CS 766 : Analysis of Concurrent Programs

Krishna. S

Outline

- Intersection of PL and Formal Methods

Concurrency

Concurrency is everywhere!

- Lowest level of system design : chips
  - Leading architectures are all multicore
  - Examples : Intel, IBM POWER, ARM
- Highest level of system design
  - Distributed databases : Facebook, Amazon Aurora
- Development of Programming Languages
  - C++11, C20, Java
  - concurrency primitives : locks, threads, release accesses, acquire accesses etc

An Example of a Concurrent Program

Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\Big\|\Big\|$	$W(x, 1);$
$a := R(x, \cdot)$		$b := R(y, \cdot)$

An Example of a Concurrent Program

Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, \cdot)$	$\|\|$	$b := R(y, \cdot)$

Outcomes

$a = 1,\ b = 1$

$a = 1,\ b = 0$

$a = 0,\ b = 1$

$a = 0,\ b = 0$

Memory Models

How do we keep track of "what's happening underneath"?

- A **memory model** is a contract between a concurrent program and the machine it runs on
- A **formal specification** of all the behaviors that can arise due to concurrency and "weak" consistency

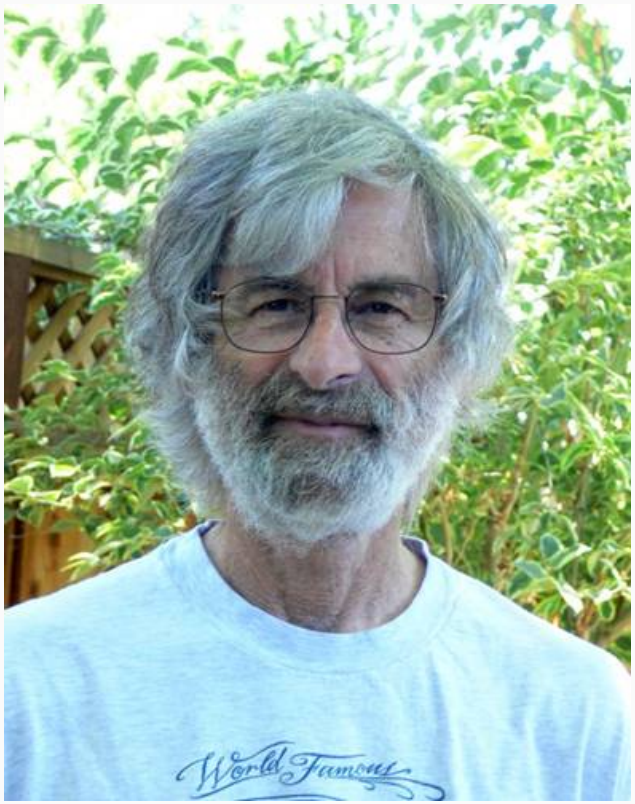
Memory Models

How do we keep track of "what's happening underneath"?

- A **memory model** is a contract between a concurrent program and the machine it runs on
- A **formal specification** of all the behaviors that can arise due to concurrency and "weak" consistency

When we think of concurrency, we typically assume **Sequential Consistency (SC)**

Sequential Consistency (SC)



Leslie Lamport

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, IEEE Trans. Computers, 1979.

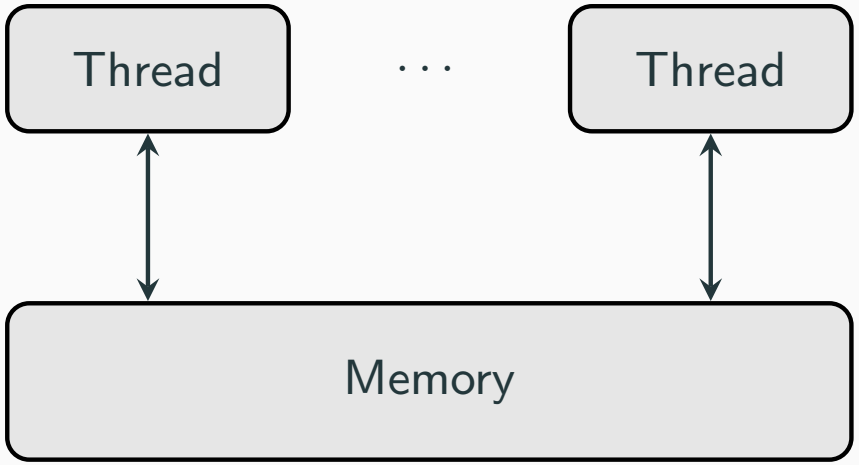
Sequential Consistency

Lamport's Formulation of Sequential Consistency

"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"

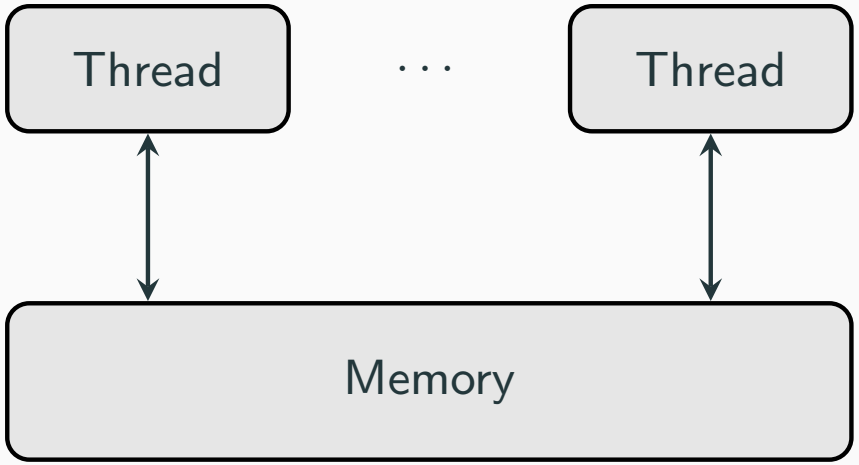
Store Buffer under SC

- Thread communication goes through the shared memory
- Under SC, all threads see the same order of writes



Store Buffer under SC

- Thread communication goes through the shared memory
- Under SC, all threads see the same order of writes



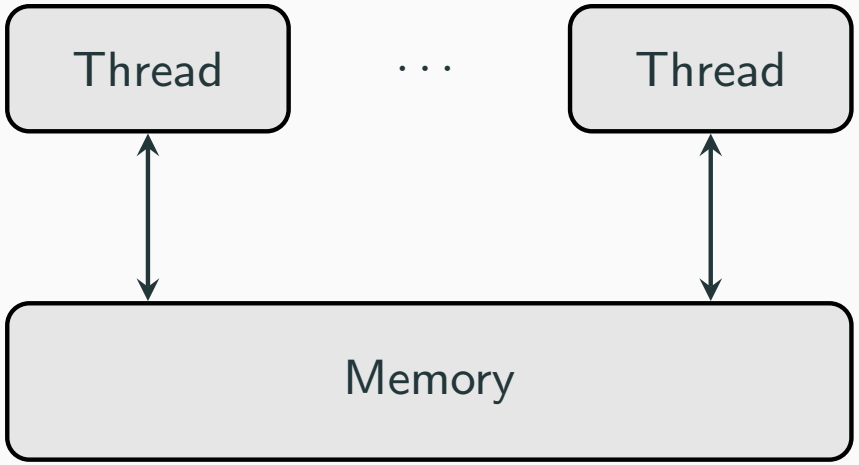
Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\Vert$	$W(x, 1);$
$a := R(x, \cdot)$	$\Vert$	$b := R(y, \cdot)$

Store Buffer under SC

- Thread communication goes through the shared memory
- Under SC, all threads see the same order of writes



Store Buffer

$$x = 0, y = 0$$

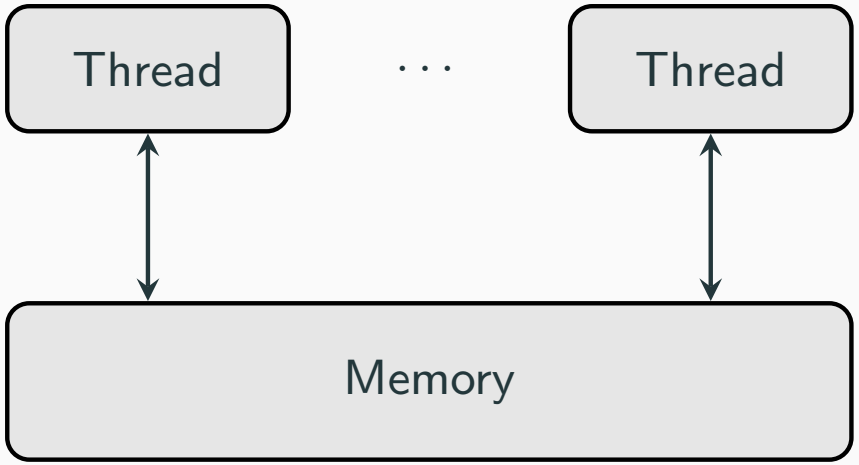
$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, 1)$		$b := R(y, 1)$

SC Outcomes

$a = 1,\ b = 1$

Store Buffer under SC

- Thread communication goes through the shared memory
- Under SC, all threads see the same order of writes



Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, 1)$		$b := R(y, 0)$

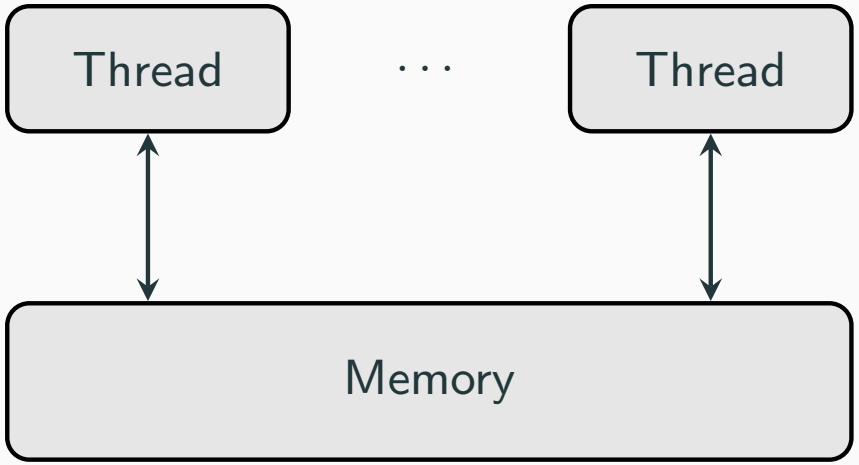
SC Outcomes

$a = 1,\ b = 1$ ✓

$a = 1,\ b = 0$

Store Buffer under SC

- Thread communication goes through the shared memory
- Under SC, all threads see the same order of writes



Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, 0)$		$b := R(y, 1)$

SC Outcomes

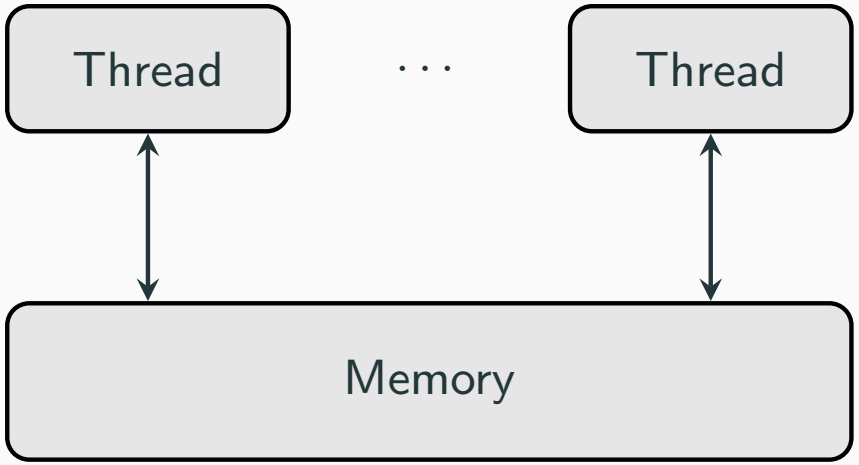
$a = 1,\ b = 1$ ✓

$a = 1,\ b = 0$ ✓

$a = 0,\ b = 1$

Store Buffer under SC

- Thread communication goes through the shared memory
- Under SC, all threads see the same order of writes



Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, 0)$		$b := R(y, 0)$

SC Outcomes

$a = 1,\ b = 1$ ✓

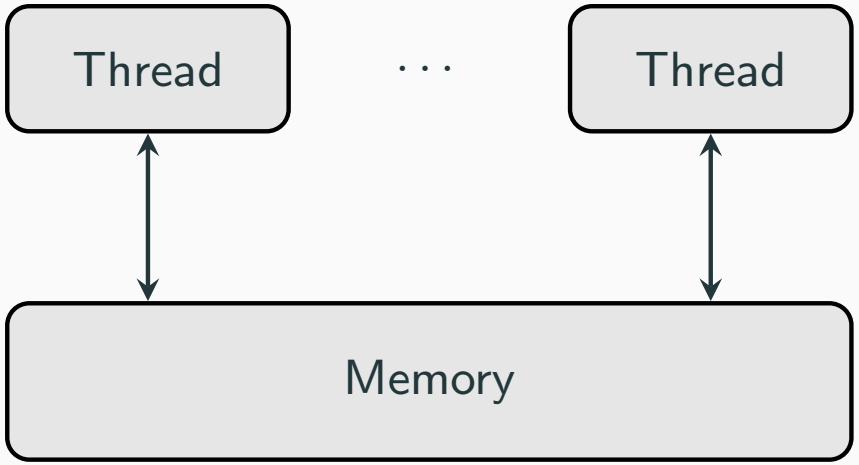
$a = 1,\ b = 0$ ✓

$a = 0,\ b = 1$ ✓

$a = 0,\ b = 0$

Store Buffer under SC

- Thread communication goes through the shared memory
- Under SC, all threads see the same order of writes



Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, 0)$		$b := R(y, 0)$

SC Outcomes

$a = 1,\ b = 1$ ✓

$a = 1,\ b = 0$ ✓

$a = 0,\ b = 1$ ✓

$a = 0,\ b = 0$ ✗

2+2W in SC

2+2W

$$x = 0, y = 0$$

$W(x, 1);$	$\|\|$	$W(y, 1);$
$W(y, 2);$	$\|\|$	$W(x, 2);$
$a := R(y, \cdot)$	$\|\|$	$b := R(x, \cdot)$

2+2W in SC

2+2W

$$x = 0, y = 0$$

$W(x,1);$	$\|\|$	$W(y,1);$
$W(y,2);$		$W(x,2);$
$a := R(y,\cdot)$		$b := R(x,\cdot)$

SC Outcomes

2+2W in SC

2+2W

$$x = 0, y = 0$$

$W(x,1);$	$\|\|$	$W(y,1);$
$W(y,2);$		$W(x,2);$
$a := R(y,1)$		$b := R(x,1)$

SC Outcomes

$a = 1,\ b = 1$ ✗

2+2W in SC

2+2W

$$x = 0, y = 0$$

$W(x,1);$	$\|\|$	$W(y,1);$
$W(y,2);$		$W(x,2);$
$a := R(y,1)$		$b := R(x,0)$

SC Outcomes

$a = 1,\ b = 1$ ✗

$a = 1,\ b = 0$ ✗

2+2W in SC

2+2W

$$x = 0, y = 0$$

$W(x,1);$	$\|\|$	$W(y,1);$
$W(y,2);$		$W(x,2);$
$a := R(y,0)$		$b := R(x,1)$

SC Outcomes

$a = 1,\ b = 1$ ✗

$a = 1,\ b = 0$ ✗

$a = 0,\ b = 1$ ✗

2+2W in SC

2+2W

$$x = 0, y = 0$$

$W(x,1);$	$\|\|$	$W(y,1);$
$W(y,2);$		$W(x,2);$
$a := R(y,0)$		$b := R(x,0)$

SC Outcomes

$a = 1,\ b = 1$ ✗

$a = 1,\ b = 0$ ✗

$a = 0,\ b = 1$ ✗

$a = 0,\ b = 0$ ✗

2+2W in SC

2+2W

$$x = 0, y = 0$$

$W(x, 1);$	$\|\|$	$W(y, 1);$
$W(y, 2);$		$W(x, 2);$
$a := R(y, 1)$		$b := R(x, 2)$

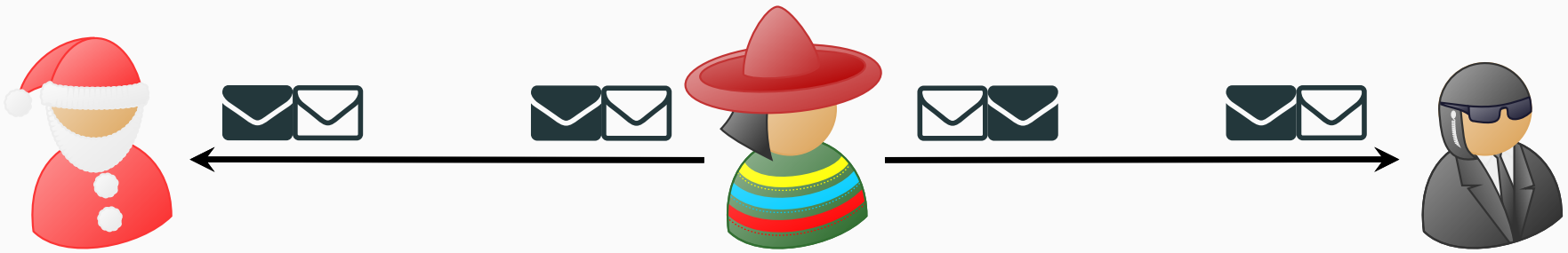
SC Outcomes

- $a = 1,\ b = 1$ ✗
- $a = 1,\ b = 0$ ✗
- $a = 0,\ b = 1$ ✗
- $a = 0,\ b = 0$ ✗
- $a = 1,\ b = 2$ ✓

Can we witness $a = 2, b = 1$?

Weak Consistency

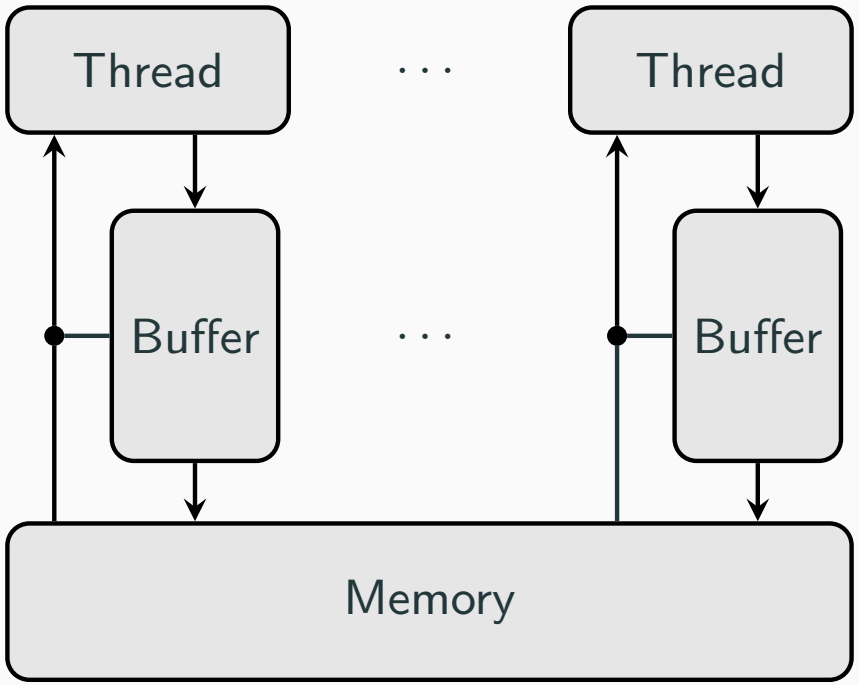
Weak Consistency



Maintaining strong consistency is hard

Intel x-86 TSO Architecture

- Thread writes go to a thread-owned buffer
- Buffers non-deterministically flush to memory
- A thread always reads from its buffer, if possible



SPARC

x86

- Every SC behavior is realizable under TSO
  - Flush each buffer immediately

Intel x-86 TSO Architecture

Optimizations using local write buffers to avoid explicit cost of synchronization

S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO (2009)

SPARC International, Inc. The SPARC Architecture Manual Version 9 (1994)

Intel. 2014. Intel 64 and IA-32 architectures software developer's manual

Store Buffer under x-86 TSO

Store Buffer

$$x = 0, y = 0$$

$$\begin{array}{l|l|l} W(y,1); & \| & W(x,1); \\ a := R(x,\cdot) & \| & b := R(y,\cdot) \end{array}$$

Store Buffer under x-86 TSO

Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, \cdot)$	$\|\|$	$b := R(y, \cdot)$

TSO Outcomes

$a = 1,\ b = 1$ ✓

$a = 1,\ b = 0$ ✓

$a = 0,\ b = 1$ ✓

$a = 0,\ b = 0$ ✓

Allows more behaviours than SC

2+2W in x-86 TSO

2+2W

$$x = 0, y = 0$$

$W(x, 1);$	$\|\|$	$W(y, 1);$
$W(y, 2);$		$W(x, 2);$
$a := R(y, \cdot)$		$b := R(x, \cdot)$

2+2W in x-86 TSO

2+2W

$$x = 0, y = 0$$

$W(x, 1);$	$\|\|$	$W(y, 1);$
$W(y, 2);$		$W(x, 2);$
$a := R(y, \cdot)$		$b := R(x, \cdot)$

TSO Outcomes

- $a = 1,\ b = 1$ ✗
- $a = 1,\ b = 0$ ✗
- $a = 0,\ b = 1$ ✗
- $a = 0,\ b = 0$ ✗
- $a = 1,\ b = 2$ ✓

Also allows $a = 2, b = 1$

Other Hardware Memory Models

- POWER Architecture
- ARM Architecture
- Uses optimizations using hierarchies of caches or speculative execution

Allows more behaviours than SC as well, implement weak consistency

Weak Consistency in Programming Languages

- Specifications of programming languages such as C++ formulate weak consistency
- Release, acquire and relaxed memory accesses
- Allows to demand SC behaviour when you need it with synchronization primitives

Release Acquire Fragment of C11

Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, \cdot)$	$\|\|$	$b := R(y, \cdot)$

Release Acquire Fragment of C11

Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\|\|$	$W(x, 1);$
$a := R(x, \cdot)$	$\|\|$	$b := R(y, \cdot)$

RA Outcomes

$a = 1,\ b = 1$ ✓

$a = 1,\ b = 0$ ✓

$a = 0,\ b = 1$ ✓

$a = 0,\ b = 0$ ✓

same behaviours as x-86 TSO

2+2W in RA

2+2W

$$x = 0, y = 0$$

$W(x, 1);$	$\|\|$	$W(y, 1);$
$W(y, 2);$	$\|\|$	$W(x, 2);$
$a := R(y, \cdot)$	$\|\|$	$b := R(x, \cdot)$

2+2W in RA

2+2W

$$x = 0, y = 0$$

$W(x,1);$	$\|\|$	$W(y,1);$
$W(y,2);$		$W(x,2);$
$a := R(y,\cdot)$		$b := R(x,\cdot)$

RA Outcomes

- $a = 1,\ b = 1$ ✓
- $a = 1,\ b = 0$ ✗
- $a = 0,\ b = 1$ ✗
- $a = 0,\ b = 0$ ✗
- $a = 1,\ b = 2$ ✓

More behaviours than SC and x-86 TSO

Part 1 : Reachability

The Reachability Problem

Reachability in Concurrent Programs : Given a concurrent program P and an assertion violation φ, does P witness the violation?

- Formal Semantics
- Undecidability of reachability in general concurrent programs (with recursion) under SC
- Under approximations
  - Bounded model checking. Goes via encoding into SMT formulae
  - Bounded context switching. Bounded reachability in concurrent pushdown systems
  - BMC, CBMC Tools
  - Recursion-free programs : PSPACE-complete. Employ reductions

The Reachability Problem

Reachability in Concurrent Programs : Given a concurrent program P and an assertion violation φ, does P witness the violation?

- x-86 TSO : Decidable, non-primitive-recursive complete
  - Formal Semantics
  - Decidability follows by showing that x-86 TSO is a Well Structured Transition System (WSTS)
  - Lower bounds by employing reductions

The Reachability Problem

Reachability in Concurrent Programs : Given a concurrent program P and an assertion violation φ, does P witness the violation?

- Release-Acquire Fragment of C11 : Undecidable
- Novel Under approximation of View Bounding and tool VBMC

The Reachability Problem

Reachability in Concurrent Programs : Given a concurrent program P and an assertion violation φ, does P witness the violation?

- Release-Acquire Fragment of C11 : Undecidable
- Novel Under approximation of View Bounding and tool VBMC

Part 2 : Consistency Checking

Some background

- Axiomatic Semantics

Consistency-checking Problems

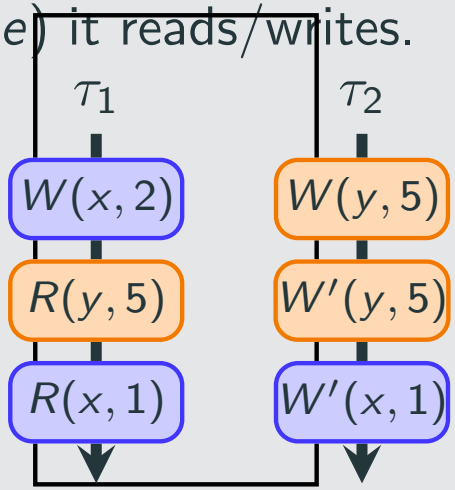
Consistency checking

Given a candidate trace (execution graph) $G = (E, po)$ and a memory model M, decide if G is consistent with M.

- G can have different forms. It typically contains a set of events E and the program order po between them

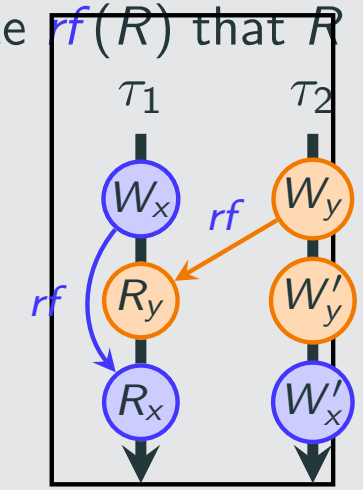
Value consistency

Additionally given a value function v, specifying for each event e the value $v(e)$ it reads/writes.



Reads-from (RF) consistency

Additionally given a reads-from function rf, specifying for each read R the write $rf(R)$ that R reads from.



Hardness of Consistency-checking in SC

- Consistency-checking is hard even for SC

Hardness of Consistency-checking in SC

- Consistency-checking is hard even for SC

Theorem

Value-consistency for SC is **NP***-hard even*

- *on inputs with just* 3 *threads*
- *on inputs over* 1 *shared variable*

Hardness of Consistency-checking in SC

- Consistency-checking is hard even for SC

Theorem

Value-consistency for SC is **NP***-hard even*

- *on inputs with just* 3 *threads*
- *on inputs over* 1 *shared variable*

Theorem

RF-consistency for SC is **NP***-hard.*

Hardness of Consistency-checking in SC

- Consistency-checking is hard even for SC

Theorem

Value-consistency for SC is **NP***-hard even*

- *on inputs with just* 3 *threads*
- *on inputs over* 1 *shared variable*

Theorem

RF-consistency for SC is **NP***-hard.*

Theorem

RF-consistency for SC is solvable in $O(k \cdot n^k)$ *time, for* n *events and* k *threads.*

- Polynomial for fixed $k = O(1)$
- What about other memory models?

Part 3 : Model Checking

Model Checking

- The two most natural model-checking questions are
  1. Plain reachability (safety)
     - Is a thread state reachable by some execution?
  2. Repeated reachability (liveness)
     - Is a thread state reachable repeatedly?

Model Checking

- The two most natural model-checking questions are
  1. Plain reachability (safety)
     - Is a thread state reachable by some execution?
  2. Repeated reachability (liveness)
     - Is a thread state reachable repeatedly?

- Naturally, reachability is undecidable even on sequential programs
- Focus on **bounded data domains**
- Reachability on sequential programs is decidable

Model Checking

- The two most natural model-checking questions are
  1. Plain reachability (safety)
     - Is a thread state reachable by some execution?
  2. Repeated reachability (liveness)
     - Is a thread state reachable repeatedly?

- Naturally, reachability is undecidable even on sequential programs
- Focus on **bounded data domains**
- Reachability on sequential programs is decidable

- What about concurrent programs (still, over bounded data domains)?
- **It largely depends on the memory model**

Part 4: Robustness

Robustness

- A program running under weak memory may or may not exhibit additional behavior compared to SC

Robustness

- A program running under weak memory may or may not exhibit additional behavior compared to SC

Robustness

A program P is **robust** under a weak memory model M if every state reachable under M is also reachable under SC.

Robustness

- A program running under weak memory may or may not exhibit additional behavior compared to SC

Robustness

A program P is **robust** under a weak memory model M if every state reachable under M is also reachable under SC.

- We are interested in **criteria that imply robustness**
- Such criteria should be easier to understand than weak-memory behavior
  - We shouldn't expect programmers to fully grasp weak-memory behavior

Further Directions

- Concurrency with GPUs
- Learning weak memory models
- Algorithms for Concurrent Data Structures

Tentative Evaluation Plan (can change)

- 30% Quizzes
- 35-40% Endsem
- 30-35% Paper presentations or Projects in groups of 2