LECTURE 13

Coherence and Atomicity

Miquel Pericàs

EDA284/DIT361 - VT 2020

What's cooking

1. **Lectures**
   - Today (9h-12h)**: Coherence and Sequential Consistency**
   - Friday (6/3, 10:30h-12h)**: Memory Consistency Models**
   - Tuesday (10/3): **The European Processor Initiative** (Guest Lecturers: Sonia Rani Gupta, Bhavishya Goel)
2. **Project Status**
   - Tomorrow deadline for revised version
3. **Problem Session**
   - Friday 9h-10:30h: **Chip Multiprocessors, GPGPU**
   - Tuesday (10/3): **Message Passing, Synchronization**
     - problems will be published soon

Overview of Lectures 12-14

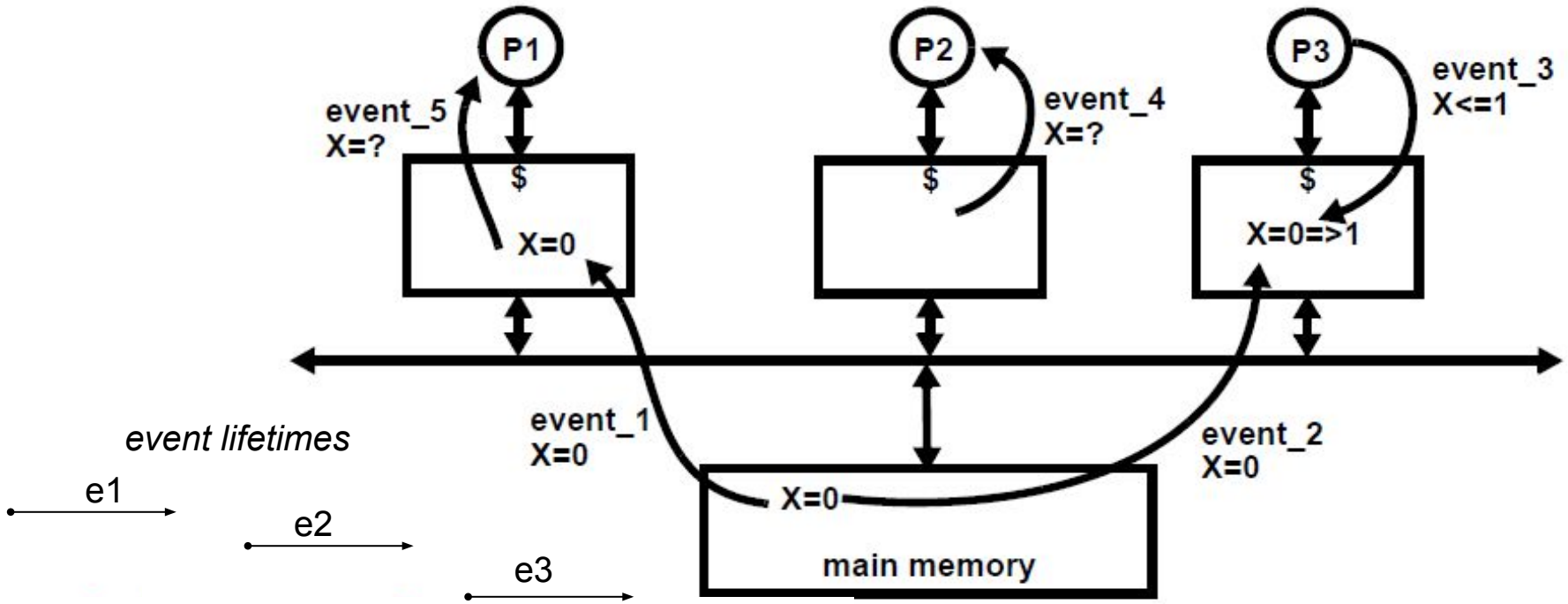
Goal: Study the correct and reliable communication of values in shared-memory multiprocessors

- **Synchronization (L12) 26/2**
- **Strict coherence (L13) 3/3**
- **Plain coherence (L13) 3/3**
- **Sequential consistency (L14, part 1) 3/3**
- **Memory consistency models (L14, part 2) 6/3**

AGENDA

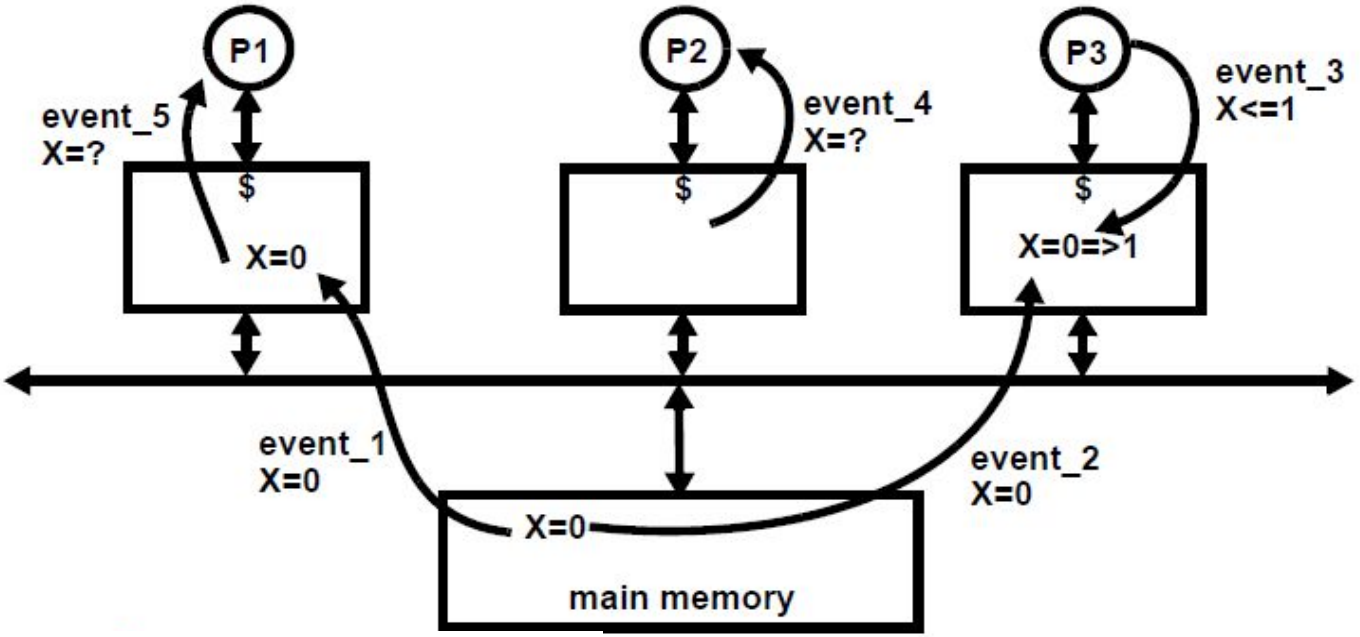
- Why is coherence so hard?
- Strict coherence and store atomicity
- Plain coherence
- Limitations of plain coherence
- Sequential Consistency

Memory coherence: what is the problem



- **Problem:** multiple copies of the same data
- **Uniprocessor definition:** Load must always return value of last store with same address in thread order
- **No such thing as "multi-thread" order in multiprocessors**
- **Assume that events 1, 2, 3, 4 and 5 are globally ordered:**
  - Do not overlap in time, or
  - Are atomic (take zero time)
- **Processors P1 and P2 "see" different values of X after event 3**

Coherence: why it is so hard

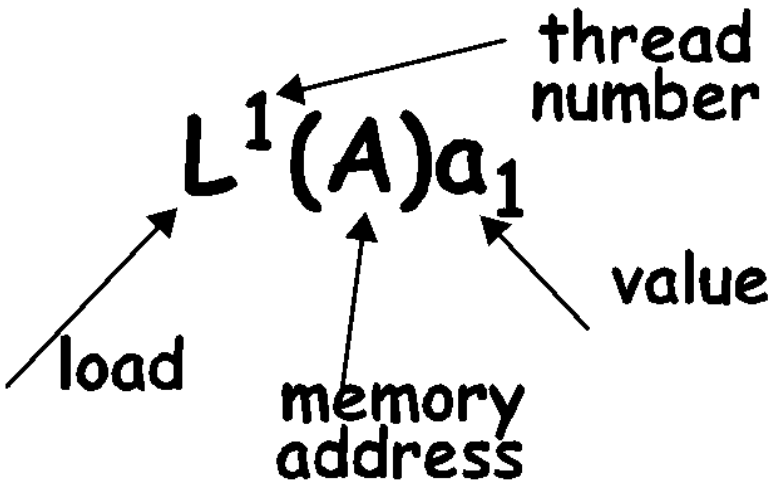


- After event 3, the caches contain different copies. Is this still coherent?
- Can the loads in events 4 and 5 return 0 or 1? Is it coherent?
- Need formal definition for coherence

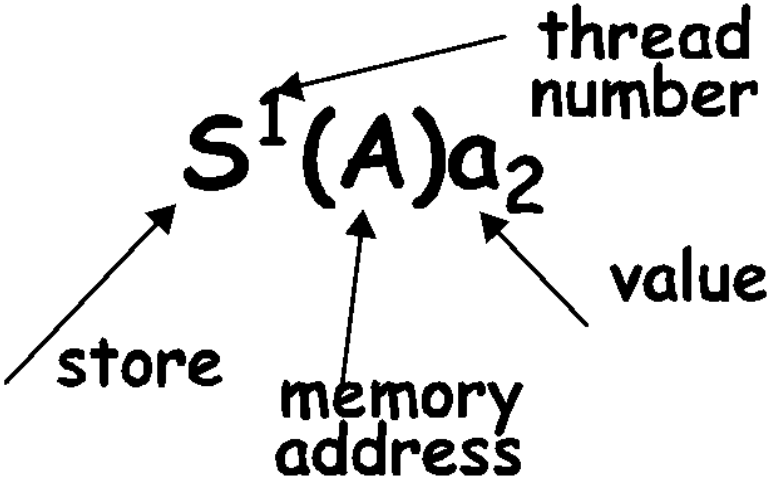
A strong definition: Strict coherence

- "A memory system is coherent if the value returned on a Load instruction is always the value given by the **latest** Store instruction with the same address"
  - Same as definition for uniprocessors
  - No multi-thread order → "latest" requires **global temporal order of stores**
- Difficult to extend as such to multiprocessors
  - Execution rate of threads is unpredictable, memory ops are overlapped
  - No instantaneous communication between processors
- Global order requires one of the following:
  - Memory accesses to same address do not overlap in time (difficult to enforce)
  - Stores are atomic so that all copies are updated instantaneously
  - Store/load orders are enforced by accesses to other memory locations (*see L14, memory consistency models*)
- Here we explore option #2: Store atomicity

Notation used throughout this lecture



Load by thread T1 of address @A, loading the value a_1



Store of thread T1 to address @A, storing the value a_2

Atomic memory accesses

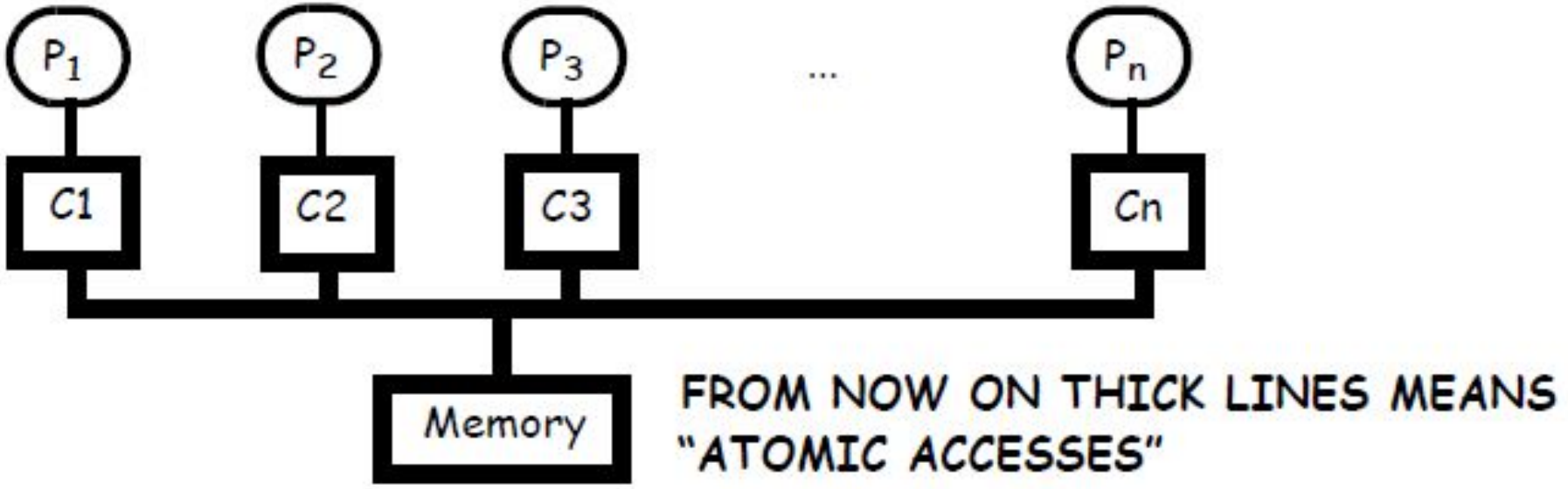
Example: MSI invalidate with Atomic Protocol Transactions (APT)

CLK	T1:	T2:	**Comments**:
t1:	$S^1(A)a_1$		Data not in C2 and
t2:	$L^1(A)a_1$		dirty in C1
t3:	$S^1(A)a_2$		
t4:	$L^1(A)a_2$		
t5:	----------	**$L^2(A)a_2$** ----------	**APT: Read Miss in C2;**
t6:	$L^1(A)a_2$		**A becomes Shared in C1**
			and C2; both threads
t7:		$L^2(A)a_2$	**can read A= a_2**
t8:	$L^1(A)a_2$		**no one can write**
t9:	----------	**$S^2(A)a_3$**----------	**APT:C1 is invalidated**
t10:		$L^2(A)a_3$	**and C2 becomes Dirty**
t11:	**$S^1(A)a_4$** ----------		**APT:Store miss in C1;**
t13:	$S^1(A)a_5$		**C2 is invalidated**

MSI-invalidate with APTs results in a single global temporal order of stores. Hence it is strictly coherent

Memory access atomicity in bus-based system

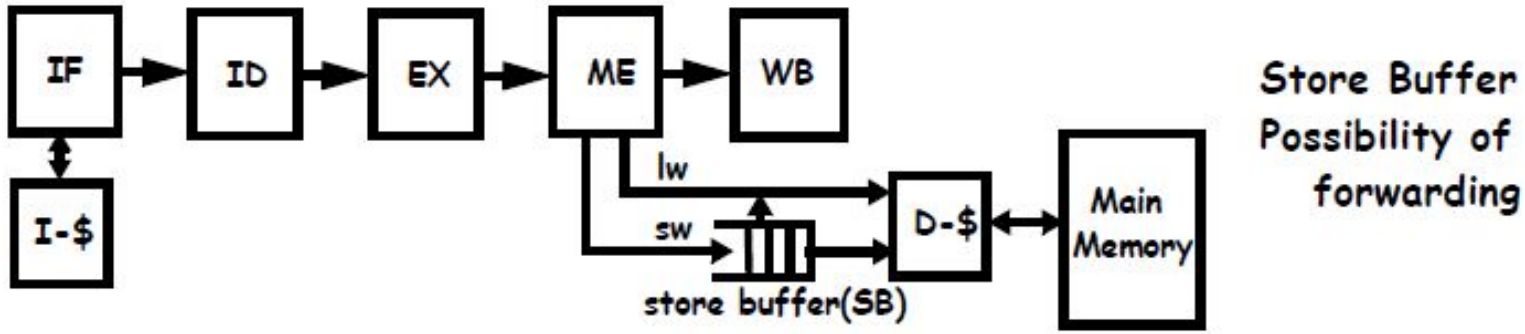
- In the early 1980s, processors were not pipelined, were connected by a single, circuit-switched bus, no store buffer

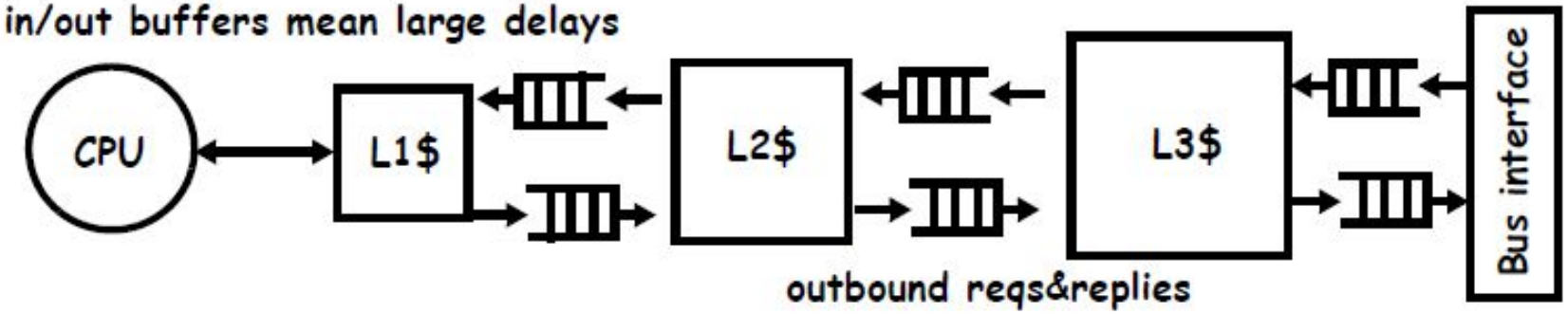


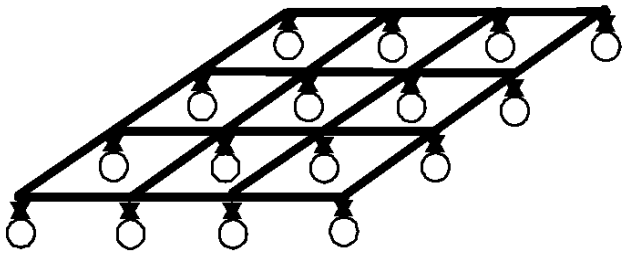
- On a coherence transaction, processor blocks, cache gets access to the bus and complete the transaction in remote caches atomically
  - Coherence transactions did not overlap in time
  - Thus the protocol worked exactly as its FSM
- **The coherence transaction is performed atomically when the bus is released**

Today we must deal with non-atomic transactions

Today coherence transactions are non-atomic







point-to-point packet-switched interconnect

many routes between two points

- adaptive routing

ordered vs. unordered networks

no order + very unpredictable transmission times

Memory access atomicity--sufficient condition

- **Coherence transactions cannot happen instantaneously**
  - Must make them look atomic to the software
  - Well-known problem: database systems, critical section
- **Sufficient condition:** make sure that only one value is accessible at any one time
- No thread can observe progressive changes. Effect becomes visible to all threads in a single operation (“atomicity”)
- Definition of Store Atomicity:
  - *Stores are atomic if different threads can never observe more than one value of the same memory location at the same time.*

Memory access atomicity--sufficient condition

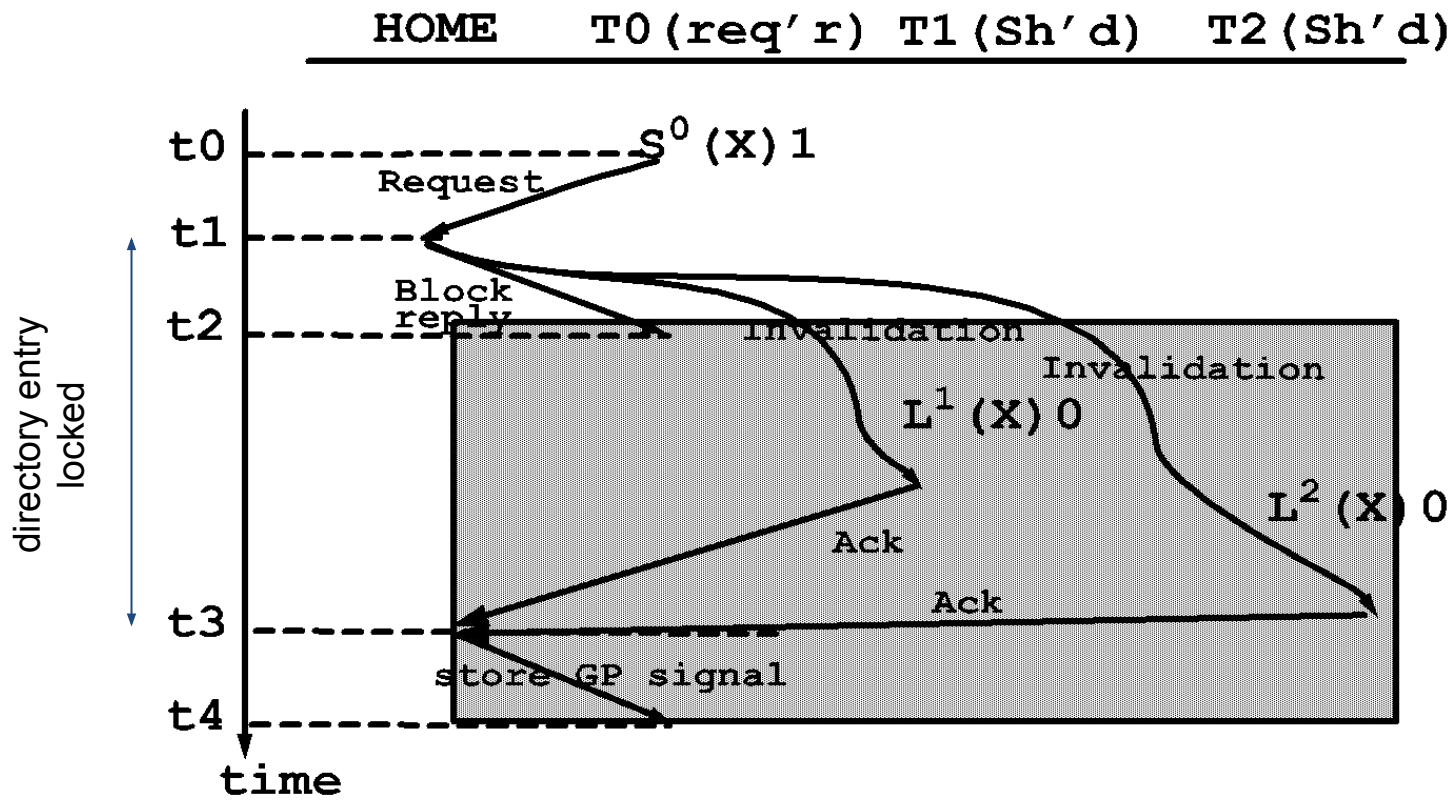
- **Definitions:** performed vs globally performed
  - A load is performed at the point in time when its value is bound and cannot be recalled
  - A store is performed with respect to thread *i* at the point in time when a load of thread *i* cannot return a value prior to the store
  - A store is globally performed when it is performed with respect to all threads
  - A load is globally performed when it is performed and the store providing the value is also globally performed
- **SUFFICIENT condition for Store Atomicity**
  - a global order of stores to each address is enforced
  - a load must be globally performed before its value can be used
    - value is bound
    - store is globally performed

This second condition means that no thread can observe a new value while any other thread can still observe the old value

store atomicity in cc-NUMAs

- In bus-based, global order of stores to same memory location is easily implemented: only one transaction per address in flight.
- In cc-NUMA: directory controller uses busy bit: ensure total order of stores!

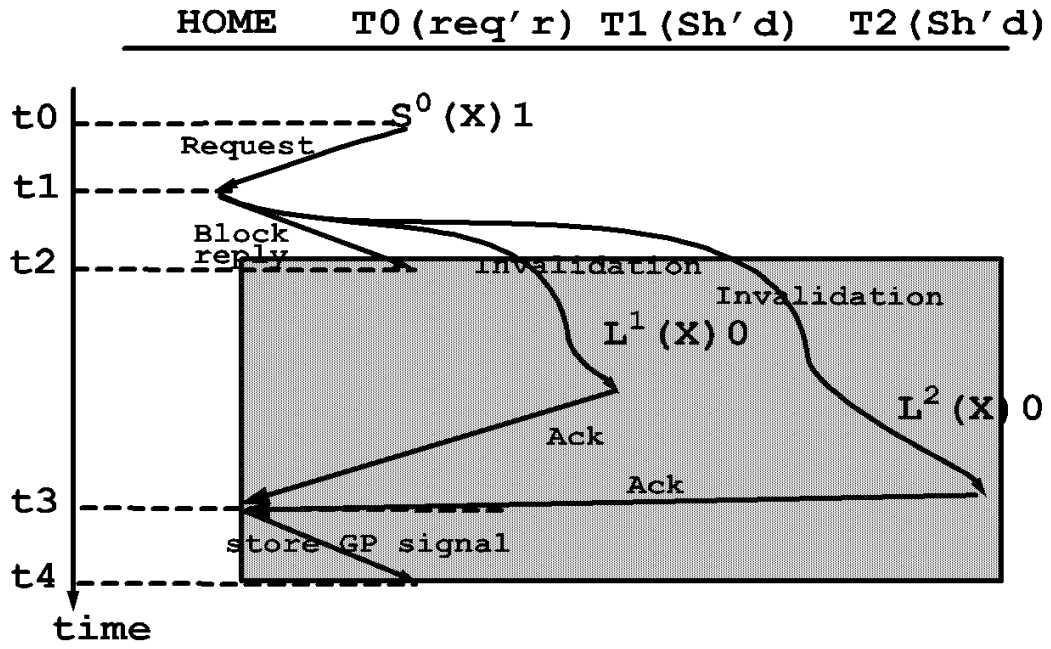
- Example. **How to enforce globally performed (GP) loads in cc-NUMA**:



write miss in thread 0 in a cc-NUMA with MSI-invalidate

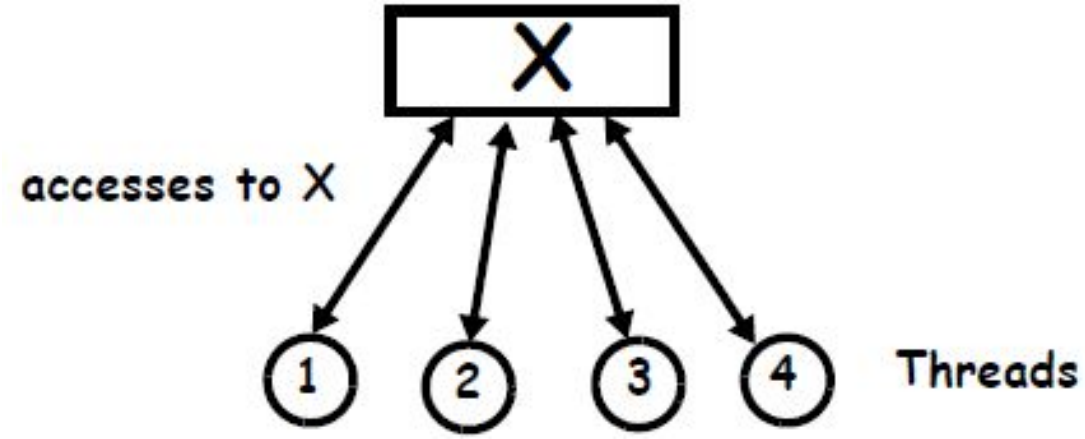
store atomicity and coherence

- The directory locks entry from t1 to t3
- Up until caches receive invalidation, T1 and T2 read latest GP values
- For store atomicity, T0 cannot return values from its own stores until t4 (nor give them away)
  - New copy becomes available atomically at t3
  - Store is globally performed at t3
- Can this be faster? Can the block be released to T0 at t2?
  - yes, but bytes in the block modified locally must be locked out for loads until t4, so that local threads (e.g. multithreading or cores w/ shared caches) load GP values only.



Strict coherence (store atomicity) is restrictive. Can it be relaxed?

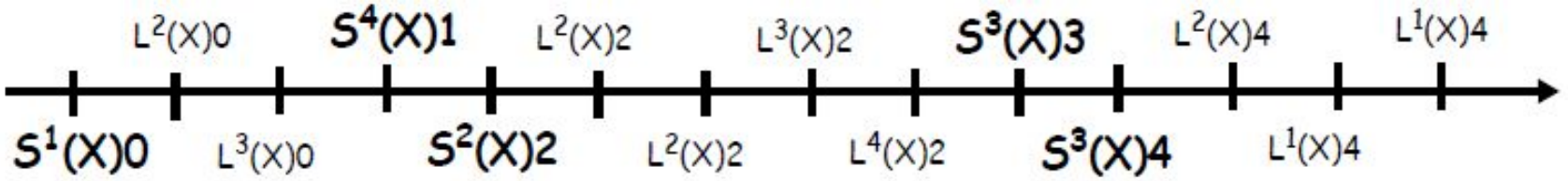
- *A system is coherent if it is equivalent to a system with a single copy of each data element*
- **Formal model:**



- **Rules of the model:**
  - Single copy of each data
  - Accesses one by one in thread order to each address
- A system is coherent if its memory accesses to each address can be executed correctly in thread order in a system with one single copy of each memory address

A weaker definition: Plain Coherence

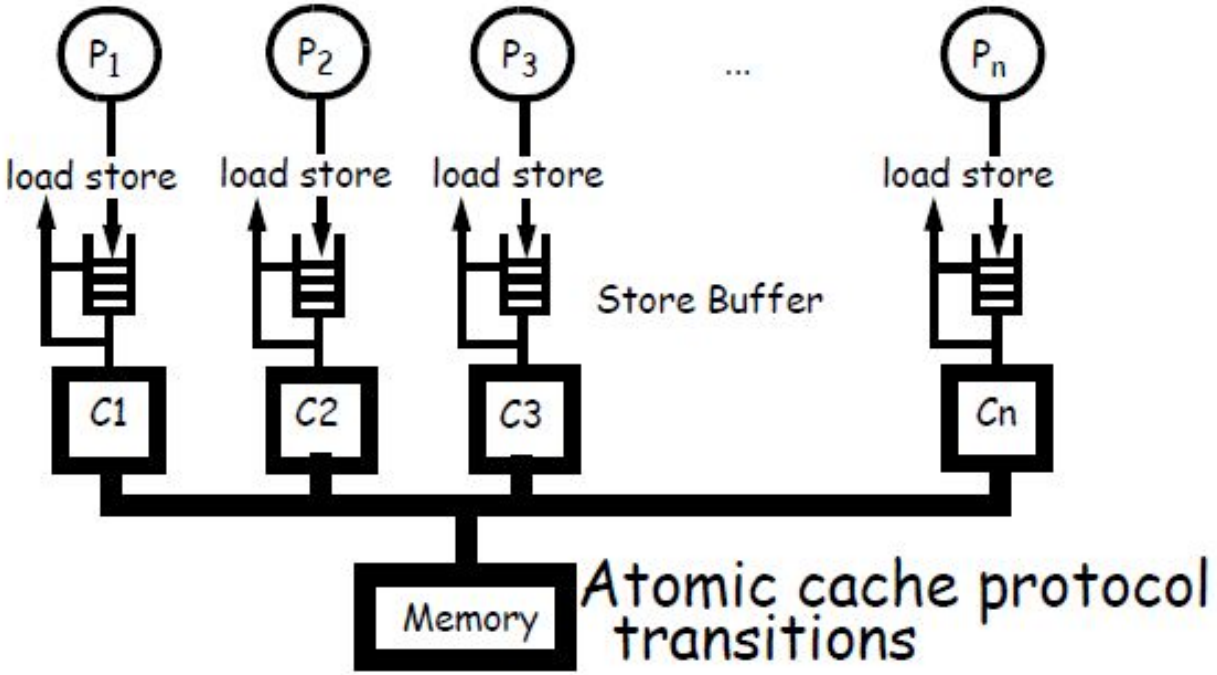
- **“A system is (plain) coherent iff, for every execution and for any memory location, it is possible to construct a serial order of all memory operations to the location such that:**
  - memory operations of each thread to the location occur in thread order
  - the value returned by a Load is the value of the latest Store to the location in the serial order”



- NOTES:
  - accesses by each thread must appear in thread order
  - serial order is not necessarily same as temporal order
- Since all accesses to every location are in thread order and every load returns the value of the latest store in the serial order, one can schedule accesses to one address one by one on the formal model and get the same values returned by all loads

Forwarding Store Buffer

- An enlightening example of hardware which is PLAIN coherent but NOT store atomic is that of a store buffer that can forward to loads



- Stores are inserted in the store buffer and issued to cache later
- Loads are satisfied by store buffer (if same address), otherwise go to memory:
  - Loads are not globally performed!

Important: store buffers are not part of cache coherence!

Forwarding Store Buffers

Stores are kept and combined in the Store Buffer & written to memory during WriteBack (WB), at which point they generate an atomic protocol transaction (APT)

	T1	T2	T3	CACHE STATES C1 C2 C3	Comments
t0			$L^3(A)a_0$	NIC NIC SHA	Miss in C3; APT1
t1	$S^1(A)a_1$			NIC NIC SHA	
t2			$S^3(A)a_2$	NIC NIC SHA	
t3	$L^1(A)a_1$			NIC NIC SHA	
t4		$S^2(A)a_3$		NIC NIC SHA	
t5			$L^3(A)a_2$	NIC NIC SHA	
t6	$S^1(A)a_4$			NIC NIC SHA	
t7		$L^2(A)a_3$		NIC NIC SHA	
t8	$L^1(A)a_4$			NIC NIC SHA	
t9		$L^2(A)a_3$		NIC NIC SHA	
t10	$WB^1(A)a_4$			MOD NIC INV	Miss in C1; APT2
t11	$L^1(A)a_4$			MOD NIC INV	Hit in C1
t12		$S^2(A)a_5$		DTY NIC INV	
t13		$WB^2(A)a_5$		INV DTY INV	Miss in C2; APT3
t14	$L^1(A)a_5$			SHA SHA INV	Miss in C1; APT4
t15			$L^3(A)a_2$	SHA SHA INV	
t16			$WB^3(A)a_2$	INV INV MOD	Miss in C3; APT5
t17	$L^1(A)a_2$			SHA INV SHA	Miss in C1; APT6
t18			$L^3(A)a_2$	SHA INV SHA	Hit in C3
t19		$L^2(A)a_2$		SHA SHA SHA	Miss in C2; APT7
t20		$S^2(A)a_6$		SHA SHA SHA	
t21		$L^2(A)a_6$		SHA SHA SHA	
t22		$WB^2(A)a_6$		INV MOD INV	Upgrade in C2; APT8

Stores are not atomic. Loads from different processors return different values at the same time

Forwarding Store Buffers

- **Aggressive Store Buffer management**
  - Stores overwrite previous values to same address (one single value in SB per address)
  - Stores forward values to loads
- **Despite lack of store atomicity, system is still coherent**
- **We first show the order of accesses to caches (GP Accesses)**
  - INIT→ $L^3(A)a_0$→ $WB^1(A)a_4$→ $L^1(A)a_4$→ $WB^2(A)a_5$→ $L^1(A)a_5$→ $WB^3(A)a_2$→ $L^1(A)a_2$→ $L^3(A)a_2$→ $L^2(A)a_2$→ $WB^2(A)a_6$
- **We then expand all WBs by local LW/SW's**
  - $L^3(A)a_0$ $S^1(A)a_1$ $L^1(A)a_1$ $S^1(A)a_4$ $L^1(A)a_4$ $L^1(A)a_4$ $S^2(A)a_3$ $L^2(A)a_3$ $L^2(A)a_3$ $S^2(A)a_5$ $L^1(A)a_5$ $S^3(A)a_2$ $L^3(A)a_2$ $L^3(A)a_2$ $L^1(A)a_2$ $L^3(A)a_2$ $L^2(A)a_2$ $S^2(A)a_6$ $L^2(A)a_6$
- **This results in the following orders:**

Temporal order: a_0→ a_1→ a_2→ a_3→ a_4→ a_5→ a_6

Coherence order: a_0→ a_1→ a_4→ a_3→ a_5→ a_2→ a_6

P1 observes a_1→ a_4→ a_5→ a_2

P2 observes a_3→ a_5→ a_2→ a_6

P3 observes a_0→ a_2

Threads skip values in coherence order because they do not observe them through Loads!

Hence system is Plain coherent

Forwarding Store Buffers

	T1	T2	T3	CACHE STATES C1 C2 C3	Comments
t0			$L^3(A)a_0$	NIC NIC SHA	Miss in C3; APT1
t1	$S^1(A)a_1$			NIC NIC SHA	
t2			$S^3(A)a_2$	NIC NIC SHA	
t3	$L^1(A)a_1$			NIC NIC SHA	
t4		$S^2(A)a_3$		NIC NIC SHA	
t5			$L^3(A)a_2$	NIC NIC SHA	
t6	$S^1(A)a_4$			NIC NIC SHA	
t7		$L^2(A)a_3$		NIC NIC SHA	
t8	$L^1(A)a_4$			NIC NIC SHA	
t9		$L^2(A)a_3$		NIC NIC SHA	
t10	$WB^1(A)a_4$			MOD NIC INV	Miss in C1; APT2
t11	$L^1(A)a_4$			MOD NIC INV	Hit in C1
t12		$S^2(A)a_5$		DTY NIC INV	
t13		$WB^2(A)a_5$		INV DTY INV	Miss in C2; APT3
t14	$L^1(A)a_5$			SHA SHA INV	Miss in C1; APT4
t15			$L^3(A)a_2$	SHA SHA INV	
t16			$WB^3(A)a_2$	INV INV MOD	Miss in C3; APT5
t17	$L^1(A)a_2$			SHA INV SHA	Miss in C1; APT6
t18			$L^3(A)a_2$	SHA INV SHA	Hit in C3
t19		$L^2(A)a_2$		SHA SHA SHA	Miss in C2; APT7
t20		$S^2(A)a_6$		SHA SHA SHA	
t21		$L^2(A)a_6$		SHA SHA SHA	
t22		$WB^2(A)a_6$		INV MOD INV	Upgrade in C2; APT8

- **Order of accesses to caches:**
  - INIT→ $L^3(A)a_0$→ $WB^1(A)a_4$→ $L^1(A)a_4$→ $WB^2(A)a_5$→ $L^1(A)a_5$→ $WB^3(A)a_2$→ $L^1(A)a_2$→ $L^3(A)a_2$→ $L^2(A)a_2$→ $WB^2(A)a_6$
- **We then expand all WBs by local LW/SW's:**
  - $L^3(A)a_0$ $S^1(A)a_1$ $L^1(A)a_1$ $S^1(A)a_4$ $L^1(A)a_4$ $L^1(A)a_4$ $S^2(A)a_3$ $L^2(A)a_3$ $L^2(A)a_3$ $S^2(A)a_5$ $L^1(A)a_5$ $S^3(A)a_2$ $L^3(A)a_2$ $L^3(A)a_2$ $L^1(A)a_2$ $L^3(A)a_2$ $L^2(A)a_2$ $S^2(A)a_6$ $L^2(A)a_6$

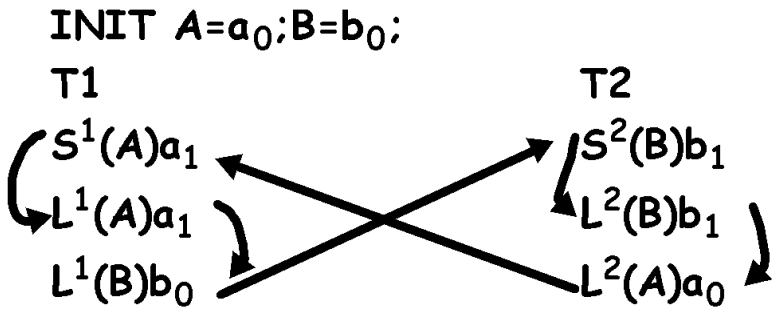
Privacy Principle

- **“Privacy Principle”**: a thread may access its own private values which are not propagated to other threads without violating coherence
  - Reason is no other thread can observe the values, so it’s easy to insert the accesses to them in a global order

the problem with plain coherence

- **Coherence is not composable with other possible orders**
  - Load-load or load-store on different locations

- **Example**



- According to plain coherence, this execution is coherent for A or B individually.
- **But**: if the loads in both threads must be ordered, then it is not possible to find a global order of all accesses while maintaining coherence
- When a global order does not exist, reasoning about executions is much more complex.

The above execution is not possible with store atomicity, since store atomicity orders accesses in real time

Are Coherence and Store Atomicity sufficient?

- **Is the following code coherent? And Store Atomic?**

```
INIT A=0,B=0;
T1              T2
S1(A)1          S2(B)1
L1(B)0          L2(A)0
```

- **Yes**, it is both coherent (because only two accesses each address) and store atomic (all loads are performed on GP values)
- **Is the following code coherent and store atomic?**

```
INIT A=0,B=0;
T1                      T2
S1(A)1                  S2(B)1
BARRIER(bar1)           BARRIER(bar1)
L1(B)0                  L2(A)0
```

- **Yes. But is it correct?**
- Not what the programmer expects!

Coherence and store atomicity deal only with single memory locations. Not enough for program correctness

Motivation for memory consistency models!

Summary so far

- why is coherence so hard
- strict coherence
- store atomicity
- plain coherence
- forwarding store buffers and generalizations
- Limitations of plain coherence and store atomicity
- next: memory consistency models