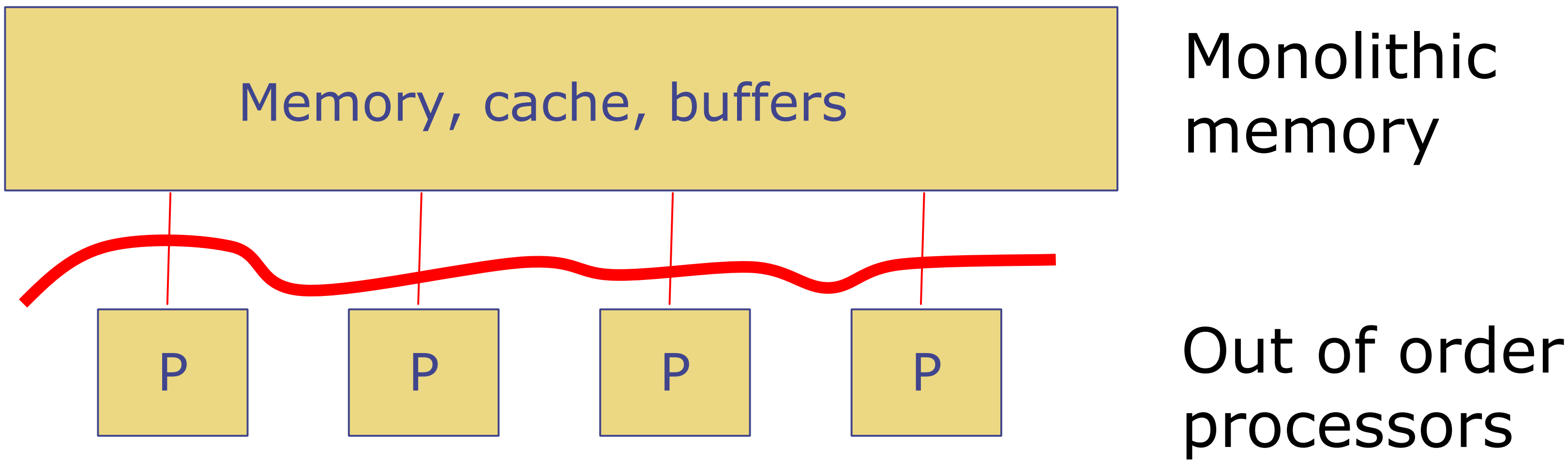
Store Atomicity

What does atomicity really require?

Jan-Willem Maessen (Sun Microsystems)
Based on joint work with Arvind from ISCA’06

From Dataflow to Synthesis
May 18, 2007

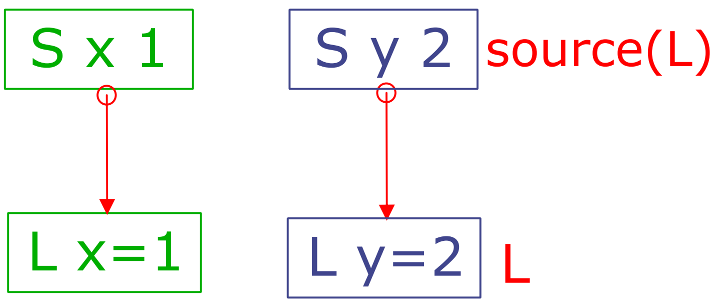
What is atomic memory?

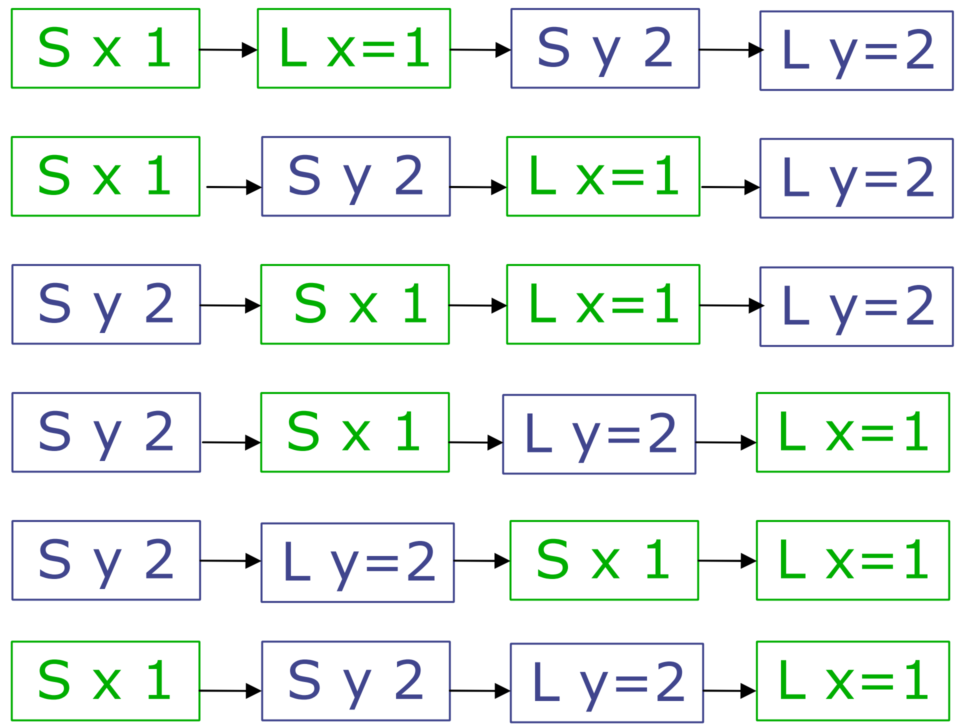


- Operational view: instruction at a time
- Declarative view: *serializability*

The Atomicity Puzzle

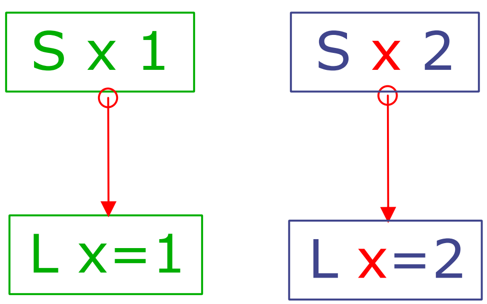
Puzzle 1: Serializability

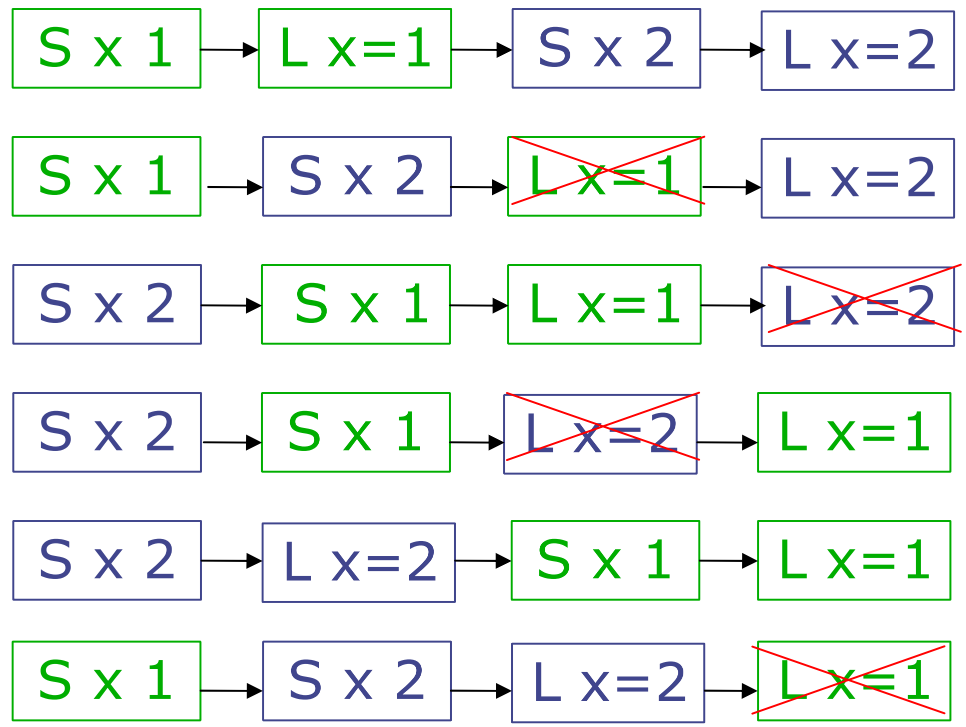




Many serializations exist for a given execution

Puzzle 1: Serializability

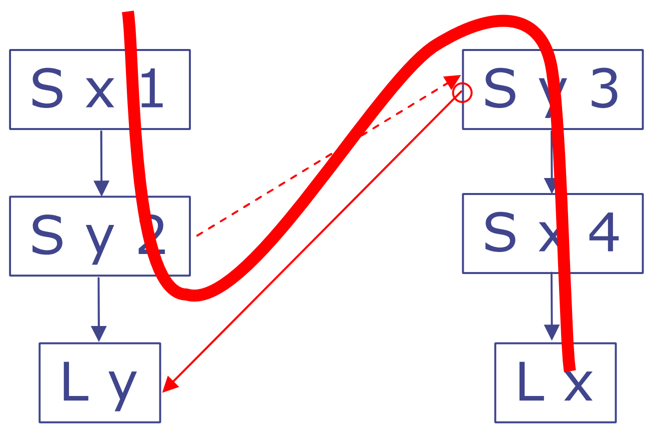




Only two serializations are possible

Potential violations of Serializability: Example 1

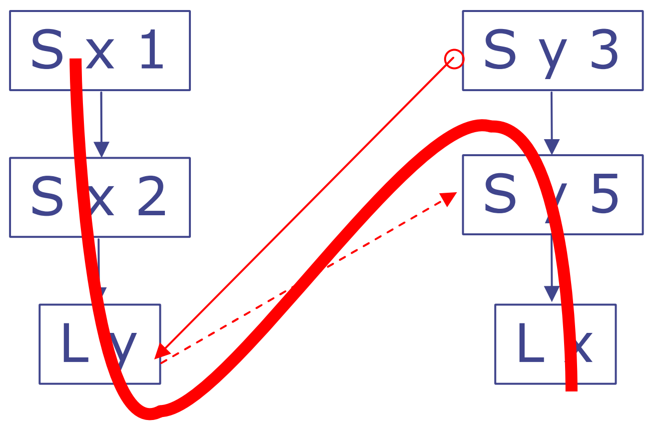
Thread 1	*Thread 2*
S x,1	S y,3
Fence	Fence
S y,2	S x,4
L y = 3	L x = 1?



Predecessor Stores of a Load are ordered before its source

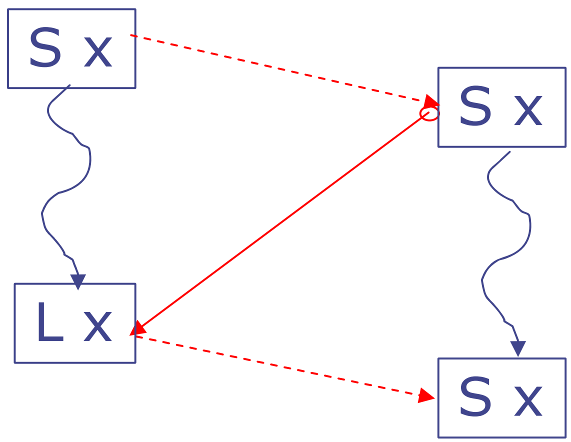
Potential violations of Serializability: Example 2

Thread 1	*Thread 2*
S x,1	S y,3
S x,2	S y,5
Fence	Fence
L y = 3	L x = 1?



Successor Stores of a Store are ordered after its observer

For Serializability we must have ...



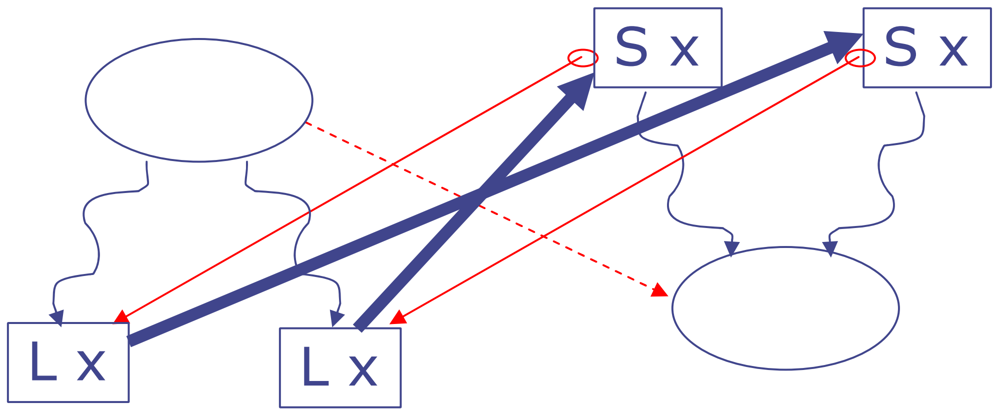
Predecessor Stores of a Load are ordered before its source

Successor Stores of a Store are ordered after its observer

Surprisingly not enough to ensure serializability!

Recognized by Hangal, Vahia, Manovit, et al. [TSOtool, ISCA '04]

Must pay attention to pairs of unrelated observations ...

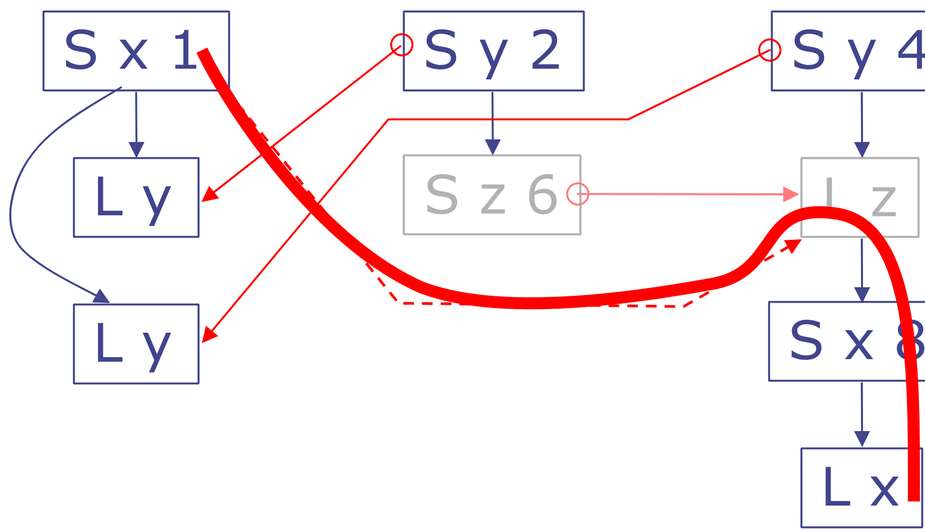


Mutual ancestors of unordered Loads are ordered before mutual successors of the Stores they observe

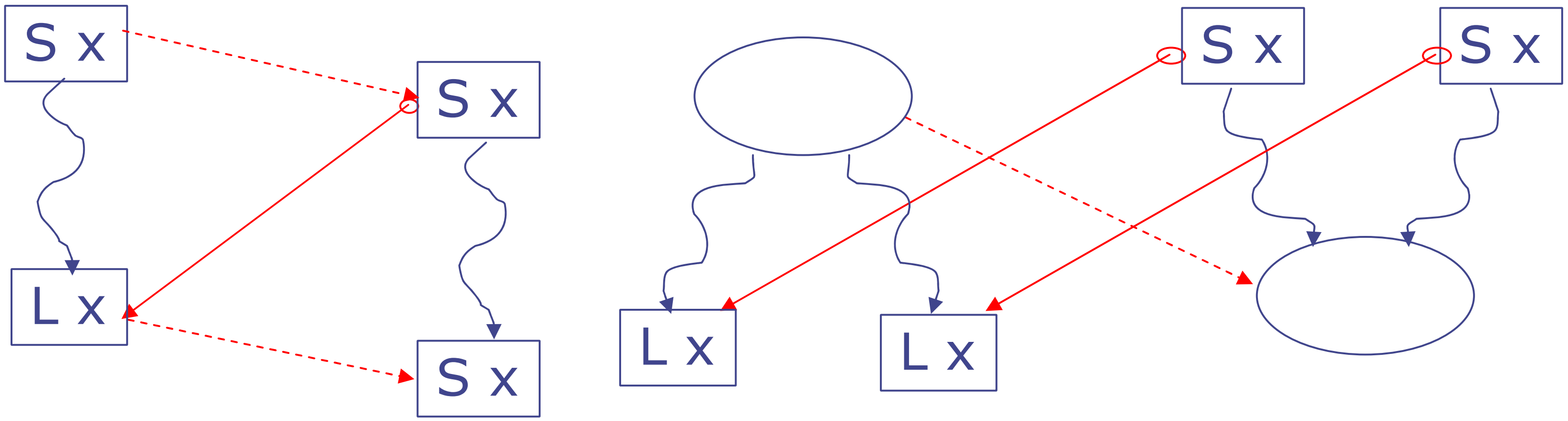
Overconstraining rules out legal executions

Potential violations of Serializability: Example 3

Thread 1	*Thread 2*	*Thread 3*
S x,1	S y,2	S y,4
Fence	Fence	Fence
L y = 2	S z,6	L z = 6
L y = 4		Fence
		S x,8
		L x = 1?



Store Atomicity



Predecessor Stores of a Load are ordered before its source

Successor Stores of a Store are ordered after its observer

Mutual ancestors of unordered Loads are ordered before mutual successors of the Stores they observe

Claim: Store Atomicity guarantees Serializability

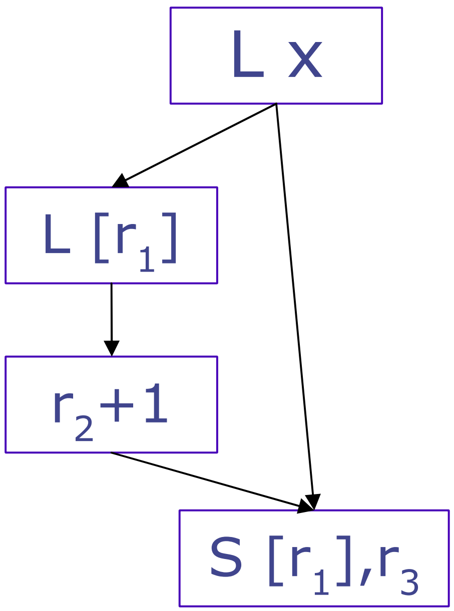
Instruction Reordering

2nd → 1st ↓	+,...	Br	L y	S y, w	Fence
+,...	indep	indep	indep	indep	✓
Br	✓	✗	✓	✗	✓
L x	indep	indep	indep	x≠y	✗
S y, w	✓	✓	x≠y	x≠y	✗
Fence	✓	✓	✗	✗	✓

Programming Language viewpoint

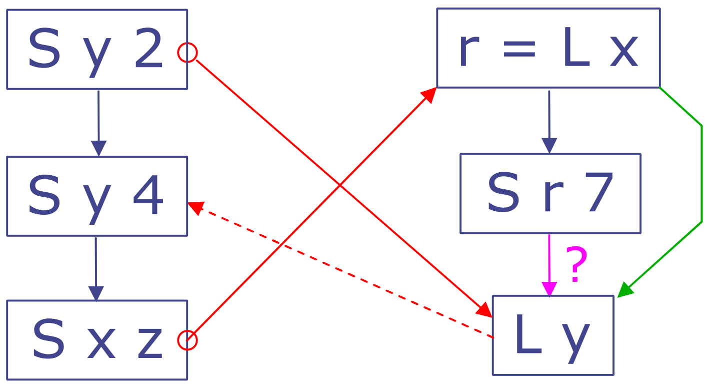
- Pointers and array indices give rise to dependent loads; these operations must be ordered.

$r_1 = L\ x$
$r_2 = L\ [r_1]$
$r_3 = r_2 + 1$
$S\ [r_1],\ r_3$



Flow of register state reflected in edges of graph; implicit register renaming

Address Speculation



S r 7 and L y are ordered if r = y

Non-speculative execution must wait until r has been computed.

Speculation assumes r ≠ y; if this fails, discard the execution

- Speculation = any decision which may break the rules down the line.
- Here we relax the reordering axioms.
- Behavior consistent with Store Atomicity

  observed by [Martin,Sorin,Cain,Hill,Lipasti 01]

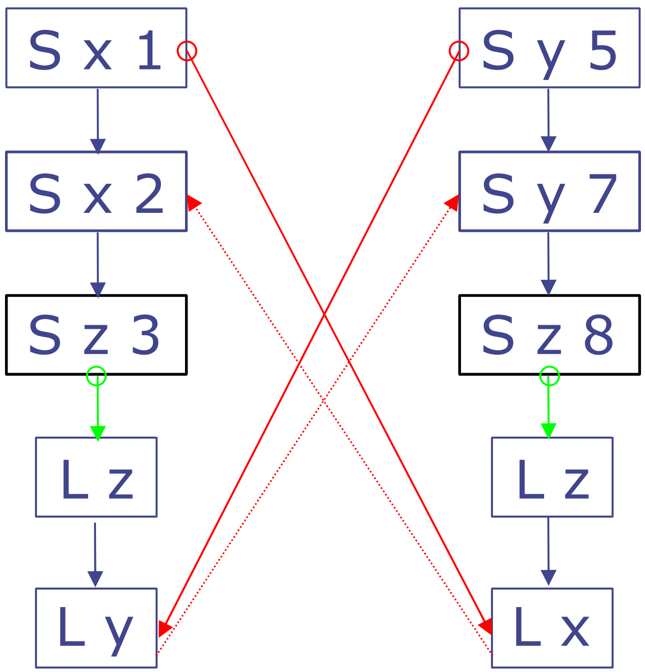
Optimizations Are Tricky

Thread 1	*Thread 2*
S x 0	S y 0
r_1 = L x = 2	r_3 = L y = 2
r_2 = L x	S x, r_3
if ~~(r_1 = r_2)~~	
S y 2	

- Ban invention of values "out of thin air"
- Permit any other imaginable optimization

  [Manson, Pugh, Adve 05]

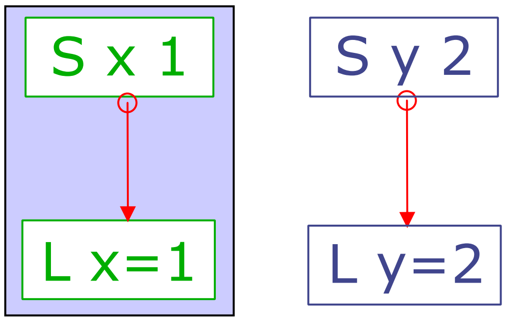
TSO is Non-Atomic



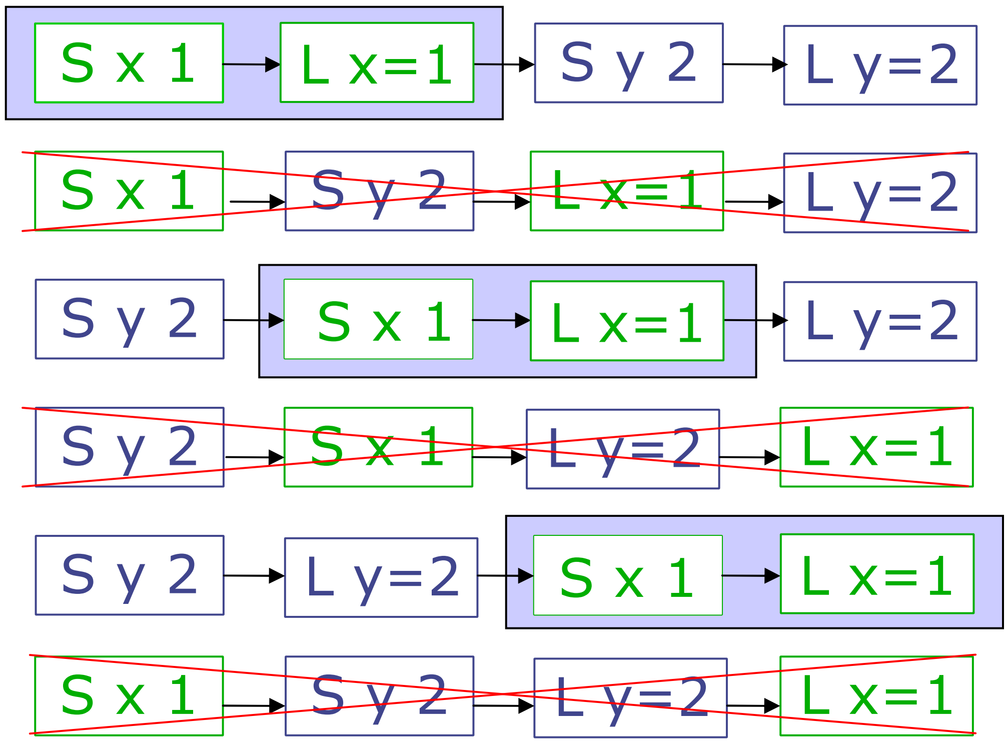
- Satisfy some Loads with local Stores
- Memory order ignores them
- Makes model non-atomic

Transactional Serializability

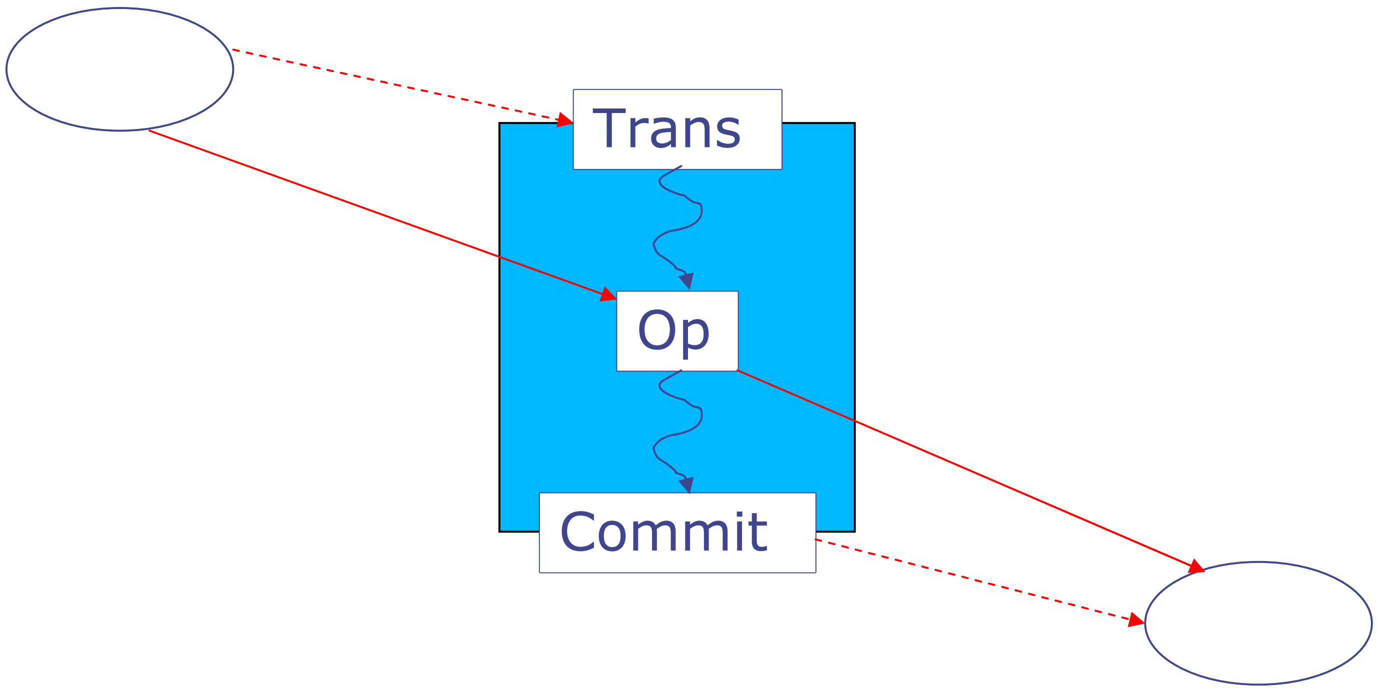
- Serialize instructions in transaction together.
  - Clearly atomic
  - Too strong; can't interleave independent operations



Disllowed executions actually are ok for this example!



Ordering and transactions



Predecessor operations precede the start of a transaction

Successor operations follow the end of a transaction

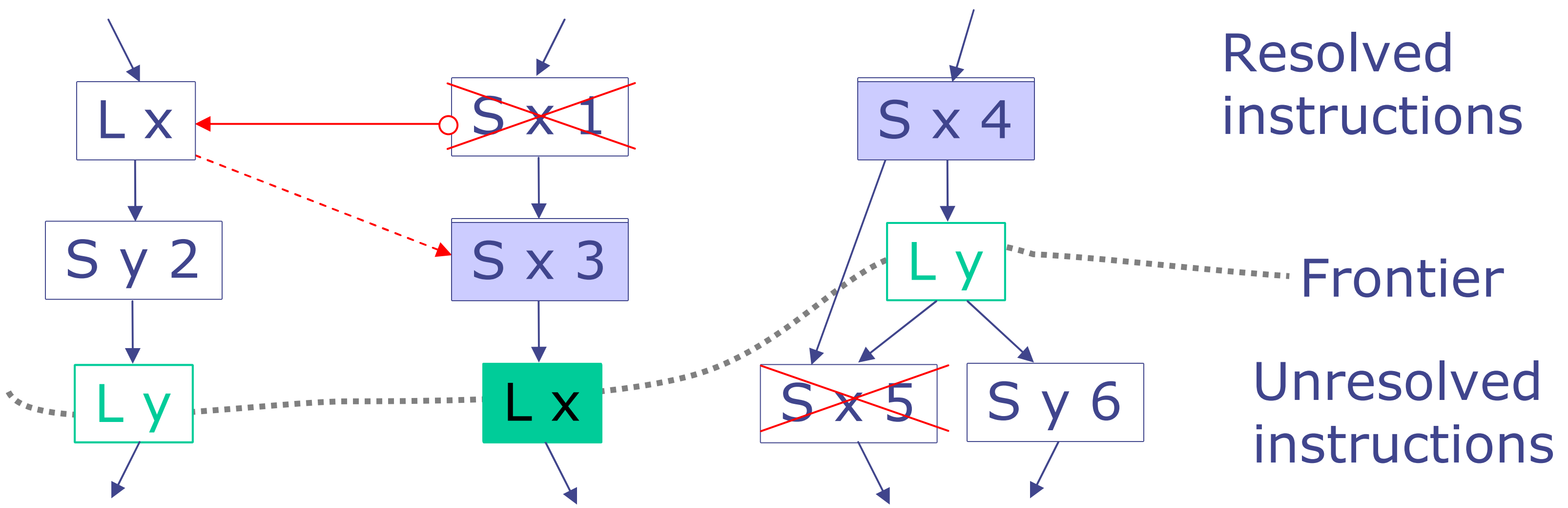
Enumeration of legal behaviors

- Find *all* legal behaviors
  - Must get the edges right

- Find *one* legal behavior
  - Can impose unnecessary ordering

  - Example: invalidation-based cache

Choosing a candidate Store



- Candidate stores for a Load must be:
  - To same address as that Load
  - Resolved
  - Not overwritten

Guarantees Store Atomicity is maintained

Store Atomicity Summary

- High-level unifying property for memory consistency protocols
- Separation between processor local, memory behavior
- Captures ordering dependencies which *must* be enforced by memory system
- A memory model with no memory

Thanks!

JanWillem.Maessen@sun.com

Implications / Applications

- Address Speculation, new behaviors but no violation of Store Atomicity (SA)
- Non-atomic models, e.g., TSO
- Properly synchronized programs
- Java Memory Model
- Transactional memory

Permit Aliasing Speculation

- New behaviors do not violate Store Atomicity
- Exploited by current architectures
- Banning complicates reordering
  - Dependency from source of Store address to any subsequent Load/Store

Overview

- Serializability, graphs
- Instruction Reordering
- Store Atomicity
- Enumerating behaviors operationally

Putting Store Atomicity to use

- Address aliasing speculation
- TSO

Drawbacks of TSO

- Complicates memory model
  - Two kinds of source edges—local, non-local
  - Must track interaction of these orderings
  - Definition of candidates(L) is subtle
- Problem on multi-core architectures
  - Separate Load/Store buffer per thread
  - Each must be large to tolerate latency

Avoid any model which treats some threads differently from others

Multithreaded Languages

- Discipline programmer must follow
  - Locks in well-synchronized programs
  - Use of synchronized and volatile in the Java™ Programming Language
- Obey discipline $\rightarrow$ Atomicity (SC)
- Every model has an atomic aspect:
  - Lock ordering
  - Volatile variables

Looking ahead

- Exploit flexible ordering constraints
  - Cache protocols
  - Cross-thread speculation
- Transactional memory
  - Serialization which reflects practice
- Programmer-level memory models
  - Well-synchronized programs
  - Implement language-level models in Store Atomic setting

Programmer's view
High-level vs. low-level models

- Store Atomicity is a very low-level property
  - Specifies what happens
  - No intuition about "how to program"
- Programmer-level models are important
  - Give a discipline for programming
  - Strong model (SC) within discipline
  - Hope: can check compliance
  - Example: Properly synchronized programs

Well synchronized programs

[Adve, Hill 90] [Keleher, Cox, Zwaenepoel 92]

- Divide the variables in two classes: synchronization variables and the rest
- In a well synchronized program a non-synchronizing Load L has only one element in candidates(L)!

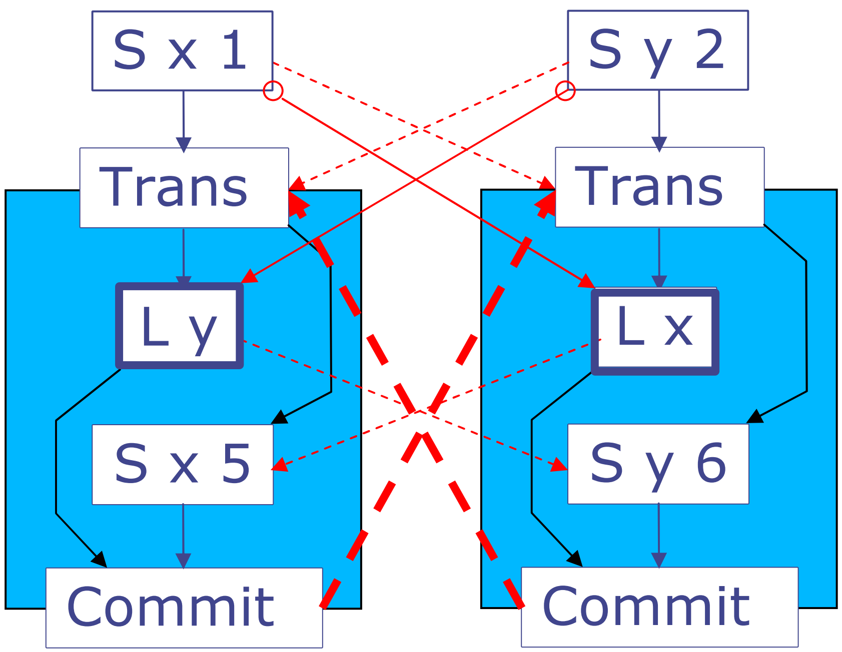
Atomicity edges can be grouped and drawn lazily

Instruction Reordering

2^{nd} → 1^{st} ↓	+,…	L y	S y, w	Fence	Trans	Commit
+,…	indep	indep	indep	✓	✓	✓
L x	indep	indep	x≠y	✗	✓	✗
S y, w	✓	x≠y	x≠y	✗	✓	✗
Fence	✓	✗	✗	✓	✓	✓
Trans	✓	✗	✗	✓	N/A	✗
Commit	✓	✓	✓	✓	✗	N/A

Partial order (dag) $\prec_{\mathbf{local}}$ on local instructions.

Resolving Transactional Loads in Parallel



We resolve a load in both transactions

Observed Stores overwritten

Results in a cycle between transactions

Roll back some Load which breaks cycle

- Bad speculation introduces cycle
- Roll back Load which break cycle
- Along with its direct dependencies