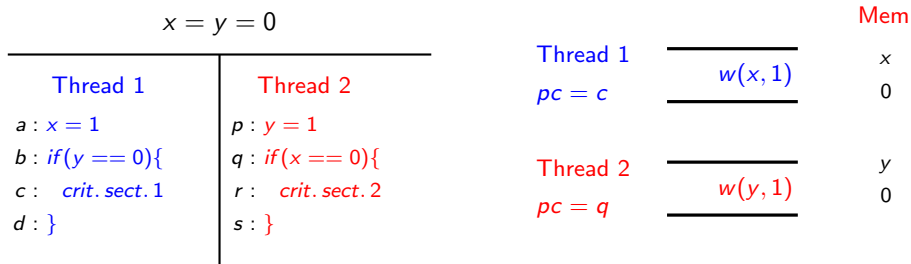
- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



$isu \cdot r(y, 0)$

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

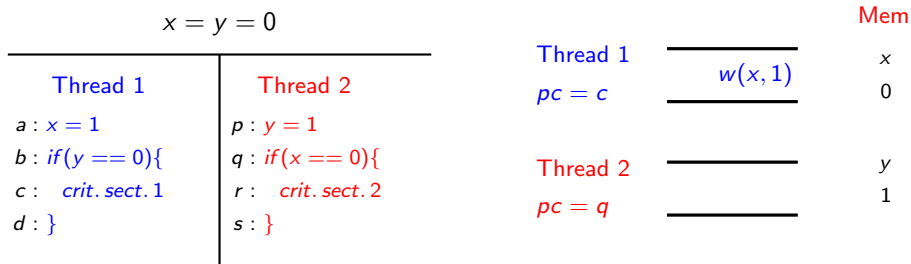
- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



$isu \cdot r(y, 0) \cdot isu$

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

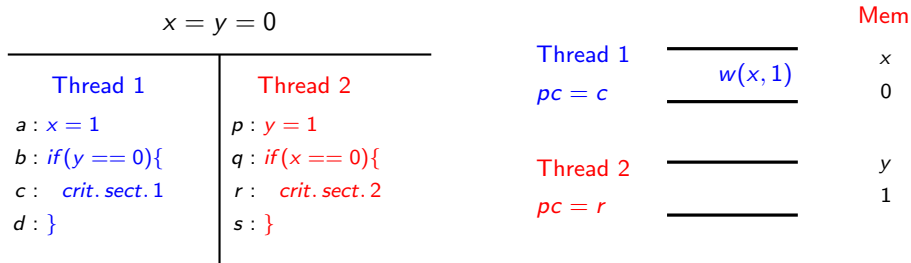
- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



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

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

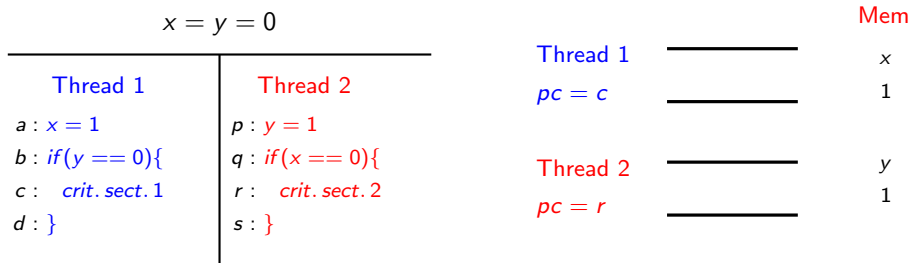
- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



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

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

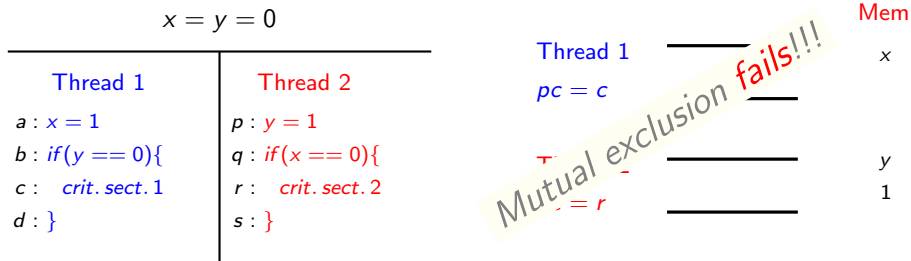
- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



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

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

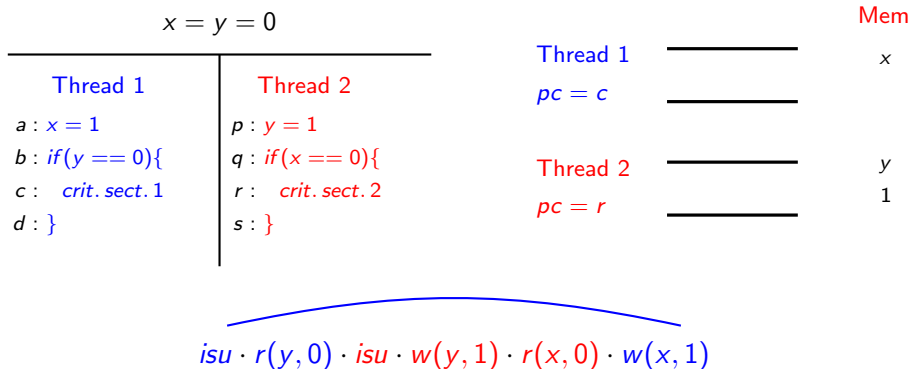
- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



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

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

- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer



Verification Required?!

Relaxed executions may lead to **bad behavior**

Verification Required?!

Relaxed executions may lead to **bad behavior**

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

Verification Required?!

Relaxed executions may lead to **bad behavior**

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 behavior**

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 behavior**

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 behavior**

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 **does have** data races

Verification Required?!

Relaxed executions may lead to **bad behavior**

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 **does have data races**

Concurrency libraries

Operating systems

HPC@Fraunhofer ITWM

Verification Required?!

Relaxed executions may lead to **bad behavior**

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 **does have data races**

Concurrency libraries

Operating systems

HPC@Fraunhofer ITWM

This is where our verification techniques apply

Reachability

[MSR, Oxford, Paris, Uppsala]

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:

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

Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP'11, ESOP'13, ICALP'14, ACM TECS'15]

Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP'11, ESOP'13, ICALP'14, ACM TECS'15]

Decision procedure for robustness that

Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP'11, ESOP'13, ICALP'14, ACM TECS'15]

Decision procedure for robustness that

- applies to most memory models (checked TSO, PSO, PGAS, Power)

Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP'11, ESOP'13, ICALP'14, ACM TECS'15]

Decision procedure for robustness that

- applies to most memory models (checked TSO, PSO, PGAS, Power)
- gives precise complexity

Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP'11, ESOP'13, ICALP'14, ACM TECS'15]

Decision procedure for robustness that

- applies to most memory models (checked TSO, PSO, PGAS, Power)
- gives precise complexity
- ... but relies on a new automaton model and lots of guessing

Robustness

Idea: SC semantics is specification

Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind

Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)

Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model *RMM*

Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model *RMM*

Input: Program *P*.

Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model RMM

Input: Program P .

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

Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

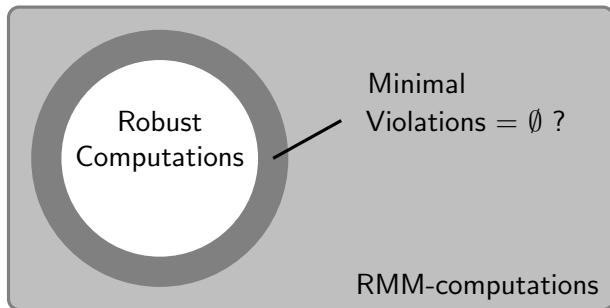
Robustness Problem against relaxed memory model RMM

Input: Program P .

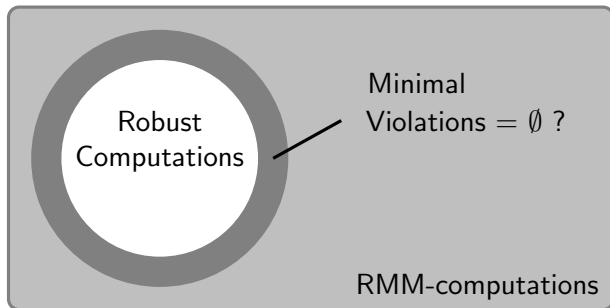
Problem: Does $Traces_{RMM}(P) \subseteq Traces_{SC}(P)$ hold?

Decidability / Complexity ?

Robustness: General Solution

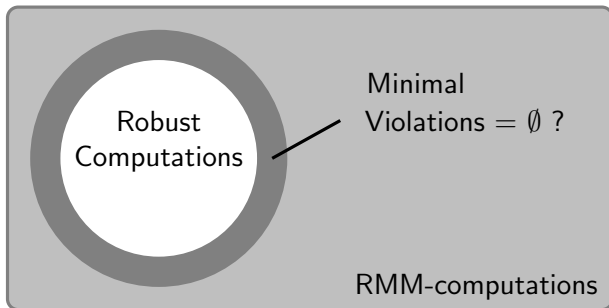


Robustness: General Solution



Combinatorics: **Violations** can be assumed to be in **normal form**

Robustness: General Solution



Combinatorics: Violations can be assumed to be in normal form

Algorithmics: Check whether normal form violations exist

Robustness: General Solution

Together: Reduce robustness to an **emptiness check**

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Robustness: General Solution

Together: Reduce robustness to an emptiness check

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

Robustness: General Solution

Together: Reduce robustness to an **emptiness check**

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

- Violations to SC (if any) have a representative in **normal form**.

Robustness: General Solution

Together: Reduce robustness to an **emptiness check**

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

- Violations to SC (if any) have a representative in **normal form**.

Algorithmics:

Robustness: General Solution

Together: Reduce robustness to an **emptiness check**

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

- Violations to SC (if any) have a representative in **normal form**.

Algorithmics:

- Language \mathcal{L}_{nf} consists of all **normal-form** computations.

Robustness: General Solution

Together: Reduce robustness to an **emptiness check**

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \mathcal{L}_{nf} consists of all normal-form computations.
- $\cap \mathcal{R}_{cyc}$ filters only **violating** computations.

Robustness: General Solution

Together: Reduce robustness to an **emptiness check**

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \mathcal{L}_{nf} consists of all normal-form computations.
- $\cap \mathcal{R}_{cyc}$ filters only violating computations.
- Decide $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$.

Robustness: General Solution

Reduce robustness to an emptiness check

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

- Violations to SC (if any) have a representative in **normal form**.

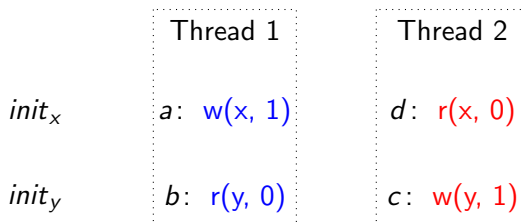
Algorithmics:

- Language \mathcal{L}_{nf} consists of all normal-form computations.
- $\cap \mathcal{R}_{cyc}$ filters only violating computations.
- Decide $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$.

Combinatorics: Normal Form Violations

Lemma (Shasha and Snir, 1988)

A computation violates SC iff it has a **cyclic happens-before relation**.




Combinatorics: Normal Form Violations

Lemma (Shasha and Snir, 1988)

A computation violates SC iff it has a **cyclic happens-before relation**.

Happens-before relation of computation


$$\tau = \textit{isu} \cdot r(y, 0) \cdot \textit{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) :$$


	Thread 1	Thread 2
\textit{init}_x	$a: w(x, 1)$	$d: r(x, 0)$
\textit{init}_y	$b: r(y, 0)$	$c: w(y, 1)$

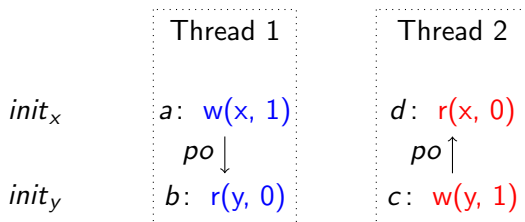
Combinatorics: Normal Form Violations

Lemma (Shasha and Snir, 1988)

A computation violates SC iff it has a **cyclic happens-before relation**.

Happens-before relation of computation


$$\tau = \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) :$$



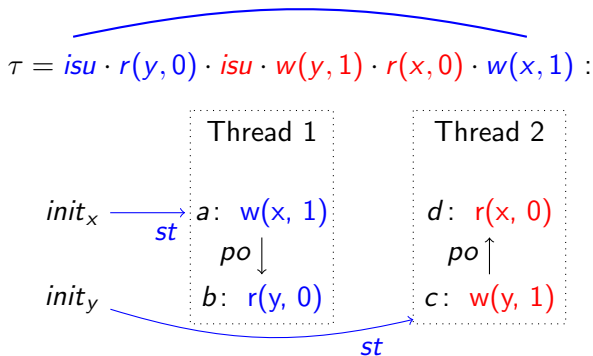
Program order

Combinatorics: Normal Form Violations

Lemma (Shasha and Snir, 1988)

A computation violates SC iff it has a **cyclic happens-before relation**.

Happens-before relation of computation



Program order, store order

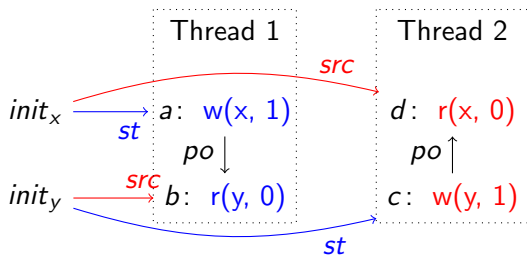
Combinatorics: Normal Form Violations

Lemma (Shasha and Snir, 1988)

A computation violates SC iff it has a **cyclic happens-before relation**.

Happens-before relation of computation

$$\tau = \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) :$$



Program order, store order, source relation

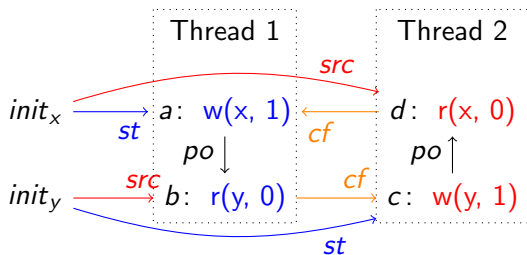
Combinatorics: Normal Form Violations

Lemma (Shasha and Snir, 1988)

A computation violates SC iff it has a **cyclic happens-before relation**.

Happens-before relation of computation

$$\tau = \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) :$$



Program order, store order, source relation, conflict relation

Combinatorics: Normal Form Violations

Normal Form:

Combinatorics: Normal Form Violations

Normal Form:

- Computation has two parts $\tau = \tau_1 \cdot \tau_2$

Combinatorics: Normal Form Violations

Normal Form:

- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- No delays within a part

Combinatorics: Normal Form Violations

Normal Form:

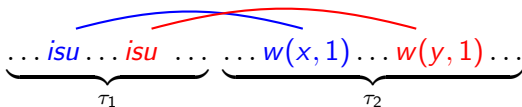
- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- No delays within a part
- Delays in τ_2 respect ordering in τ_1

Combinatorics: Normal Form Violations

Normal Form:

- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- No delays within a part
- Delays in τ_2 respect ordering in τ_1

In normal form

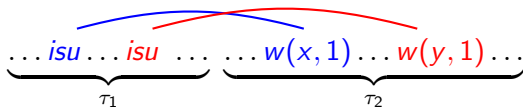


Combinatorics: Normal Form Violations

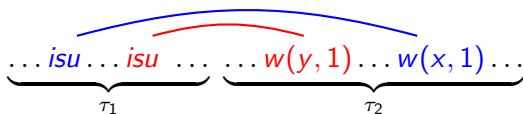
Normal Form:

- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- No delays within a part
- Delays in τ_2 respect ordering in τ_1

In normal form



Not in normal form



Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Proof:

- Take a **shortest** computation τ with cyclic happens-before relation.

Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Proof:

- Take a shortest computation τ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be **cancelled**:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Proof:

- Take a shortest computation τ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter

Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Proof:

- Take a shortest computation τ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence **not violating**.

Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Proof:

- Take a shortest computation τ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation σ with same happens-before relation.

Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Proof:

- Take a shortest computation τ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation σ with same happens-before relation.
- Now

$$(\sigma \downarrow \tau_1) \cdot a \cdot (\sigma \downarrow \tau_2)$$

Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Proof:

- Take a shortest computation τ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation σ with same happens-before relation.
- Now

$$(\sigma \downarrow \tau_1) \cdot a \cdot (\sigma \downarrow \tau_2)$$

is **in normal form**

Combinatorics: Normal Form Violations

Theorem (Normal form):

If a program is not robust, it has a violation in normal form.

Proof:

- Take a shortest computation τ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation σ with same happens-before relation.
- Now

$$(\sigma \downarrow \tau_1) \cdot a \cdot (\sigma \downarrow \tau_2)$$

is in normal form and **violating**.

Robustness: General Solution

Reduce robustness to an emptiness check

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \mathcal{L}_{nf} consists of all **normal-form** computations.
- $\cap \mathcal{R}_{cyc}$ filters only violating computations.
- Decide $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$.

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Need a language class that

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Need a language class that

- includes \mathcal{L}_{nf} ,

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Need a language class that

- includes \mathcal{L}_{nf} ,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Need a language class that

- includes \mathcal{L}_{nf} ,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has **decidable emptiness** problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$).

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Need a language class that

- includes \mathcal{L}_{nf} ,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$).

Properties of \mathcal{L}_{nf}

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Need a language class that

- includes \mathcal{L}_{nf} ,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$).

Properties of \mathcal{L}_{nf}

- Number of concurrently executed instructions is unbounded

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Need a language class that

- includes \mathcal{L}_{nf} ,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$).

Properties of \mathcal{L}_{nf}

- Number of concurrently executed instructions is **unbounded**
- May include computations like $isu^n \cdot w(x_i, 1)^n$

Algorithmics: Generating Normal-Form Computations

Challenge

Describe language \mathcal{L}_{nf} of all normal-form computations

Need a language class that

- includes \mathcal{L}_{nf} ,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$).

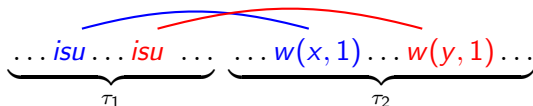
Properties of \mathcal{L}_{nf}

- Number of concurrently executed instructions is **unbounded**
- May include computations like $isu^n \cdot w(x_i, 1)^n$
 \Rightarrow **not context-free** (language $\sigma \cdot \sigma$)

Algorithmics: Generating Normal-Form Computations

Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

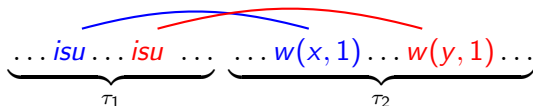


Algorithmics: Generating Normal-Form Computations

Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata



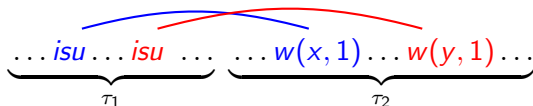
Algorithmics: Generating Normal-Form Computations

Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA



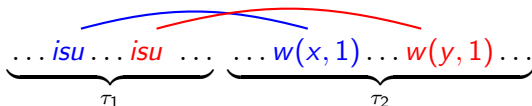
Algorithmics: Generating Normal-Form Computations

Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates **parts** τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ **simultaneously**



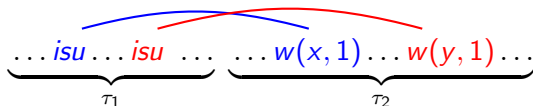
Algorithmics: Generating Normal-Form Computations

Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates parts τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ **labeled by head** $i = 1, 2$



Algorithmics: Generating Normal-Form Computations

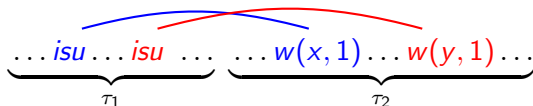
Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates parts τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:



Algorithmics: Generating Normal-Form Computations

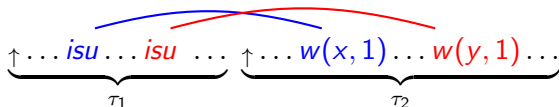
Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates parts τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:



Algorithmics: Generating Normal-Form Computations

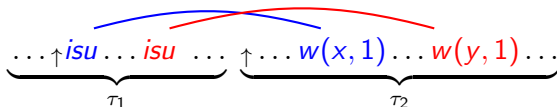
Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates parts τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:



Algorithmics: Generating Normal-Form Computations

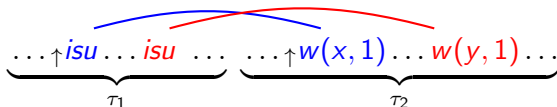
Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates parts τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:



Algorithmics: Generating Normal-Form Computations

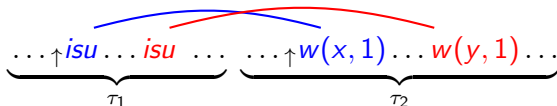
Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates parts τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:



Transitions: $q_1 \xrightarrow{1, isu} q_2 \xrightarrow{2, w(x, 1)} q_3$

Algorithmics: Generating Normal-Form Computations

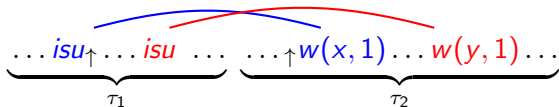
Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates parts τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:



Transitions: $q_1 \xrightarrow{1, isu} q_2 \xrightarrow{2, w(x, 1)} q_3$

Algorithmics: Generating Normal-Form Computations

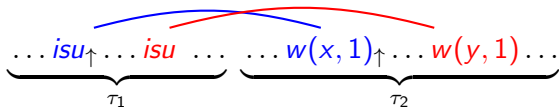
Solution

Define \mathcal{L}_{nf} as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA
- Generates parts τ_1 and τ_2 of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:



Transitions: $q_1 \xrightarrow{1, isu} q_2 \xrightarrow{2, w(x, 1)} q_3$

Robustness: General Solution

Reduce robustness to an emptiness check

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

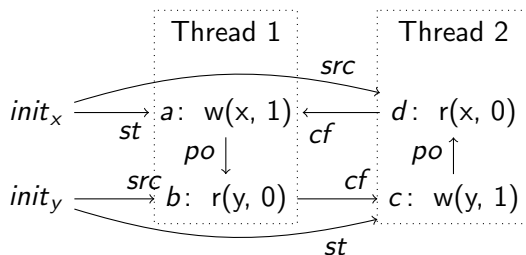
- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \mathcal{L}_{nf} consists of all normal-form computations.
- $\cap \mathcal{R}_{cyc}$ filters only **violating** computations.
- Decide $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$.

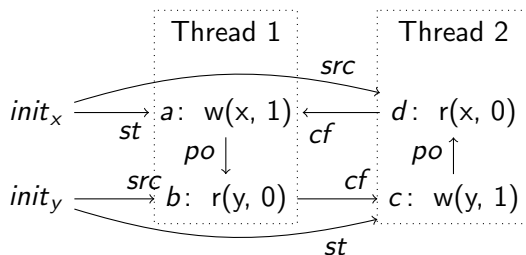
Algorithmics: Checking Cyclicality

Happens-before relation from the example:



Algorithmics: Checking Cyclicality

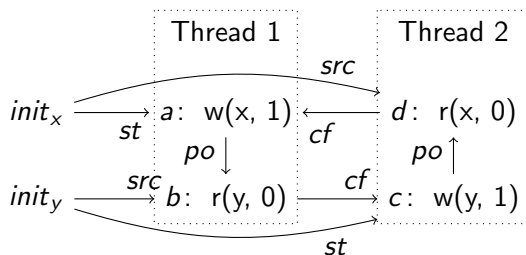
Happens-before relation from the example:



Checking cyclicality

Algorithmics: Checking Cyclicality

Happens-before relation from the example:

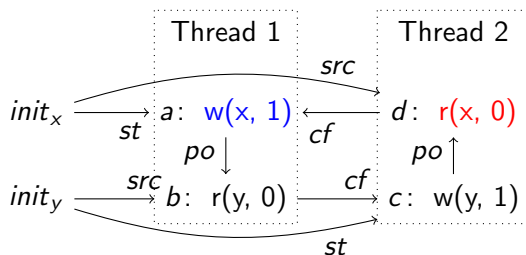


Checking cyclicality

- Finitely many types of cycles

Algorithmics: Checking Cyclicality

Happens-before relation from the example:

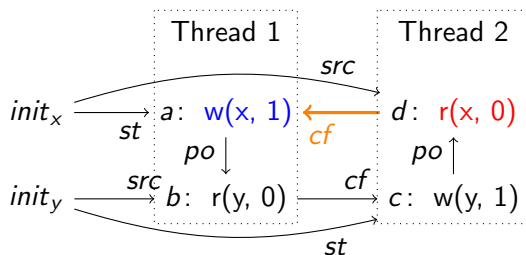


Checking cyclicality

- Finitely many types of cycles
- **Guess** per thread two instructions in program order

Algorithmics: Checking Cyclicity

Happens-before relation from the example:



Checking cyclicity

- Finitely many types of cycles
- Guess per thread two instructions in program order
- **Finite automata check edges** between guessed instructions from different threads

Robustness: General Solution

Reduce robustness to an emptiness check

$$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset.$$

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \mathcal{L}_{nf} consists of all normal-form computations.
- $\cap \mathcal{R}_{cyc}$ filters only violating computations.
- Decide $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$.

Algorithmics: Emptiness

Theorem:

Assuming finite memory, **robustness** is **PSPACE-complete**.

Algorithmics: Emptiness

Theorem:

Assuming finite memory, **robustness** is **PSPACE-complete**.

Proof:

Algorithmics: Emptiness

Theorem:

Assuming finite memory, **robustness** is **PSPACE-complete**.

Proof:

- Upper bound: $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$.

Algorithmics: Emptiness

Theorem:

Assuming finite memory, **robustness** is **PSPACE-complete**.

Proof:

- Upper bound: $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \stackrel{?}{=} \emptyset$.
- Lower bound: SC state reachability [Kozen 1977].