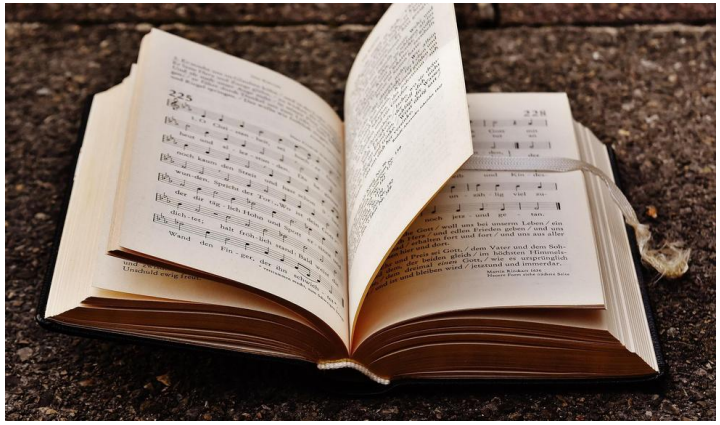


Chapter 9: Multicore Systems

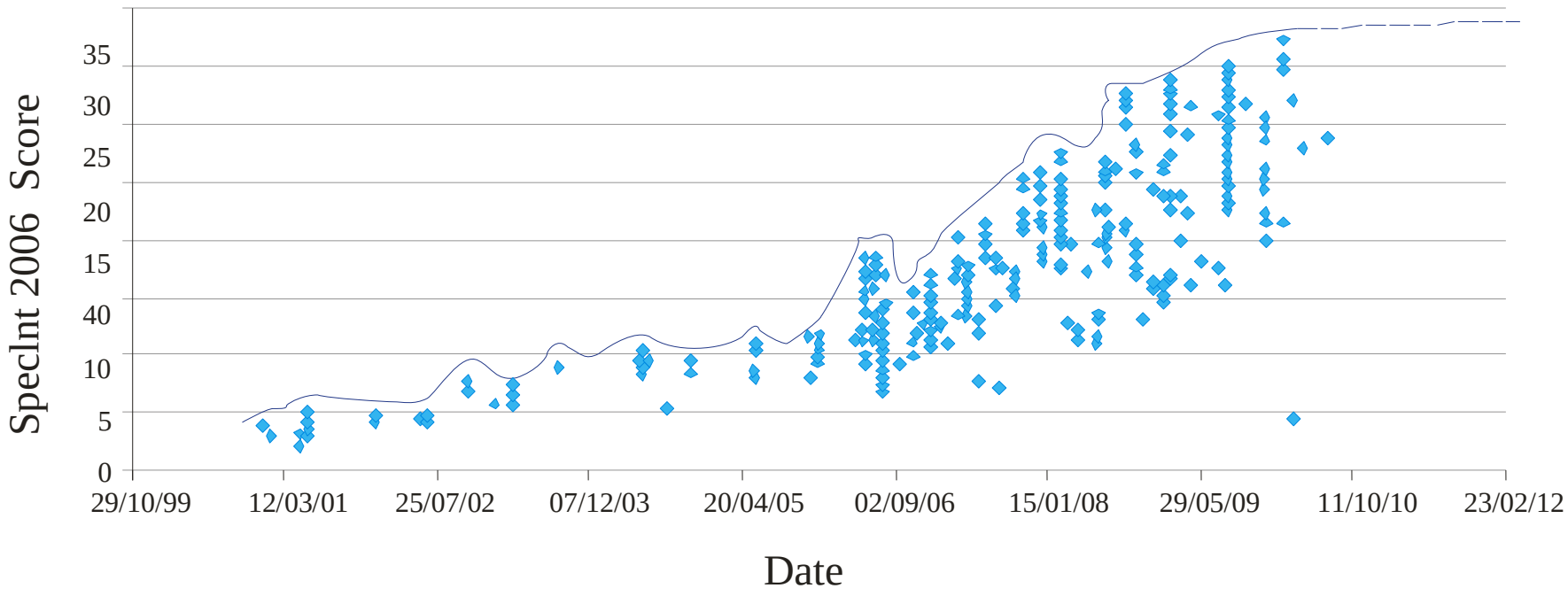
Background Required to Understand this Chapter

- Caches
- On-Chip Network
- Graph Algorithms

Chapters 7 and 8

Contents

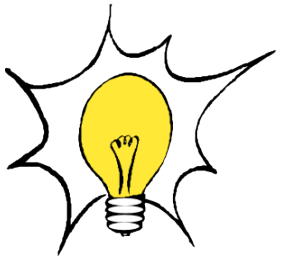
Era of Sequential Computing is Over



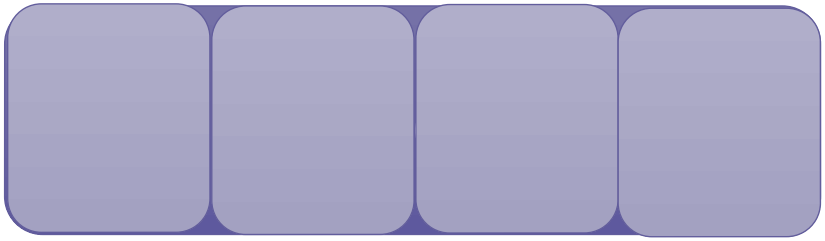
Single core performance from 2001 to 2010

It has clearly saturated after 2008.

Solution



Have multiple cores that collaboratively solve a task.



Map each sub-problem to a core

Smaller problems

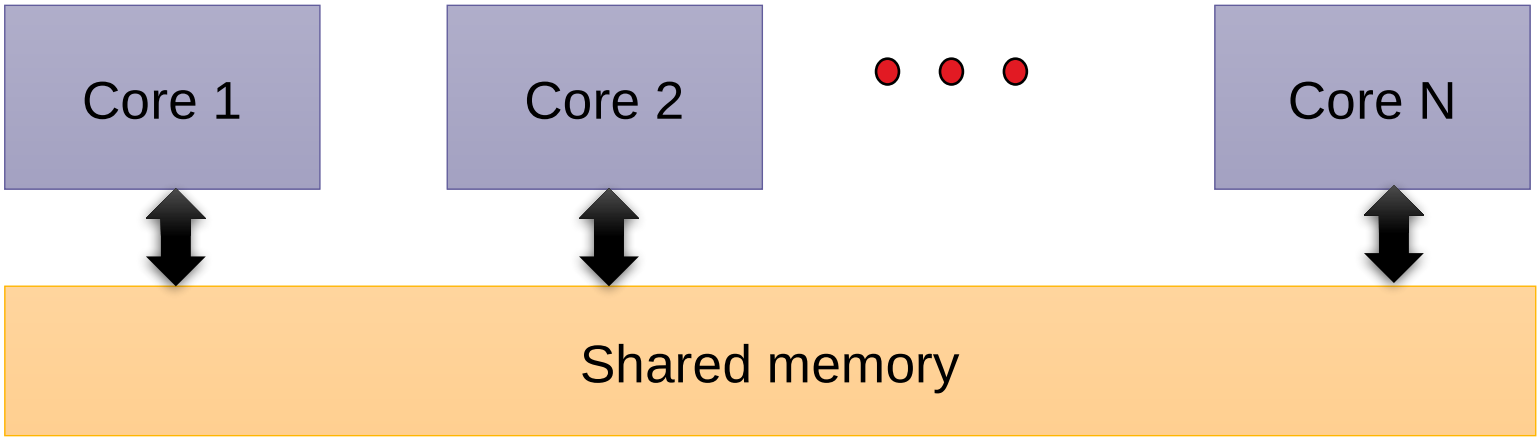
Problem

- Take an array *numbers* with *SIZE* elements.
- Compute the sum of all of its elements.
- Distribute the work among *N* threads.

Approach

- Divide the array into *N* parts.
- Assign each part to a core.
- Compute the partial sum (sum of each part)
- Add the partial sums

Shared Memory based Programming

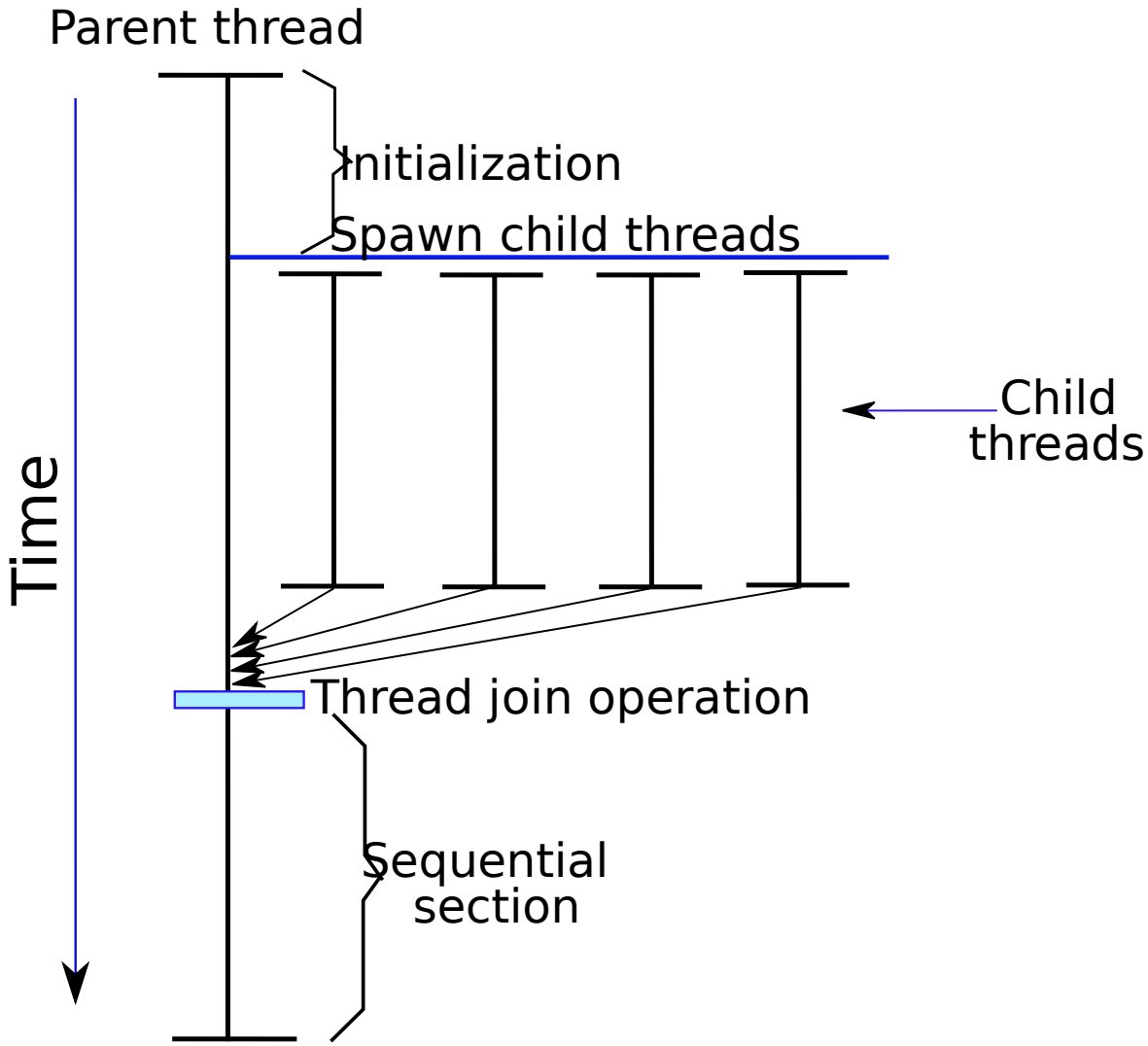


- Each core runs a thread.
- Different threads share parts of the virtual memory space
- They communicate by reading and writing to shared variables

```
/* variable declaration */
int   partialSums [N];
int   numbers [SIZE];
int   result   =   0;

/* initialise arrays */
...
/* parallel section */ # pragma omp parallel {
                /* get my processor id */
            int   myId   =   omp_ get_ thread_ num ();

            /* add my portion of numbers */
            int   startIdx   =   myId   *   SIZE/ N;
            int   end Idx   =   startIdx   +   SIZE/ N;

            for( int   jdx   =   startIdx;   jdx   <   end Idx;   jdx++)
                        partialSums[ myId]   +=   numbers[ jdx];
}

/* sequential section */
for( int   idx=0;   idx  < N;   idx++)
            result   +=   partialSums[ idx];
```

OpenMP code

A Typical Shared Memory Program



Message Passing Code

- Cores communicate by sending messages to each other
- They do not share memory.
- There are two basic functions.

Function	Semantics
send(pid, val)	Send the integer *val* to the process with id, *pid*
receive(pid)	1. Receive an integer from process *pid* 2. This is a blocking call 3. If the *pid* is equal to *ANYSOURCE*, then the *receive* function returns with the value sent by any process

```
/* start all the parallel processes */
Spawn AllParallelProcesses();

/* For each process execute the following code */
int myId = getMyProcessId ();

/* compute the partial sums */
int startIdx = myId * SIZE/ N;
int end Idx = startIdx + SIZE/ N;
int partialSum = 0;

for( int jdx = startIdx; jdx < end Idx; jdx++)
        partialSum += numbers[ jdx];

/* All the non - root nodes send their partial sums to
the root */
if( myId != 0) {
                /* send the partial sum to the root */
                send (0 , partialSum );

}
```

Message passing code

Continuation

```
else {

        /* for the root */ int sum = partialSum ;
        for ( int pid = 1; pid < N; pid ++) {
                sum += receive( ANYSOURCE );
        }

        /* shut down all the processes */
        shutDown AllProcesses();

        /* return the sum */
        return sum;
}
```

Shared Memory vs Message Passing

Shared memory

- Easy to program.
- Issues with scalability
- The code is portable across machines.

Message passing

- Hard to program.
- Scalable
- The code may not be portable across machines.

Amdahl's Law

How much can we parallelize?

We are limited by the fraction of the sequential section: f_{seq}

For P processors

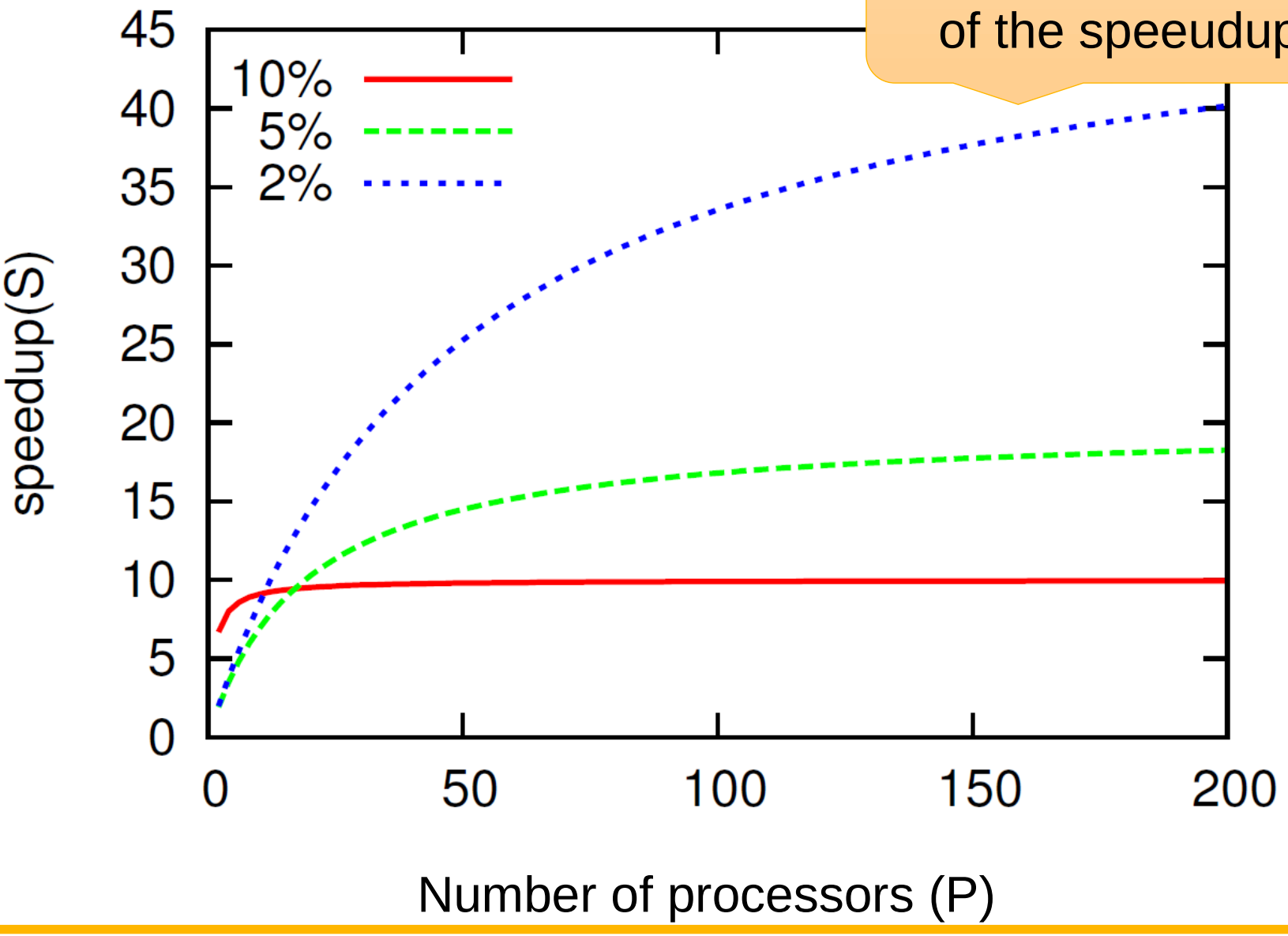
$$T_{par} = T_{seq} \times \left(f_{seq} + \frac{1 - f_{seq}}{P} \right)$$

Speedup

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{f_{seq} + \frac{1 - f_{seq}}{P}}$$

As ,

Speedup vs #Processors



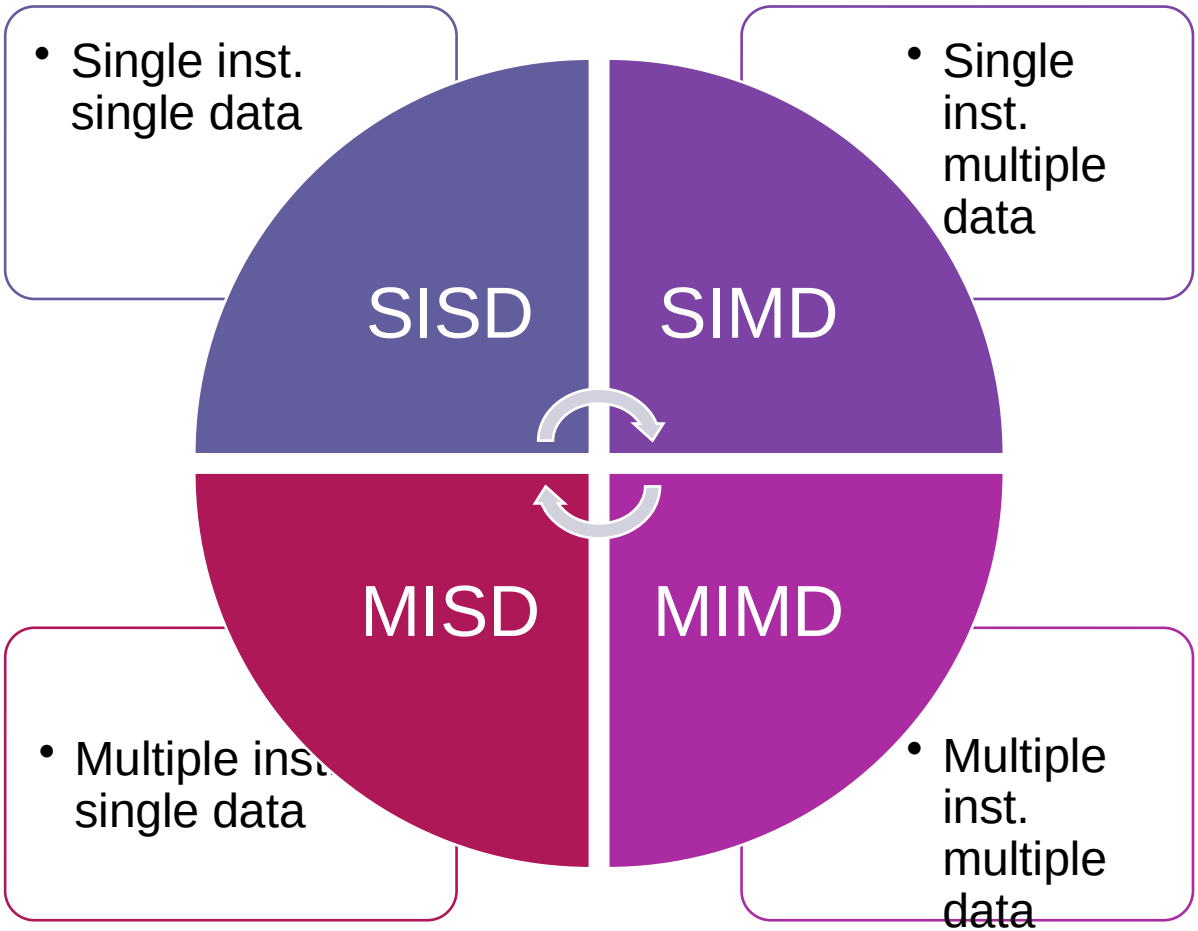
Gustafson-Barsis's Law

Old workload: W

New workload: Scale the parallel part by #procs (P)

Entity	Mathematical expression
New workload	
Sequential Time () is a constant of proportionality	
Parallel Time ()	

Design Space of Multiprocessors: Flynn's Classification



Explanation of the Flynn's Classification

SISD Processors

- A regular single core processor

SIMD Processors

- Single instruction stream, multiple data streams
- Vector instruction set. **Example**: add v1, v2, v3
  - v1, v2, and v3 are **vector registers**
  - They pack multiple integers (let's say 4)
  - A pairwise addition is performed
  - **Summary**: We can do 4 additions using just a single instruction

Explanation II

MISD Processors

- Used in airplanes: Run the same program on three separate processors that have different instruction sets. Compare the outputs and decide by voting.

MIMD Processors

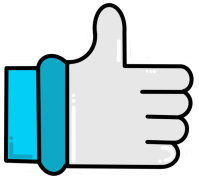
- **SPMD Processors**: Single program, multiple data. Run the same program on different cores with different data streams (most common)
- **MPMD Processors**: Consider a processor with different accelerators. Each core or accelerator runs a different program.

Hardware Threads

Notion of Hardware Threads

- We traditionally have processes that run on different cores.

- Let's say we have cores with large issue widths
  - If processes have low ILP, then we waste issue slots



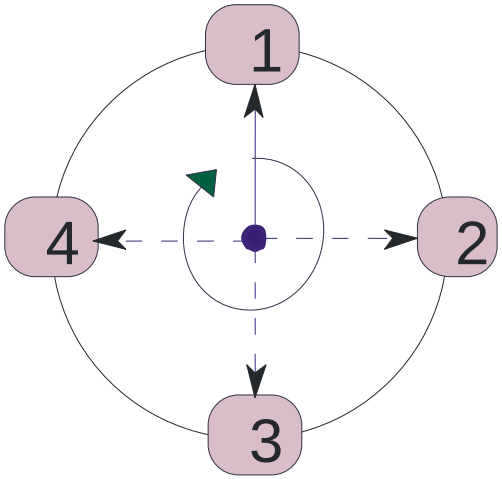
- Run multiple processes simultaneously on the same core

Definition

Such processes that share cores are known as hardware threads.

- They are not the same as software threads that share the virtual address space.

Coarse-grained Multithreading



- Run instructions from thread 1 for *k* cycles.
- Then switch to thread 2, then to thread 3, thread 4, thread 1, ...
- Separate program counters, ROBs and retirement register files per thread
- Each instruction packet, rename table entry, LSQ entry, physical register is tagged with the thread id
- If there is a high-latency event like an L2 miss, the core can switch to a new thread.

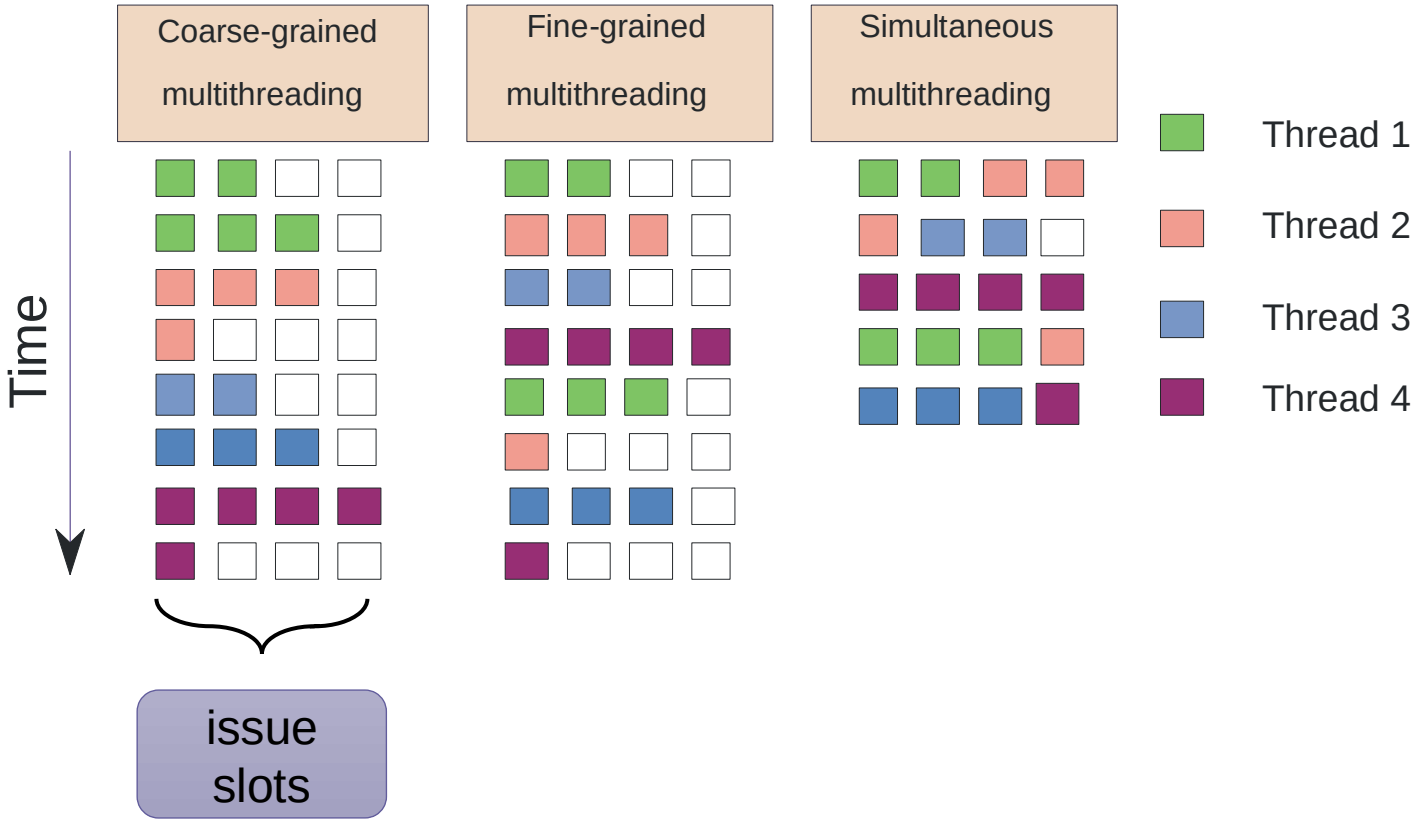
Fine-grained Multithreading

- Coarse-grained multithreading does not let the processor idle, if there is a high-latency event.
- Fine-grained multithreading reduces *k* to 1-5 cycles.
- We can tolerate low-latency events like L1 misses.
- There is an additional overhead of excessive thread switching.

Simultaneous Multithreading (SMT)

- Run multiple hardware threads simultaneously
- Dynamically split the issue slots between the threads
- There are several heuristics for partitioning the issue slots
  - Fairness
  - Instruction criticality (higher priority to loads)
  - Thread criticality: real time, non-real time
  - Based on instruction type (expected ILP, etc.)
- Hyperthreading ⎵ SMT with static partitioning (typically 50:50)

Comparison of Different Hardware Multithreading Schemes



Contents

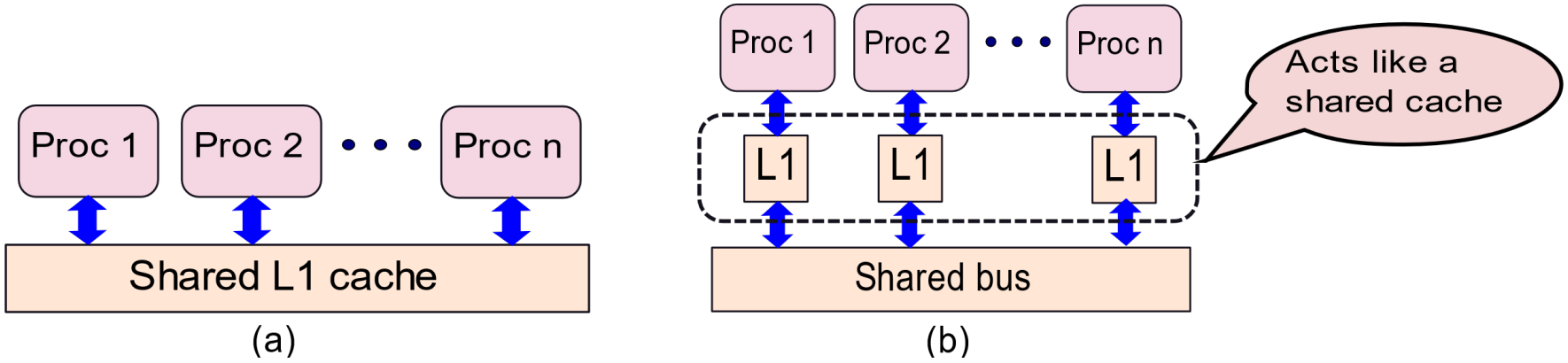
Issues with Large Caches

- Large caches' access times can be between 10-50 cycles
  - Requests need to traverse the NoC
- If we have many cores, they will make parallel accesses.

We are dealing with large, slow, multi-ported caches.
This is not possible to build.

Create large distributed caches.

Notion on a Distributed Cache



Make a set of small caches act like one single, large cache.

Each sister cache is small and fast.

Many parallel accesses.

Shared vs Distributed Caches

We want a set of small L1 caches (sister caches) act like one, single, large L1 cache. How is this possible?

Consider accesses to a single memory address/variable, *x*

- If we have a single physical location for *x,* then there is no advantage of a distributed design.
- If we have multiple locations for storing the variable *x*, then the replicas have to contain the same value.
- This is the problem of coherence.

Coherence ⎵ Make a set of locations behave like a single location.

Consider Multiple Locations

Assumptions

- All global variables are initialized to 0. They start with u, v, x, y, or z.
- All local variables are mapped to registers. They start with a ‘t’.
- The delay between the execution of consecutive instructions can be indefinitely large. Instructions across threads can execute in any order.

Thread 1	Thread 2
x = 1	t1 = y
y = 1	t2 = x

- Is the outcome <t1, t2> = <1,0> possible?
- It for some reason bothers us because no sequential ordering of instruction executions can produce this outcome.

This outcome is indeed valid on many machines

Thread 1	Thread 2
x = 1	t1 = y
y = 1	t2 = x

- Thread 1 sets *x* to 1
- It sends an update message on the NoC. The message gets caught in congestion.
- Thread 1 sets *y* to 1. The corresponding message on the NoC is swiftly delivered.
- Thread 2 reads *y* to be 1.
- Thread 2 reads *x* as 0 (initialized value)

This is indeed possible!

One More Example

Thread 1	Thread 2
x = 1	y = 1
t1 = y	t2 = x

Can we read <t1, t2> = <0,0> ?

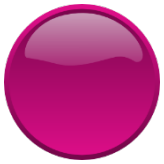
The loads are issued early, and the writes happen at commit time.

Set of Valid Outcomes

Given a parallel program, what are the set of valid outcomes?

This depends on the system on which the program is running. It depends on ....

- The pipeline
- Memory system
- NoC
- Memory controller (for off-chip memory)



Every processor has a set of specifications that specify the allowed outcomes/behaviors. If the behavior is consistent with the specifications, the execution is said to be consistent.

This specification is known as the memory model or memory consistency model.

Difference between Memory Consistency and Memory Coherence

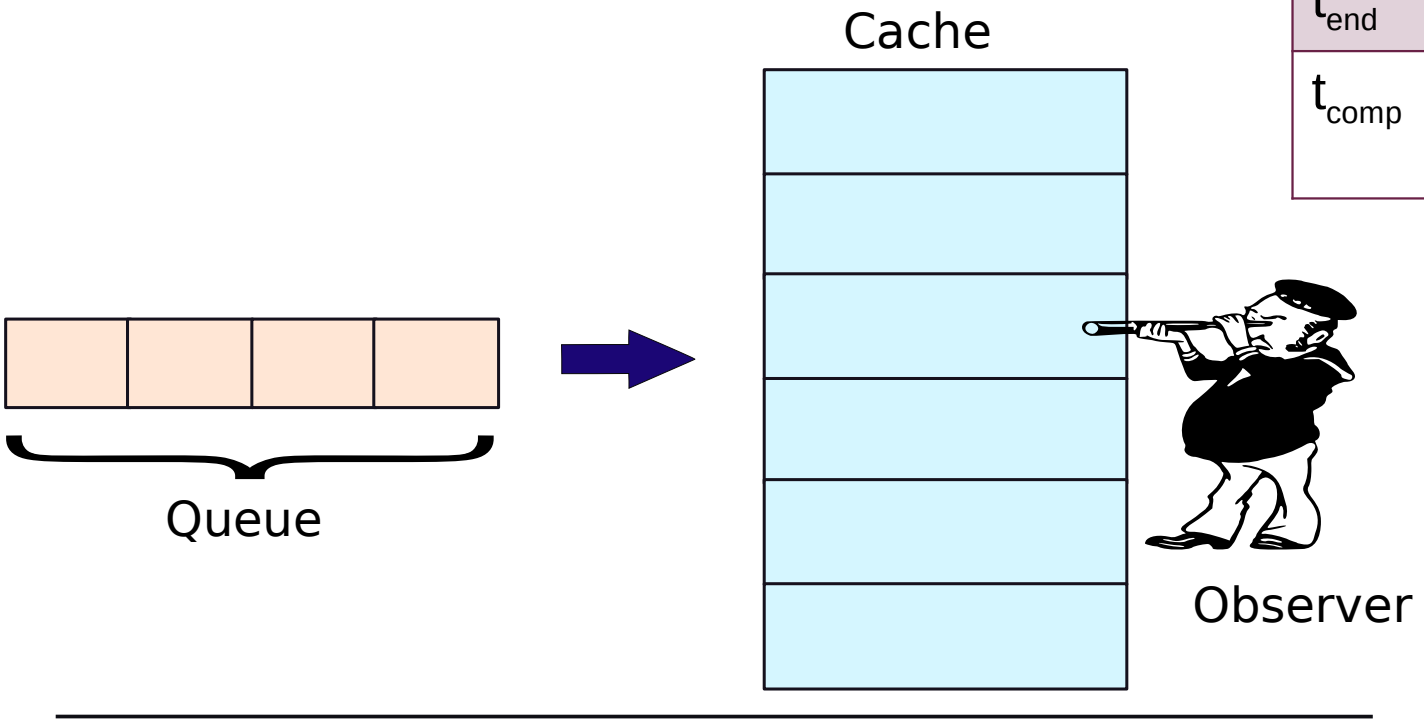
A memory consistency model or a memory model is a policy that specifies the behavior of a parallel, multithreaded program. In general, a multithreaded program can produce a large number of outcomes depending on the relative order of scheduling of the threads, and the behavior of memory operations. A memory consistency model restricts the set of allowed outcomes for a given multithreaded program. It is a set of rules that defines the interaction of memory instructions between each other.

Coherence is a subset of the overall memory model that specifies the behavior with respect to a single location.

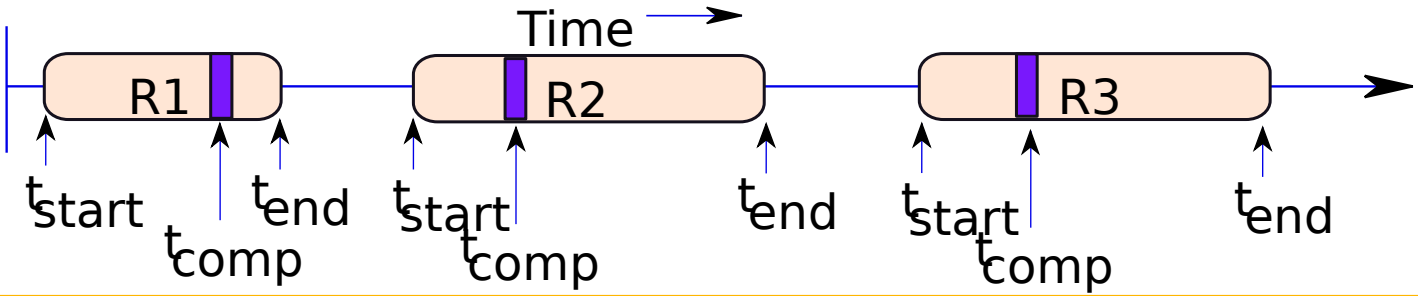
Theoretical Foundations

Every Observer has a Point of View

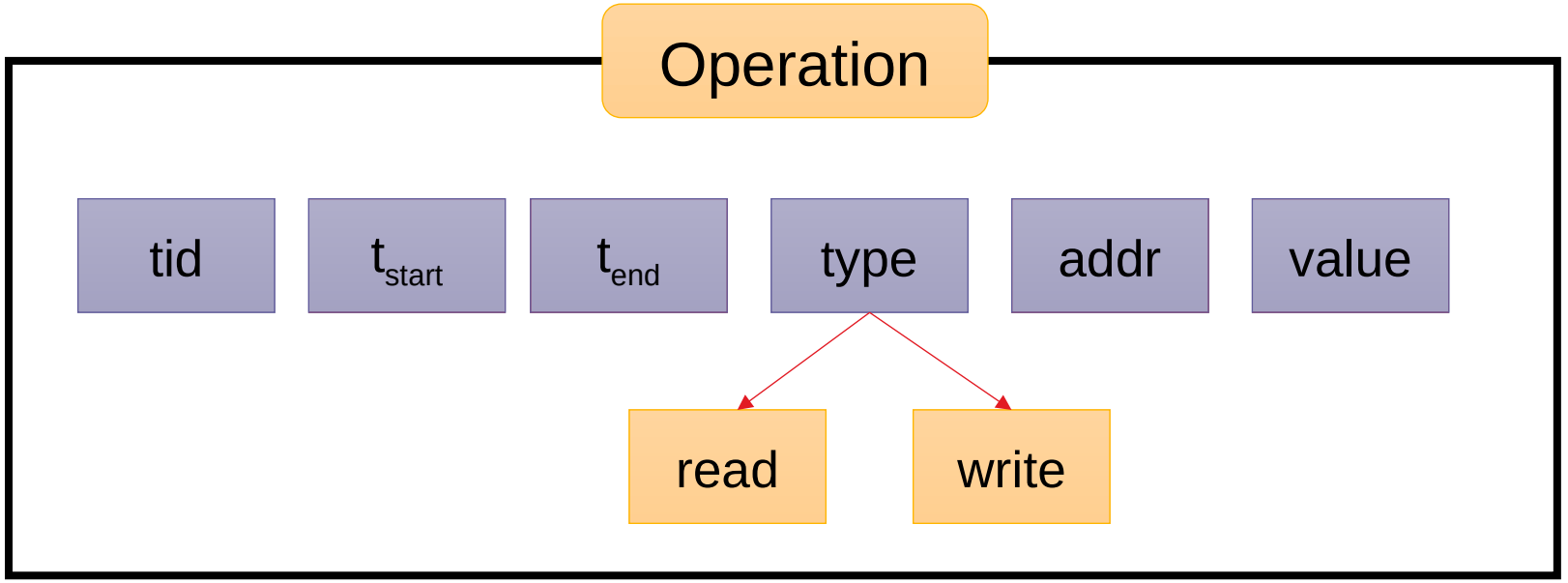
t_{start}	Start time
t_{end}	End time
t_{comp}	Completion time

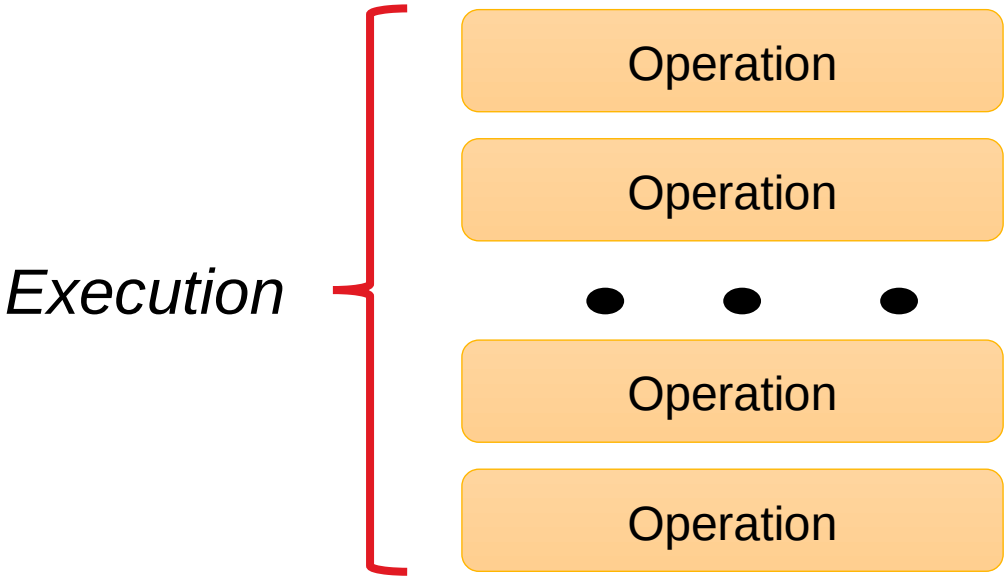


neline of memory requests seen by the observer:



Definition of an Execution





More about Executions

Sequential Execution → All the operations are ordered.

Legal Sequential Execution → Every read operation returns the value of the latest write operation to the same address.

The observer sitting on a memory location needs to observe a legal sequential execution.

Observer sitting on a Core

- In the case of a load, this relationship still holds ▭
- However, things change for a store.
- We don’t know when it will complete. The store will be put on the NoC. It may reach the cache bank a long time later.
- The operation ends when the store leaves the ROB.
- We can very well have ▭
- Even in this case, the observer on the core expects to see a legal sequential execution

Otherwise, the execution will not make any sense ....

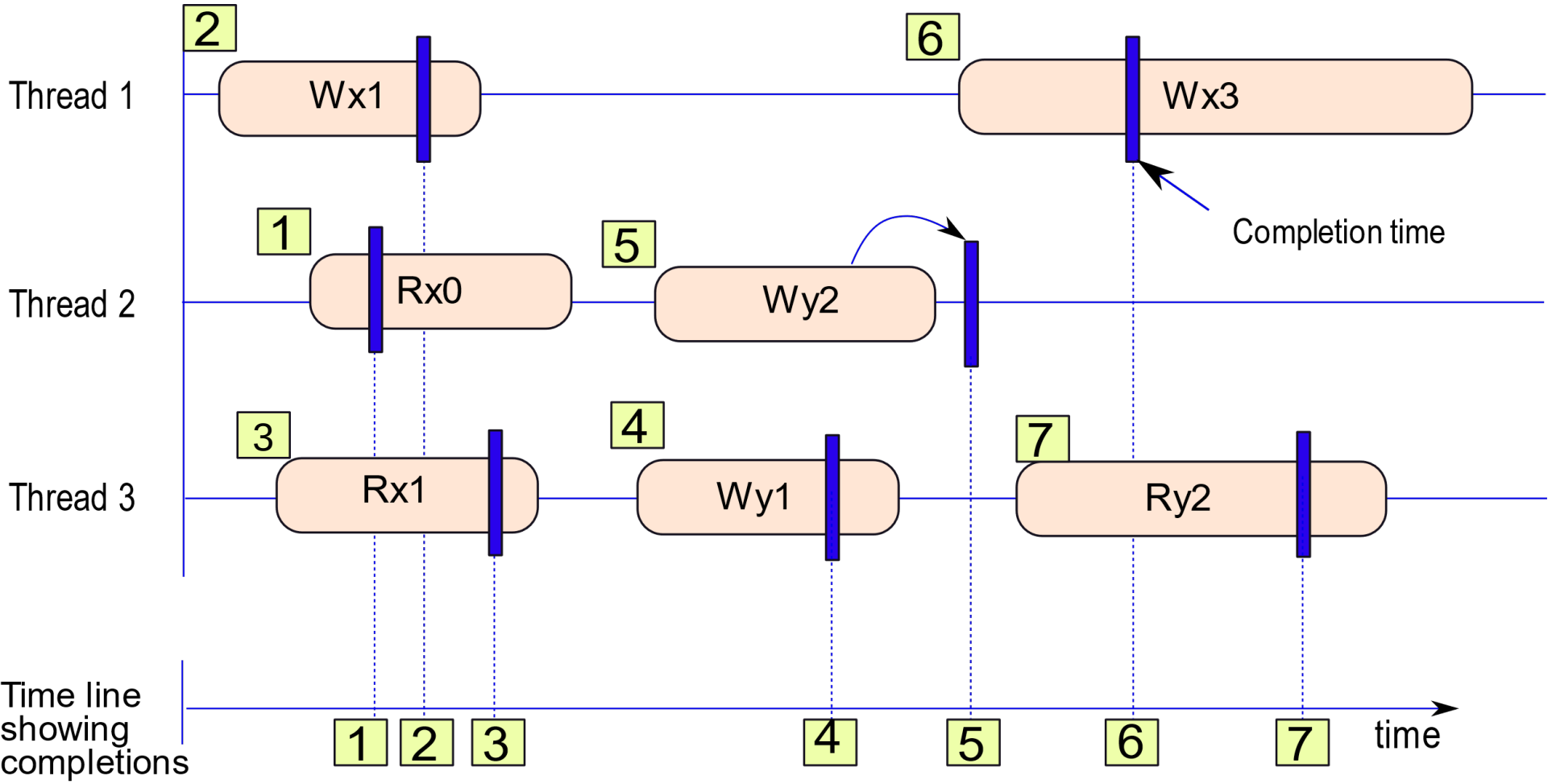
Parallel Executions

Consider a parallel execution.

Term	Meaning
Rx1	Read the value of *x* as 1
Wy2	Set y = 2

Multiple threads: one observer per thread. Each observer records the local execution history of a thread.

A Parallel Execution



Note the time line: ordered by the completion time.

Issues with Parallel Executions

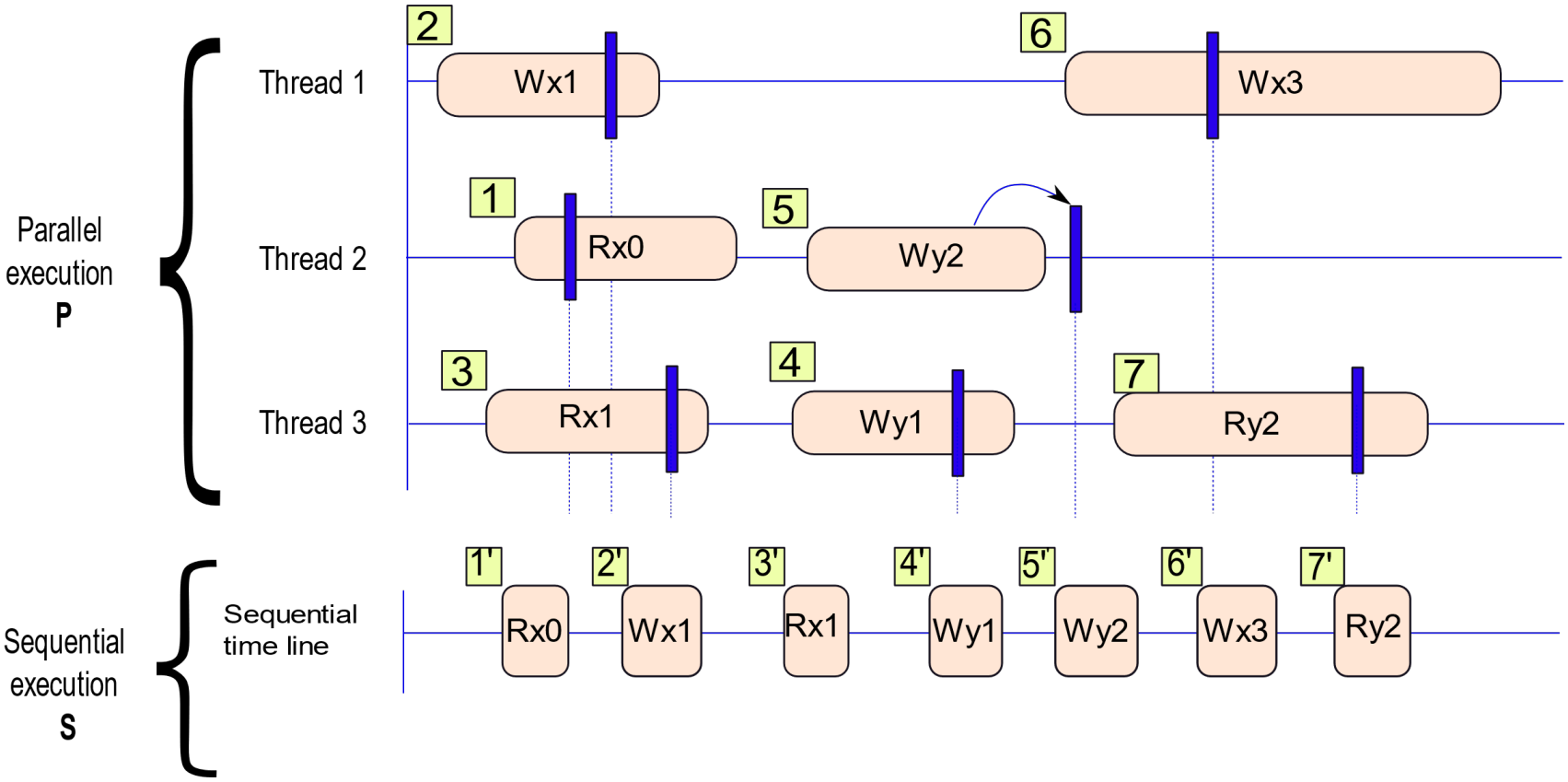
- **Atomicity** ␣ Every operation has a single completion time. It appears to execute instantaneously at that point of time.
- Have multiple observers (one per each thread) with their points of view does not help.
- It is better if we can think of parallel executions in terms of sequential executions ␣ It is very intuitive.
- For this, we need to introduce the notion of the equivalence of executions: P and S.

Equivalence of Two Executions

Equivalence of two executions

Expression	Meaning
P | T	All the operations issued by thread T (in the same order). This is an ordered sequence.
	There is a one-to-one mapping between the two sequences of operations.
	For all *T,*

Example of Execution Equivalence



Mapping

$1 \sqsubset 1', 2 \sqsubset 2', \ldots, 7 \sqsubset 7'$

Sequential Consistency

When is a parallel execution equivalent to some sequential execution?

Sequential Consistency (SC)

When a parallel execution is equivalent to a legal sequential execution and the order of operations in the sequential execution is as per program order, we say that the execution is sequentially consistent.

We can interleave the executions of different threads such that they are arranged sequentially, every read receives the value of the latest write, and for each thread the operations are arranged in program order.

Key Properties of SC

- Writes are atomic
- Program order is preserved
- SC = Atomicity + Program Order

Reordering accesses to different addresses might create havoc in multiprocessor systems.

Examples of Single-variable Programs

T1	T2	T3
Wx1	Rx0	Rx1
Wx2	Rx2	Rx2

Execution in SC

Rx0 $\sqsubset$ Wx1 $\sqsubset$ Rx1 $\sqsubset$ Wx2 $\sqsubset$ Rx2 (T2) $\sqsubset$ Rx2 (T3)

T1	T2	T3
Wx1	Rx1	Rx2
Wx2	Rx2	Rx1

Execution that is not in SC

This behavior is non-intuitive. For *x* there appear to be different storage locations.

PLSC

If we consider all the accesses to a single variable (memory location), let such an execution be always in SC. This is known as the PLSC (Per Location Sequential Consistency) constraint. It is needed to provide the illusion of a single memory location, even if we have a distributed cache.

SC for Multi-variable Programs

T1	T2
Wx1	Wy1
Ry0	Rx0

Can <x,y> = <0,0>?

This execution is not in SC

PLSC does not guarantee SC?

Accesses w.r.t *x*

T1	T2
Wx1	
	Rx0

Accesses w.r.t *y*

T1	T2
	Wy1
Ry0	

Both are in PLSC

Why not make all machines guarantee SC?

T1	T2
x = 1	y = 1
t1 = y	t2 = x

Threads can reorder accesses to different addresses.

- The outcome <x,y> = <0,0> is clearly not in SC.
- This is because
  - The loads will be sent early to the cache
  - The stores will only be sent at commit time
  - This means that both the loads will read *x* and *y* to be 0
- If we still need SC then
  - Loads will have to be issued at commit time
  - All the benefits of OOO execution and the LSQ will go away

For high performance we need to sacrifice SC

- It prohibits many performance-enhancing optimizations

Non-atomic Writes

T1	T2	T3
Wx1	while (x != 1) {}	while (y != 1) {}
	Wy1	Rx0

T1	T2	T3
Wx1	Rx1	Ry1
	Wy1	Rx0

T1	T2	T3
Wx1	Rx1	
		Rx0

T1	T2	T3
		Ry1
	Wy1	

- This execution is clearly not in SC
- It is however in PLSC. Verify
- However, the write to *x* appears to have different completion times for different threads. The write is non-atomic.

Non-Atomic Writes - II

- This is possible in many architectures such as IBM and ARM machines
- *Thread 2* is basically reading the write to *x* early ␣ before it is visible to *Thread 3*
- This is possible if multiple locations store the variable *x*, and Core 2 is closer to Core 1 and Core 3 is much farther away.
- In a system with an NoC, this can easily happen.

Given that this execution is in PLSC, let us accept it.

Behaviors not in PLSC

T1	T2	T3
Wx1	Rx1	Rx2
Wx2	Rx2	Rx1

- This behavior is not in PLSC
- SC implies PLSC, PLSC does not necessarily imply SC
- Writes atomic w.r.t. one location need not be atomic w.r.t. accesses for multiple locations.
- Such behaviors break the notion of shared memory completely

Let us thus allow non-atomic writes as long as PLSC is preserved.

Summary Till Now

SC is intuitive yet impractical

- SC = Atomicity + Program order
- In OOO machines, it is impractical

PLSC is definitely required

- SC implies PLSC (not the other way round)
- PLSC holds in systems that have non-atomic writes.
- It is needed to provide the illusion of a single memory location in a distributed cache

From PLSC to Coherence

- Akin to the definition of SC, we can define PLSC as per-location program order + atomicity (viewed per location)
- Program order basically means that processors or cache controllers are not allowed to **reorder** accesses to the same address.
- Since caches have FIFO queues for requests, they already ensure program order.
- Consider atomicity. Because of PLSC an observer at a memory address always sees atomic writes, but an observer on a core may see non-atomic writes because of intervening accesses to other addresses.

- For a given address, let us look at the following orders of operations
  - Global order ⎵ All observers record this order between operations
  - Local order ⎵ A few observers do, and a few don't

Ordering between Accesses to the Same Variable across Threads (Observer at the Core)

Order	Implication
Read $\sqsubset$ Read	Does not matter
Write $\sqsubset$ Read	Given that a core can read another core's write early, while other cores may not even see the write, this order may not be global all the time. A core does not know when a write reaches the sister caches present in other cores. Hence, it may not agree about the values read by other cores; this order is local.
Write $\sqsubset$ Write	If this order is not global, then the same variable will end up with multiple final states. This is not allowed. Hence, in all systems this order is global.
Read $\sqsubset$ Write	Writes are globally ordered. Assume one core records $W_i \sqsubset R_i \sqsubset W_j$. All the cores will record $R_i \sqsubset W_j$ because the core that issued R_i could not have seen W_j else it would have read a different value.

Implications of PLSC (Observer at the Core)

1. **Write** $\sqsupset$ **Write** order is global
2. **Read** $\sqsupset$ **Write** order is global
3. **Write** $\sqsupset$ **Read** order is global only for machines with atomic writes.

1 2

Axioms of Coherence

Write Serialization	Writes to the same location are globally ordered.
Write Propagation	A write is eventually seen by all the threads.

SC Using Synchronisation Instructions

Communicating a Value between Threads

T1	T2
value = 3	while (status != 1) {}
status = 1	temp = value

- This code will work correctly on an SC machine. It will not work correctly on other machines with other memory models.
- The ordering between the read and write in T2 may not hold.
- A store instruction fully completes when all the threads can read the stored value

fence instruction ⇨ All the instructions fetched before the *fence* need to fully complete before the *fence* instruction executes. All the instructions after the *fence* instruction cannot execute until the *fence* instruction has completed.

Code with *fence* Instructions

T1	T2
value = 3	while (status != 1) {}
fence	*fence*
status = 1	temp = value

- Regardless of the underlying memory model this code will always work
- Memory barriers
  - A *fence* is an example of a memory barrier
  - Specifies rules of completion for instructions before and after the barrier (in program order)
  - Store barrier ⊔ Ensures an ordering between store instructions before and after the barrier instruction.

Acquire and Release Instructions

- Acquire instruction ⊔ No instruction after the acquire instruction in program order can execute before it has completed.

- Release instruction ⊔ The release instruction can only complete if all the instructions before it have been fully completed. Note that the release instruction allows instructions after it to execute before it has completed.

Theory of Memory Models

Where do we stand ...

- Processors reorder instructions that access different addresses
- Very problematic in multithreaded systems ⇒ non-intuitive behaviors
- Writes can be non-atomic

How do we reason about the correctness of these systems?

SC gave us a mechanism where we could map a parallel execution to a legal sequential execution (LSE). We could then reason on the LSE.

What about non-SC executions?

Method of Execution Witnesses

- A given piece of parallel code can have many different executions.
- The valid outcomes are dependent on the memory model of the machine.
- For each execution (valid or invalid), we can create an execution witness (a *graph* containing nodes and edges).
- If we can convert the execution witness to a sequential execution that obeys the memory model, the execution is valid, otherwise it is invalid.

Execution Witness

T1	T2
x = 1	y = 1
t1 = y	t2 = x

SC Execution	
1	x = 1
2	t1 = y
3	y = 1
4	t2 = x
<t1,t2> = <0,1>	

It is a **graph**

- The nodes are the instructions
- We can have edges between the instructions based on the orders that are guaranteed by the memory model.
- We can have two edges: global and local
- These are happens-before edges ▭ If means that event *A* happened first and then event *B* happened after that. It could be immediately later or after a very long time.
- A global *hb* edge (***ghb***) is agreed to by all threads.

Program Order (po) Edge

All the edges are between memory operations issued by the same thread regardless of their address.

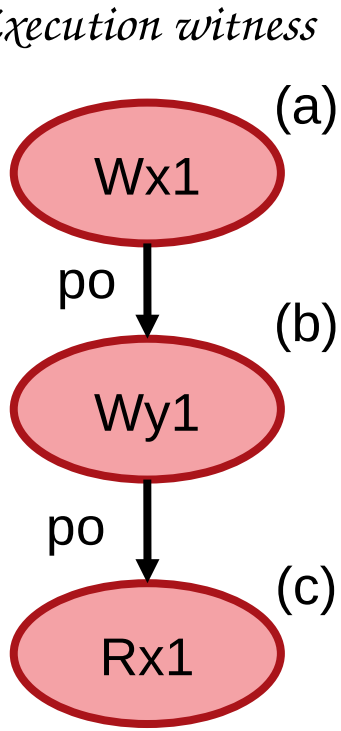
Edge	Description
po_{RW}	Read $\sqsubset$ Write edge
po_{RR}	Read $\sqsubset$ Read edge
po_{WR}	Write $\sqsubset$ Read edge
po_{WW}	Write $\sqsubset$ Write edge
po_{IS}	read/write $\sqsubset$ synch operation
po_{SI}	synch $\sqsubset$ read/write operation

Not always global. Global only in SC.

Global

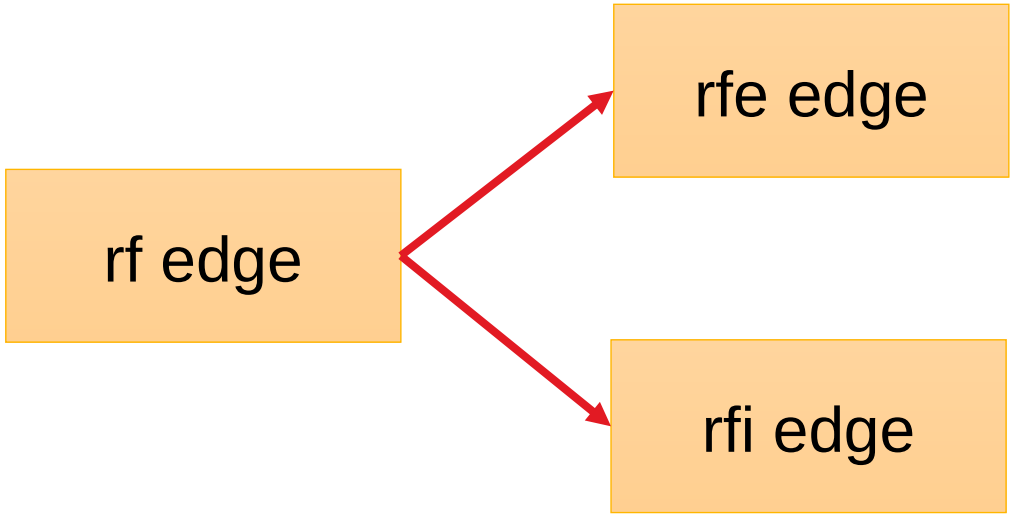
Example of a *po edge*

T1	
(a)	x = 1
(b)	y = 1
(c)	t1 = x



- We create a node for each memory operation
- We add edges between them (as per the memory model)

rf (read from) edge



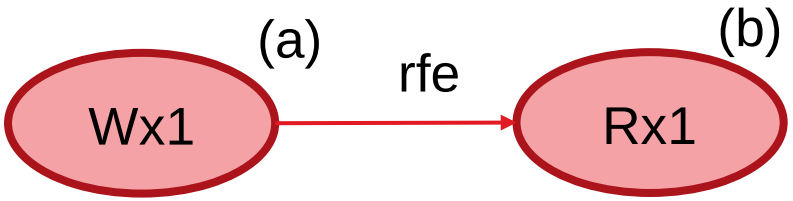
Read and write ops in different threads

Read and write ops in the same thread

- If writes are atomic, the *rfe* edge is global. Not otherwise.
- *rfi* is not global when we have optimizations like load-store forwarding where a core can read its own value before other cores can.

T1	T2
(a) x = 1	(b) t1 = x

t1 = 1

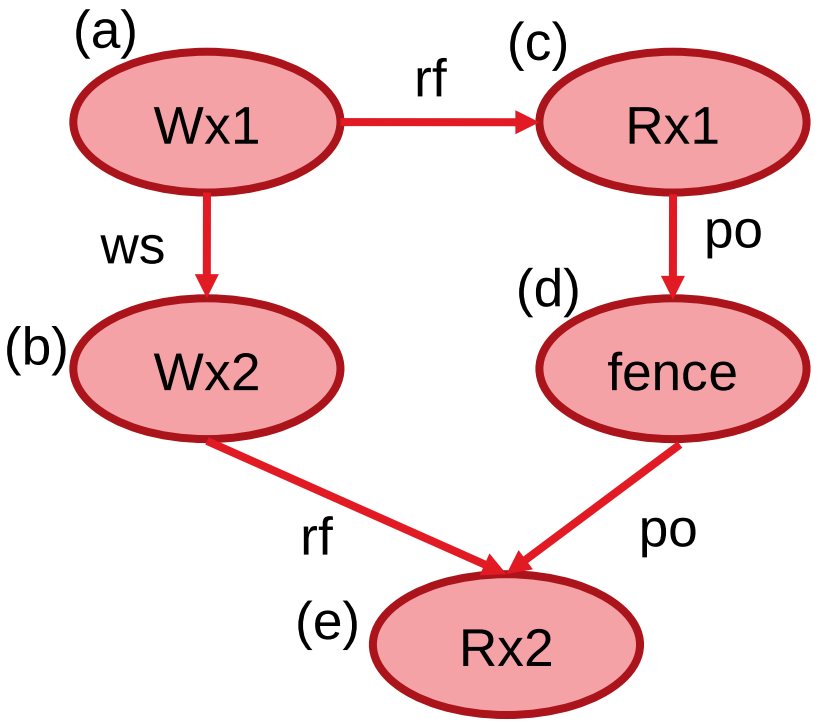


Write Serialization (ws) Edge

T1	T2	T3
(a) x = 1	(b) x = 2	(c) t1 = x (d) fence (e) t2 = x

t1 = 1, t2 = 2

Execution witness

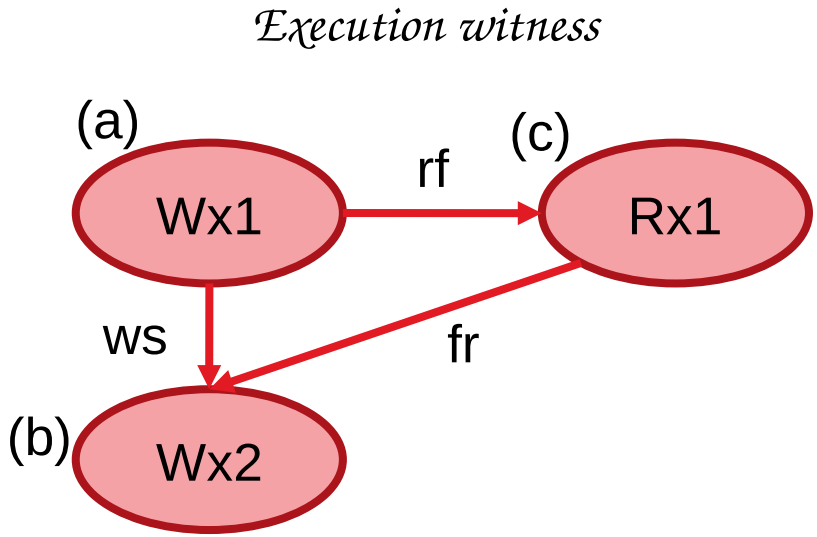


- This is always **global**. This is a direct consequence of PLSC and the requirements of coherence.

From-read (fr) Edge

T1	T2
(a) x = 1 (b) x = 2	(c) t1 = x

t1 = 1

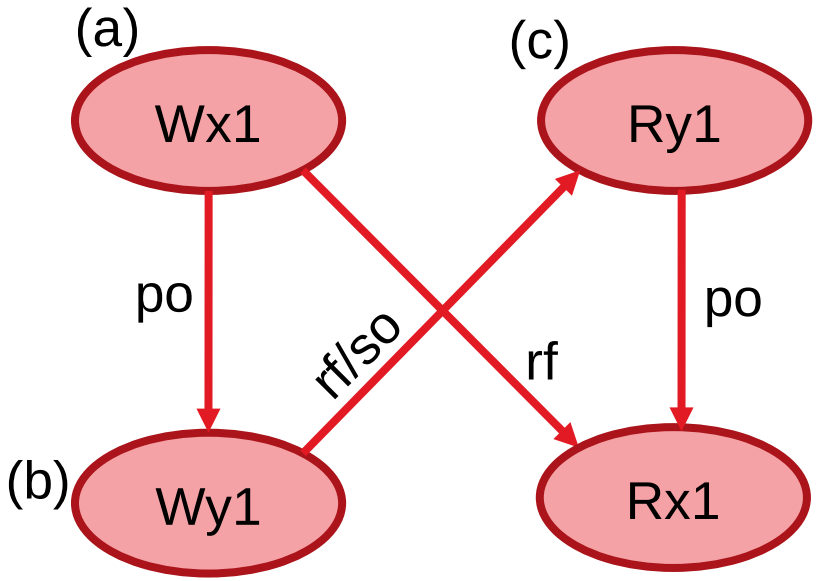


- This edge is global because of PLSC.

Synchronization Edge: *so*

T1	T2
(a) x = 1 (b) y = 1	(c) t1 = y (d) t2 = x

t1 = 1, t2 = 1



Assume *y* is a synchronization variable.

All updates to such synch variables are globally ordered.

Relationships between memory accesses

po	• program order edge (same or different addresses)
rf	• write $\sqsubset$ read, dependence (same address)
ws	• write $\sqsubset$ write (same address)
fr	• read $\sqsubset$ write (same address)
so	• Synchronisation edge between synch operations.

Summary: Memory Models

ws, *fr*, and *so* are always global

Let be global

Let be global

grf=rf and ppo=po only for SC

$$ghb = ppo \cup ws \cup fr \cup grf$$

Cycles in an Execution Witness

If a graph does not have a cycle, we can arrange all the nodes in a linear sequence such that

- If there is a path from operation *A* to *B* in the graph, then *A* appears before *B* in the linear sequence.
- This is a topological sort.

An acyclic execution witness → Sequential execution that respect *ghb*

Implications of a Sequential Execution

- For an arbitrary memory model
  - We can now establish an equivalence between a parallel execution and a sequential execution.

It may not be legal.

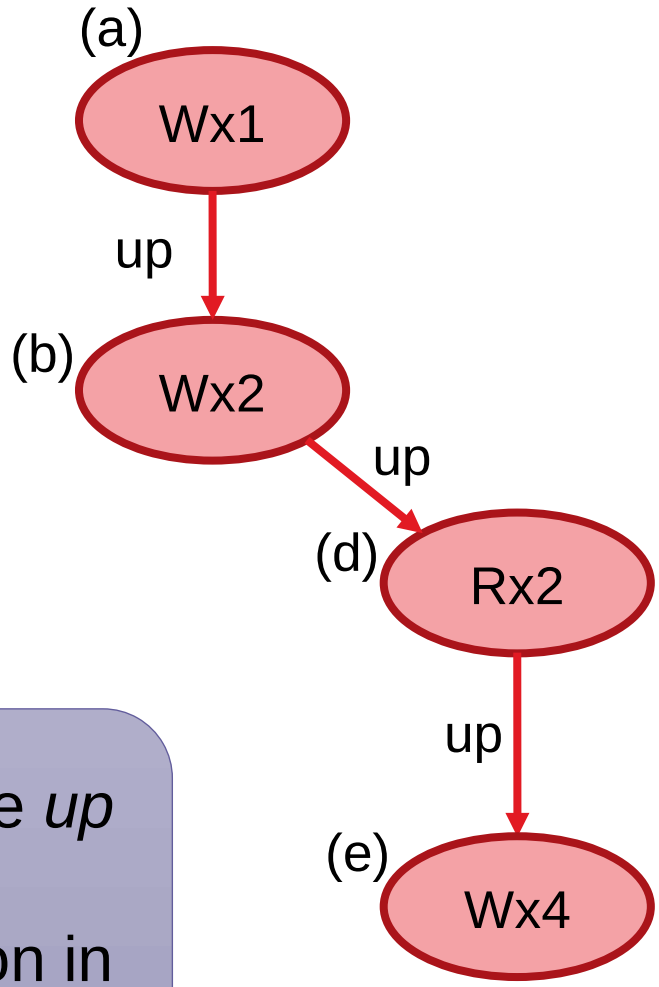
- The sad part is that it may not be legal if writes are not atomic.

At least the fact that we established an equivalence means that the execution is feasible according to the memory model.

Access to a Single Location Data and Control Dependences

Access Graphs: All accesses to the same location

T1
(a) x = 1
(b) x = 2
(c) y = 3
(d) z = x + y
(e) x = 4



Instead of *po* edges, we have *up* edges ▭ Edge between accesses to the same location in the same thread.

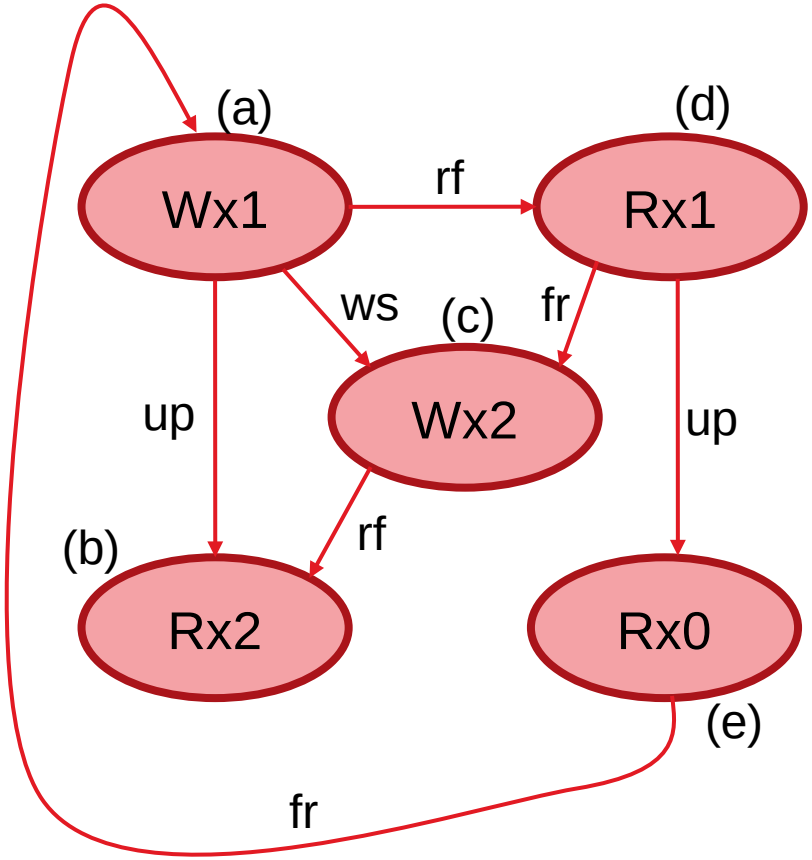
$$PLSC = po \cup rf \cup fr \cup ws$$

- PLSC is always respected regardless of the memory model.
- Hence, the access graph never has cycles.

Example

T1	T2	T3
(a) x = 1	(c) x = 2	(d) t2 = x
(b) t1 = x		(e) t3 = x

<t1,t2,t3> = <2,1,0>?

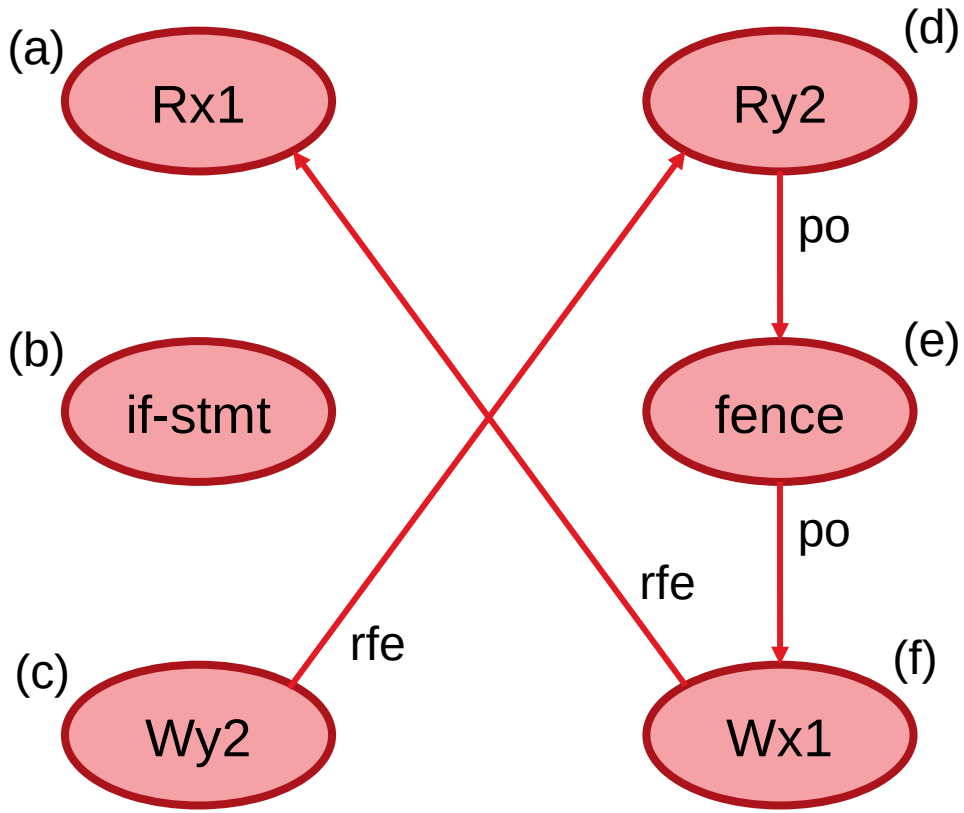


Note the *up* edges and the presence of the cycle.

Data and Control Dependences

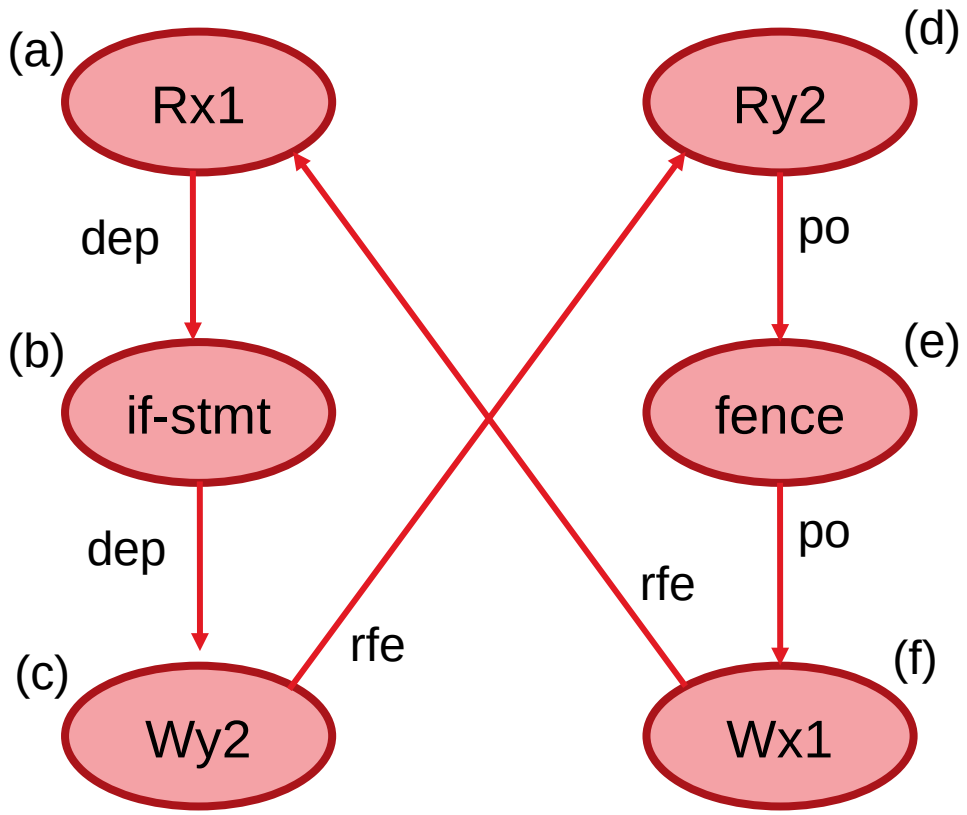
T1	T2
(a) t1 = x	(d) t2 = y
(b) if (t1 == 1) {	(e) fence
(c) y = 2 }	(f) x = 1

t1 = 1, t2 = 2



There is a clear breakdown of causality. *Rx1* seems to be happening without a preceding write. This is a *thin air read*.

Causal Graph



Three kinds of global edges: rf, gpo (all global program order edges), dep (dependences)

To stop thin air reads

$$acyclic(rf \cup gpo \cup dep)$$

Putting it all Together

Condition	Test
Satisfies the memory model	The execution witness is acyclic.
PLSC holds for all memory locations	All access graphs are acyclic.
No thin air reads	The causal graph is acyclic.

Contents

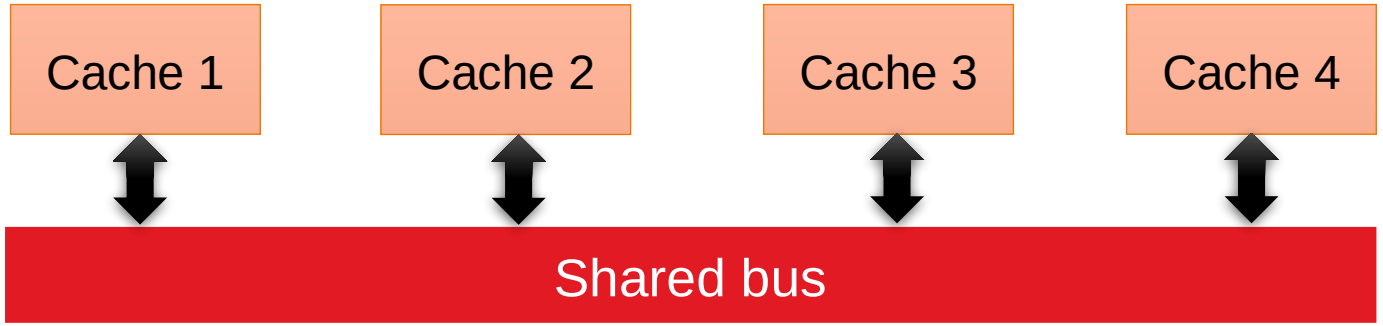
Write Update Protocol

Enforce PLSC in Hardware

All writes to the same location are seen in the same order

A write ultimately completes

Bus based Model



Write-Update Protocol

- Every cache line has three states

State	Meaning
M	Modified
S	Shared with other sister caches
I	Invalid

Event | Message Notation

Event Type	Meaning
Rd	Read request
Wr	Write request
Evict	Evict the block
Wb	Write back data to the lower level
Update	Update the copy of the block

Message Types

Message Type	Meaning
RdX	Generate a read miss message. Send it on the bus/NoC if required.
WrX	Generate a write miss message. Send it on the bus/NoC if required.
WrX.u	Get permission to write to a block that is already present in the cache.
Broadcast	Broadcast a write on the bus.
Send	Send a copy of the block to the requesting cache.

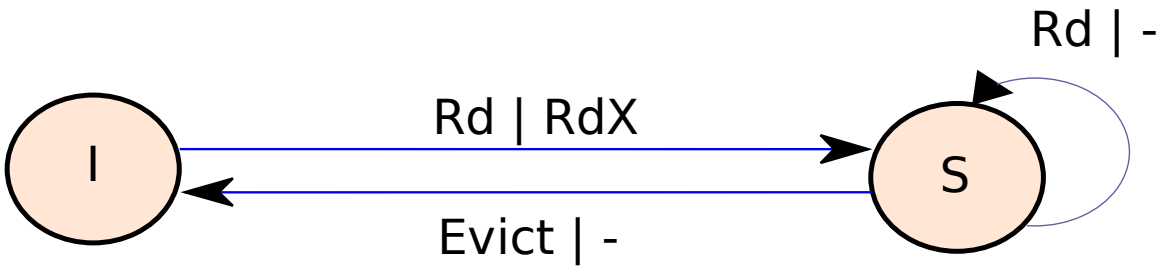
Snoopy Protocols

- Cache coherence protocols that use buses are known as Snoopy protocols.
- All messages are essentially broadcast messages.
- All the caches can read them (snoop on them).
- These are easy to design

There are serious scalability issues. Such systems do not scale beyond 4 to 8 cores.

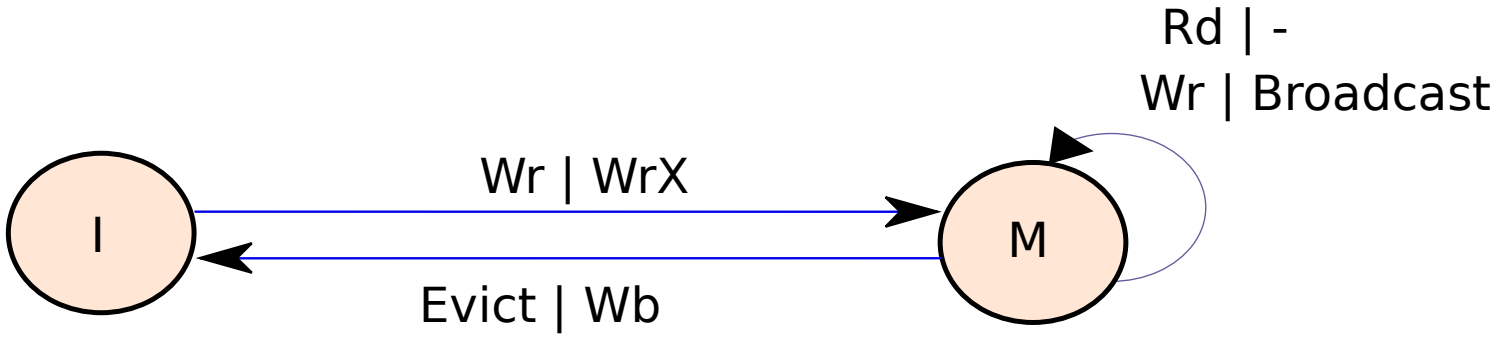
Reads

- If the block is not present, send a read miss message on the bus.
- *S* state ␣ If it is already present, don't do anything. Just read it.
- The *S* state allows seamless evictions



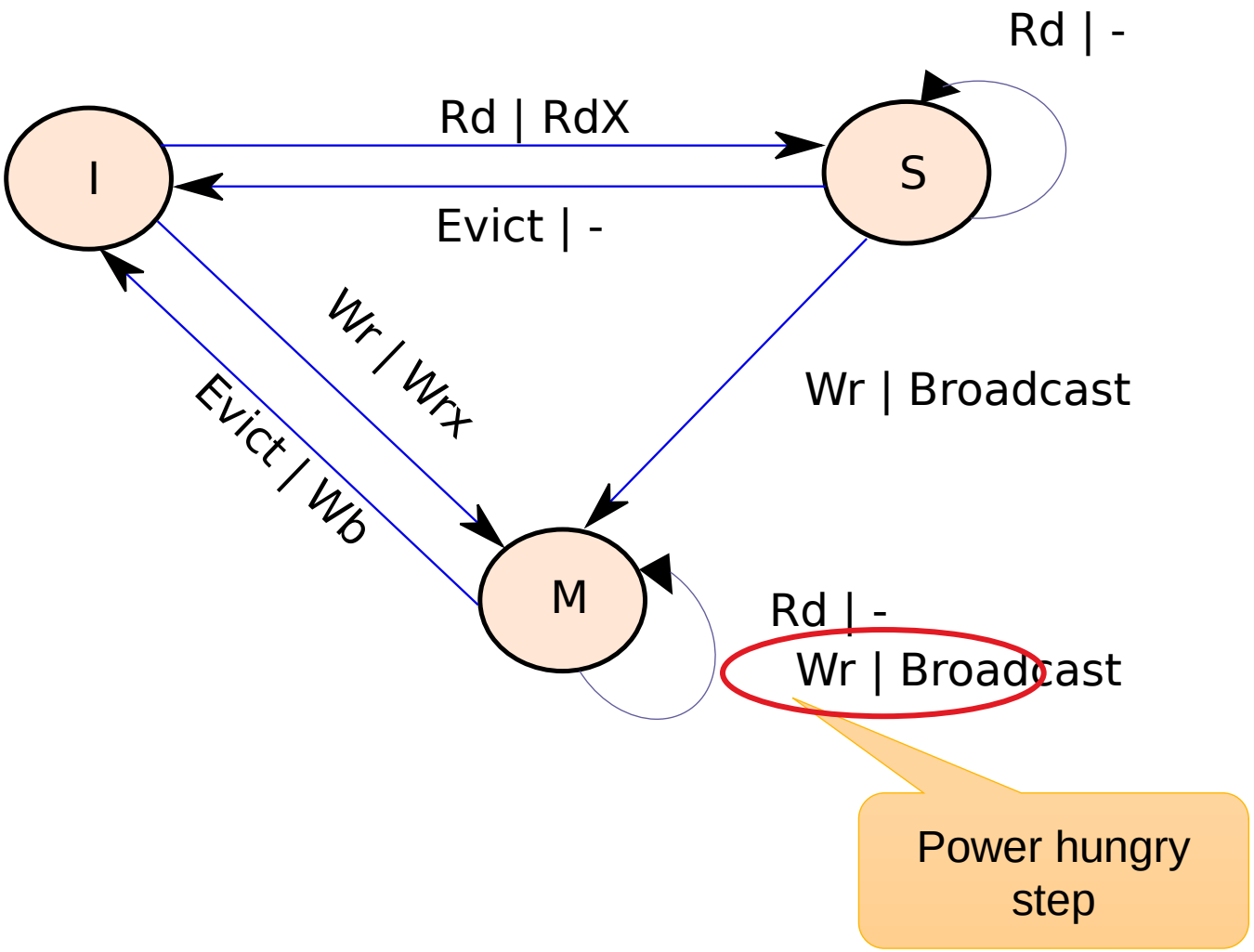
Writes

- If the block is not present, send a write miss message on the bus. Transition to the *M* state.
- If it is already present, don't do anything. Just read it.
- *M* state ⎵ If we need to write, the write is broadcasted first to the rest of the sister caches.
- The *M* state does not allow seamless evictions. We write back the data to the lower level, while evicting ⎵ Otherwise, the updates will be lost.

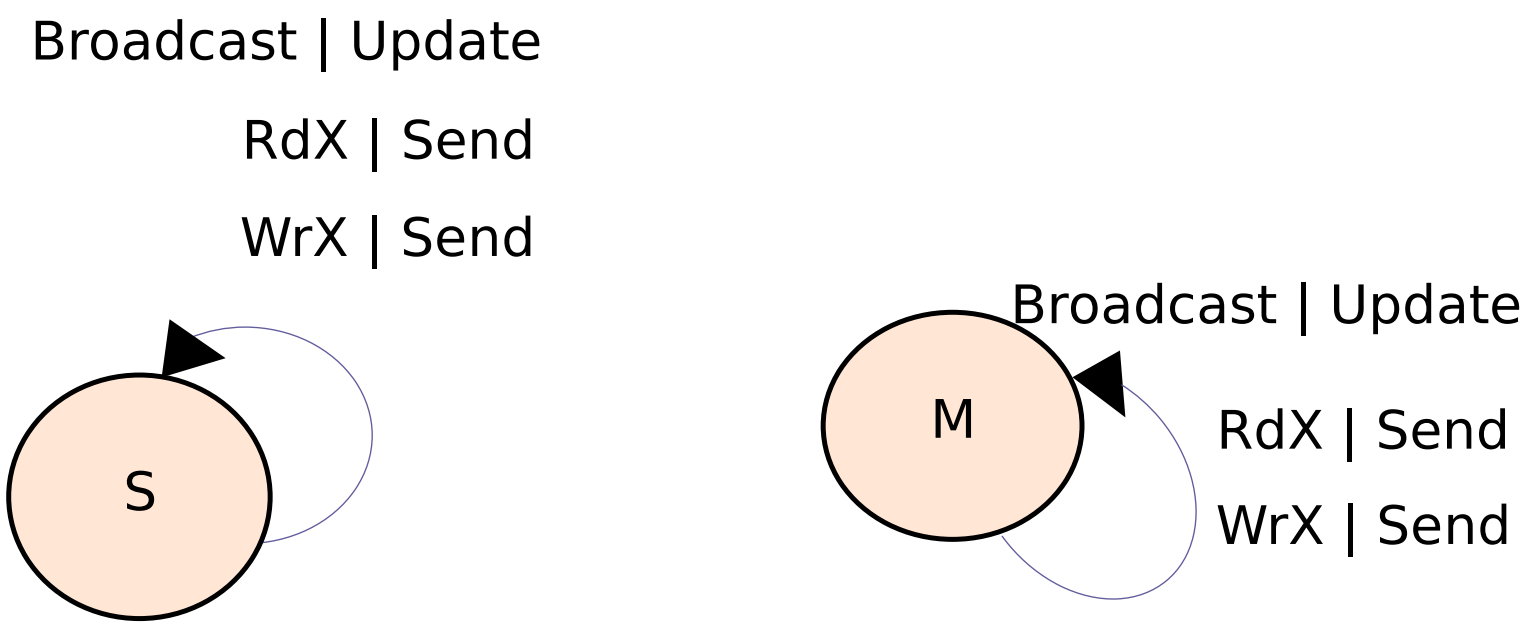


Broadcasting after every write is the issue.

All the transitions due to read, write, and evict events



Events Received from the Bus



- Given that only one sister cache can use the bus at any time, a global order of writes is automatically enforced.
- If the bus master disallows starvation, then all writes will ultimately complete.

Write Invalidate Protocol

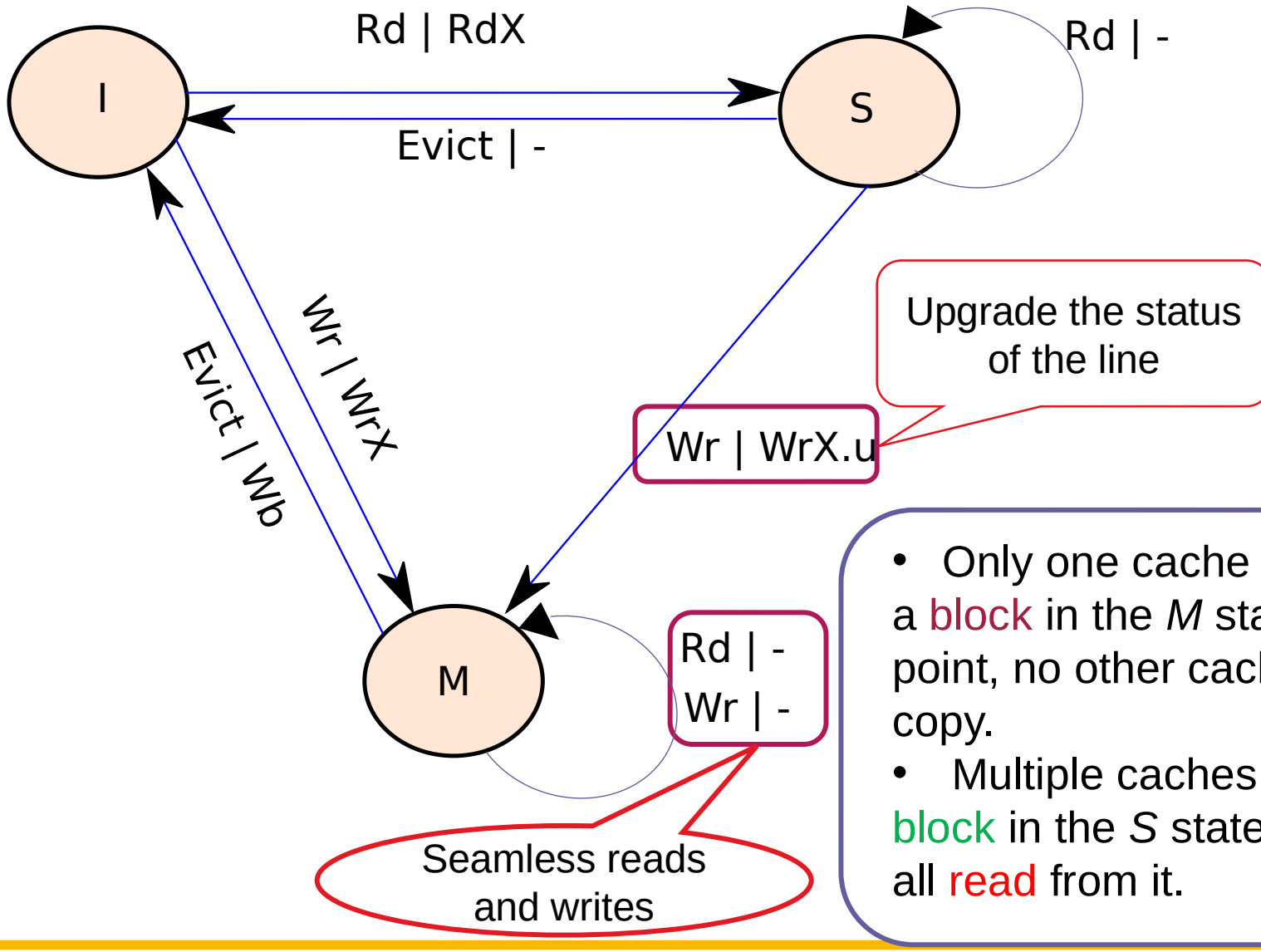
Single Writer Multiple Reader Model for each Cache Line

Single Writer

Multiple Readers

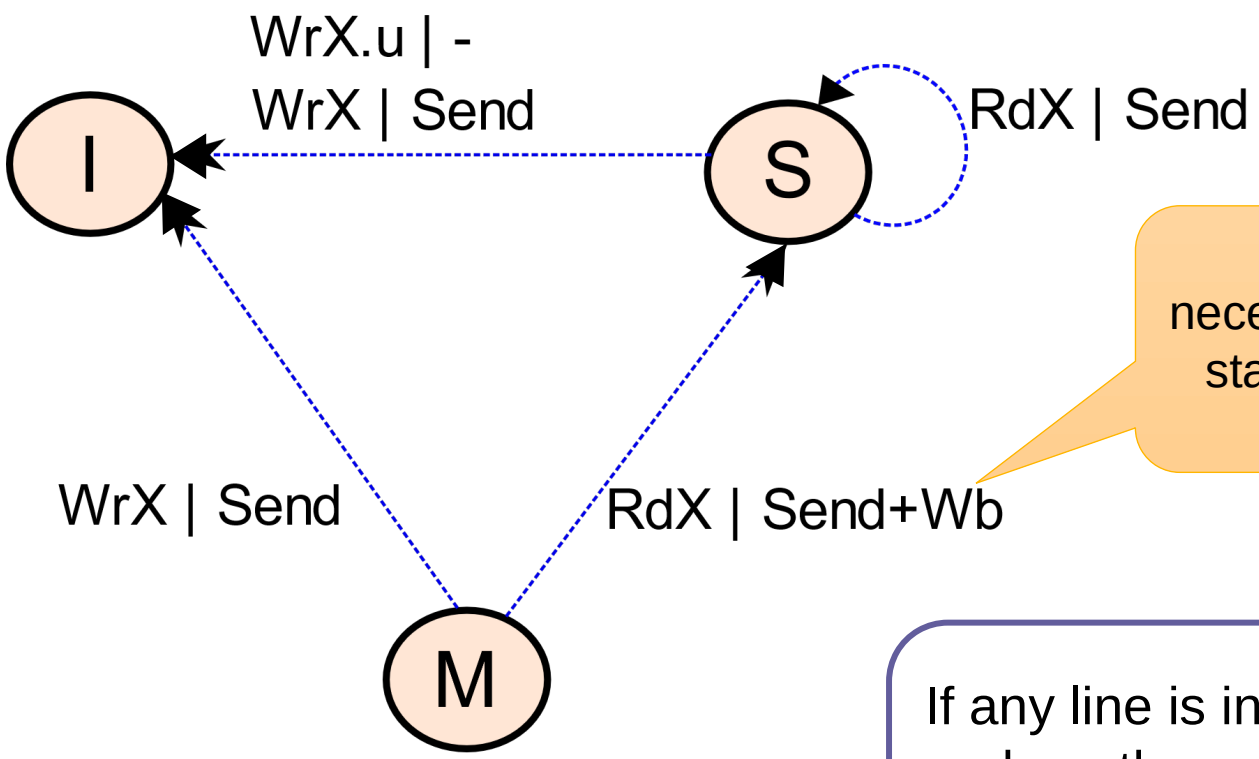
Ensures a global order of writes to the same memory location

Write Invalidate Protocol (MSI Protocol)



- Only one cache can contain a block in the *M* state. At that point, no other cache has a valid copy.
- Multiple caches can have the block in the *S* state. They can all read from it.

Transitions because of Messages Received on the Bus



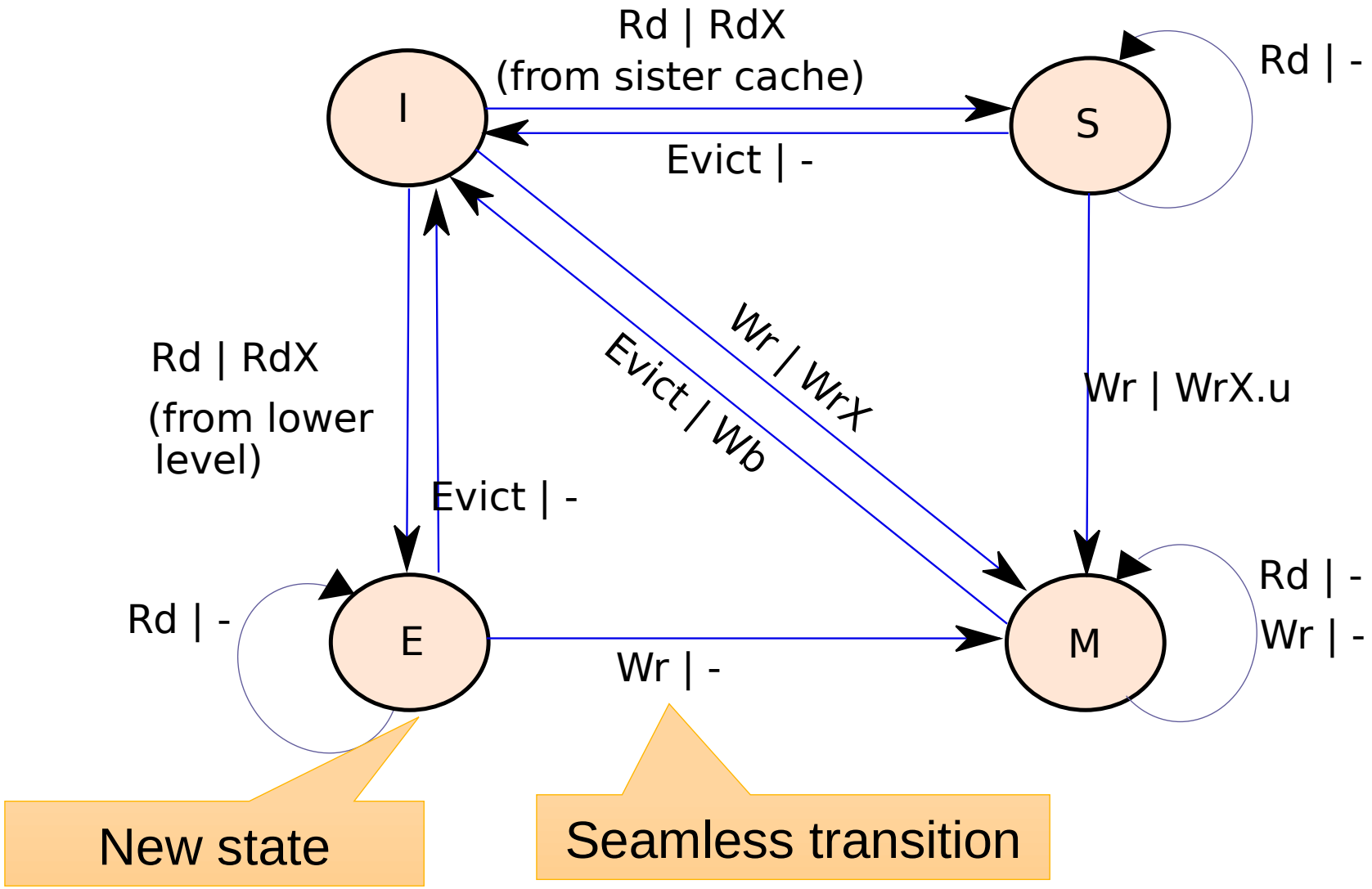
A writeback is necessary here. The *S* state has seamless evictions.

If any line is in the modified state and another cache requests for the block, then a state transition is necessary.

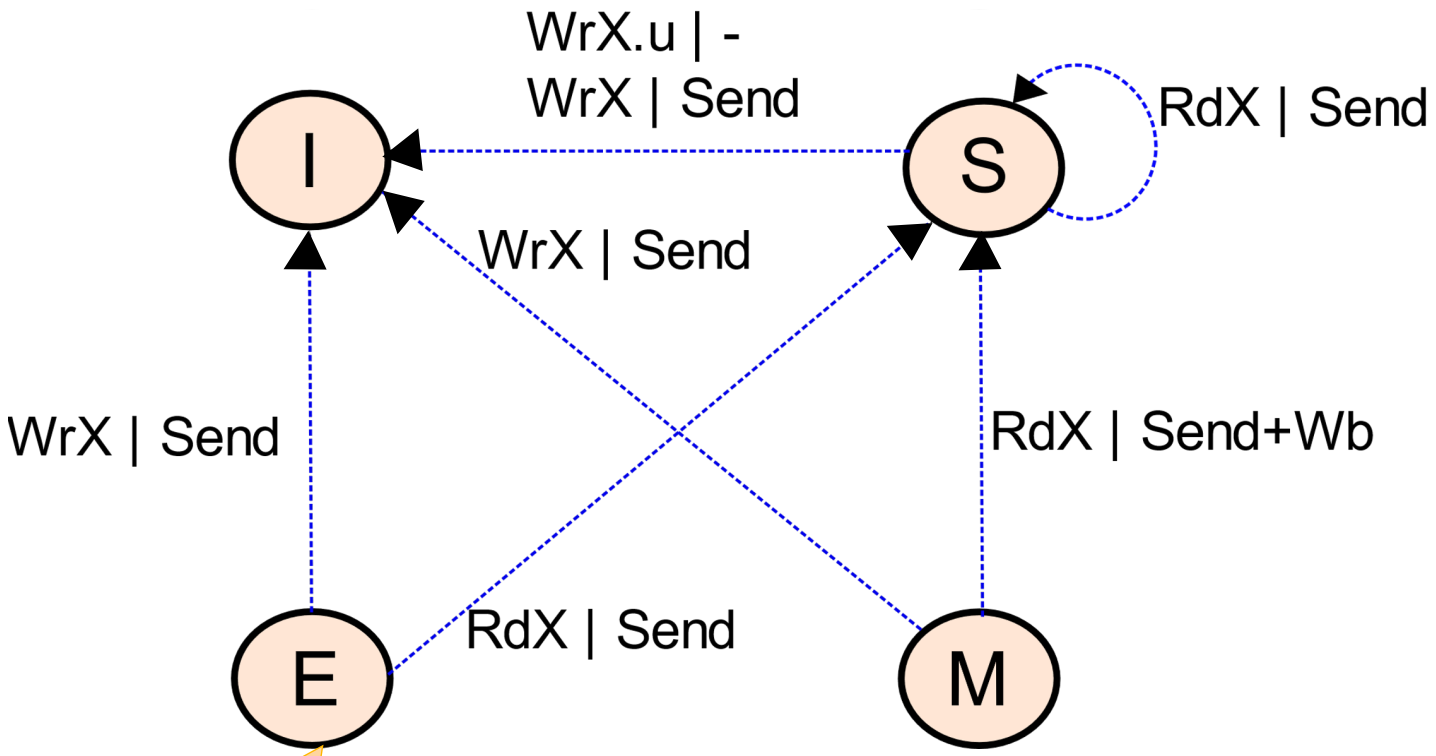
MESI Protocol

- Insight: Consider a core that reads a block, and the block is never shared. It will be read first in the shared state, and then an additional message (WrX.u) is required to transition to the *M* state.
- Can we avoid this?
- Add an additional **Exclusive (E)** state ⊏ The block is only present in one cache. If we are reading a block from the lower level, it enters the *E* state.
- The rest remains the same.
- MSI protocol ⊏ MESI protocol

MESI (Read, Write, and Evict events)



MESI (messages received from the bus)



The moment another cache requests for the data (read-only), we need to transition to the *S* state.

Important Engineering Questions

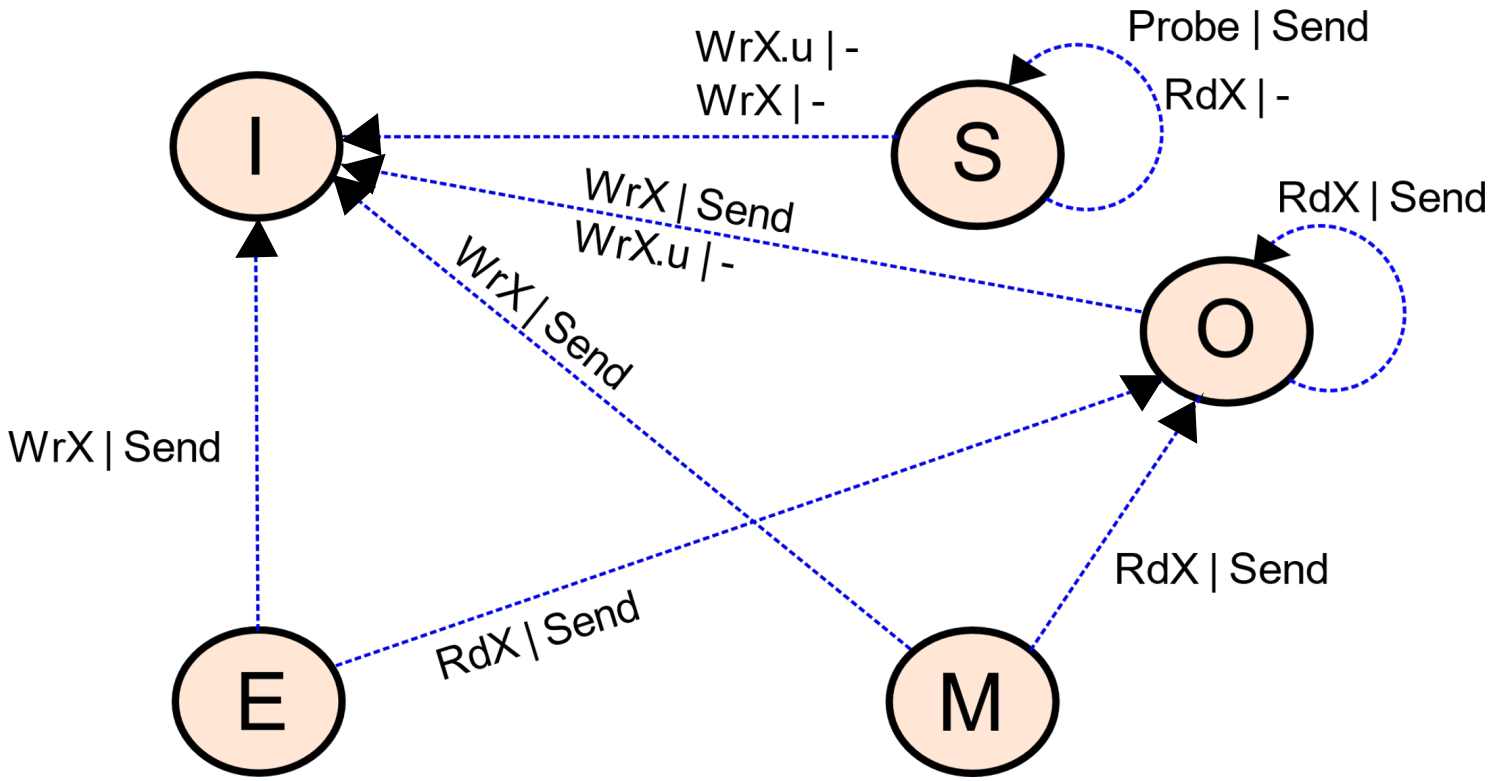
Who supplies data if a sister cache sends a read miss or write miss message?

Answer: The caches who have a copy of the block, arbitrate for the bus. The one who gets access to the bus first, sends the data. The rest snoop this value, and then cancel their request. This is an overhead in terms of time and power.

Do we have to write data back on an M $\sqsubset$ S transition?

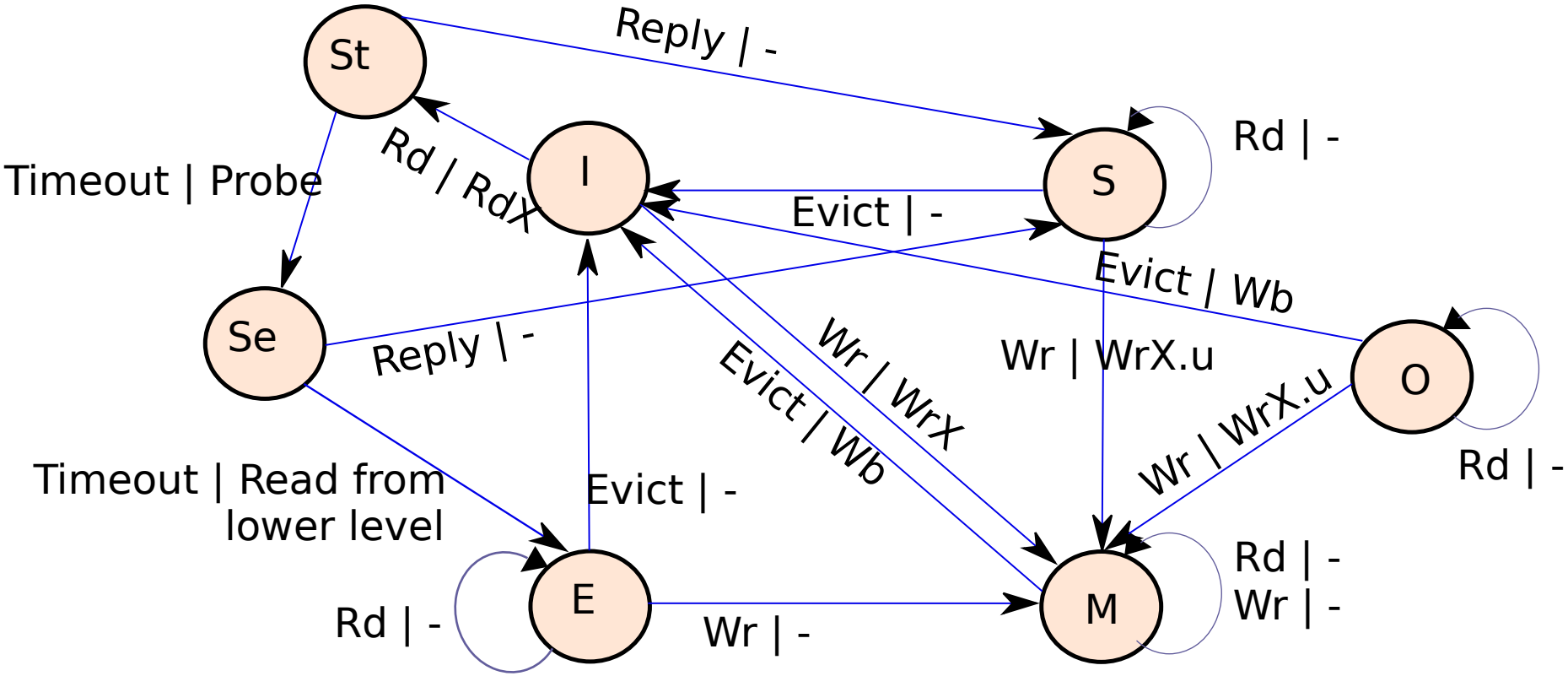
Answer: In the interest of performance and power, it will be the best if we can avoid it.

MOESI (messages received from the bus)



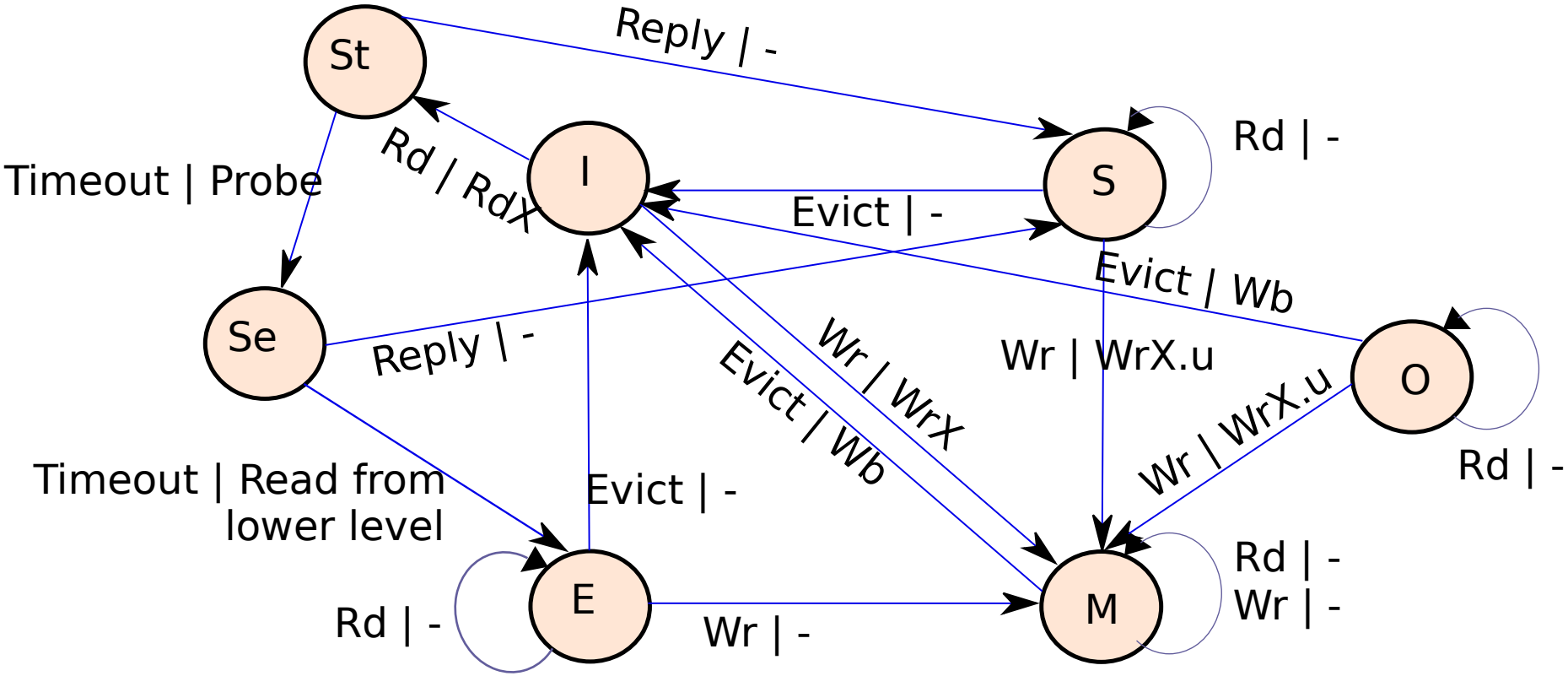
- Whenever, another cache requests for data, from an *E* or *M* state, the line moves to the *O* state.
- In the *O* state, it sends data to other requesting caches.
- We will discuss the *Probe* message in the next slide.
- From the *S* state, no data is sent.

MOESI



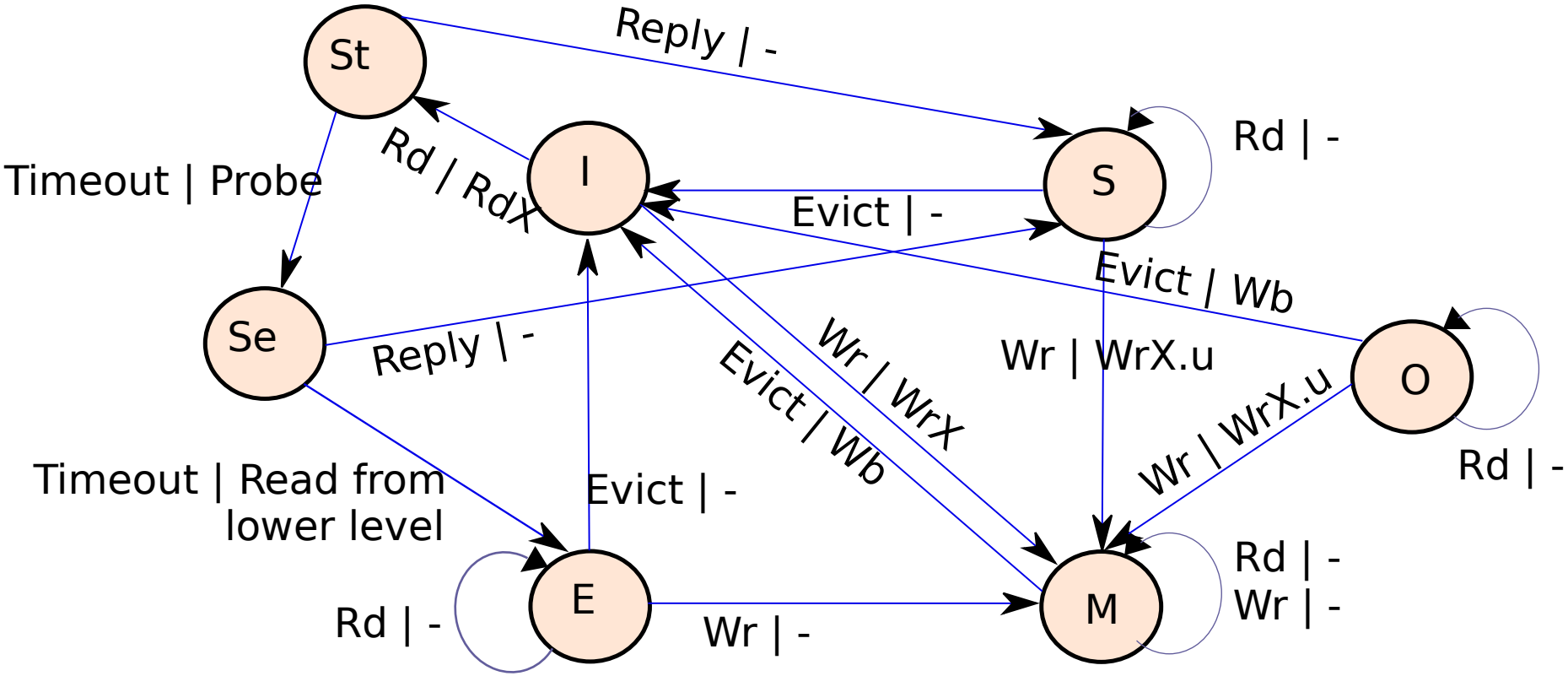
- Ignore the two temporary states – **St** and **Se** – for the time being.
- Main problem: An eviction from the *O* state, leaves us with a state where there is no owner.
- This makes the transitions from the *I* state tricky

MOESI



- We first transition to the *St* state upon encountering a read miss
- If a *Reply* is received, then the state transitions to the *shared(S)* state.
- Otherwise, there is a timeout. A *Probe* message is sent. We transition to the *Se* state.
- Again, if we receive a *Reply* from a cache (line in the *S* state), we transition to the *S* state.

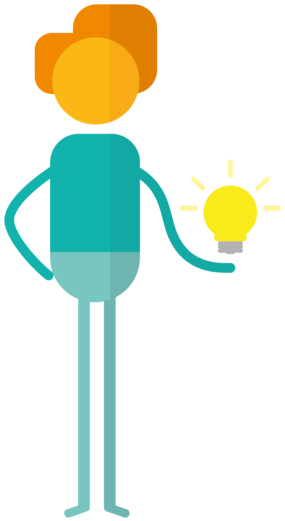
MOESI



- If there is a timeout in the *Se* state, we read from the lower level.
- Transition to the *E* state.

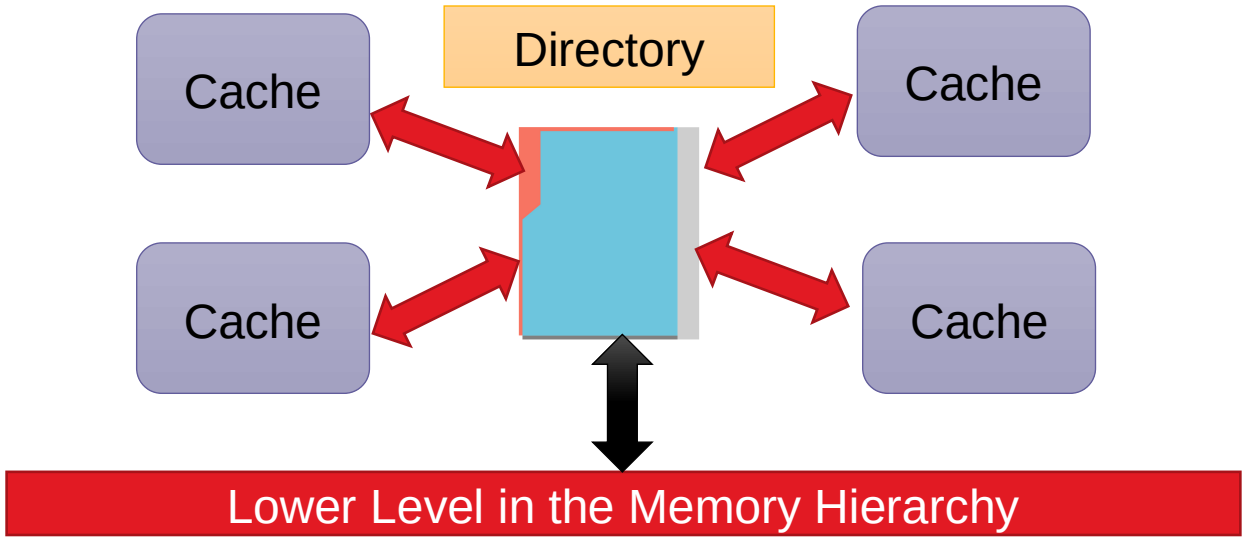
Directory Protocol

A Scalable Solution: Directory Protocol

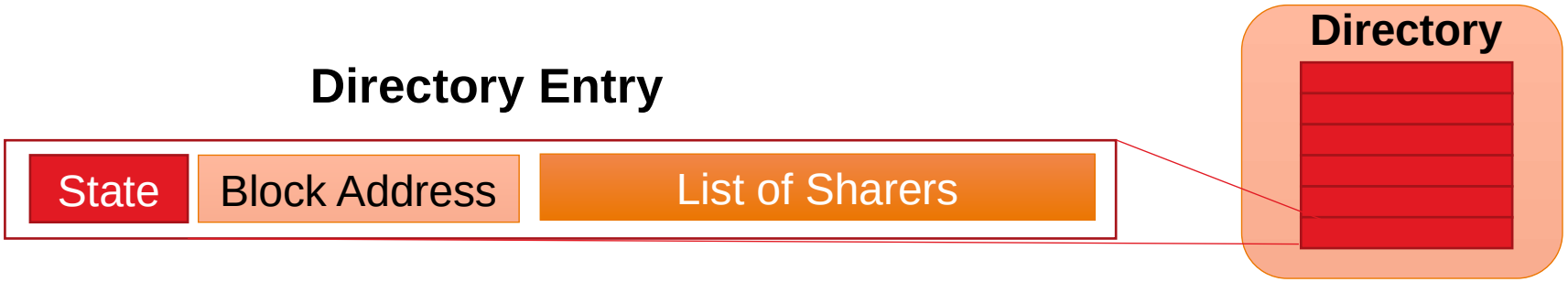


Do not have a bus

- Have a dedicated structure called a directory
- The directory co-ordinates the actions of the coherence protocol
- It sends and receives messages to/from all the caches and the lower level in the memory hierarchy
- Scalable



Structure of the Directory



- State of the entry
- Block address
- List of sharers
  - List of caches that contain a copy of the block.
  - Typically, stored as a bit vector. If the i^{th} bit is set, it means that the i^{th} cache has a copy of the block.

Design of the Directory

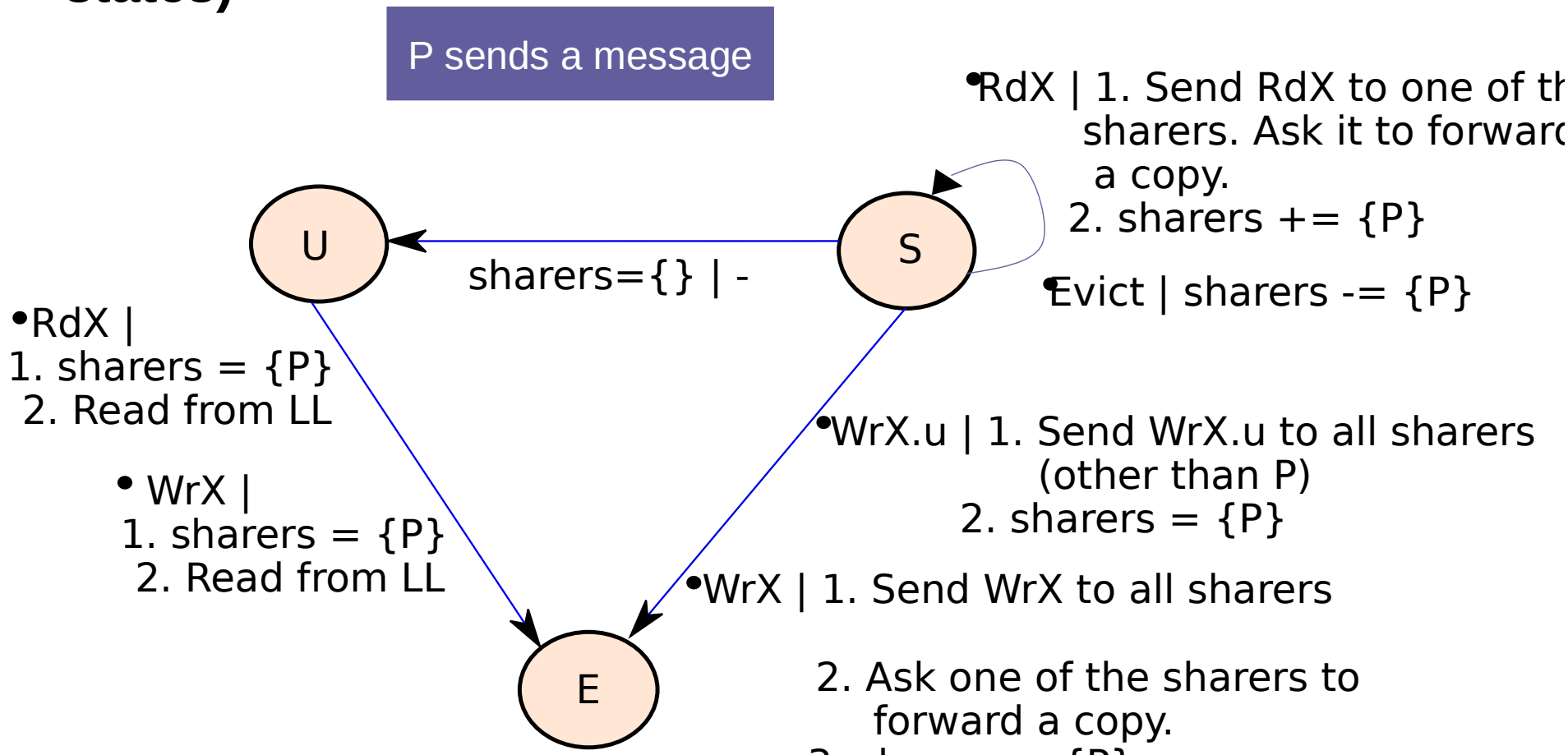
What are the messages that the directory receives?

- RdX
- WrX and WrX.u (regular and upgrade)
- Evict

What should the directory do:

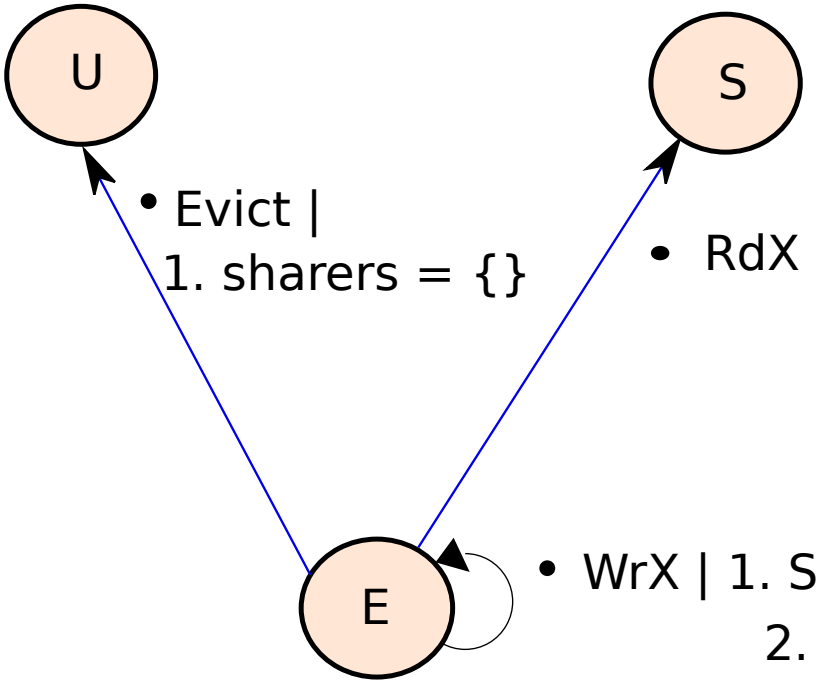
- RdX ▭ Locate a cache that contains the block (sharer) and fetch the block
- WrX and WrX.u ▭ Ask all sharers to invalidate their lines, give exclusive rights to the cache that wants to write
- Evict ▭ Delete the cache from the list of sharers
- The basic protocol at the level of the caches remains the same. The state transitions of directory entries are as follows.

State Transitions of a Directory Entry (from the U and S states)



LL ⎵ Lower level
U ⎵ Uncached

State Transitions of a Directory Entry (from the E state)



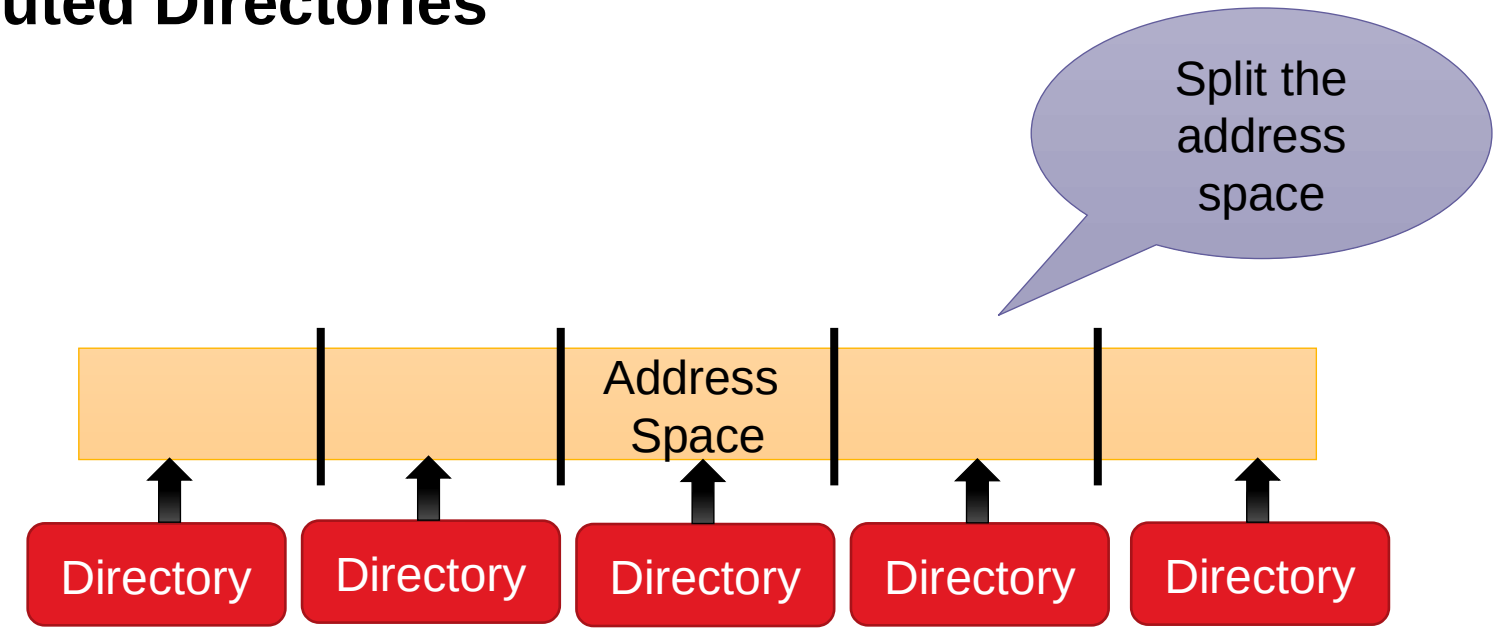
- Evict |
  1. sharers = {}
- RdX | 1. Send RdX to the sharer. Ask it to forward a copy of the block.
  2. sharers += {P}
- WrX | 1. Send WrX to the sharer
  2. Ask the sharer to forward a copy
  3. sharers = {P}

Enhancements to the Directory Protocol

Let us list some of the common problems associated with directories

- We need an entry for each block in a program's working set (lot of storage)
- In each directory entry, we need an entry for each constituent cache (storage overheads)
- The directory itself can become a point of contention
- Let us look at ␣

Distributed Directories



- Split the physical address space
- A directory handles all the requests for the part of the address space it owns
- Resolves the issue of the single point of contention.

List of Sharers

- How to maintain the list of sharers?
- Solution 1 [Fully mapped scheme]:
  - If there are *N* processors, have a bit vector of *N* processors.
  - Each block is associated with a bit vector of sharers

Space-efficient Solutions

- Maintain a bit for a set of caches. Run a **snoopy** protocol inside the set.
- Store the ids of only *k* sharers. Have an overflow bit to indicate that there are more than *k* sharers. In this case, every message needs to be broadcasted. [Partially mapped scheme]

Size of the Directory

The directory should ideally be as large as the number of blocks in the programs' working sets

Having an entry for every block in the physical address space is impractical

Practical Solution:

- Design a directory as a cache
- Keep the state of a limited number of blocks
- If an entry is evicted from the directory, invalidate it in all the caches

A Few More Issues

- False sharing ⎵ Consider a 64-byte block. It is possible that different threads access different memory words within the block.
  - The block will keep bouncing between caches.
  - These are false sharing misses.
  - Use a smart compiler to place data more intelligently.
- Race conditions
  - In real hardware, there are a lot of interactions. It is possible that multiple messages of different types for the same block might arrive at the same time.
  - Such concurrent events (race conditions) need to be handled. Hence, a practical cache coherence protocol has close to a 100 states.

Critical Sections and Atomic Operations

Consider this piece of code

```
t1 = account.balance;
t2 = t1 + 100;
account.balance = t2;
```

Can this code be executed in parallel by multiple threads?

What is the problem?

100	t1 = account.balance;
200	t2 = t1 + 100;
200	account.balance = t2;

100	t1 = account.balance;
200	t2 = t1 + 100;
200	account.balance = t2;

- Each line corresponds to a line of assembly code
- Assume corresponding lines execute in parallel.
- The final answer is wrong. It should be 300, it is 200.

Solution: Use Locks

Only one thread can execute this piece of code at any single point in time.

```
lock();
        t1 = account.balance;
        t2 = t1 + 100;
        account.balance = t2;
unlock();
```

Use Atomic Instructions to Implement Locks

For implementing lock and unlock functions.

- We need atomic instructions
- Either execute completely or not at all. Nobody observes a partial state.
- Most atomic operations also act as a fence.

Atomic exchange operation

xchg reg, <mem>

Atomically exchanges their contents

Assembly Level Implementation of the Lock and Unlock Functions

```
.lock:
        mov  r1, 1
        xchg r1, 0[r0]
        cmp r1, 0
        bne .lock
        ret
```

- r0 contains the lock address
- The *xchg* instruction contains a *fence*
- Contains 0 if the lock is not acquired
- 1 if acquired

```
.unlock:
        mov  r1, 0
        xchg r1, 0[r0]
        ret
```

- We store 0 at the lock address

Implementation of Atomic Exchange

xchg r1, [r2]

- temp = r1, r1 = [r2], [r2] = temp

It involves 3 steps

- 1 memory read + 1 memory write + register move
- All the operations need to happen atomically
- This is called a read-modify-write instruction (RMW)

Method

- Get exclusive access (M state) with write permissions for the memory address in *r2*
- Perform the read-modify-write operation
- Do not respond to any other requests from the local cache, or other caches, or the directory when the operation is in progress
- Respond to the directory or other caches only when the operation is over

Spin Locks

- We need to repeatedly try to acquire the lock
- This is a spin lock
- There are a lot of overheads:
  - Every time we access the lock, it needs to be read in the *M* state.
  - Too many invalidation messages

- Test and Exchange Lock 

Test and Exchange Lock

```
.lock:
        mov  r1, 1

.test
        /* test if the lock is free */
        ld r2, 0[r0]
        cmp r2, 0
        bne .test

        /* attempt an exchange only when the
                lock is free */
        xchg r1, 0[r0]
        cmp  r1, 0
        bne  .test
        ret
```

```
.unlock:
        mov  r1, 0
        xchg r1, 0[r0]
        ret
```

Atomic Operation	Example	Explanation

Test and Set	tas r1, 8[r0]	if (8[r0] == 0) {
		  8[r0] = 1;
		  r1 = 1;
		}
		else r1 = 0;

Fetch and Increment	fai r1, 8[r0]	r1 = 8[r0]; 8[r0] = r1 + 1;
Fetch and Add	faa r1, r2, 8[r0]	r1 = 8[r0]; 8[r0] = r1 + r2;

Compare and Set	cas r1, r2, r3, 8[r0]	if (8[r0] == r3) {
		  8[r0] = r2;
		  r1 = 1;
		} else r1 = 0;

Load linked (ll)
Store conditional (sc)
	ll r1, 8[r0]
	....
	mov r2, 1
	sc r3, r2, 8[r0]
		r1 = 8[r0] /* Use the ll instruction */
		...
		/* sc */
		if (8[r0] is not written to since the last ll){
		  8[r0] = r2;
		  r3 = 1;
		} else r3 = 0;

Lock-free Algorithms

- Let us write the same code without locks.
- If the thread goes to sleep after acquiring the lock, all the threads wait.

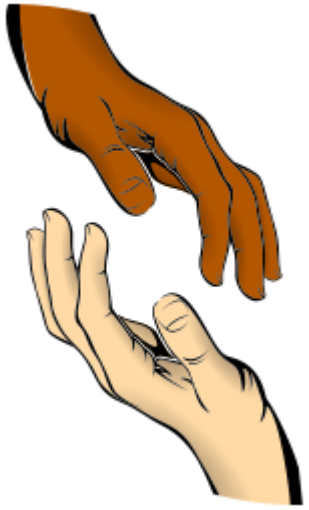
```
while (1) {
        t1 = account.balance;
        t2 = t1 + 100;
        if (CAS (account.balance, t1, t2))
                break;
}
```

It is possible for a thread to starve – never get the lock.

How do we eliminate starvation?

Answer:

Wait free algorithms



Basic Idea

- A request, T, first finds another request, R, that is waiting for a long time
- T decides to help R
- This strategy ensures that no request is left behind
- Also known as an altruistic algorithm

Summary of Consensus Numbers

Are all atomic operations equally powerful?

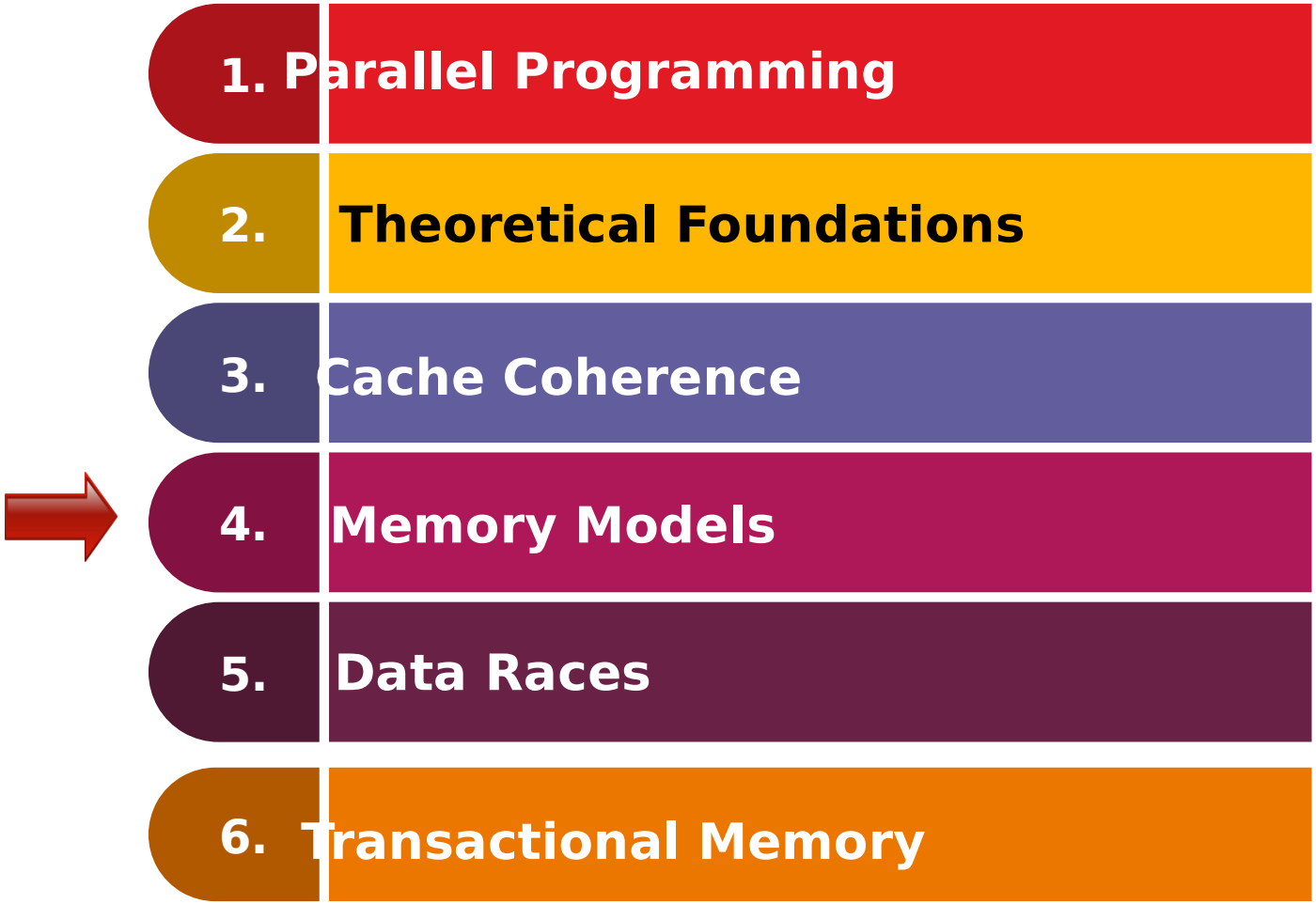
Consensus number

Consensus problem: each thread proposes a value – one among the proposed values is chosen. The consensus number is the maximum number of threads that can solve this problem using a wait-free algorithm.

Type of Operation	Consensus Number
Atomic exchange	2
Test and Set	2
Fetch and add	2
CAS (Compare and Set)	
LL/SC	

The consensus problem forms the theoretical basis of most concurrent algorithms.

Contents


1. Parallel Programming
2. Theoretical Foundations
3. Cache Coherence
4. Memory Models
5. Data Races
6. Transactional Memory

Memory Models

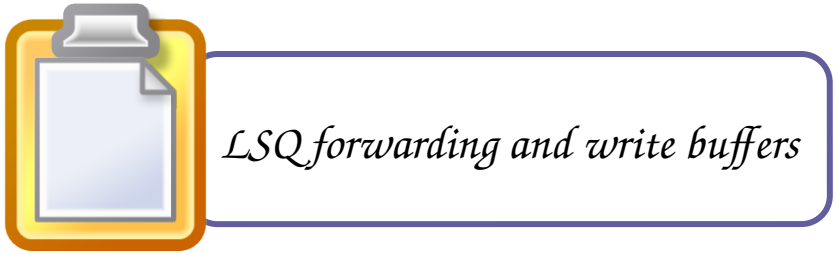
- The memory model is dependent on the processor architecture. If there are very aggressive optimizations, then the memory model has to be very weak.

- PLSC requires the ws and fr orders to be global. All popular architectures follow PLSC, and many disallow thin air reads.

- The only orders that can be local are *rf* and *po*.

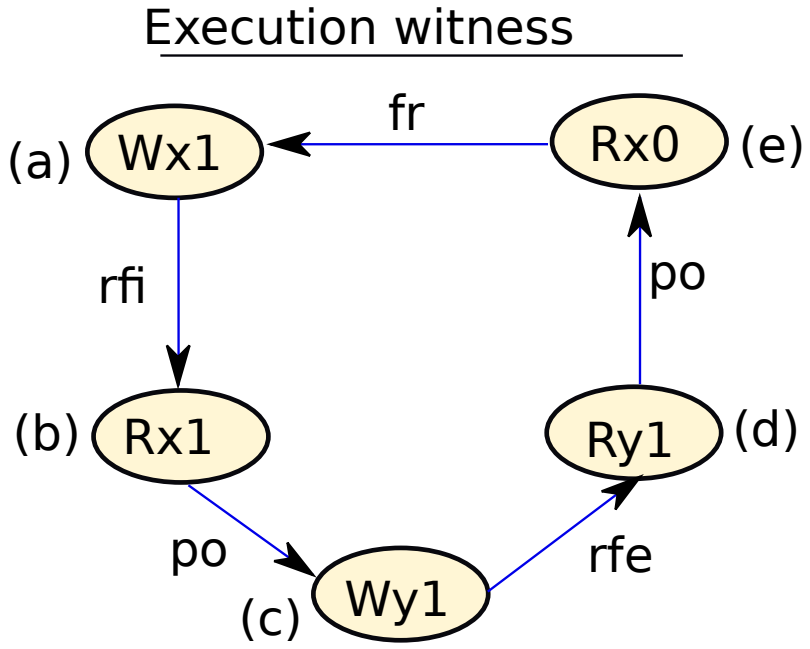
Write-to-Read Order

rfi



T1	T2
(a) x = 1 (b) t1 = x (c) y = 1	(d) t2 = y (e) t3 = x

t1 = 1, t2 = 1, t3 = 0 ?



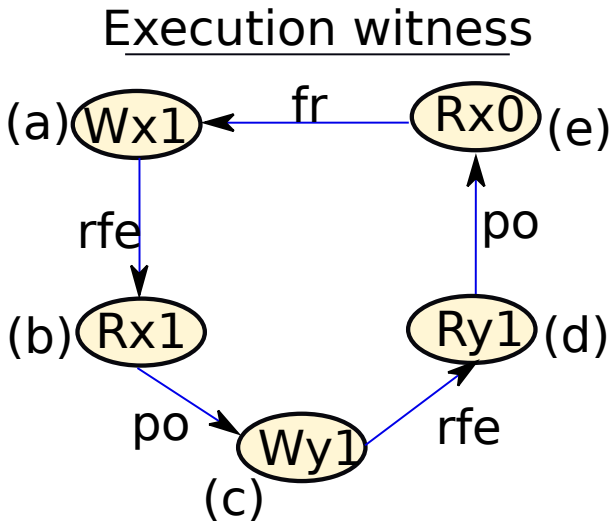
Non-atomic Writes

Create local tiles of caches.

rfe

T1	T2	T3
(a) x = 1	(b) t1 = x	(d) t2 = y
	(c) y = 1	(e) t3 = x

t1 = 1, t2 = 1, t3 = 0 ?



If writes are atomic, this behavior is not allowed, otherwise it is.

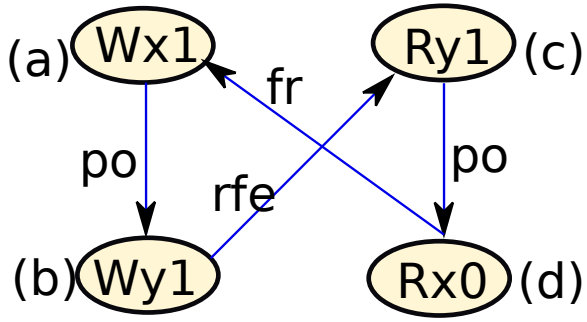
We are seeing writes to different locations in different orders.

Write-to-Write Order

T1	T2
(a) x = 1	(c) t1 = y
(b) y = 1	(d) t2 = x

t1= 1, t2 = 0 ?

Execution witness



Even if writes are atomic, this behavior is not allowed.

If the $W_{\square}$ W ordering is relaxed, we will see writes to different locations in different orders.

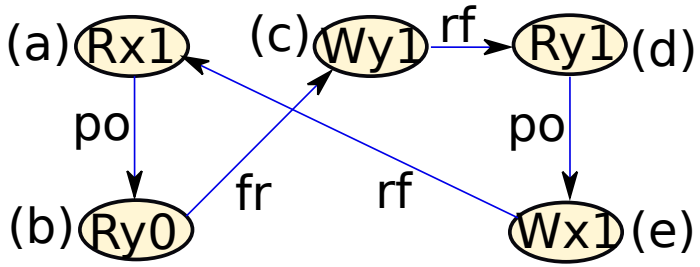
Read-to-Read Order

Issues loads OOO in the LSQ

T1	T2	T3
(a) t1 = x	(c) y = 1	(d) t3 = y
(b) t2 = y		(e) x = 1

t1 = 1, t2 = 0, t3 = 1?

Execution witness



If the R $_{\square}$ R ordering is relaxed, we will observe this behavior.

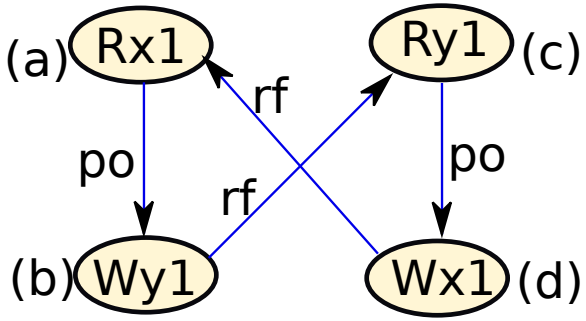
Read-to-Write Order

Speculative writes

T1	T2
(a) t1 = x	(c) t2 = y
(b) y = 1	(d) x = 1

t1= 1, t2 = 1 ?

Execution witness



If the R ▭ W ordering is relaxed, we will observe this behavior.

Special case of *rfi* in SC.

Can the *rfi* relation be relaxed in SC?

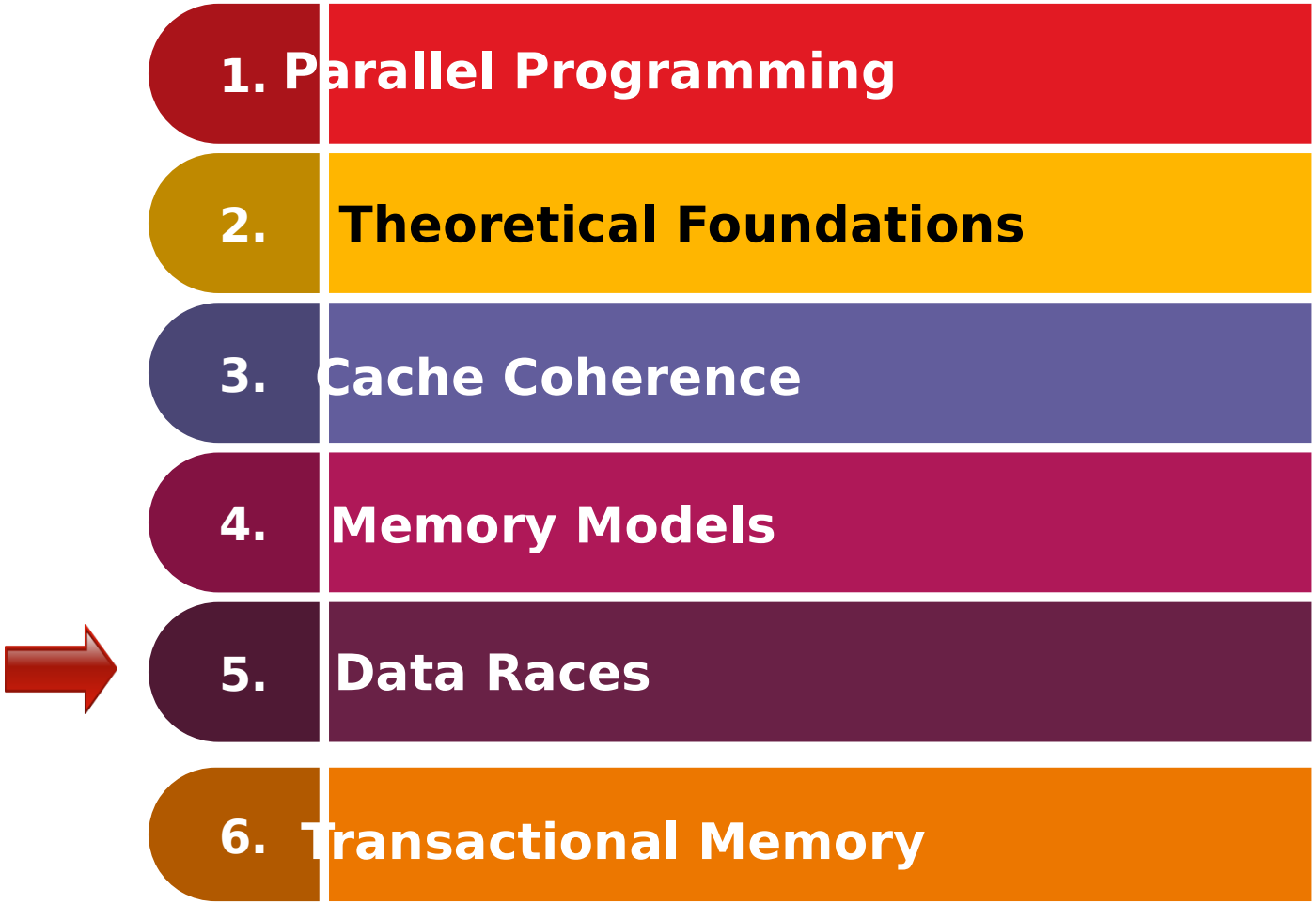
The proof is there in the book

Summary of Memory Models

Relaxation	W ⊐ R	W ⊐ W	R ⊐ R	R ⊐ W	rfe	rfi
SC						✔
TSO (Intel)	✔					✔
Processor consistency	✔				✔	✔
PSO	✔	✔				✔
Weak Ordering/ RC	✔	✔	✔	✔		✔
IBM PowerPC	✔	✔	✔	✔	✔	✔
ARM	✔	✔	✔	✔	✔	✔

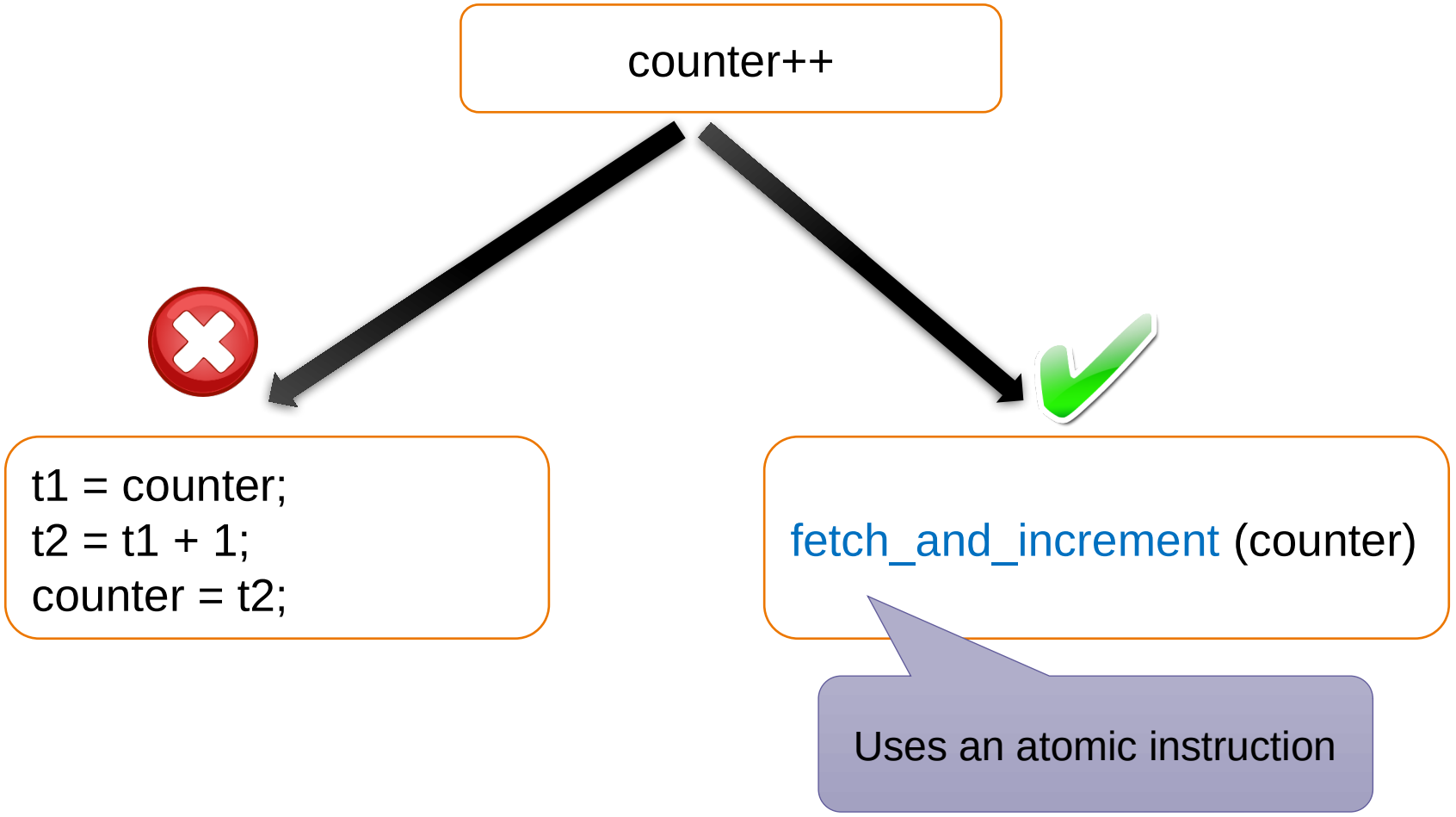
✔ Ordering is relaxed

Contents


1. Parallel Programming
2. Theoretical Foundations
3. Cache Coherence
4. Memory Models
5. Data Races
6. Transactional Memory

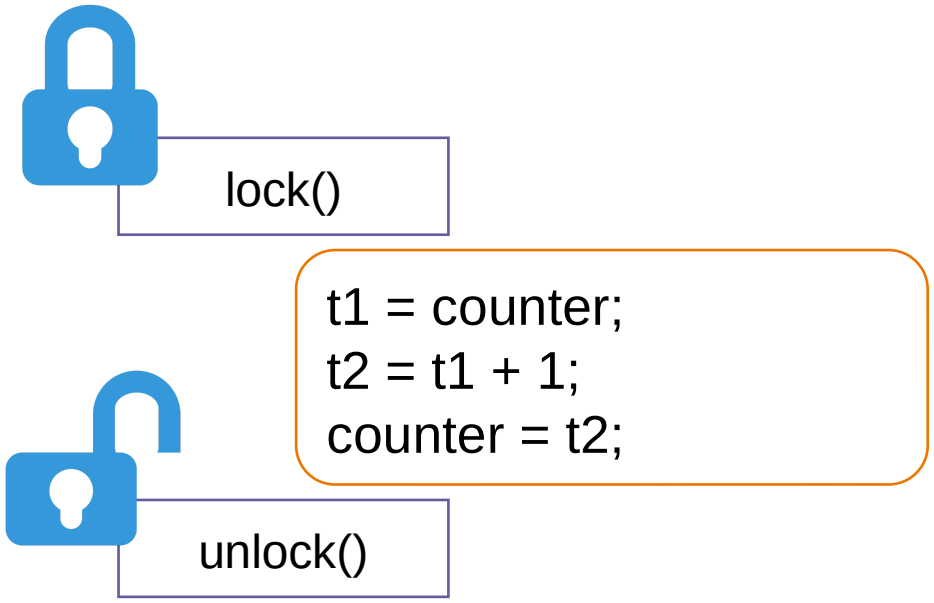
Increment a Counter

Let us say that all we want to do is

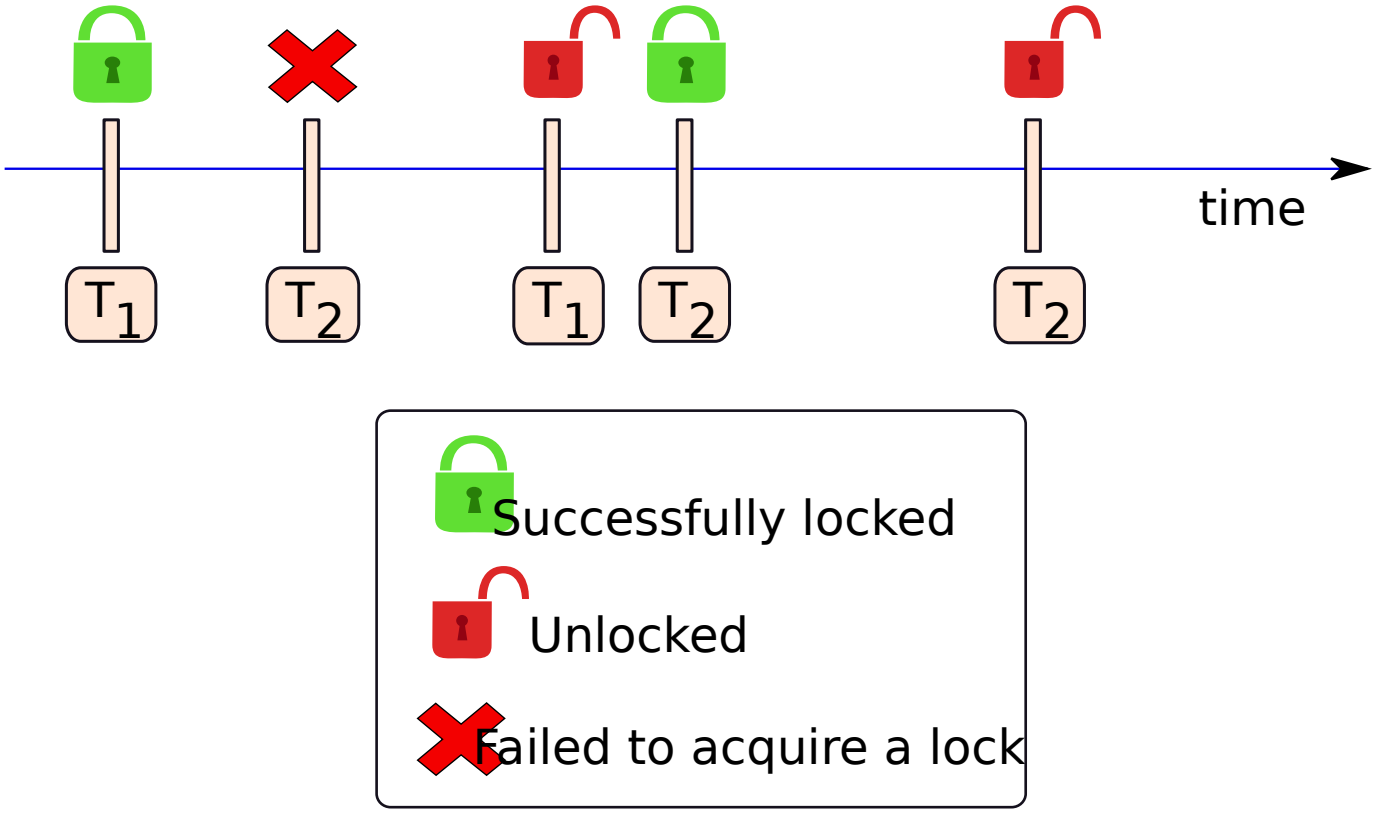


Code with Locks

If we do not want to write code using atomic instructions, we need to encapsulate this code within a critical section.



Critical Sections



Why do we need to use locks?

If there are no shared variables, do we need to use locks?

Answer: 

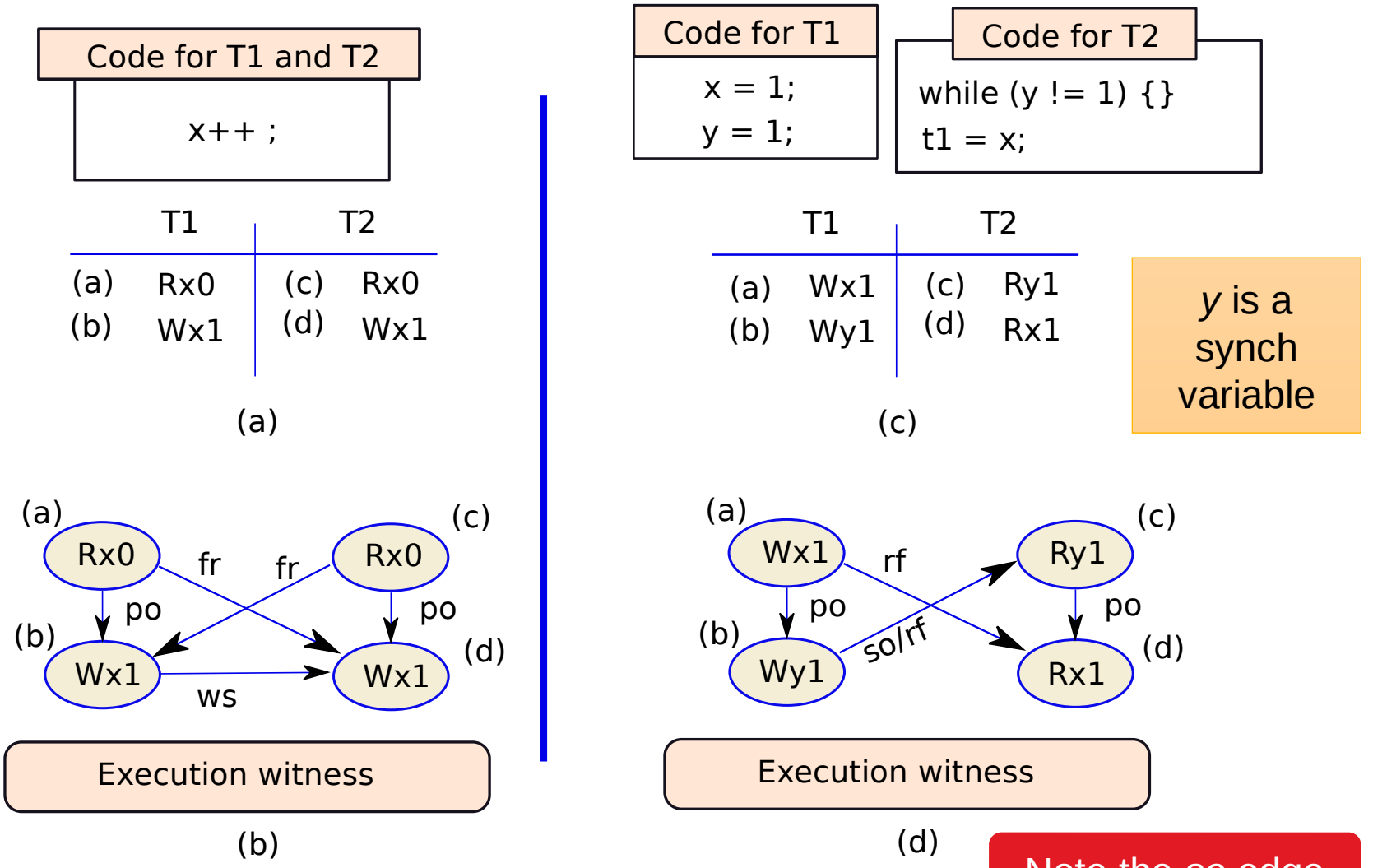
When do we need locks then?

Answer: Two blocks of code need to be making conflicting and concurrent accesses to the same address.

Conflicting accesses → At least one access is a write

Concurrent Accesses

Common sense meaning ⎵ At the same time.



Data Races

Two accesses are said to be concurrent when there is no path between them in the execution witness that contains an *so* edge.

Data Race

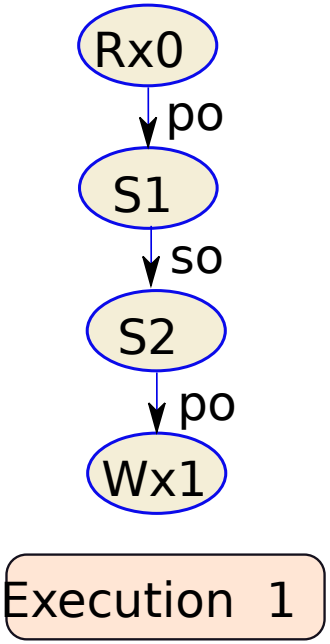
A pair of conflicting and concurrent accesses to the same regular variable constitute a data race.

If a piece of code does not have data races, what does it mean?

Does SC Imply Data-Race-Freedom?

T1	T2
Rx0	S2
S1	Wx1

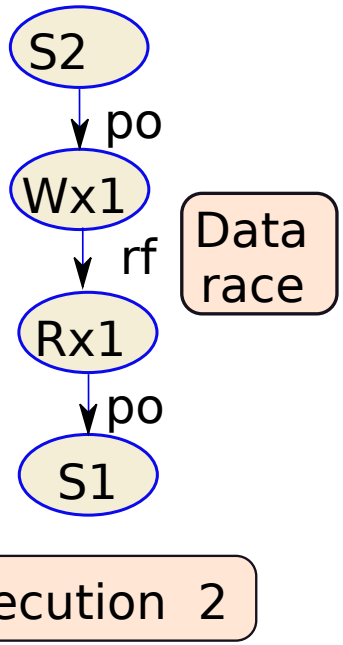
(a)



(b)

T1	T2
Rx1	S2
S1	Wx1

(c)



(d)

Does Data-Race-Freedom imply SC?

Refer to the book for the detailed proof.

Salient Points

- If there are two conflicting accesses, there will be an *so* edge on at least one path between them in the execution witness.
- They will thus be ordered by the *so* edge.
- Let us add all the SC edges to the execution witness.
- A cycle implies that there is a cycle between synch operations.
- This is not possible. Synch operations follow SC.
- Proof by contradiction.

What does having data races imply?

Theorem

If we have a data race in a program, we can construct an SC execution that also has a data race.

Proof ⊟ refer to the book

What does it imply?

If an automated tool cannot construct an SC execution that has a data race, then it means that the program is data race free.

Method to detect data races

Summary of All the Results

Properties
SC does not imply data-race-freedom.
Data-race-freedom implies SC
Non-SC execution implies data races
If an automated tool cannot construct an SC execution that has a data race, then it means that the program is data race free.

Moral of the story → If there are no data races, the memory model doesn't matter

- Programming languages need to define data-race-free memory (DRF) memory models that provide synchronization primitives.
- Programmers need to use them to write *properly synchronized* programs.
- This means that all accesses to shared variables are protected with critical sections such that there are no concurrent, conflicting accesses.

Methods to Detect Data Races

Data Race Detection Algorithm

How do we detect data races?

- We need an algorithm that tests a piece of code for data races
  - Exercise all possible control paths
  - Create as many interleavings as possible
- A data race must show up in an SC execution
  - Try to find it …

Notion of Lock Sets

L(v) — Set of locks held by this memory location, *v*.

L(T) — Set of locks held by thread, *T*.

A lock is identified by the lock address.

After each access we modify the lock set.

$$L(v) = L(v) \cap L(T)$$

Compute an intersection

Standard Approach

1. Add annotations to multithreaded code. These annotations modify the lock set.
2. The locksets of threads are initially empty.
3. For each variable, its lockset initially has all the locks.
4. When a thread acquires a lock, we add the lock to its lock set.
5. When it releases a lock, we remove the lock.
6. As the program executes, , keeps getting updated.
7. At the end, if there is a variable with an empty lock set, it is probably involved in a data race.

If the lock set is empty, it means that there is no synchronization between accesses to the variable.

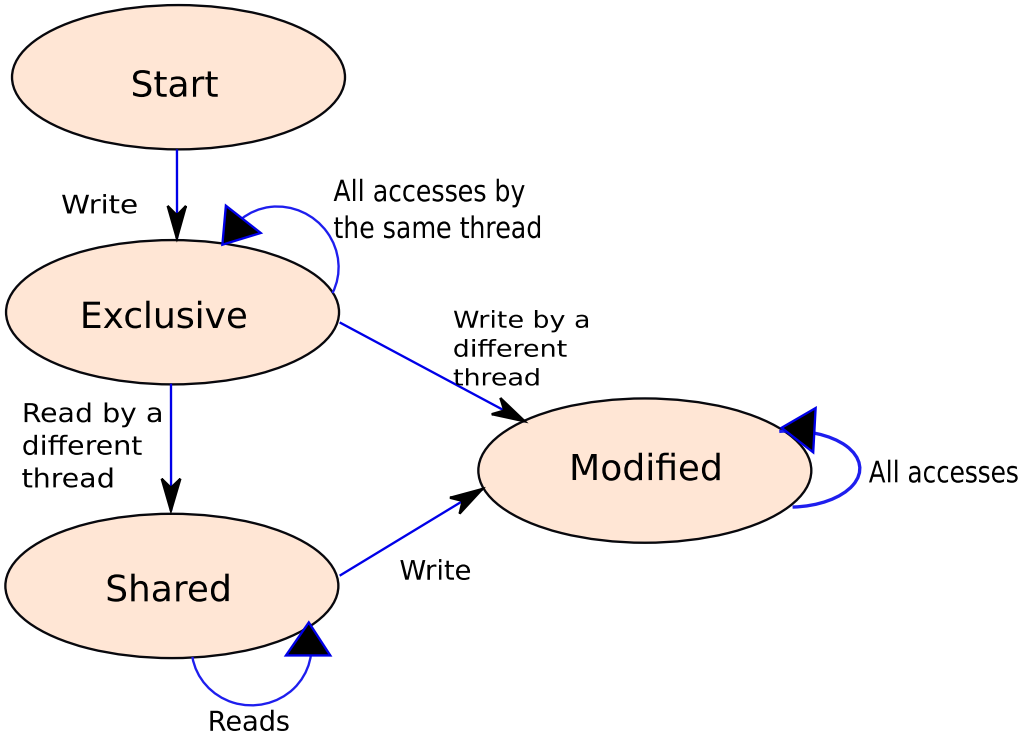
Notion of False Positives

Note that we will detect many scenarios that are actually not data races. For example, the basic algorithm will flag a data race when we consider read-only variables.

A few more examples

Example	Description
Initialization	We typically initialize variables without using locks
Read-only variables	Written once (during initialization) and read many times
Reader-writer pattern	Multiple threads read a variable concurrently.

Modified State Diagram for each Variable



1. The first access has to be a write. We then move to the *Exclusive* state.
2. If a different thread reads it, move to the *Shared* state.
3. The *Shared* state allows reads (irrespective of the thread).
4. After a write, move to the *Modified* state.
5. In the *Modified* state, run the regular lock set algorithm.

Notion of Vector Clocks

- We can think of a multithreaded execution environment as a classical distributed system
- Here, there is no notion of global time.
- Every thread has a local clock.
- The local clocks are updated any time there is an interaction between threads.
- Assume there are *n* threads. Every thread maintains an *n*-element vector clock. Thread *i*'s vector clock is .
- is the best estimate that *i* has for j's local clock

Comparability of Clocks

- Two vector clocks are equal when

$$V_i = V_j \iff \forall k, V_i[k] = V_j[k]$$

- Two vector clocks are totally ordered when

$$V_i \prec V_j \iff (V_i \neq V_j) \wedge (\forall k, V_i[k] \leq V_j[k])$$

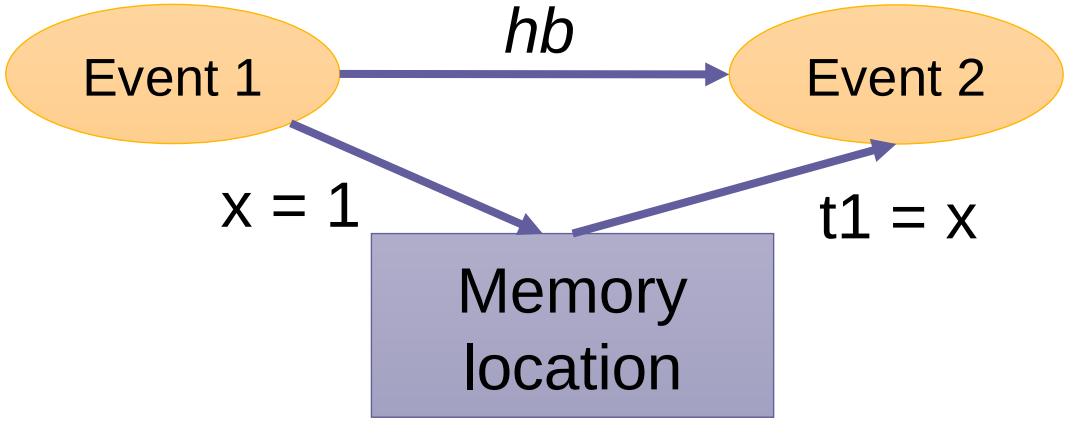
$$V_i \leq V_j \iff (V_i = V_j) \vee (V_i \prec V_j)$$

- Two vector clocks may not always be comparable

Notion of a Message

A *message* represents an interaction between two threads.

Say if a thread *i* accesses a variable in memory, it updates its state, and then thread *j* accesses the variable (and its state) ∎ This is an interaction that falls under the theoretical definition of a message.



More about Events

Increment the local clock (e.g., for thread *i*) before sending a message and after receiving a message.

Let's say, thread *i* sends a message to thread *j*. When *j* receives the message, it updates its clock as follows.

$$\forall k, V_j[k] = max\left(V_i[k], V_j[k]\right)$$

The receiver is as at least as up to date as the sender. It records an additional receive event.

Vector Clocks and Causality

Let's say that there is a happens-before relationship between events e_i and e_j. We use vector clocks to track the interaction.

Event	Time
e_i	Event in thread i at time V_i
e_j	Event in thread j at time V_j

We have the following relationship

$$V_i < V_j \iff e_i \overset{hb}{\longrightarrow} e_j$$

This happens because the receiver has all the sender's updates and without a chain of happens-before edges, two vector clocks will not remain comparable.

Vector Clock based Algorithm

Symbol	Meaning
	Vector clock of the current thread
	Vector clock of the current lock
	Read clock of variable *v*
	Write clock of variable *v*
	Thread id

$$C_T[tid] \leftarrow C_T[tid]+1$$
$$C_T \leftarrow C_T \cup C_L$$
$$C_L \leftarrow C_T$$
$$C_T.inLock \leftarrow \textbf{True}$$

Increment the vector clock of C_T and C_L

$$C_T.inLock \leftarrow \textbf{False}$$

Read Operation

$$\textbf{if } (\sim C_T.inLock) \textbf{ then}$$
$$C_T[tid] \leftarrow C_T[tid]+1$$
$$\textbf{end}$$
$$\textbf{if } (W_v \leq C_T) \textbf{ then}$$
$$R_v \leftarrow R_v \cup C_T$$
$$\textbf{end}$$
$$\textbf{else}$$
$$\textbf{DeclareDataRace}$$
$$\textbf{end}$$

Increment the local count of the **thread clock**

If no thread has overwritten the value after the transaction started, then the read is successful. Update the read clock.

Write Operation

$$\textbf{if } (\sim C_T.inLock) \textbf{ then}$$

$$C_T[tid] \leftarrow C_T[tid]+1$$

$$\textbf{end}$$

$$\textbf{if } (W_v \leq C_T) \wedge (R_v \leq C_T) \textbf{ then}$$

$$R_v \leftarrow R_v \cup C_T$$

$$W_v \leftarrow W_v \cup C_T$$

$$\textbf{end}$$

$$\textbf{else}$$

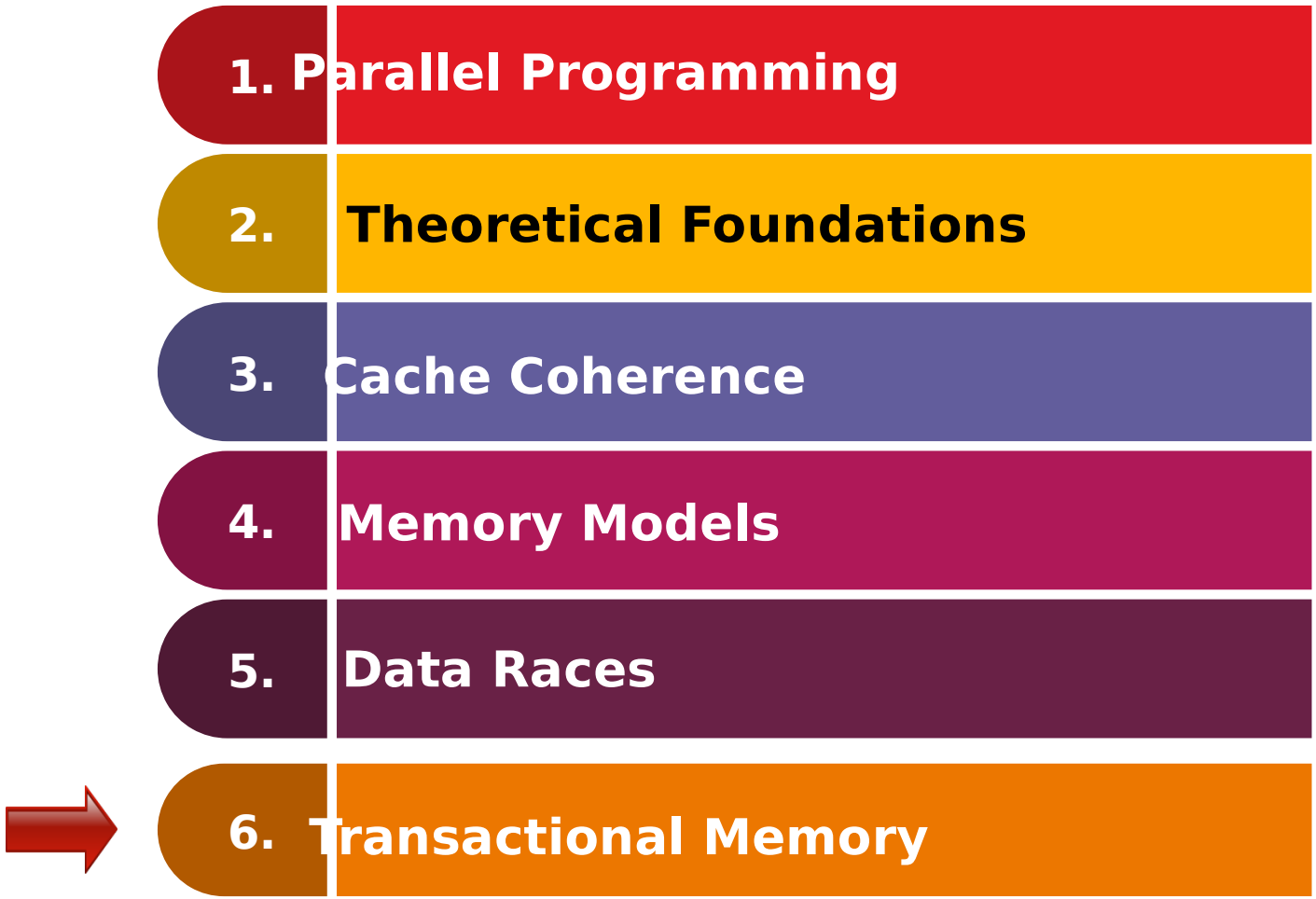
$$\textit{DeclareDataRace}$$

$$\textbf{end}$$

Increment the local count of the **thread clock**

If no thread has overwritten or read the value after the transaction started, then the read is successful. Update the read and write clocks.

Contents


1. Parallel Programming
2. Theoretical Foundations
3. Cache Coherence
4. Memory Models
5. Data Races
6. Transactional Memory

Lock and *Unlock* functions have a problem

```
void updateBalance ( int amount , Account account ) {
        lock ();

        int temp = account.balance ;
        temp = temp + amount ;
        account.balance = temp ;

        unlock ();
}
```

- If we have one lock, then there is no parallelism in the system.
- We can afford much more parallelism.
- The only constraint is:
  - We should not have two parallel accesses to the same *account*.

Disjoint access parallelism

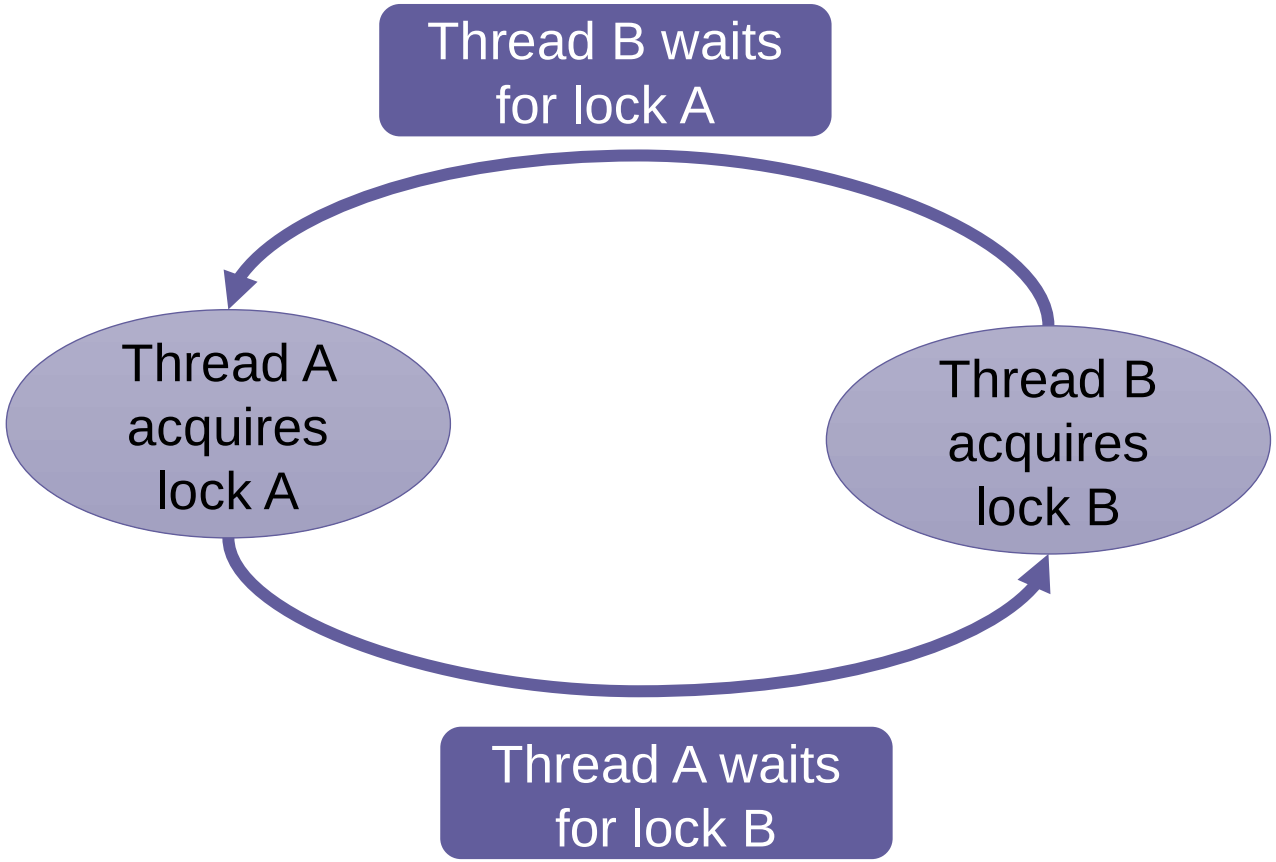
Code that Allows Disjoint Access Parallelism

```
void updateBalance ( int amount , Account account ) {
        account.lock ();

        int temp = account.balance ;
        temp = temp + amount ;
        account.balance = temp ;

        account.unlock ();
}
```

Account specific locks

- This is a scalable solution.
- But, there is a problem. If we have code where we acquire multiple locks, we may have a deadlock.

Deadlock Situation



Deadlock

Create a Transaction

Provides disjoint access parallelism

Automatically manages all the locks and avoids deadlocks.

```
void updateBalance ( int amount , Account account ){
        atomic { /* an atomic transaction */
                int temp = account . balance ;
                temp = temp + amount ;
                account.balance = temp ;
        }
}
```

The *atomic* block implements a transaction

Properties of a Transaction

Atomicity: Either the entire transaction completes or fails.

Consistency: A *consistent* state is a valid state that is as per a given set of specifications (consistency model). The property of consistency says that if the full system was consistent before a transaction started, it should be consistent after it ended.

Isolation: It appears that while the transaction was executing, regular instructions or other transactions were not executing.

Durability: Once a transaction has finished, its results are written to stable storage.

Transactional Memory

A memory system that provides support for transactions is known as a transactional memory (TM).

Two Types

Hardware TM

Software TM

Basics of Transactional Memory

read set

set of variables that are read by the transaction

write set

set of variables that are written by the transaction

Term	Meaning
R_i	Read set of transaction *i*
W_i	Write set of transaction *i*
R_j	Read set of transaction *j*
W_j	Write set of transaction *j*

When do transactions conflict?

There is a conflict if and only if 

$$W_i \cap W_j \neq \varphi$$

OR

$$W_i \cap R_j \neq \varphi$$

$$R_i \cap W_j \neq \varphi$$

Abort and Commit

Commit

- A transaction completed without any conflicts
- Finished writing its data to main memory

- *Abort*
  - A transaction could not complete due to conflicts
  - Did not make any of its writes visible

What happens after an abort?

- The transaction restarts and re-executes
- Might wait for a random duration of time to minimize future conflicts

- do {

  …

  …

  } while (! Tx.commit());

This is automatically handled by the transactional memory system

Basics of Concurrency Control

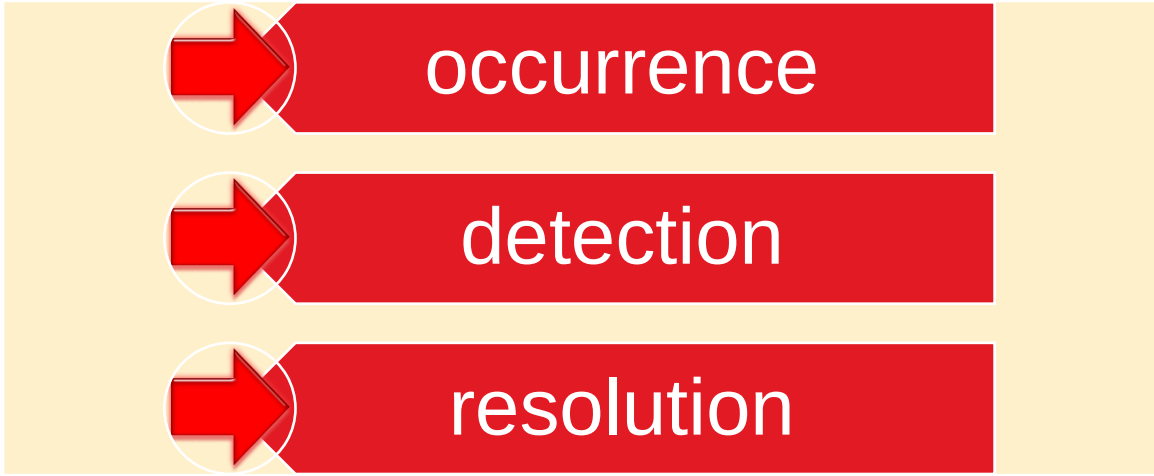
A conflict occurs when the read-write sets overlap

A conflict is detected when the TM system

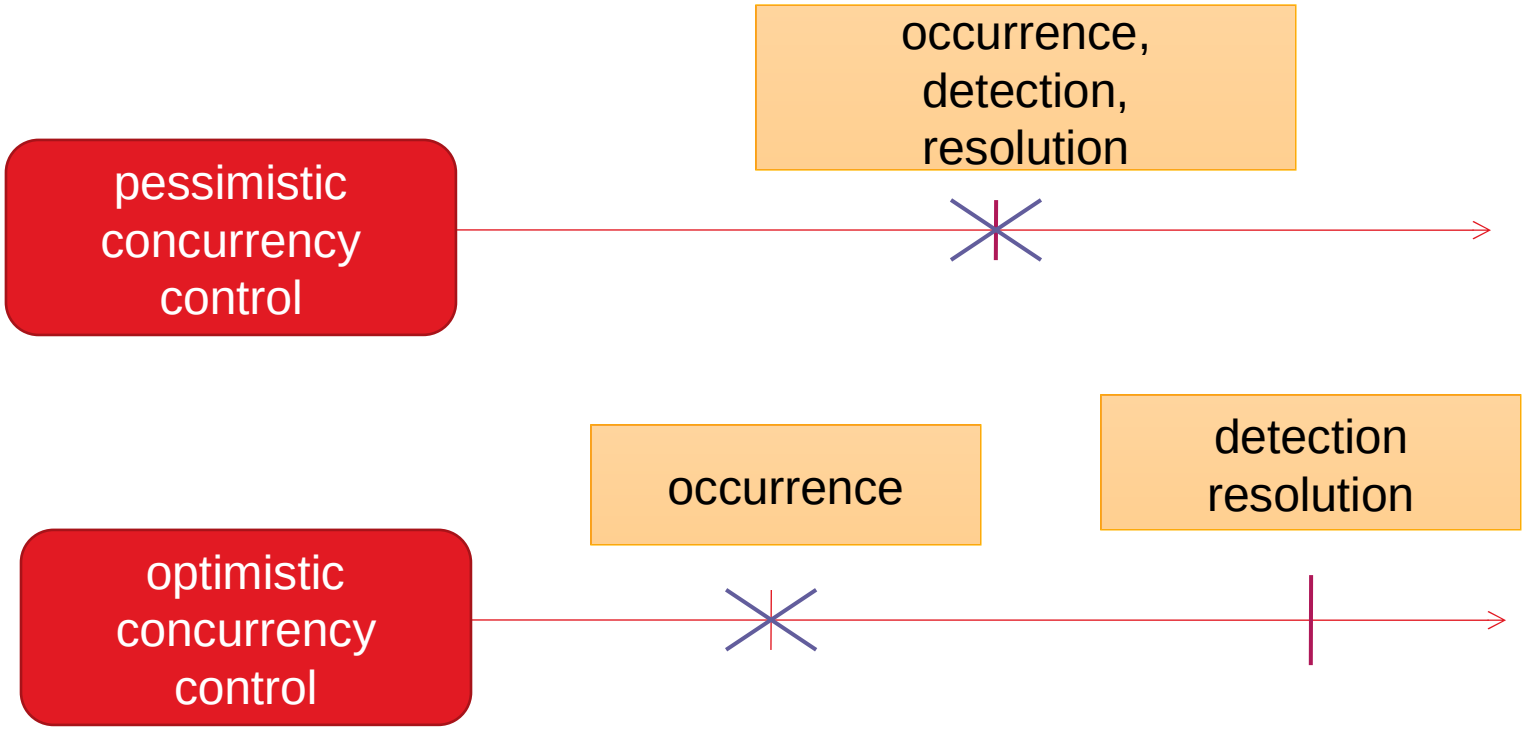
becomes aware of it

A conflict is resolved when the TM system either

- delays a transaction
- aborts it



Pessimistic vs Optimistic Concurrency Control



Version Management

Eager version management

- Write directly to memory
- Maintain an undo log

commit	flush undo log	**abort**	writeback undo log

Lazy version management

- Write to a buffer (redo log)
- Transfer the buffer to memory on a commit

commit	writeback redo log	**abort**	flush redo log

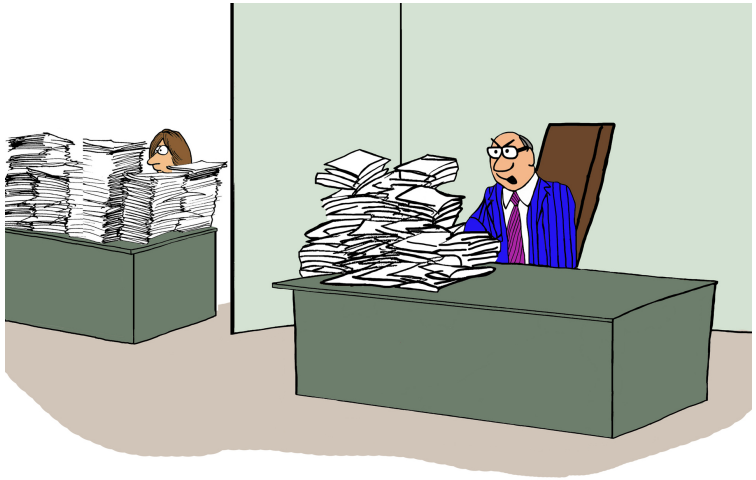
Conflict Detection

Eager

- Check for conflicts as soon as a transaction accesses a memory location

Lazy

- Check at the time of committing a transaction



"Hey, why can't I get any help with this paperwork?!"

Semantics of Transactions

Serializable

- Sequential consistency at the level of transactions

Strictly Serializable

- The sequential ordering is consistent with the real time ordering
- This means that if Transaction *A* starts after Transaction *B* ends, it should be ordered after it in the equivalent sequential ordering
- For concurrent transactions, their ordering does not matter

Opacity

- Even aborted transactions need to see a consistent state – one produced by only committed transactions

What about aborted transactions?

Opacity

Thread 1

```
atomic {
        t1 = x;
        t2 = y;
        while (t1 != t2) {}
}
```

Thread 2

```
atomic {
        x = 5;
        y = 5;
}
```

- If one transaction executes after the other, *x* will always be equal to *y*
- Assume optimistic concurrency control: resolution at the end
- Assume *Thread 1* reads *x=0*
- Then the transaction on thread 2 finishes
- The transaction on thread 1 needs to be aborted (will read *y=5*)
- However, it will be stuck in the *while* loop, it will never reach the end
- Opacity will not *y* to be read as *5*

Mixed Mode Accesses: Transactional and Non-Transactional

Single Lock Atomicity (SLA)

- Assume all the transactions are protected by a single lock
- Transactions first acquire a hypothetical (global) lock
- Same definition of data races
- Reduces concurrency

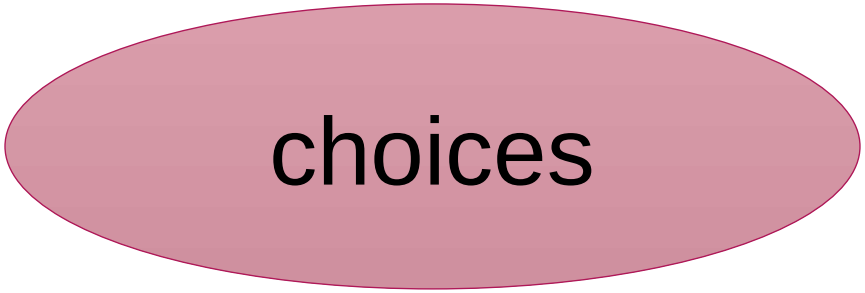
Disjoint Lock Atomicity (DLA)

- Uses more locks than SLA: (let's say one per variable)
- We a priori need to know the locks that a transaction is going to use

Transactional Sequential Consistency (TSC)

- We can order all transactions (committed or aborted) and regular instructions in a sequential order
- All the instructions are in program order (incl. within transactions)

Software Transactional Memory



Concurrency Control

- Optimistic or Pessimistic

Version Management

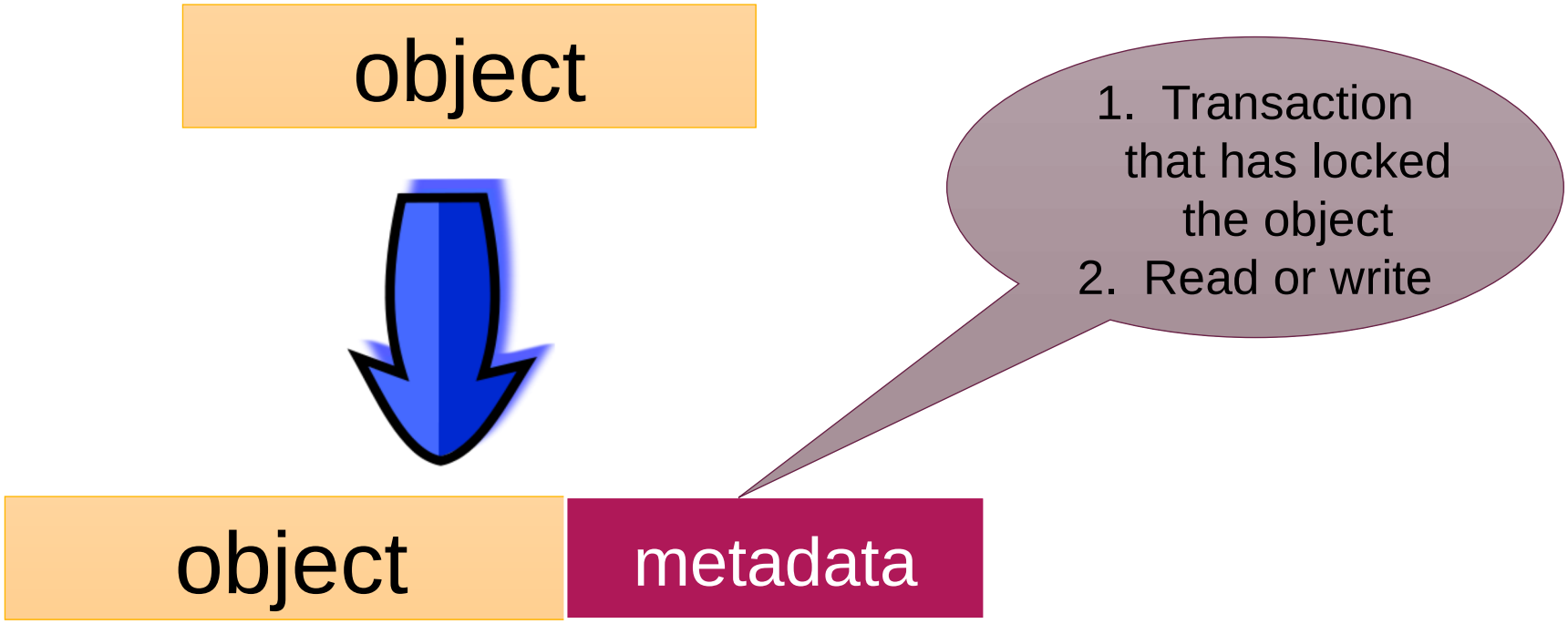
- Lazy or Eager

Conflict Detection

- Lazy or Eager

Support Required

Augment every transactional object/ variable with meta data



Maintaining Read–Write Sets

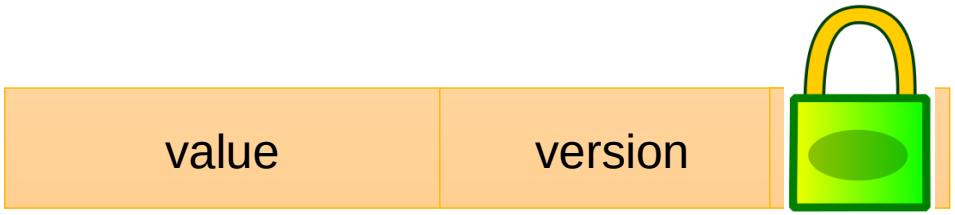
- Each transaction maintains a list of locations that it has
  - read in the read-set
  - written in the write-set
- Every memory read or write operation is augmented
  - *readTX* (read, and enter in the read set)
  - *writeTX* (write, and enter in the write set, make changes to the undo/redo log)

Bartok STM

Eager version management, lazy conflict detection

Every variable has the following fields

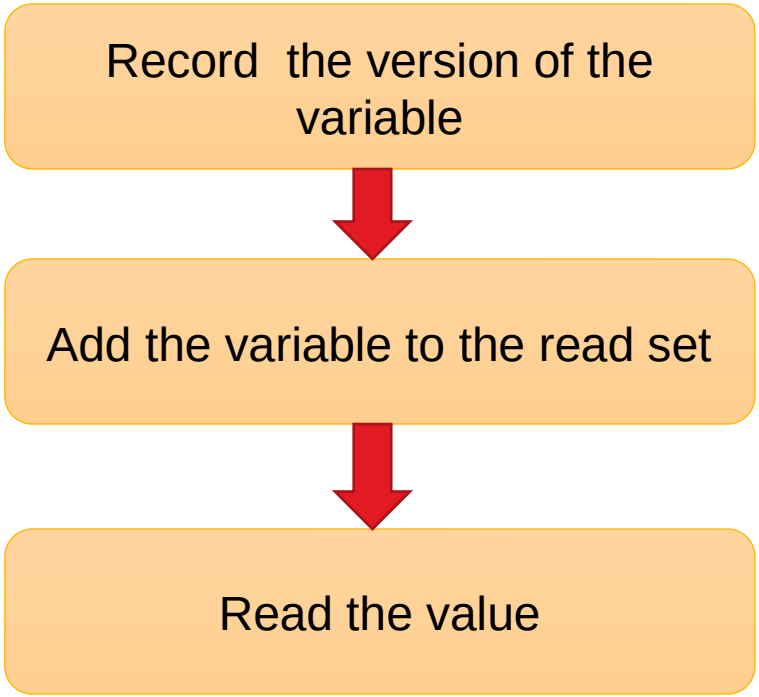
- version
- value
- lock



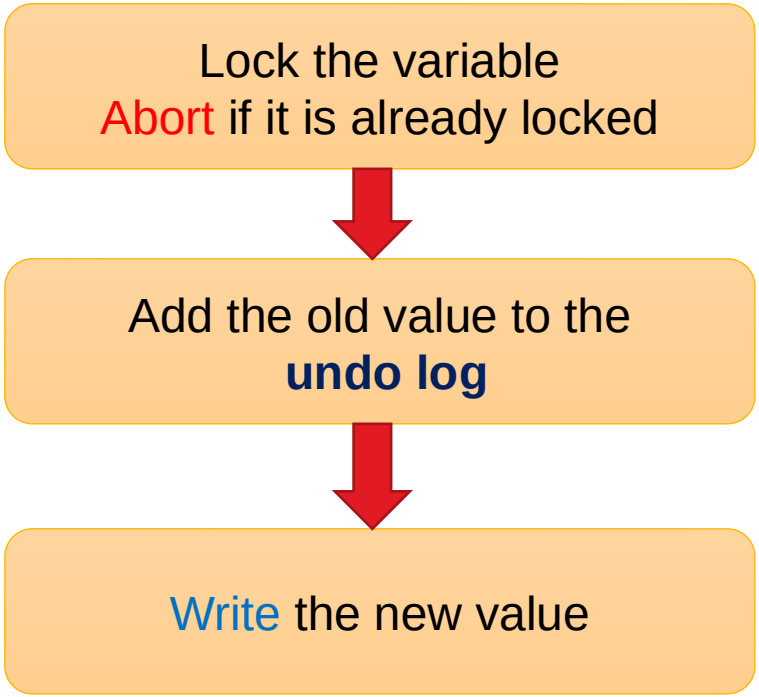
Transactional Variable

Read Operation

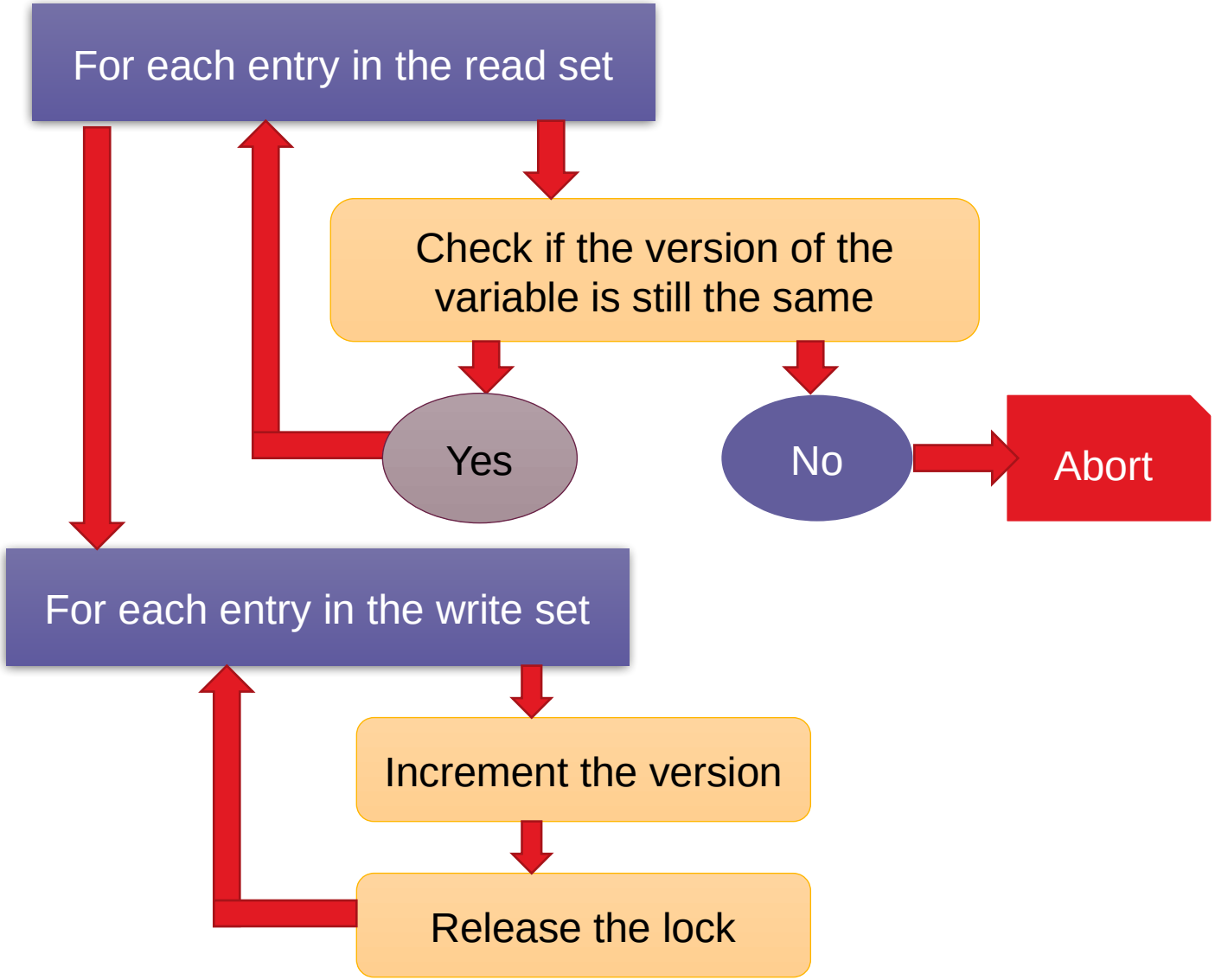
Read Operation



Write Operation



Commit Operation



Pros and Cons

- simple
- reads are simple
- provide a strong semantics for transactions

- does not provide opacity
- uses locks
- writes are slow

Subtle Points

- With an undo log, aborts are more expensive than commits
- A transaction can read intermediate values written by transactions
- Opacity is thus not guaranteed
- Locks are kept for a long time: lock $\sqsubset$ commit

TL2 STM

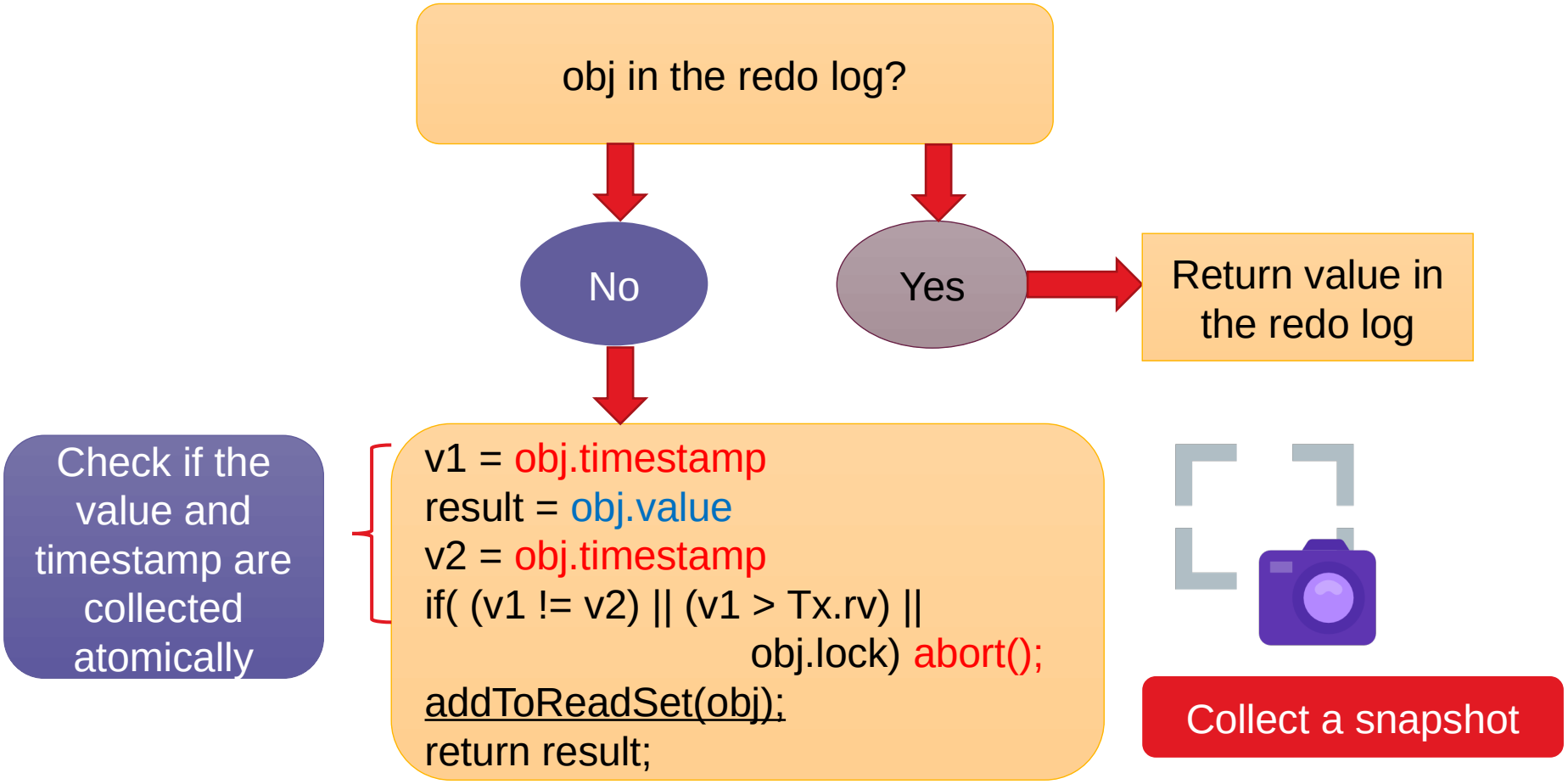
- Uses lazy version management ⎵ redo log
- Uses a global timestamp (*globalClock*)
- Locks variables only at commit time
- Every transaction does the following (*atomically*) when it starts:

```
globalClock++ ;
Tx.rv = globalClock ;
```

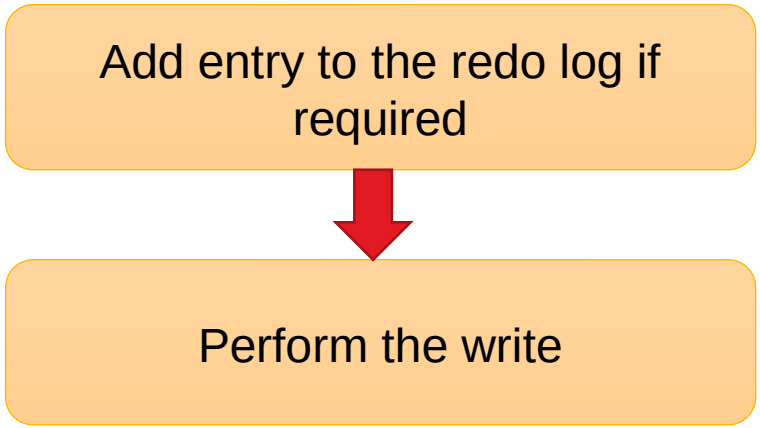
Set the value of the transaction's Tx.rv timestamp (read version)

Read Operation

Read Operation: *read* (tX, obj)

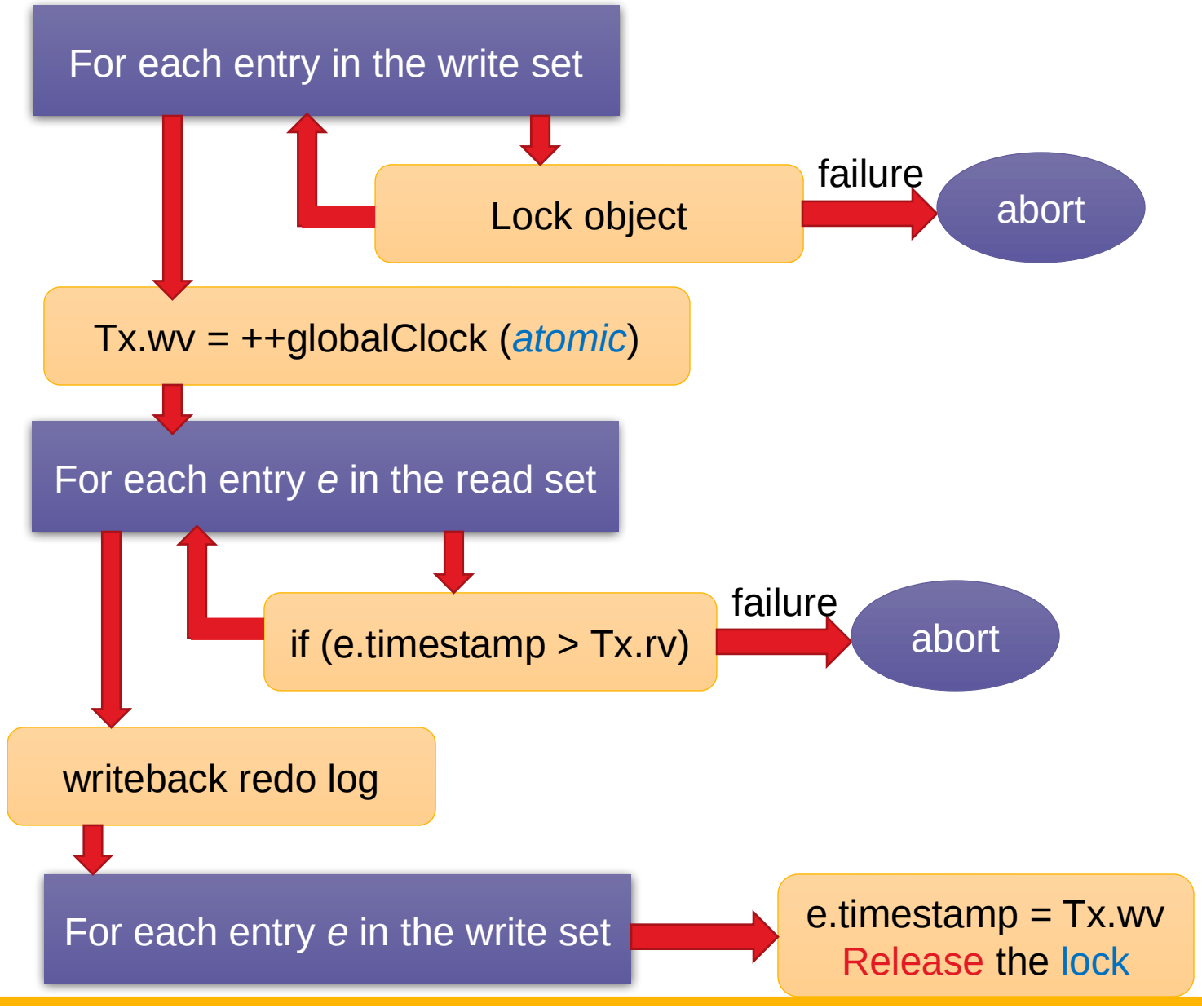


Write Operation



Writes are

Commit Operation



Pros and Cons

simple

provides opacity

holds locks for a lesser amount of time

A redo log is slower

Commits are more expensive than aborts

uses locks

Subtle Points

- We use two timestamps per transaction: Tx.rv and Tx.wv
  - We have,
- We first write the variables to permanent state
  - Then, we update their timestamps
  - If another transaction sees an updated timestamp, it is sure that the variable has been written to
  - Finally, we release the locks. This allows later reads.

Provides opacity

Hardware Transactional Memory

Case for Hardware

- STM systems do not handle non-transactional accesses
- Acquiring and releasing locks is expensive
- Maintaining undo and redo logs is difficult
- Hardware is much faster …

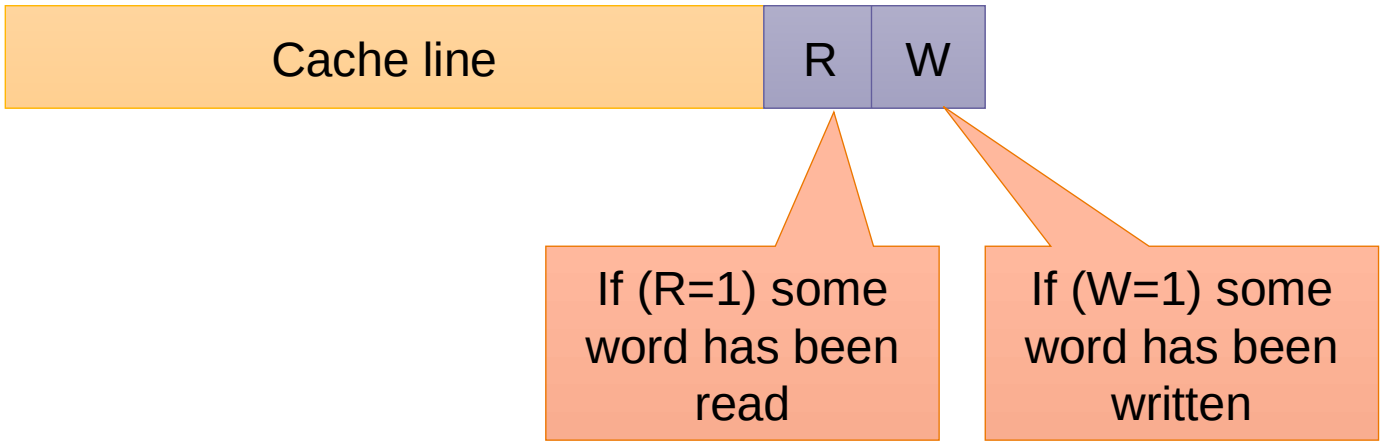
Hardware Support (mostly based on LogTM)

ISA Support

- Add three new instructions: begin, *abort, and commit*

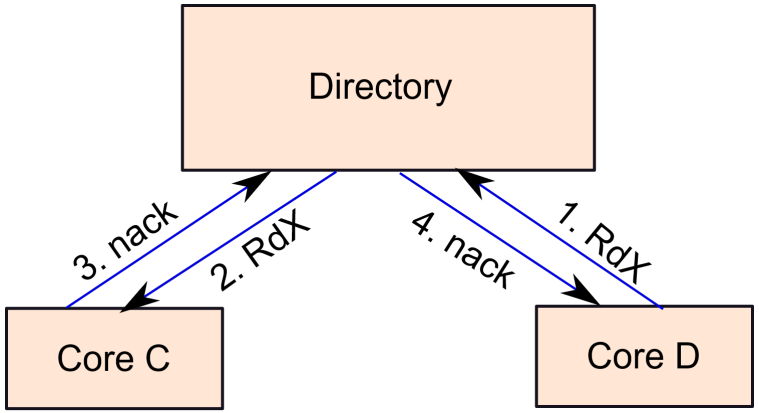
Version Management

- HW schemes mostly use eager version management – undo log
- The log has its dedicated set of addresses in virtual memory



Conflict Detection

Use the coherence protocol to detect conflicts



1. Let us say core *D* has a miss. It sends a read-miss to the directory.
2. The directory forwards it to core *C*.
3. *C* detects a conflict.
4. It sends a *nack* message to *D* (via the directory).
5. The transaction at *D* aborts.

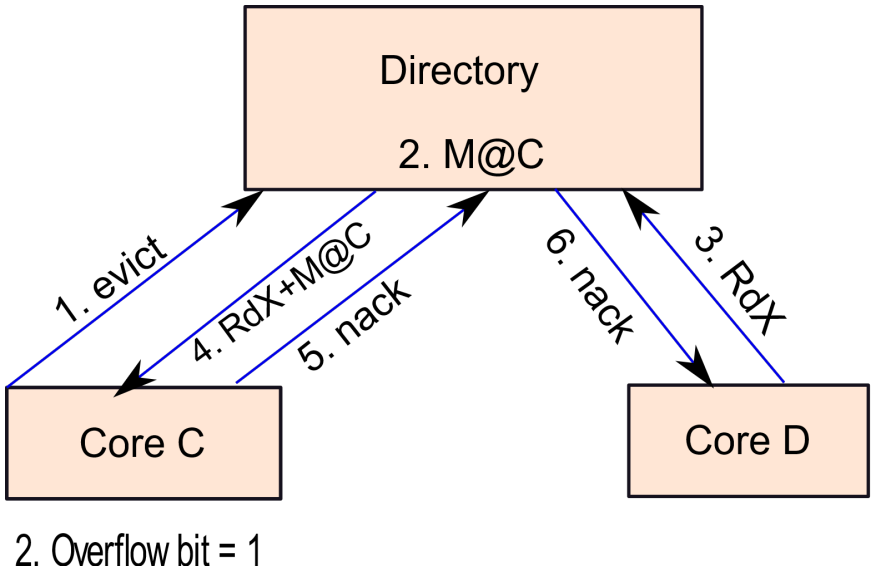
Eviction

What if there is an eviction?

- If there is an eviction in the *M* state, the state at the directory is set to *M@C*. *C* is the number of the core that evicted the block.
- Sets the overflow bit to 1.
- Let us assume silent evictions from the *S* state.

- Whenever the directory gets a request for a block in the M@C state, it forwards it to core *C*.
- Core *C* may not have the block in the cache.
- It will however infer a conflict because of the state of the block in the directory (the block is in its write set).
- If it gets a normal request (not @C), it will assume that the line was in the *S* state.

Example

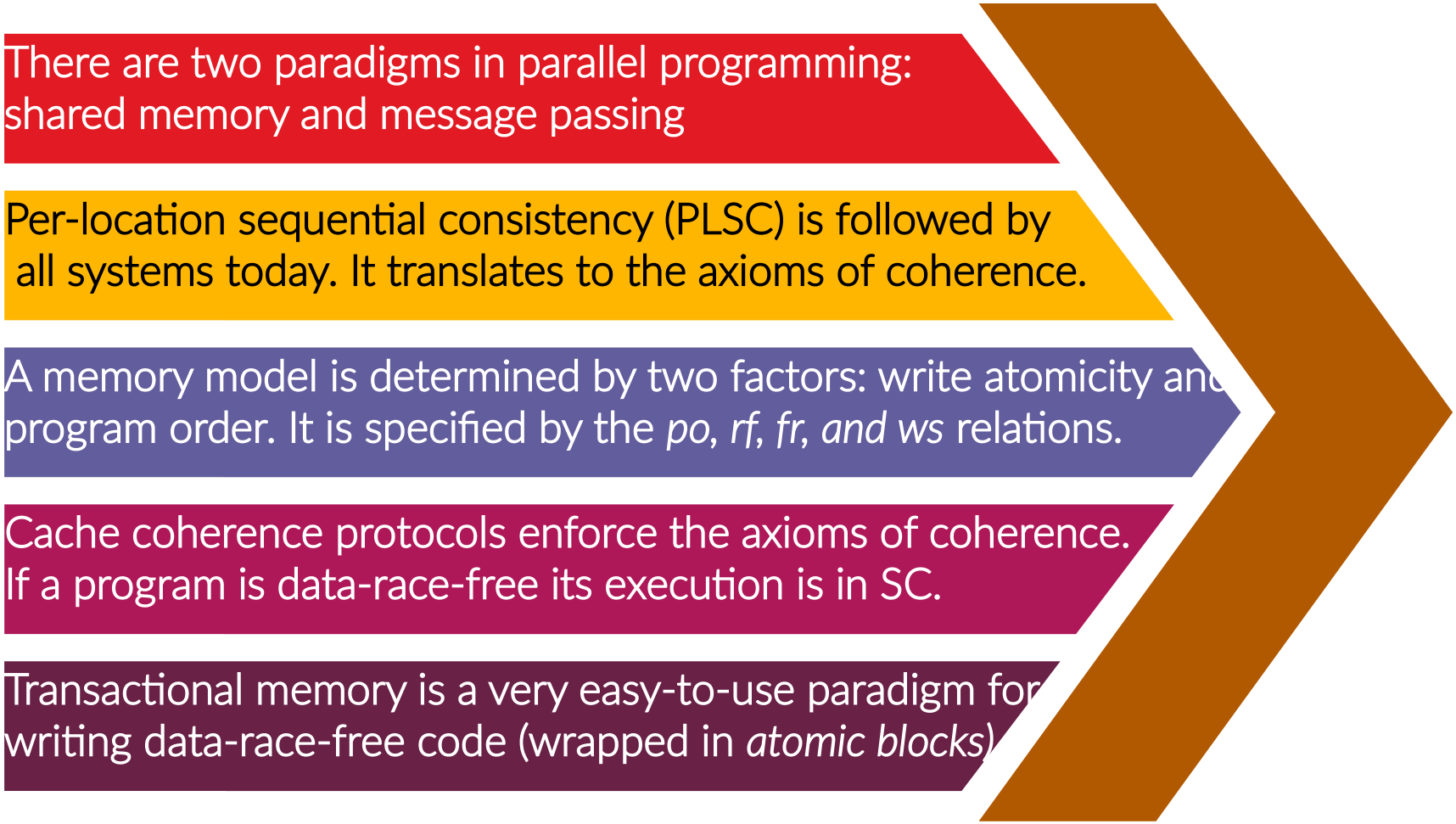


1. Core *D* sends a read-miss message to the directory.
2. The directory forwards it to core *C* with the (M@C) directive
3. Core *C* infers a conflict
4. Sends a *nack* back to *D* (via the directory)

Subtle Issues

- Assume a block has been evicted.
- If there is a conflicting request by a non-transactional access, the current transaction has to abort
- If a transaction aborts, then we need to clean the read/write sets. Some blocks might have gone to lower levels of the memory hierarchy.
- Since we restore the entire undo log, the correct state of the blocks is restored (in the L1 cache). Wrong values at lower levels do not matter.
- After a transaction finishes, all the R, W, M@C, and overflow bits need to be cleared.

Conclusion


There are two paradigms in parallel programming: shared memory and message passing
Per-location sequential consistency (PLSC) is followed by all systems today. It translates to the axioms of coherence.
A memory model is determined by two factors: write atomicity and program order. It is specified by the *po*, *rf*, *fr*, *and ws* relations.
Cache coherence protocols enforce the axioms of coherence. If a program is data-race-free its execution is in SC.
Transactional memory is a very easy-to-use paradigm for writing data-race-free code (wrapped in *atomic blocks*)

The End