Cache Coherence, memory consistency, and their interaction

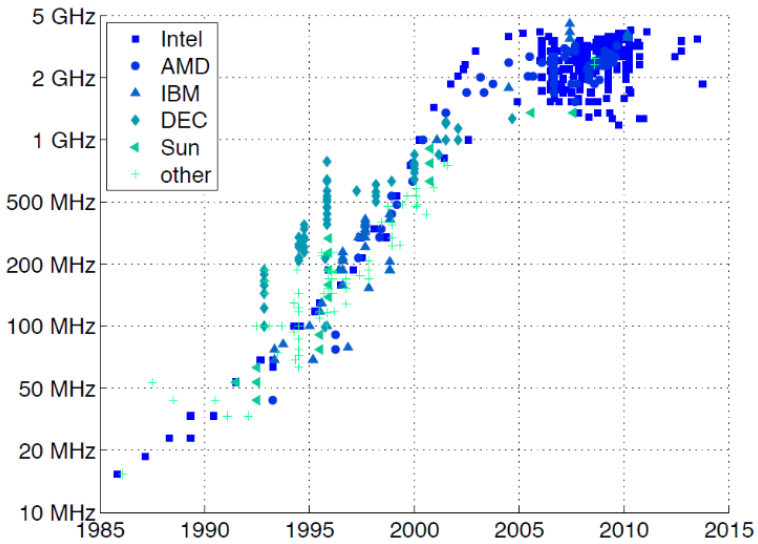
Alberto Ros

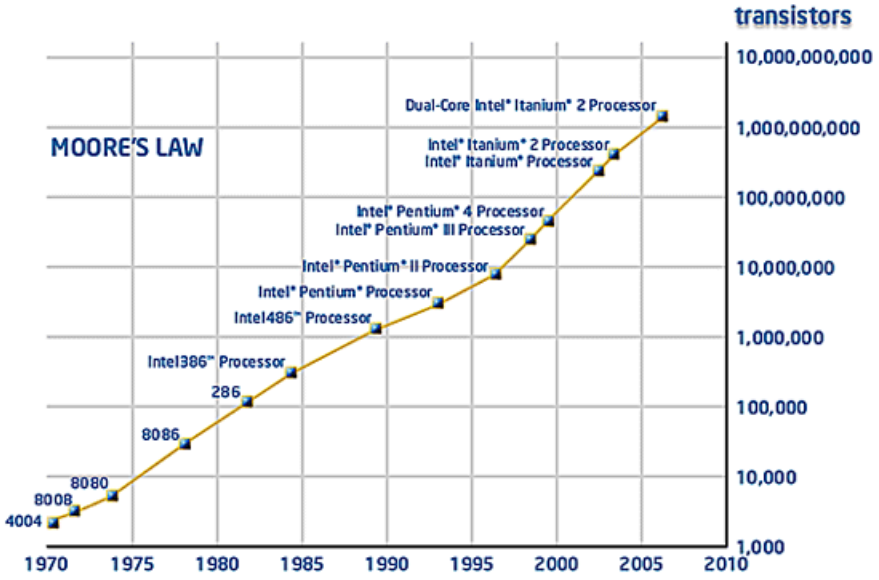
Universidad de Murcia
aros@um.es

15th of June, 2017

WHY MULTICORE?

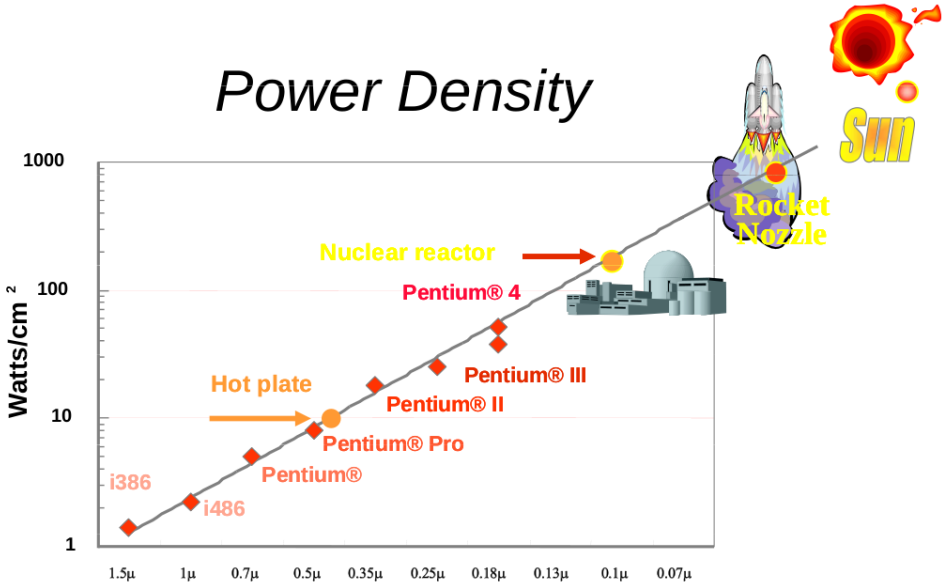
- UPMARC Summer School on Multicore Computing
- Why do we need multicores?
  - Need for higher performance
    - Increase frequency
    - Increase transistor count → Moore's law
  - ... but also for energy efficient

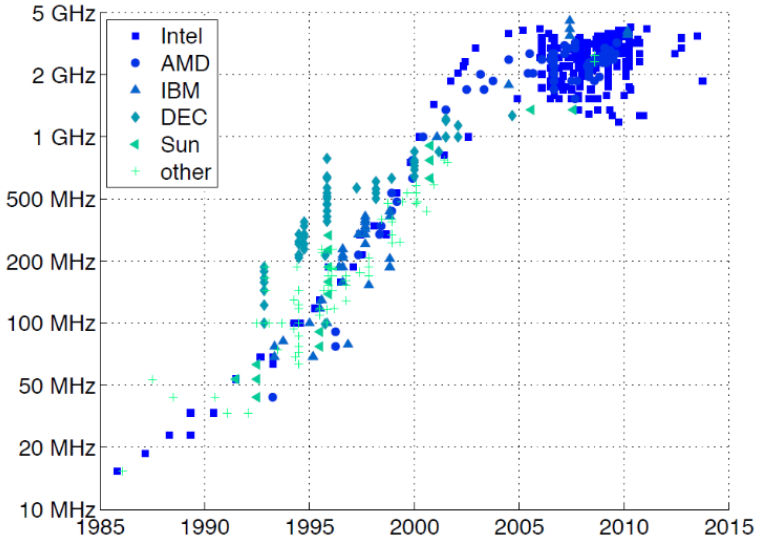


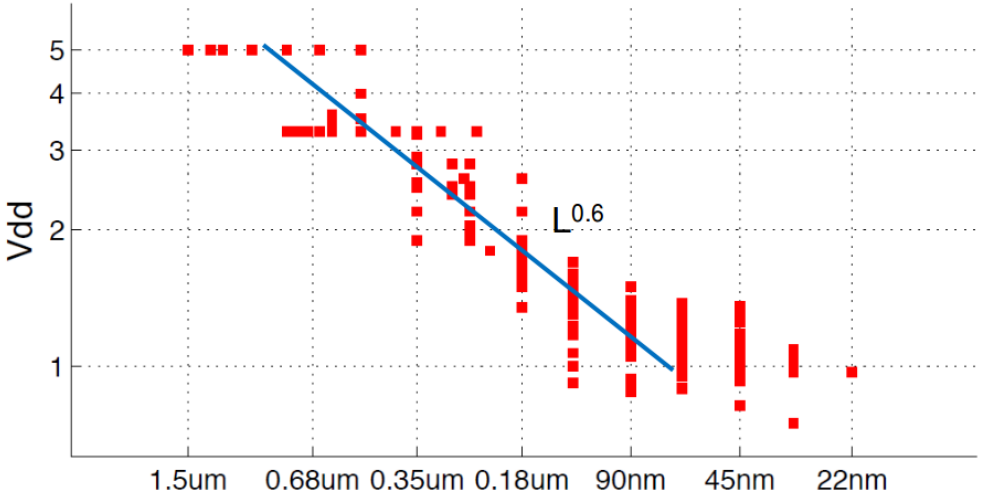


Power consumption problem

- Dynamic power $P_{dyn} = \alpha C V^2 f$
  - α: activity factor
  - C: capacitance
  - V: **voltage**
  - f: **frequency**

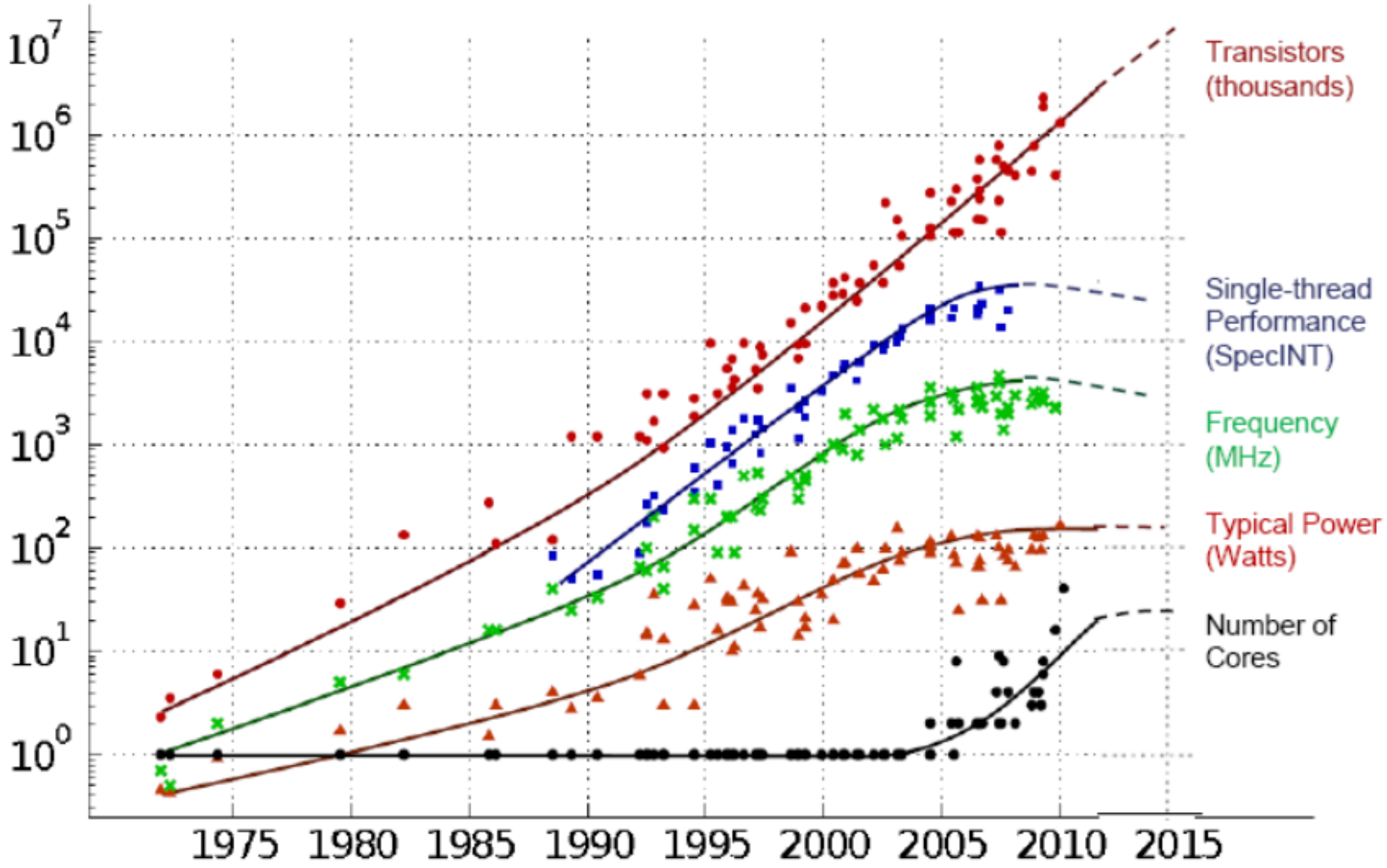






NEED FOR MULTICORES

35 YEARS OF MICROPROCESSOR TREND DATA



Building and programming multicores

- Most multicores support shared memory in hardware
  - Single shared address space for all cores
  - Communication among cores through reads and writes
  - Eases programming

Building and programming multicores

- Most multicores support shared memory in hardware
  - Single shared address space for all cores
  - Communication among cores through reads and writes
  - Eases programming
- How to write correct (parallel) programs?
- What is a correct (parallel) program?

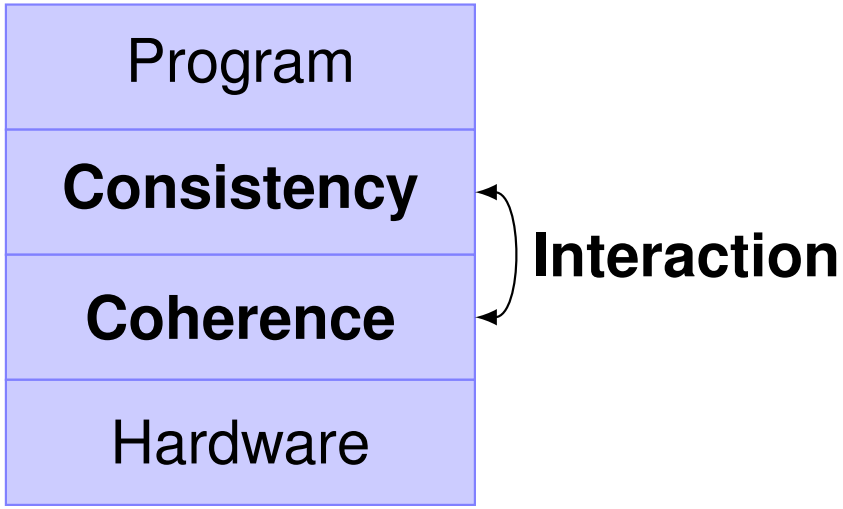
Building and programming multicores

- Most multicores support shared memory in hardware
  - Single shared address space for all cores
  - Communication among cores through reads and writes
  - Eases programming
- How to write correct (parallel) programs?
- What is a correct (parallel) program?
  - Memory consistency model: defines correct behavior of reads and writes (e.g. the value(s) that each read can get)
  - Must be simple / intuitive

Building and programming multicores

- Memory consistency model: defines correct program behavior
- Cache coherence protocol: eases the implementation of a consistency model
  - Makes caches in a shared memory multicore functionally invisible (timing can be inferred)

- Memory consistency model: defines correct program behavior
- Cache coherence protocol: eases the implementation of a consistency model
  - Makes caches in a shared memory multicore functionally invisible (timing can be inferred)



This lecture is about *coherence, consistency, and their interaction*

Outline

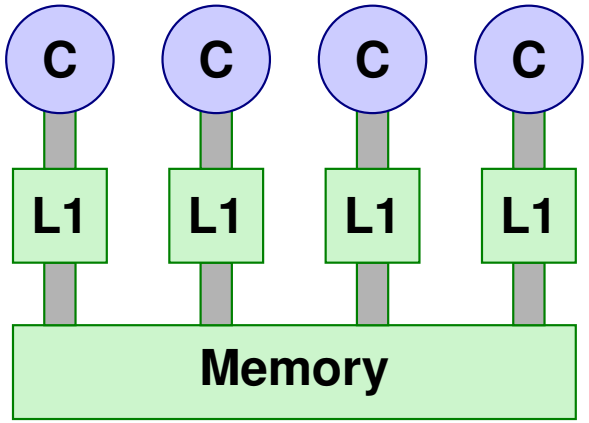
1. Cache coherence
2. Memory consistency
3. Coherence-Consistency Interaction
4. Open research questions

Outline

1. Cache coherence
2. Memory consistency
3. Coherence-Consistency Interaction
4. Open research questions

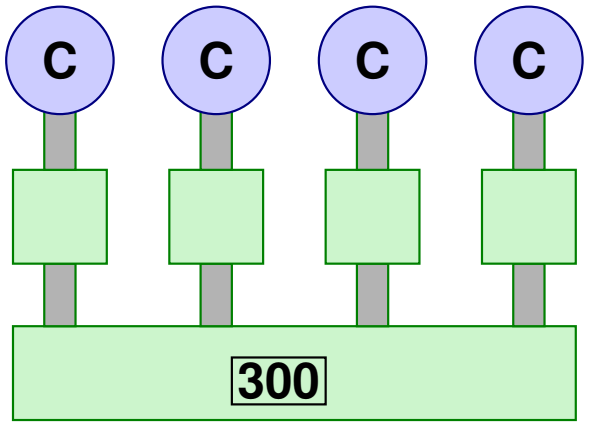
Introduction to cache coherence

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



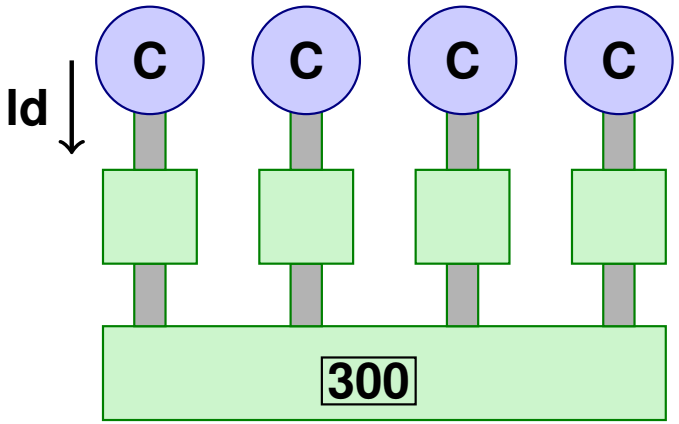
Introduction to cache coherence

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



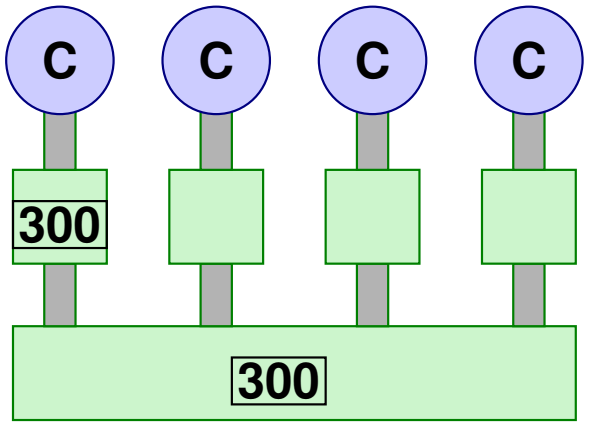
Introduction to cache coherence

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



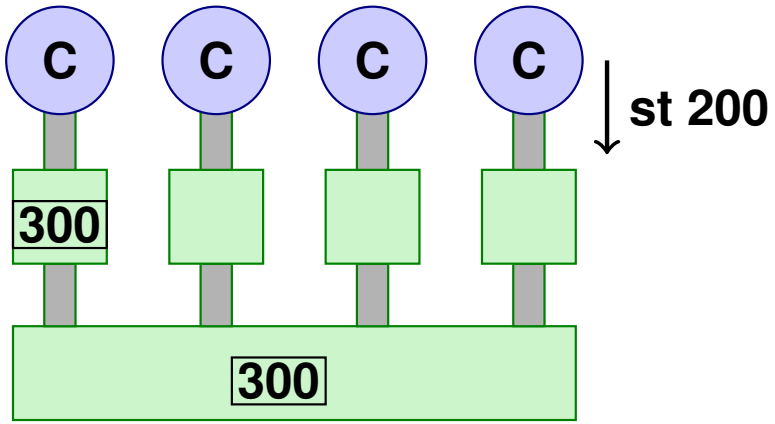
Introduction to cache coherence

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



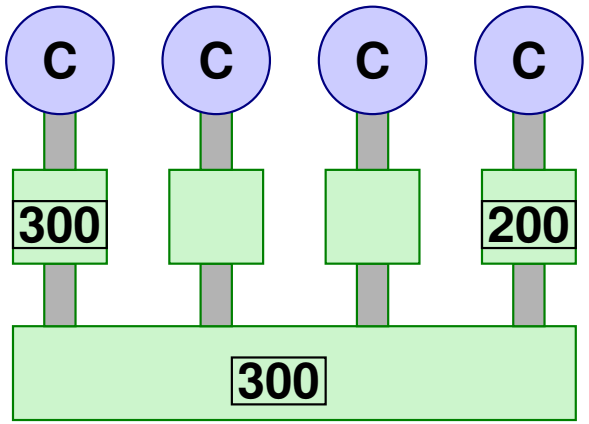
Introduction to cache coherence

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



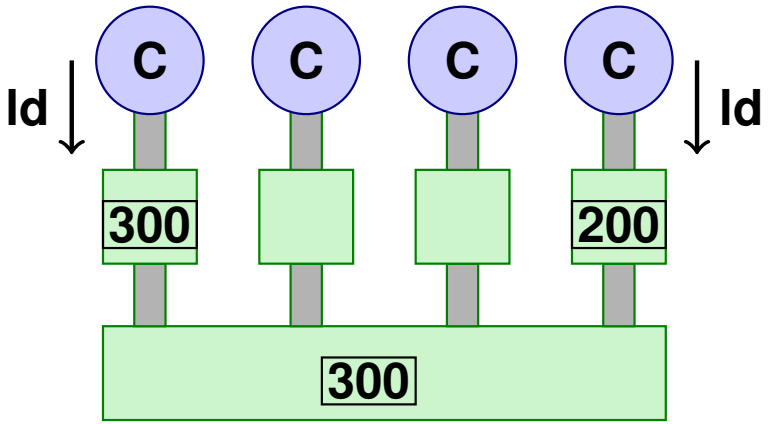
Introduction to cache coherence

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



Introduction to cache coherence

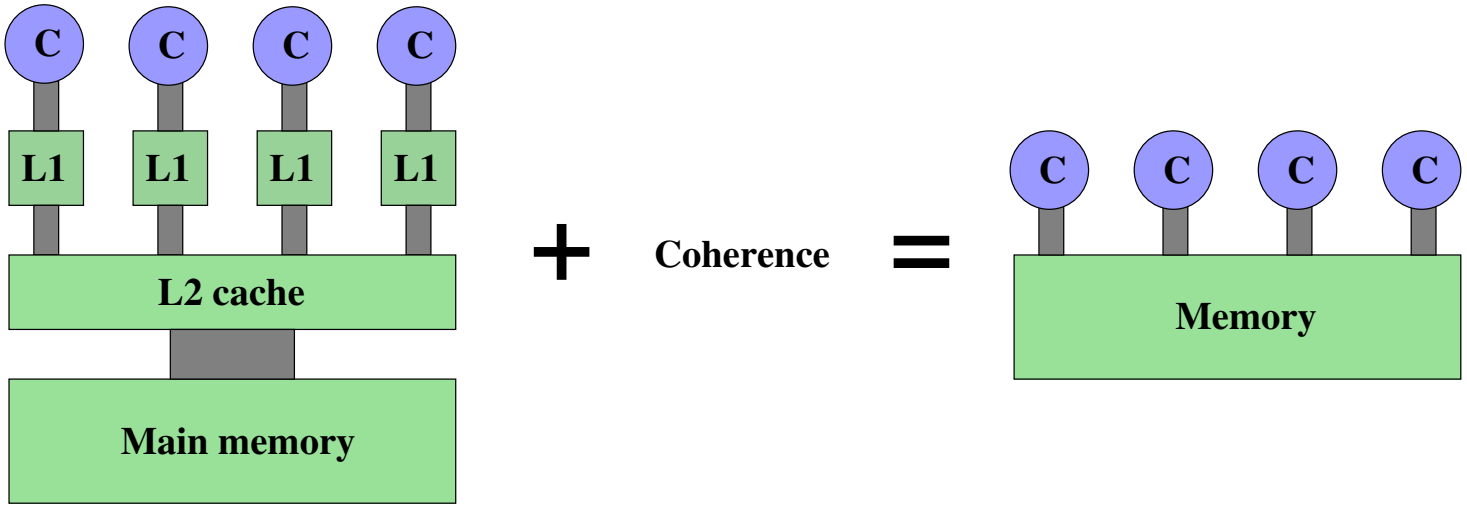
- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



- Incoherence!

Introduction to cache coherence

- The cache coherence protocol makes caches in a shared memory functionally invisible



- Data appear to be in a single location
- Operates per address → each address is treated as independent

Coherence invariants

Two invariants suffice to accomplish the coherence goal

Coherence invariants

Two invariants suffice to accomplish the coherence goal

1. SWMR: Single-writer–multiple-reader

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it

Coherence invariants

Two invariants suffice to accomplish the coherence goal

1. SWMR: Single-writer–multiple-reader

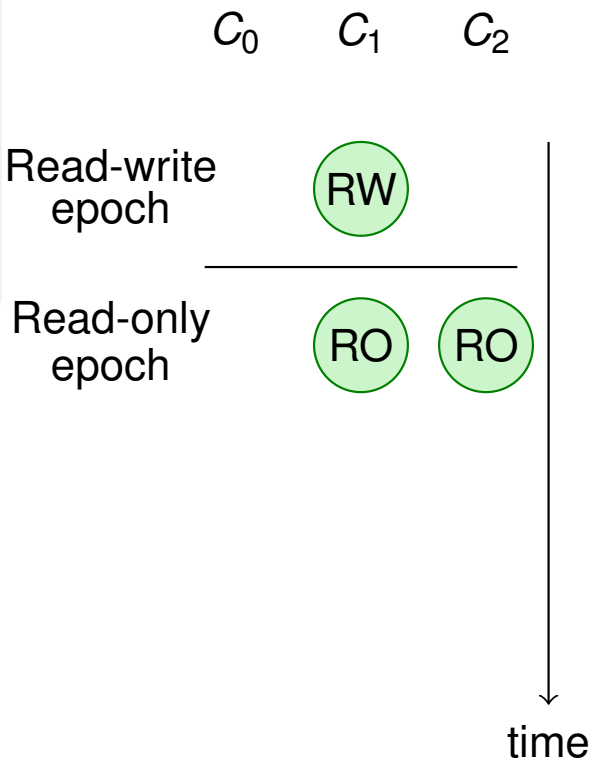
For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it

Coherence invariants

Two invariants suffice to accomplish the coherence goal

1. SWMR: Single-writer–multiple-reader

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it

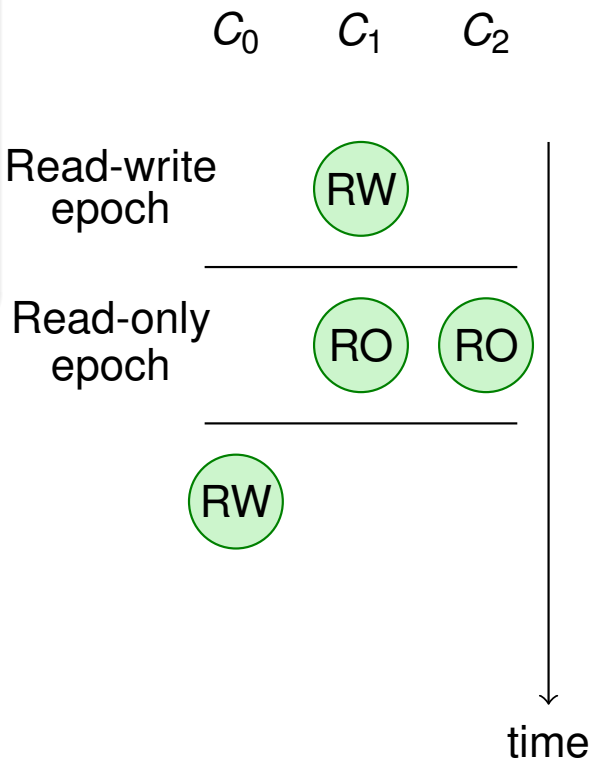


Coherence invariants

Two invariants suffice to accomplish the coherence goal

1. SWMR: Single-writer–multiple-reader

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it

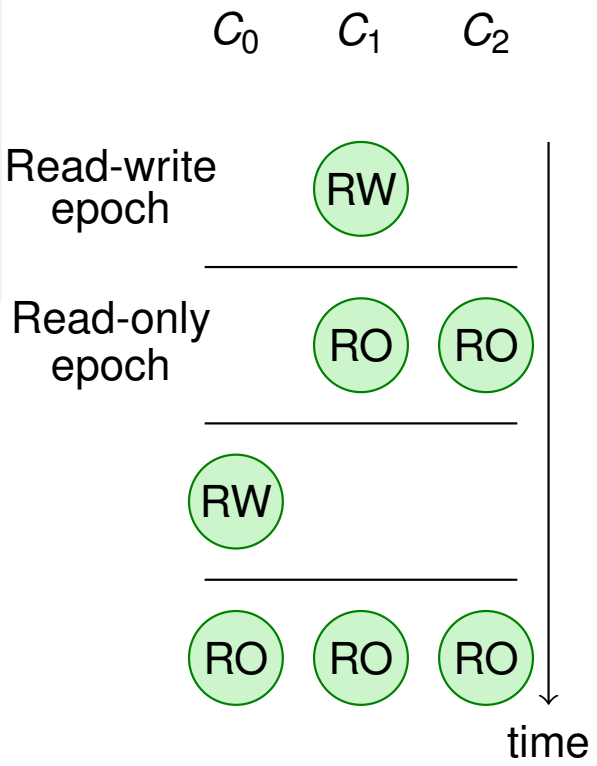


Coherence invariants

Two invariants suffice to accomplish the coherence goal

1. SWMR: Single-writer–multiple-reader

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it



Coherence invariants

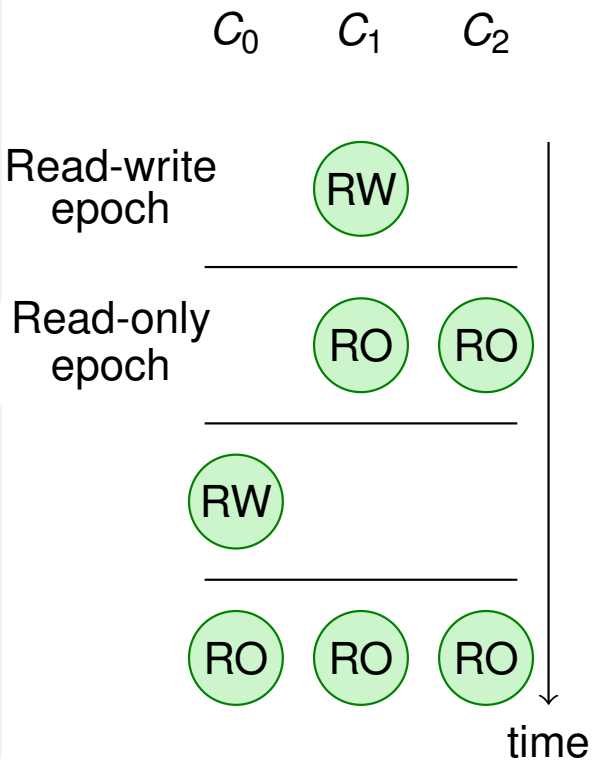
Two invariants suffice to accomplish the coherence goal

1. SWMR: Single-writer–multiple-reader

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it

2. Data value

The value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch



Coherence invariants

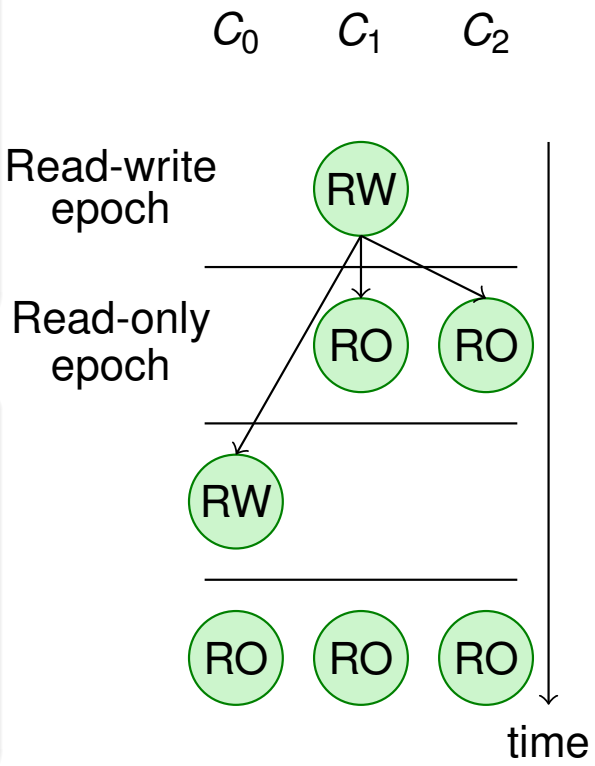
Two invariants suffice to accomplish the coherence goal

1. SWMR: Single-writer–multiple-reader

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it

2. Data value

The value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch



Standard implementation

- Read-only permission by all cores
  - Load operations require read-only permission to perform
    - Perform: load the data from memory into a register
- Read-write permission by one core
  - Store operations require read-write permission to perform
    - Perform: write the data to memory

Standard implementation

- Read-only permission by all cores
  - Load operations require read-only permission to perform
    - Perform: load the data from memory into a register
- Read-write permission by one core
  - Store operations require read-write permission to perform
    - Perform: write the data to memory
- Single-core read-write permission obtained through invalidation

Standard implementation

- Read-only permission by all cores
  - Load operations require read-only permission to perform
    - Perform: load the data from memory into a register
- Read-write permission by one core
  - Store operations require read-write permission to perform
    - Perform: write the data to memory
- Single-core read-write permission obtained through invalidation
- At which granularity are permissions granted?
  - In theory, it is possible at the finest granularity (1 byte)

Standard implementation

- Read-only permission by all cores
  - Load operations require read-only permission to perform
    - Perform: load the data from memory into a register
- Read-write permission by one core
  - Store operations require read-write permission to perform
    - Perform: write the data to memory
- Single-core read-write permission obtained through invalidation
- At which granularity are permissions granted?
  - In theory, it is possible at the finest granularity (1 byte)
  - In practice, this is too expensive.
    - A directory keeps track of all permissions per core and coherence unit
    - Caches store data at block granularity
    - Solution: the coherence unit is the memory block, e.g., 64 bytes

Standard implementation

- Each memory block can have the following attributes

Standard implementation

- Each memory block can have the following attributes
  - Validity: it has a valid copy (may have read-only permission)

Standard implementation

- Each memory block can have the following attributes
  - Validity: it has a valid copy (may have read-only permission)
  - Exclusivity: it is the one copy (may have read-write permission)

Standard implementation

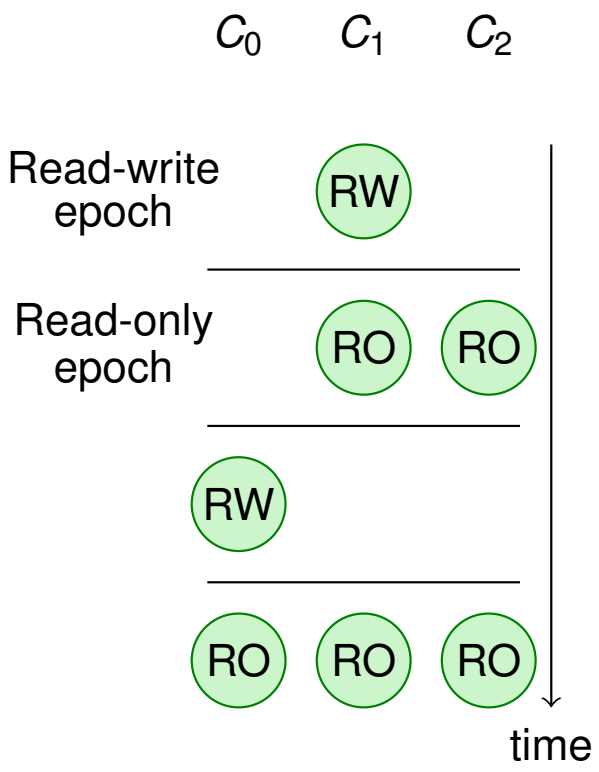
- Each memory block can have the following attributes
  - Validity: it has a valid copy (may have read-only permission)
  - Exclusivity: it is the one copy (may have read-write permission)
  - Dirtiness: it has been locally modified

Standard implementation

- Each memory block can have the following attributes
  - Validity: it has a valid copy (may have read-only permission)
  - Exclusivity: it is the one copy (may have read-write permission)
  - Dirtiness: it has been locally modified
  - Ownership: it will be provide the data on a request

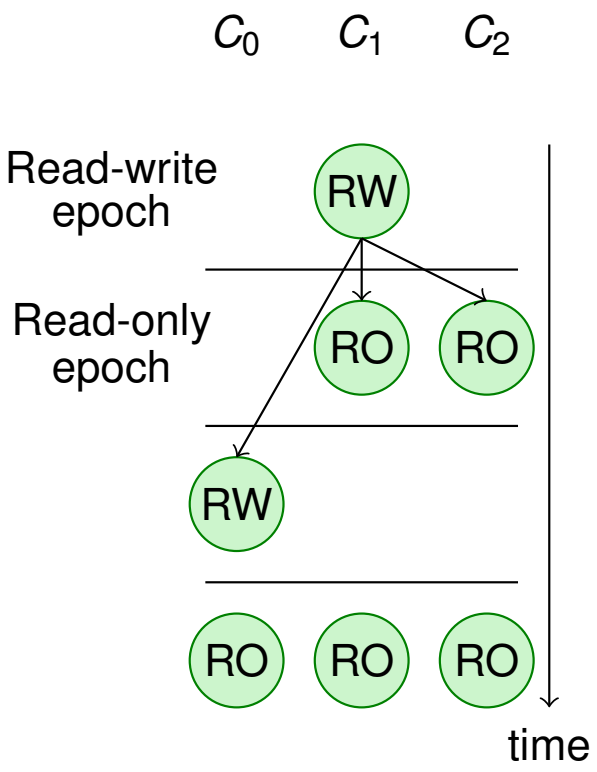
Standard implementation

- Each memory block can have the following attributes
  - Validity: it has a valid copy (may have read-only permission)
  - Exclusivity: it is the one copy (may have read-write permission)
  - Dirtiness: it has been locally modified
  - Ownership: it will be provide the data on a request
- SWMR invariant → Validity and Exclusivity attributes



Standard implementation

- Each memory block can have the following attributes
  - Validity: it has a valid copy (may have read-only permission)
  - Exclusivity: it is the one copy (may have read-write permission)
  - Dirtiness: it has been locally modified
  - Ownership: it will be provide the data on a request
- SWMR invariant → Validity and Exclusivity attributes
- Data value invariant → Dirtiness and Ownership attributes



A simple MSI protocol

- Blocks can be in a particular state (a collection of attributes)
- Each memory has a coherence controller
  - State, event, action
- One of the simplest protocols have 3 states: MSI
  - M (Modified): Validity, Exclusivity, Dirtiness, and Ownership
  - S (Shared): Validity
  - I (Invalid): None

A MESI PROTOCOL

- MSI is inefficient for sequential applications
- The E state was introduced to avoid write-after-read to incur two cache misses (read-only permission and read-write permission)
- The states of the MESI protocol are:
  - M (Modified): VALIDITY, EXCLUSIVITY, DIRTINESS, and OWNERSHIP
  - E (Exclusive): VALIDITY, EXCLUSIVITY, and possibly OWNERSHIP
  - S (Shared): VALIDITY
  - I (Invalid): None

A MOESI PROTOCOL

- The owner state was introduced for multiprocessors with high access latency to shared memory
  - But it increases indirection
- The states of the MOESI protocol are:
  - M (Modified): VALIDITY, EXCLUSIVITY, DIRTINESS, and OWNERSHIP
  - O (Owner): VALIDITY, DIRTINESS, and OWNERSHIP
  - E (Exclusive): VALIDITY, EXCLUSIVITY, and possibly OWNERSHIP
  - S (Shared): VALIDITY
  - I (Invalid): None

Transient states and complexity

TABLE 8.1: MSI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}										
IS^D	stall	stall	stall			stall		-/S		-/S		
IM^{AD}	stall	stall	stall	stall	stall			-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	stall	stall						ack--	-/M
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^A			send Inv-Ack to Req/I						
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	stall	stall						ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I							
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I					
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I					
II^A	stall	stall	stall				-/I					

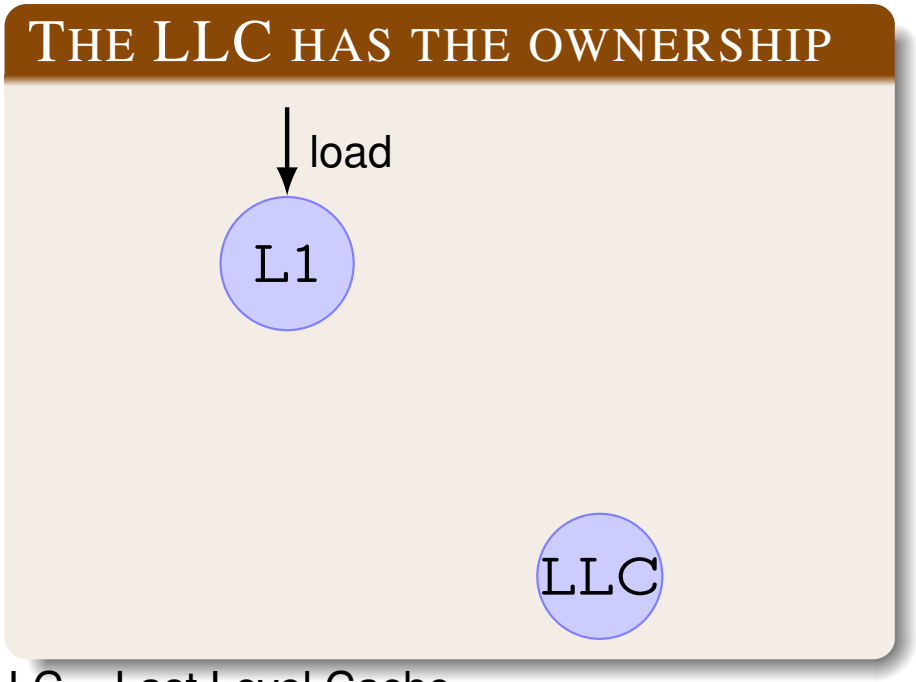
Source: [1] D. Sorin, M. D. Hill, D. A. Wood “A Primer on Memory Consistency and Cache Coherence”, 2011.

Getting read-only permission

- Load operations require read permission to perform
- If the cache has not read permission ⇒ cache miss
- A read miss coherence transaction is generated

Getting read-only permission

- Load operations require read permission to perform
- If the cache has not read permission ⇒ cache miss
- A read miss coherence transaction is generated

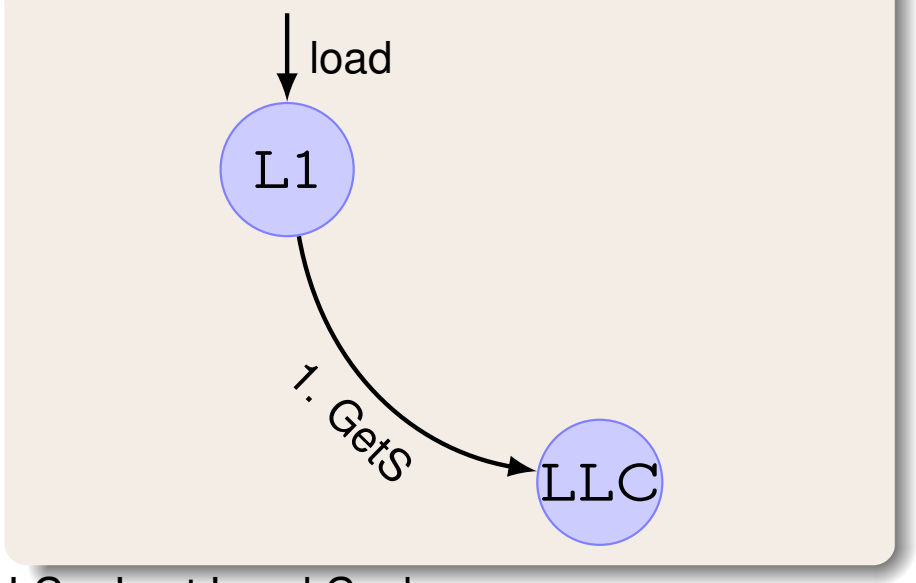


LLC = Last Level Cache

Getting read-only permission

- Load operations require read permission to perform
- If the cache has not read permission ⇒ cache miss
- A read miss coherence transaction is generated

The LLC has the ownership

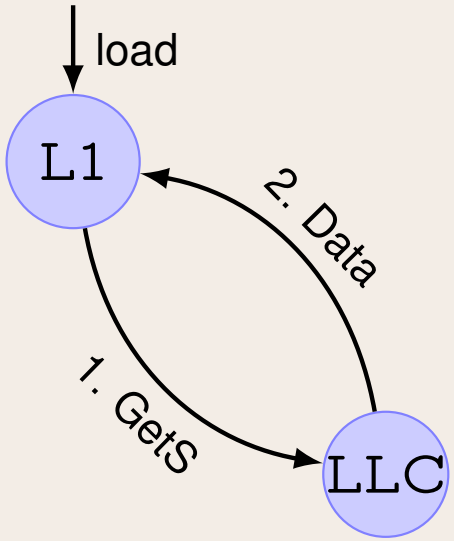


LLC = Last Level Cache

Getting read-only permission

- Load operations require read permission to perform
- If the cache has not read permission ⇒ cache miss
- A read miss coherence transaction is generated

The LLC has the ownership

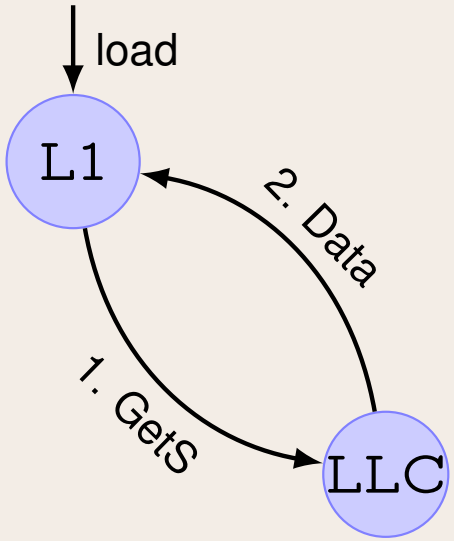


LLC = Last Level Cache

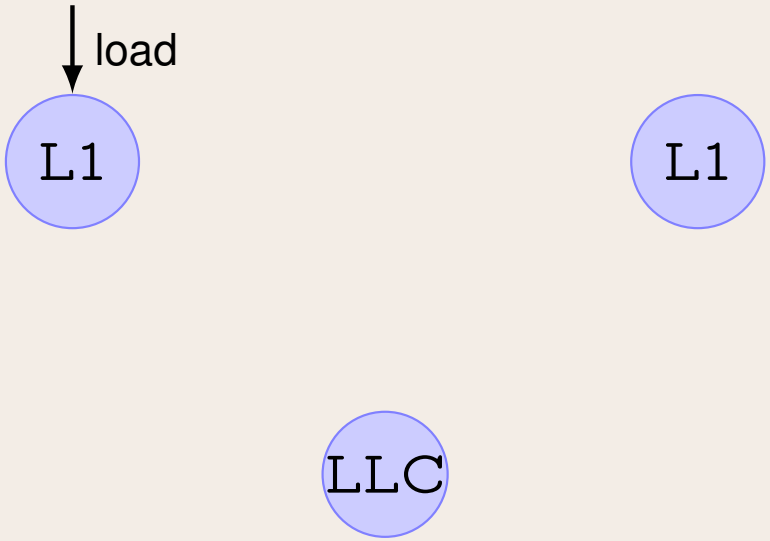
Getting read-only permission

- Load operations require read permission to perform
- If the cache has not read permission ⇒ cache miss
- A read miss coherence transaction is generated

The LLC has the ownership



An L1 has the ownership

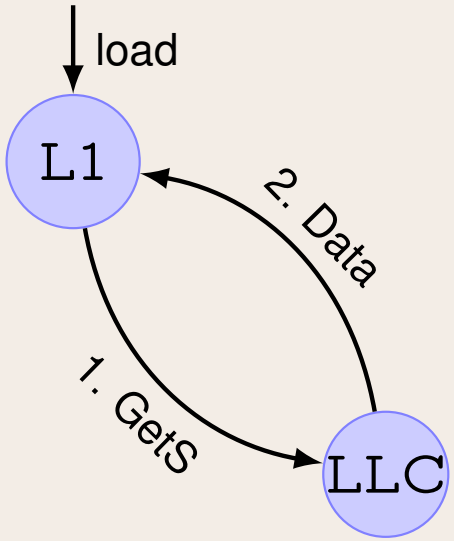


LLC = Last Level Cache

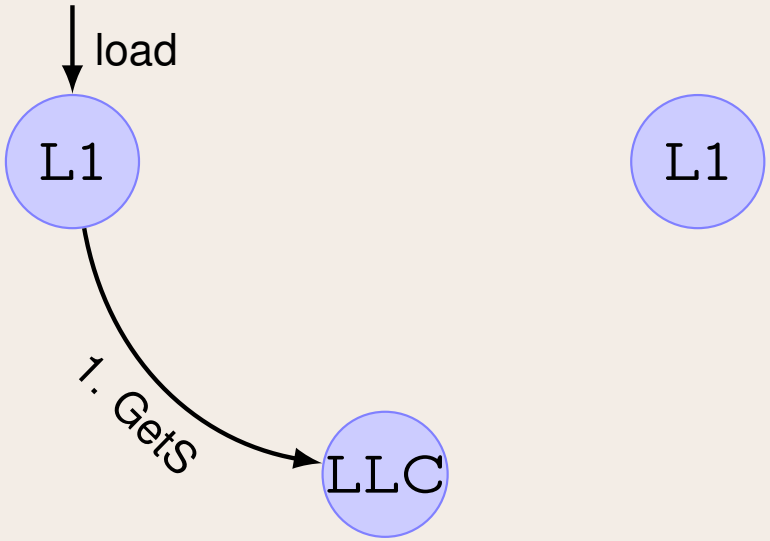
Getting read-only permission

- Load operations require read permission to perform
- If the cache has not read permission ⇒ cache miss
- A read miss coherence transaction is generated

The LLC has the ownership



An L1 has the ownership

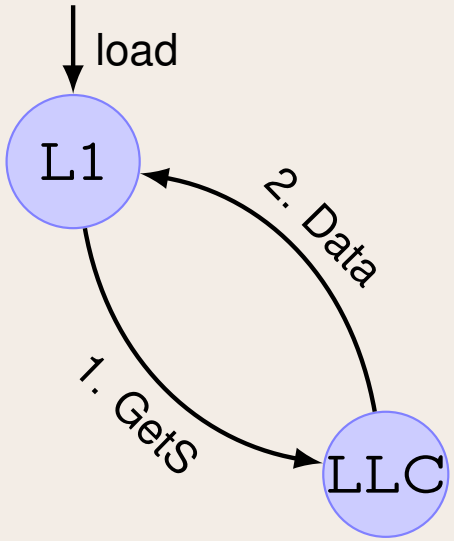


LLC = Last Level Cache

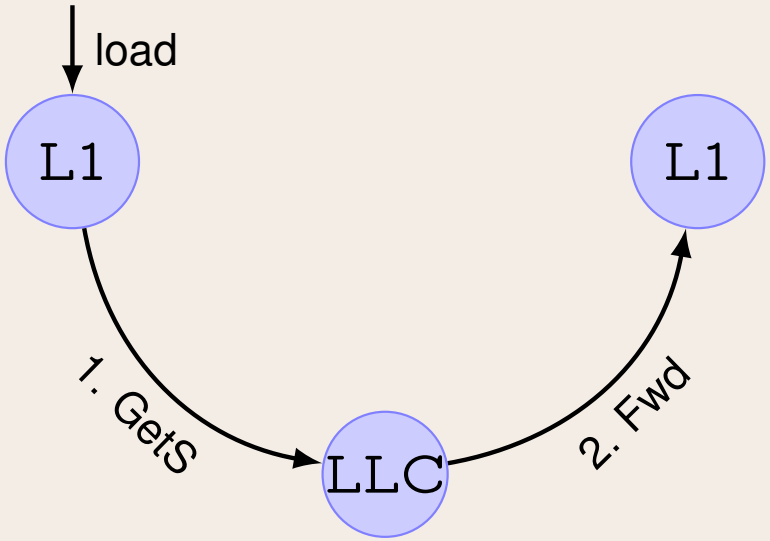
Getting read-only permission

- Load operations require read permission to perform
- If the cache has not read permission ⇒ cache miss
- A read miss coherence transaction is generated

The LLC has the ownership



An L1 has the ownership

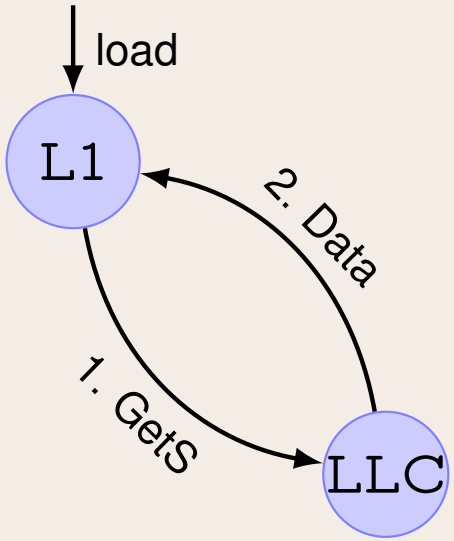


LLC = Last Level Cache

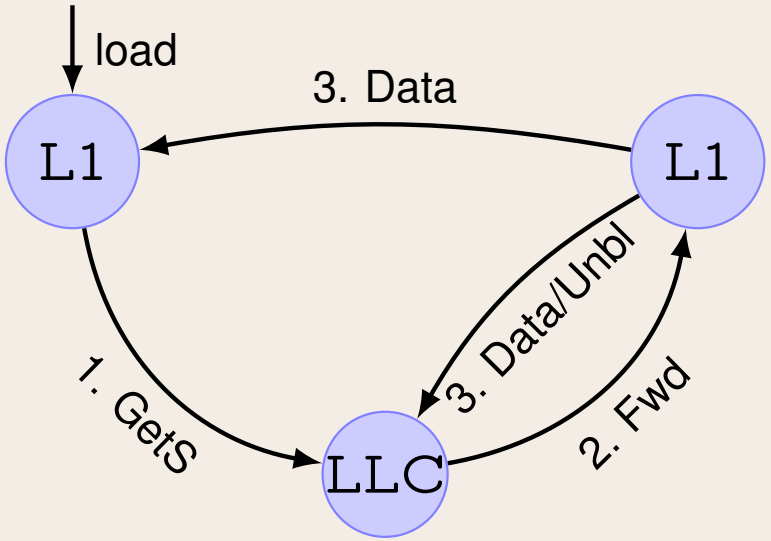
Getting read-only permission

- Load operations require read permission to perform
- If the cache has not read permission ⇒ cache miss
- A read miss coherence transaction is generated

The LLC has the ownership



An L1 has the ownership



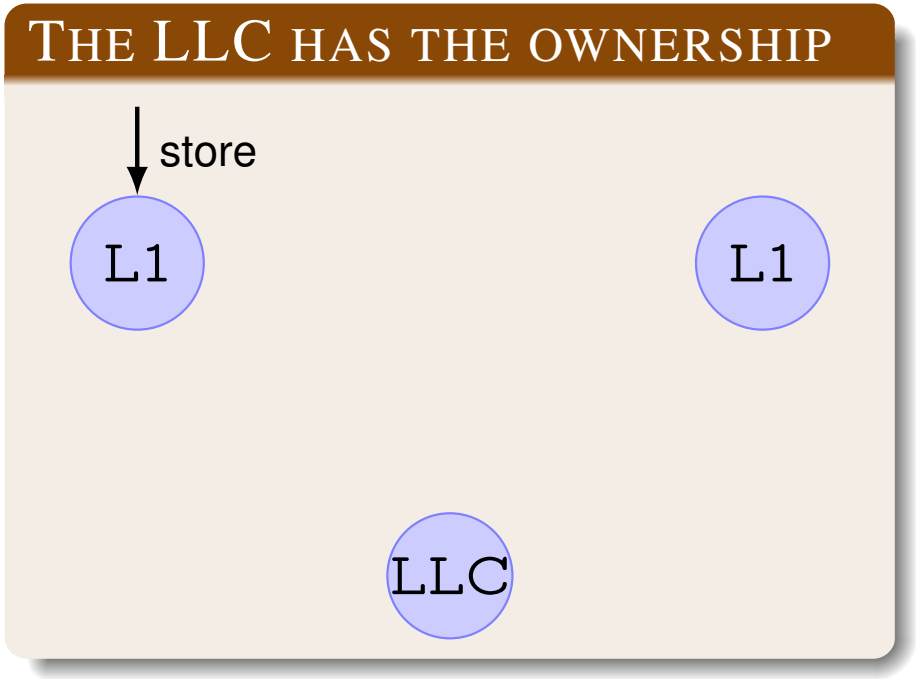
LLC = Last Level Cache

Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission $\Rightarrow$ cache miss
- A write miss coherence transaction is generated

Getting read-write permission

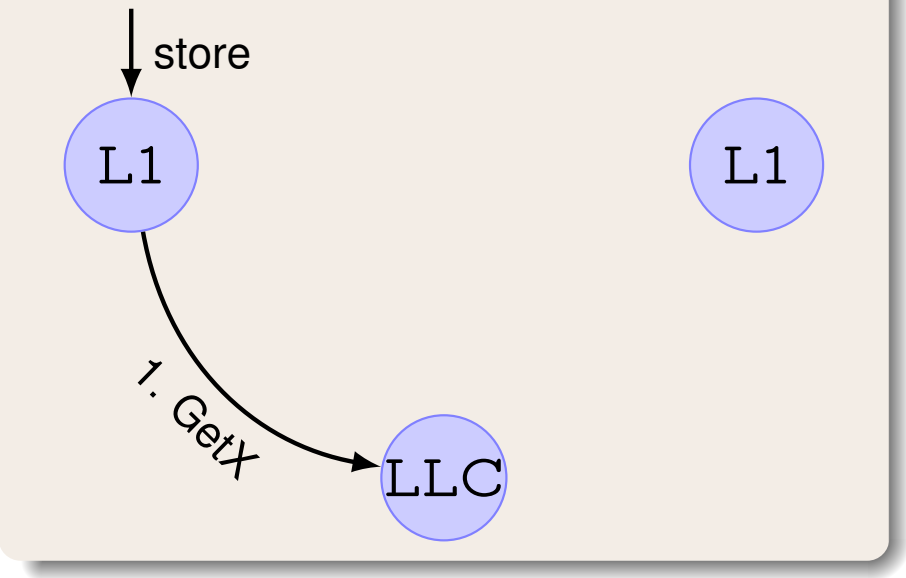
- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated



Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated

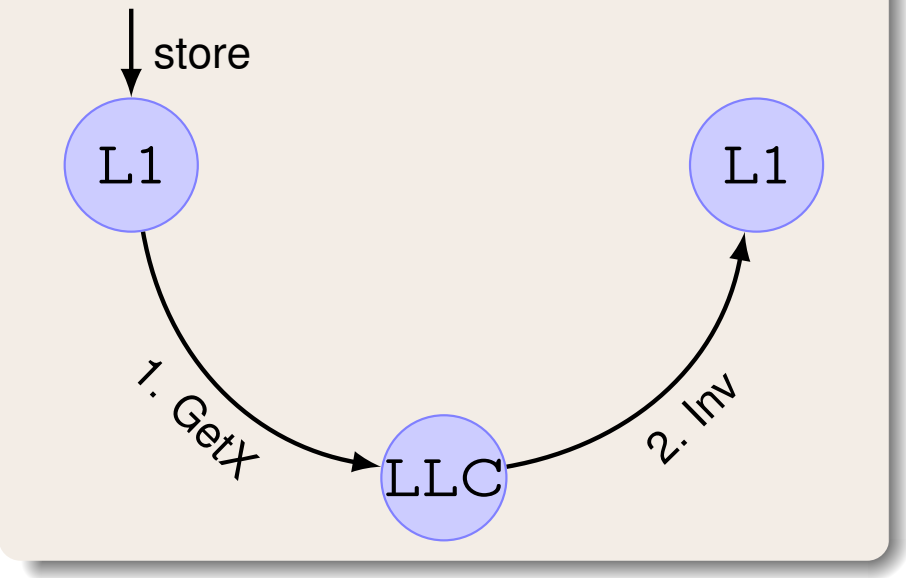
The LLC has the ownership



Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated

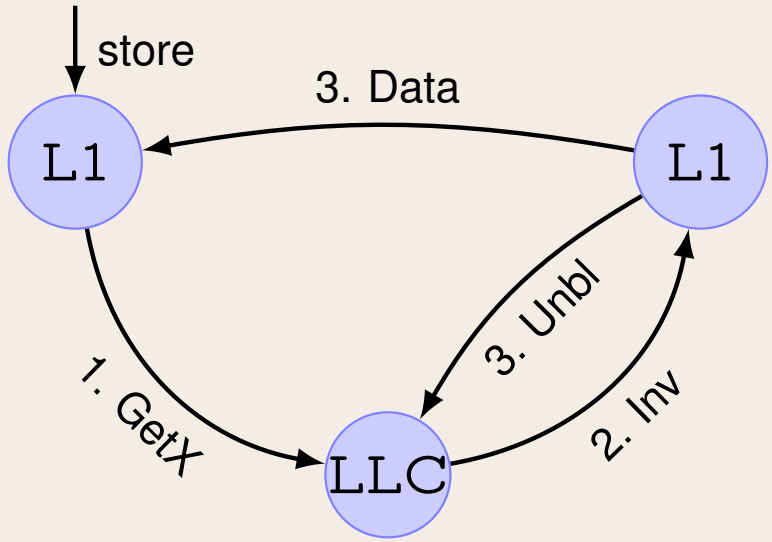
The LLC has the ownership



Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated

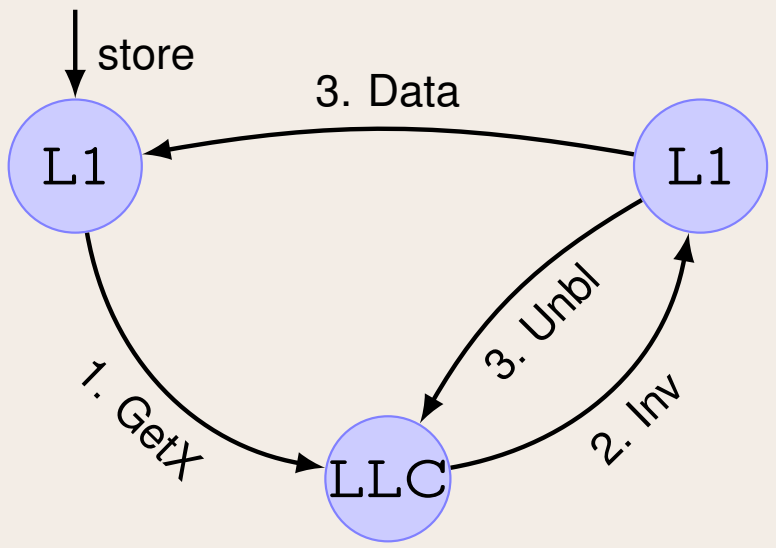
The LLC has the ownership



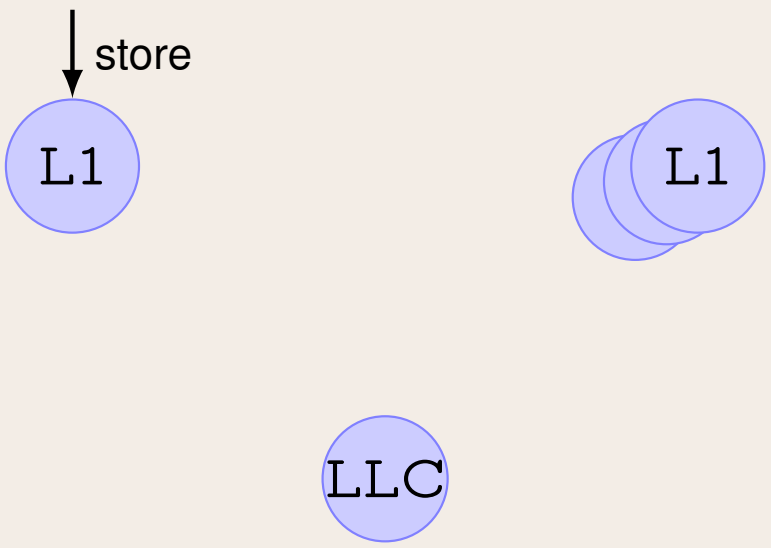
Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated

The LLC has the ownership



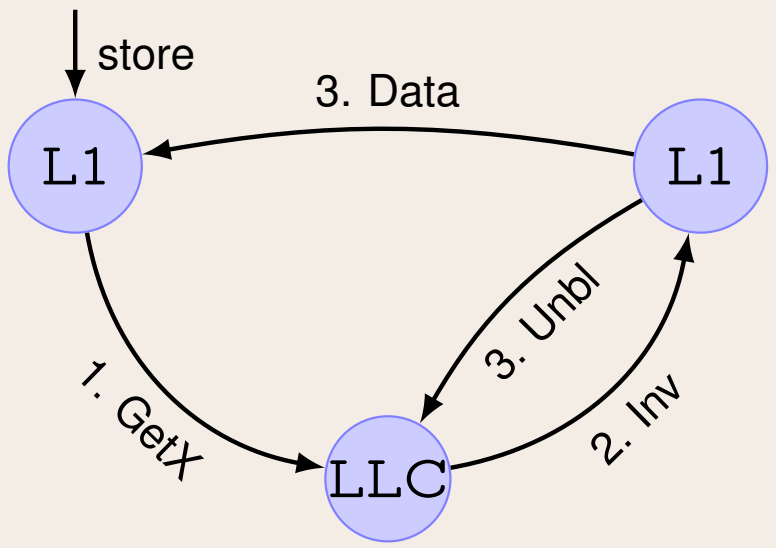
An L1 has the ownership



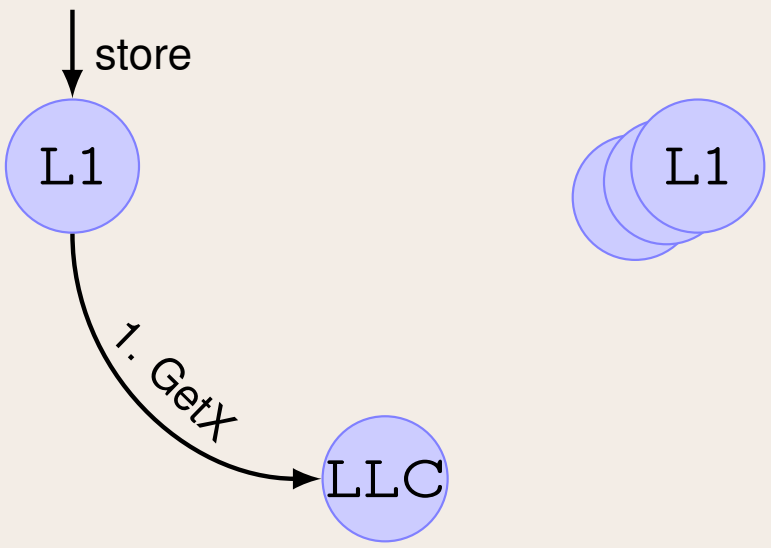
Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated

The LLC has the ownership



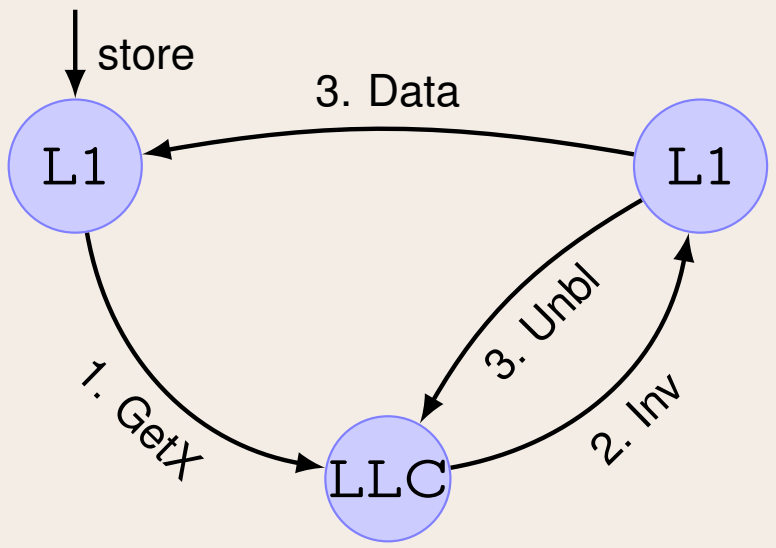
An L1 has the ownership



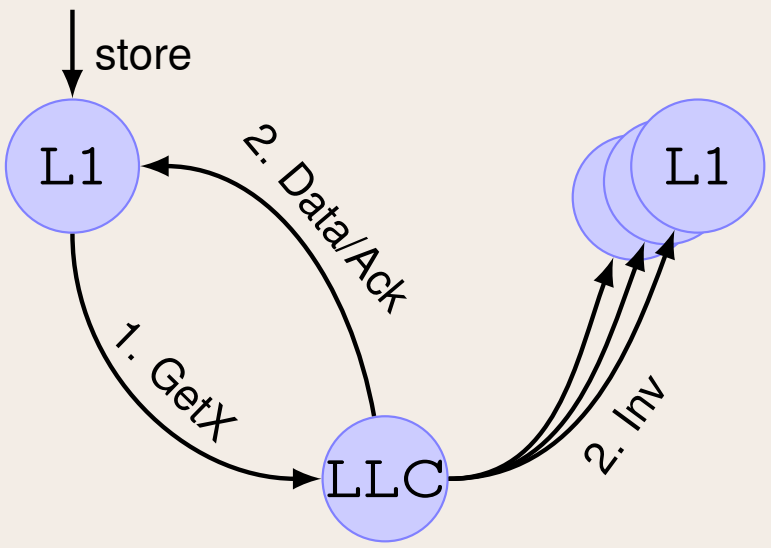
Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated

The LLC has the ownership



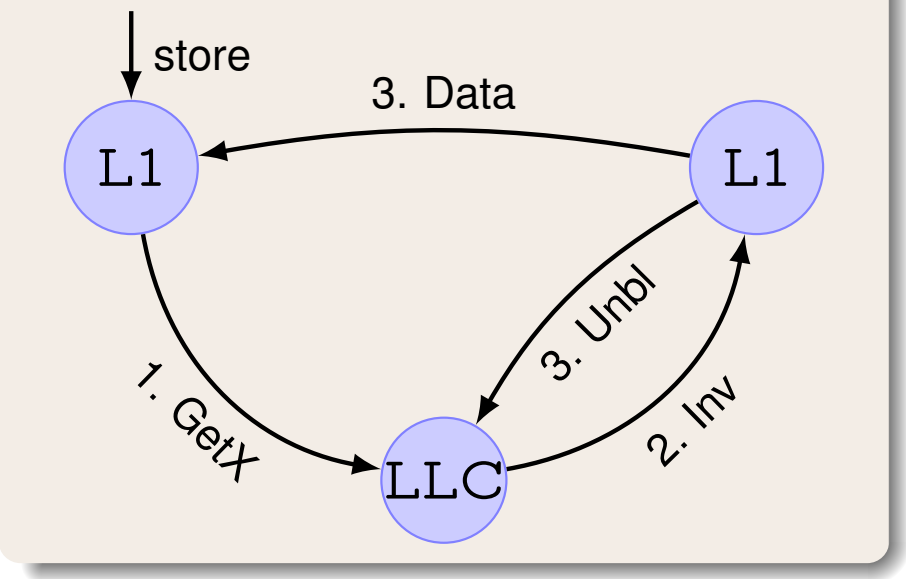
An L1 has the ownership



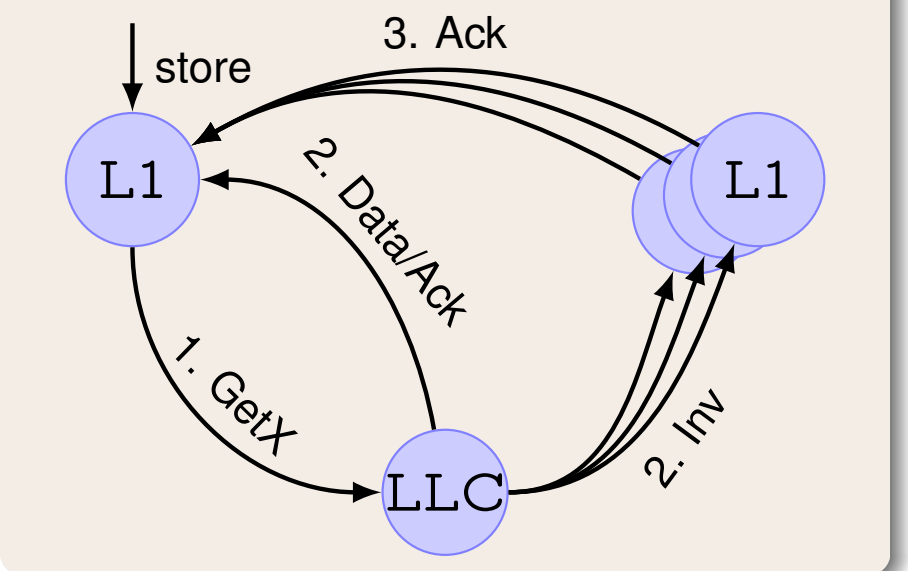
Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated

The LLC has the ownership



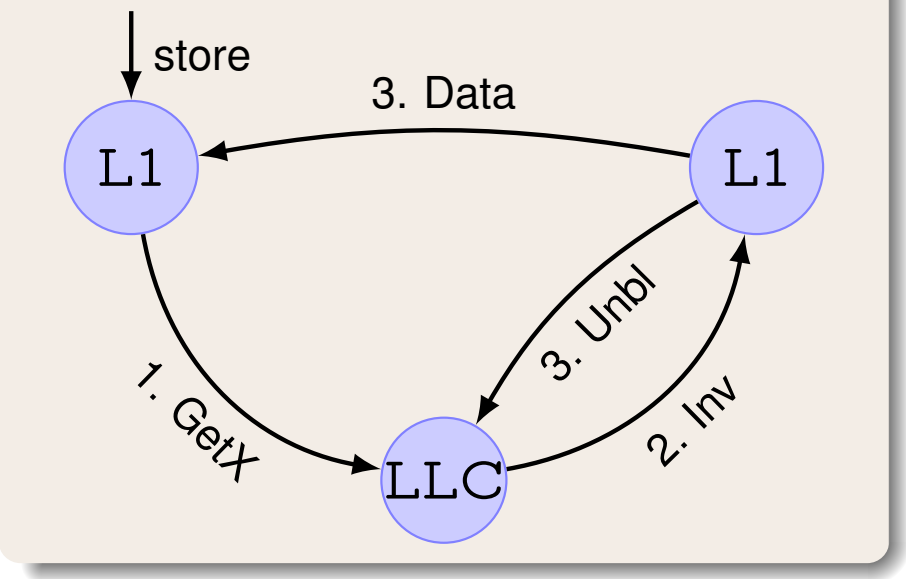
An L1 has the ownership



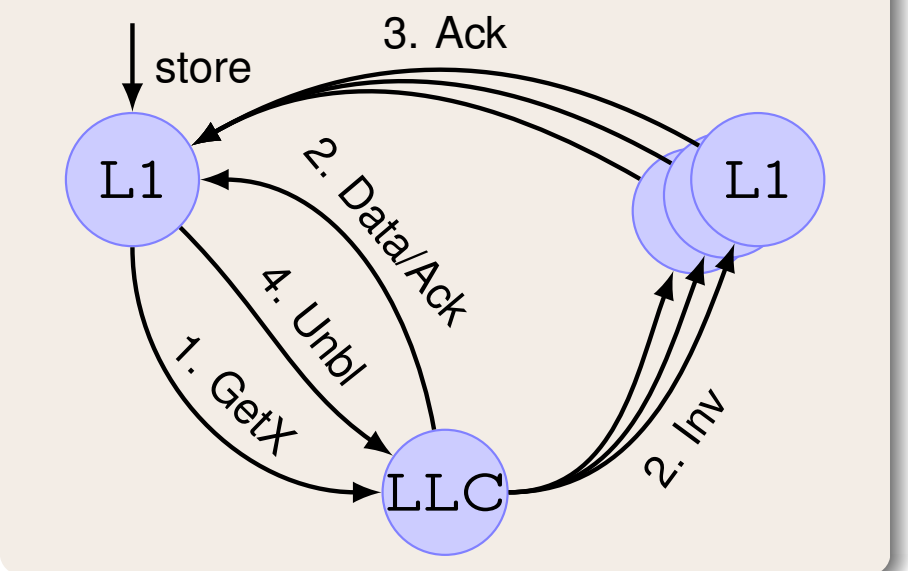
Getting read-write permission

- Store operations require write permission to perform
- If the cache has not write permission ⇒ cache miss
- A write miss coherence transaction is generated

The LLC has the ownership



An L1 has the ownership

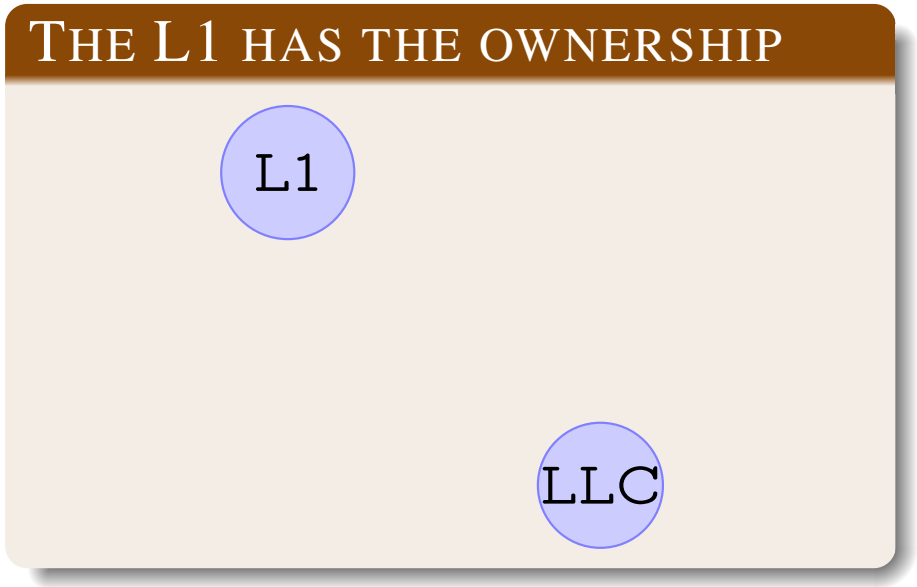


Evicting cache blocks: Silent vs. Noisy

- Noisy cache evictions generate coherence messages
- Noisy cache evictions are mandatory when the block has Dirtiness or Ownership properties
  - To ensure the data value invariant
- They tell the directory not to track the copy anymore

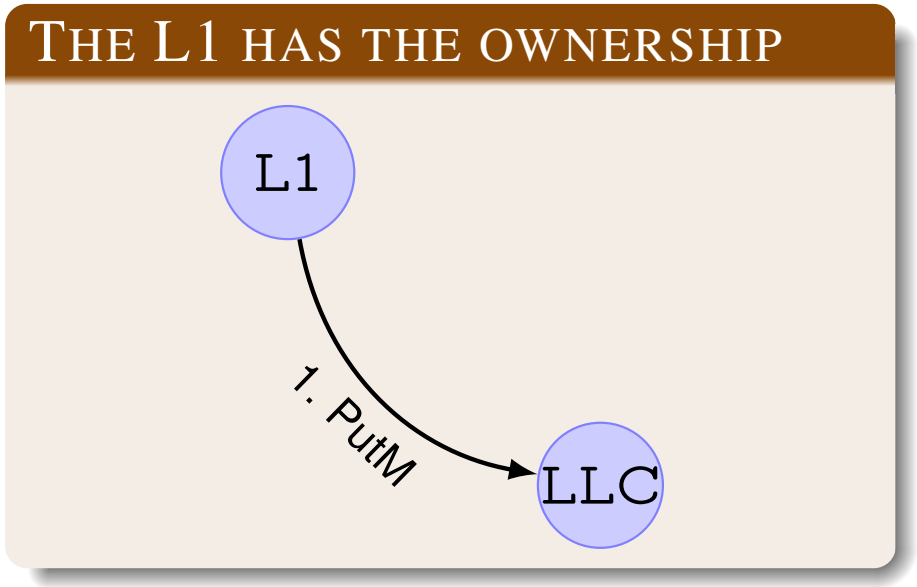
Evicting cache blocks: Silent vs. Noisy

- Noisy cache evictions generate coherence messages
- Noisy cache evictions are mandatory when the block has Dirtiness or Ownership properties
  - To ensure the data value invariant
- They tell the directory not to track the copy anymore



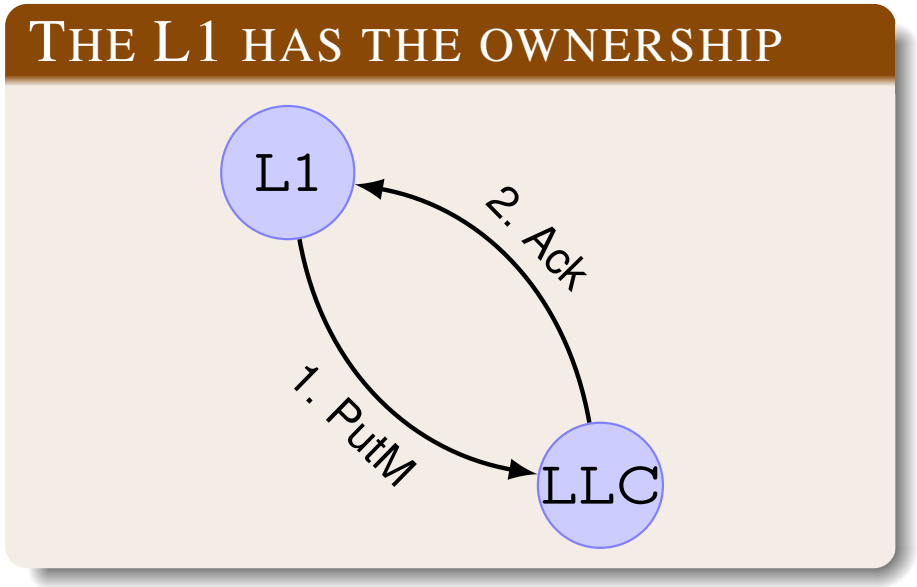
Evicting cache blocks: Silent vs. Noisy

- Noisy cache evictions generate coherence messages
- Noisy cache evictions are mandatory when the block has Dirtiness or Ownership properties
  - To ensure the data value invariant
- They tell the directory not to track the copy anymore



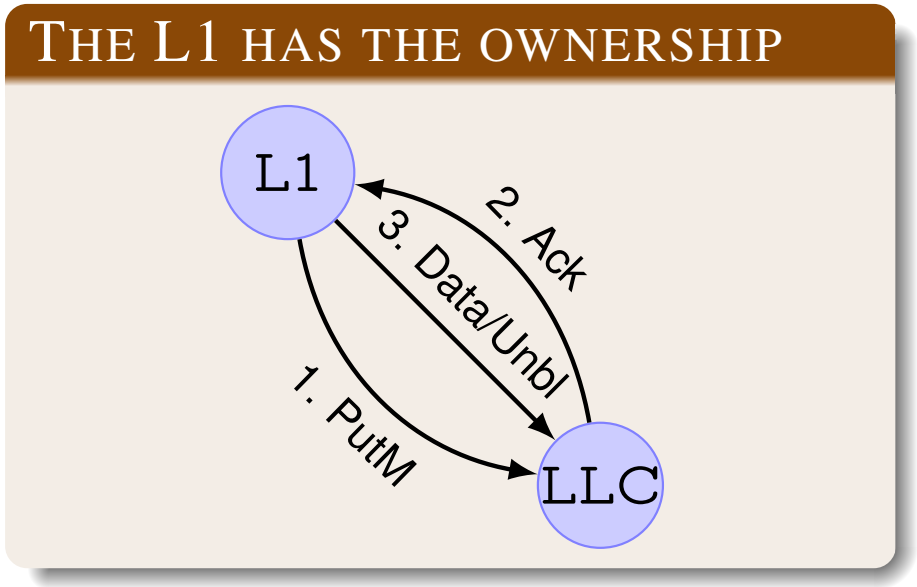
Evicting cache blocks: Silent vs. Noisy

- Noisy cache evictions generate coherence messages
- Noisy cache evictions are mandatory when the block has Dirtiness or Ownership properties
  - To ensure the data value invariant
- They tell the directory not to track the copy anymore



Evicting cache blocks: Silent vs. Noisy

- Noisy cache evictions generate coherence messages
- Noisy cache evictions are mandatory when the block has Dirtiness or Ownership properties
  - To ensure the data value invariant
- They tell the directory not to track the copy anymore



Evicting cache blocks: Silent vs. Noisy

- Silent cache evictions do not generate coherence messages
- Silent cache evictions are possible when the block has not Dirtiness or Ownership properties
- Directory information is not updated
  - The directory will eventually send an invalidation

Evicting cache blocks: Silent vs. Noisy

- Replacement of clean and no-owner blocks (e.g. S) can be implemented either using silent or noisy evictions[2]
- Advantages of silent evictions
  - Less traffic on load-evict-load cases (no writes involved)
- Advantages of noisy evictions
  - Faster write misses (less invalidations)
  - Updated directory
    - Less directory space
    - GetS sends exclusive copy if no sharers
- But there are also more implications

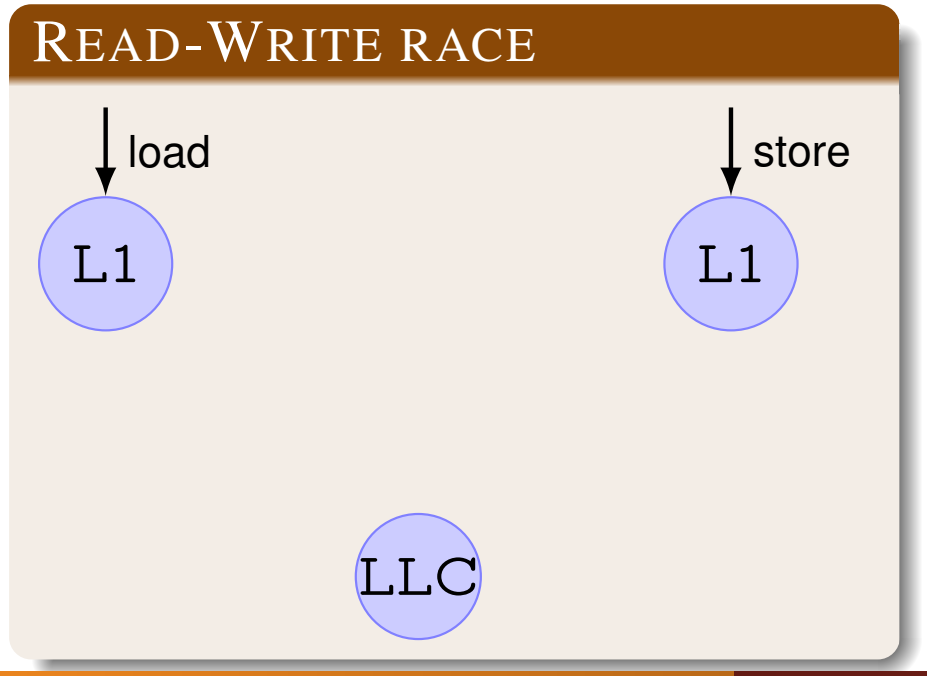
[2] R. Fernandez-Pascual, A. Ros, and M. E. Acacio, "To Be Silent or Not: On the Impact of Evictions of Clean Data in Cache-Coherent Multicores", Journal of Supercomputing, 2017.

Dealing with protocol races

- Simultaneous read and write misses.

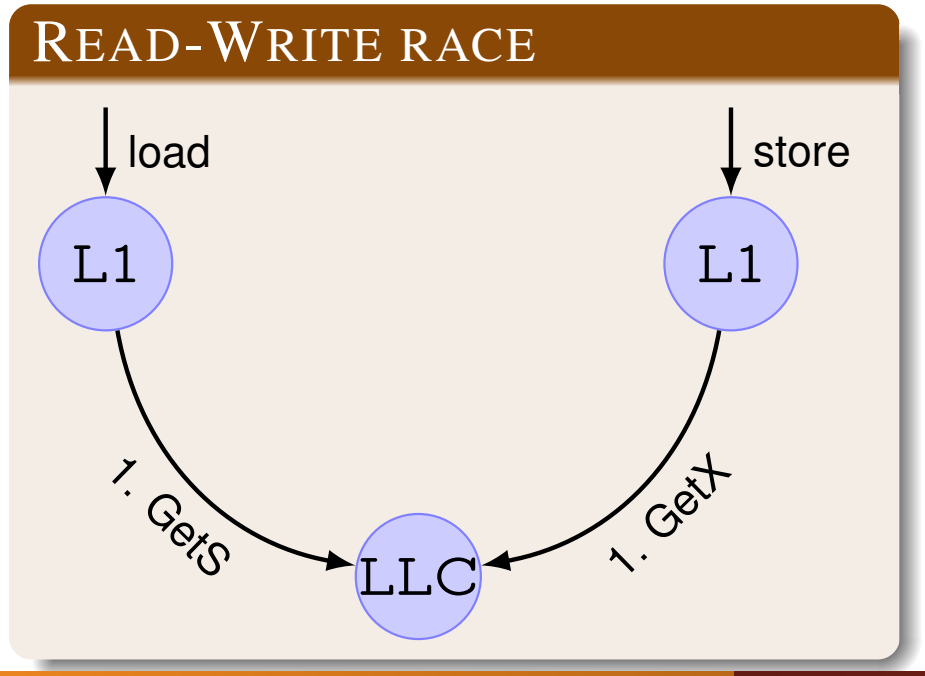
Dealing with protocol races

- Simultaneous read and write misses.



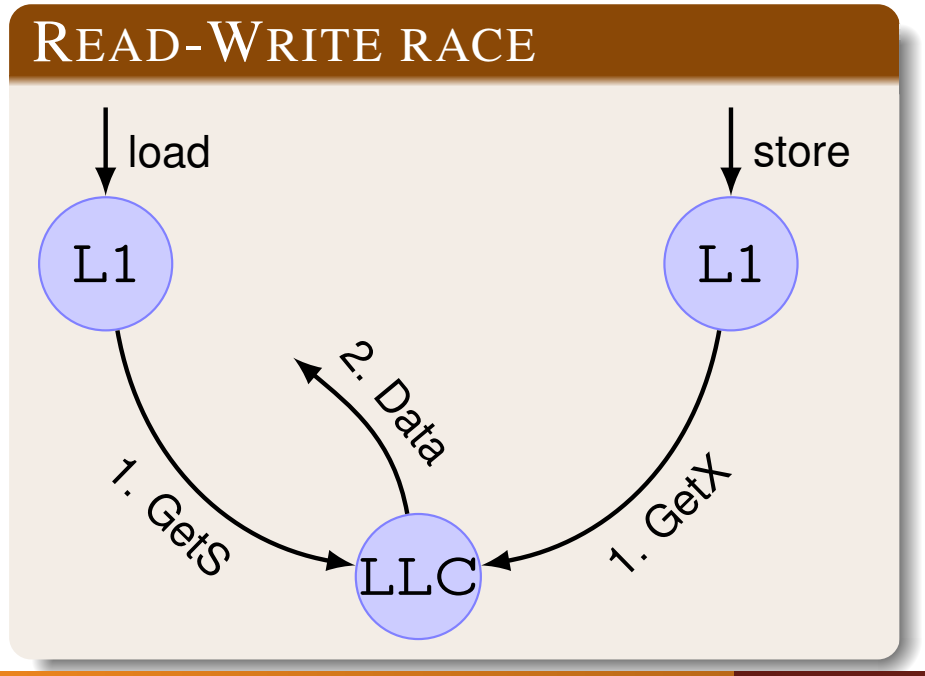
Dealing with protocol races

- Simultaneous read and write misses.



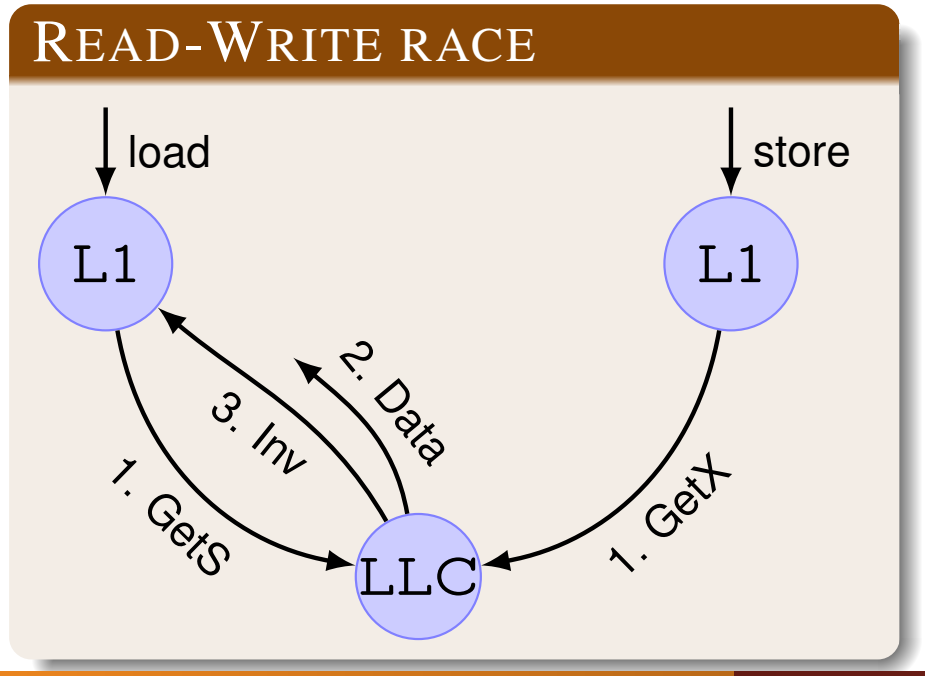
Dealing with protocol races

- Simultaneous read and write misses.



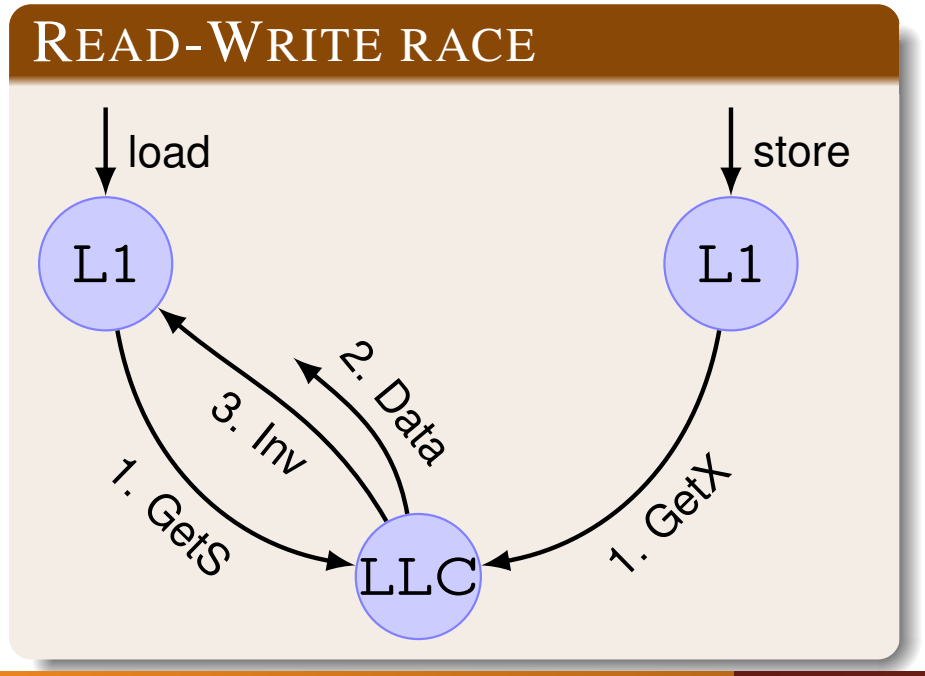
Dealing with protocol races

- Simultaneous read and write misses.
- How can the core (load) know if the read happened before or after the write, in case of silent evictions?



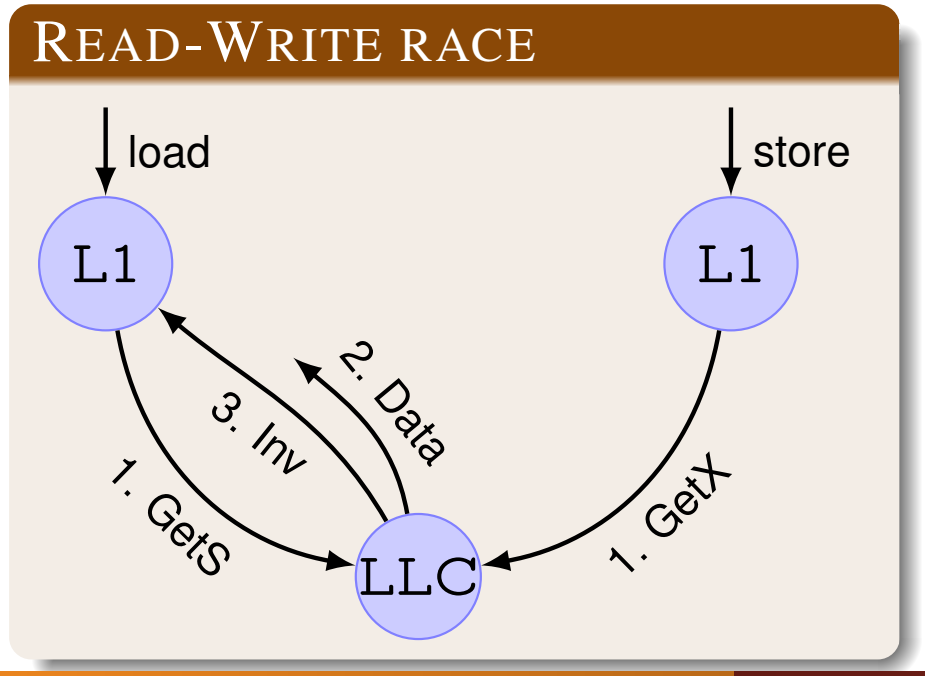
Dealing with protocol races

- Simultaneous read and write misses.
- How can the core (load) know if the read happened before or after the write, in case of silent evictions?
  - It cannot ... but it should
  - Conflict order: the order in which load/store operations perform



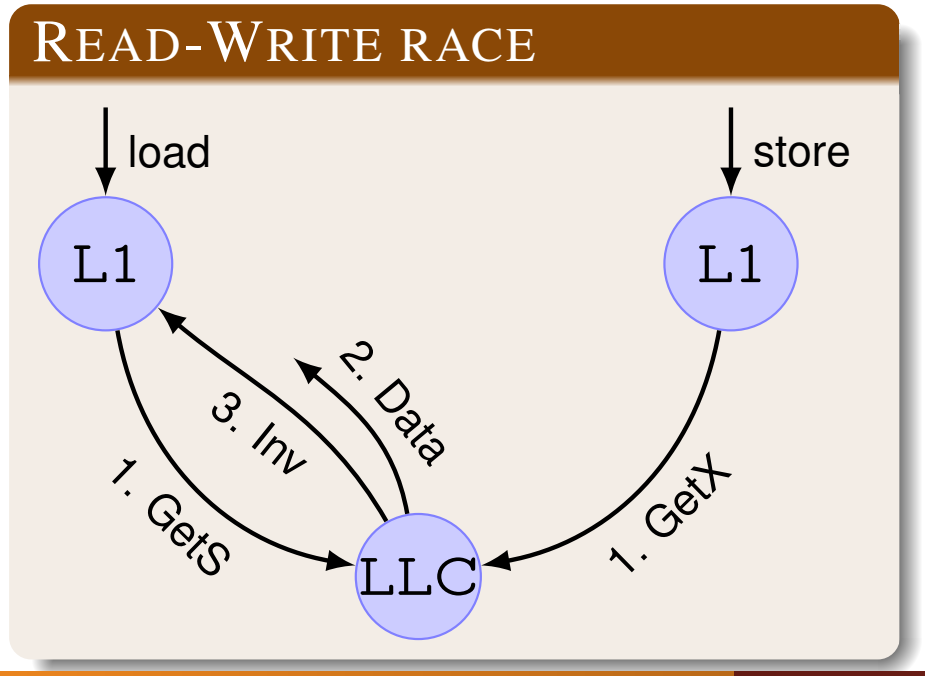
Dealing with protocol races

- Simultaneous read and write misses.
- How can the core (load) know if the read happened before or after the write, in case of silent evictions?
  - It cannot ... but it should
  - Conflict order: the order in which load/store operations perform
- Solution 1: Invalidate the data when it comes



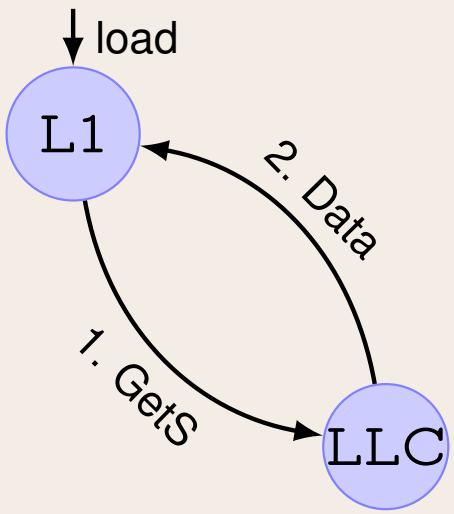
Dealing with protocol races

- Simultaneous read and write misses.
- How can the core (load) know if the read happened before or after the write, in case of silent evictions?
  - It cannot ... but it should
  - Conflict order: the order in which load/store operations perform
- Solution 1: Invalidate the data when it comes
- Solution 2: Unblock the directory after the load performs

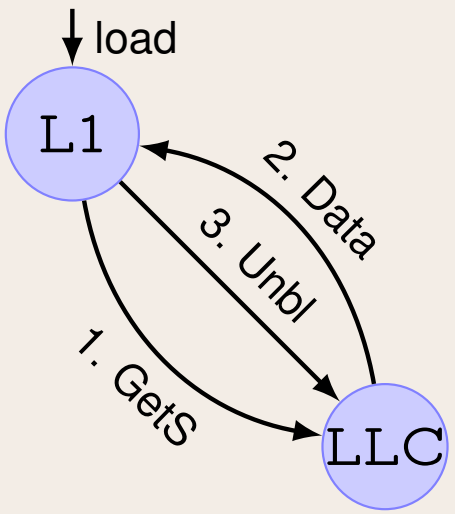


Dealing with protocol races

Early unblock



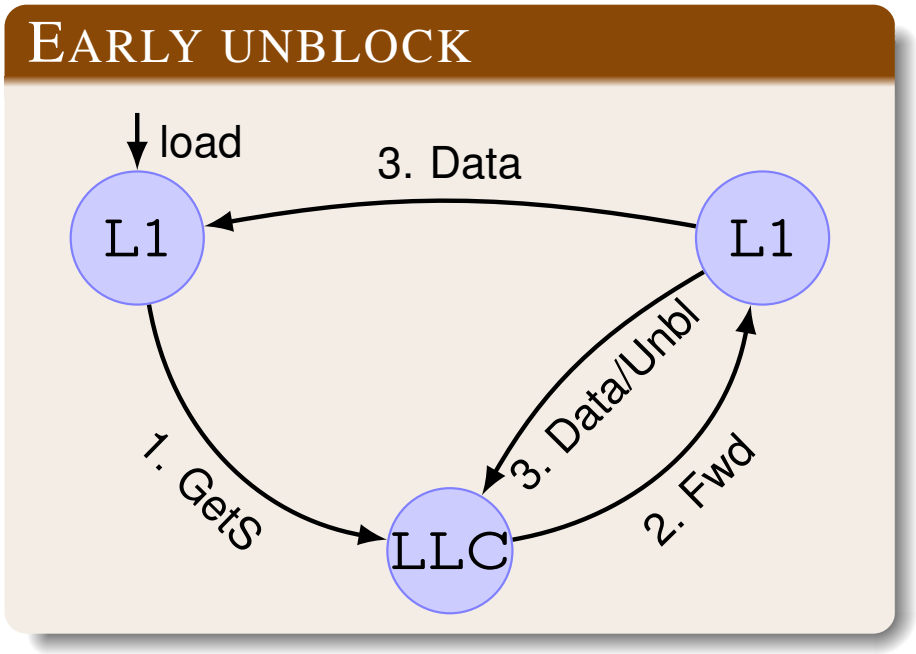
Late unblock

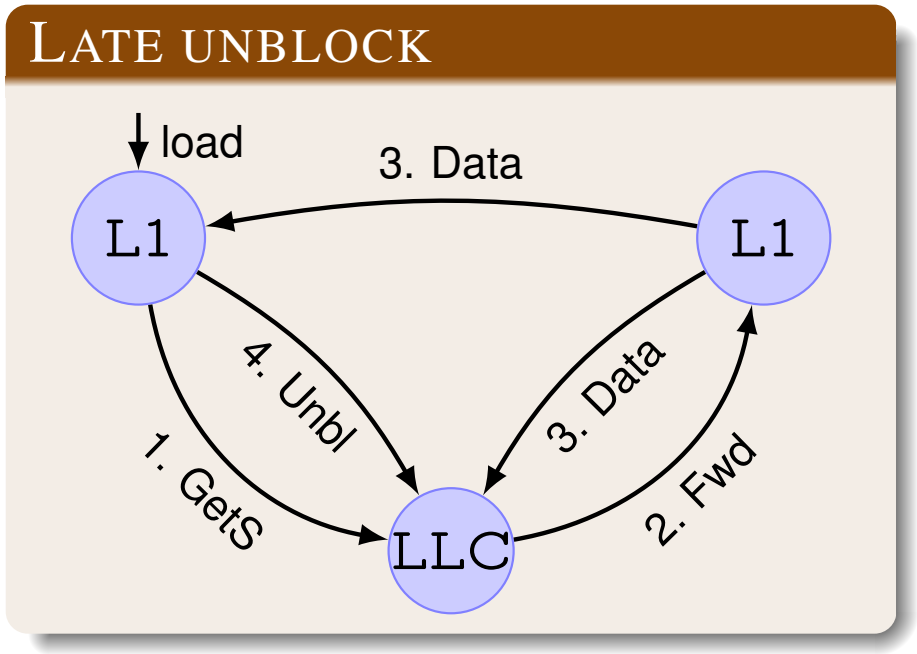


- Early unblock
  - cannot infer ordering on races
  - requires conservative invalidation to guarantee conflict order
- Late unblock
  - can infer ordering on races
  - blocks the directory for longer
    - but read misses can be processed in parallel

Dealing with protocol races

- Early vs. Late unblock when the owner of the block is an L1

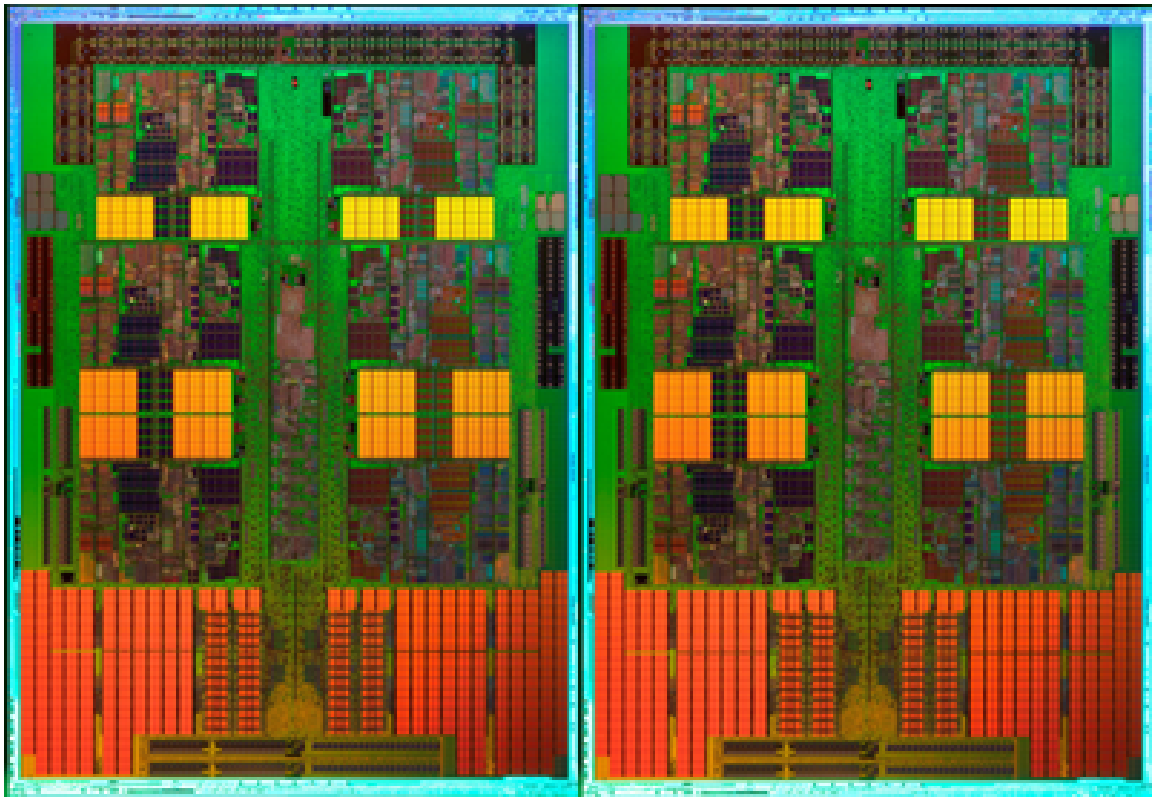




- This decision also affects the implementation of the consistency model

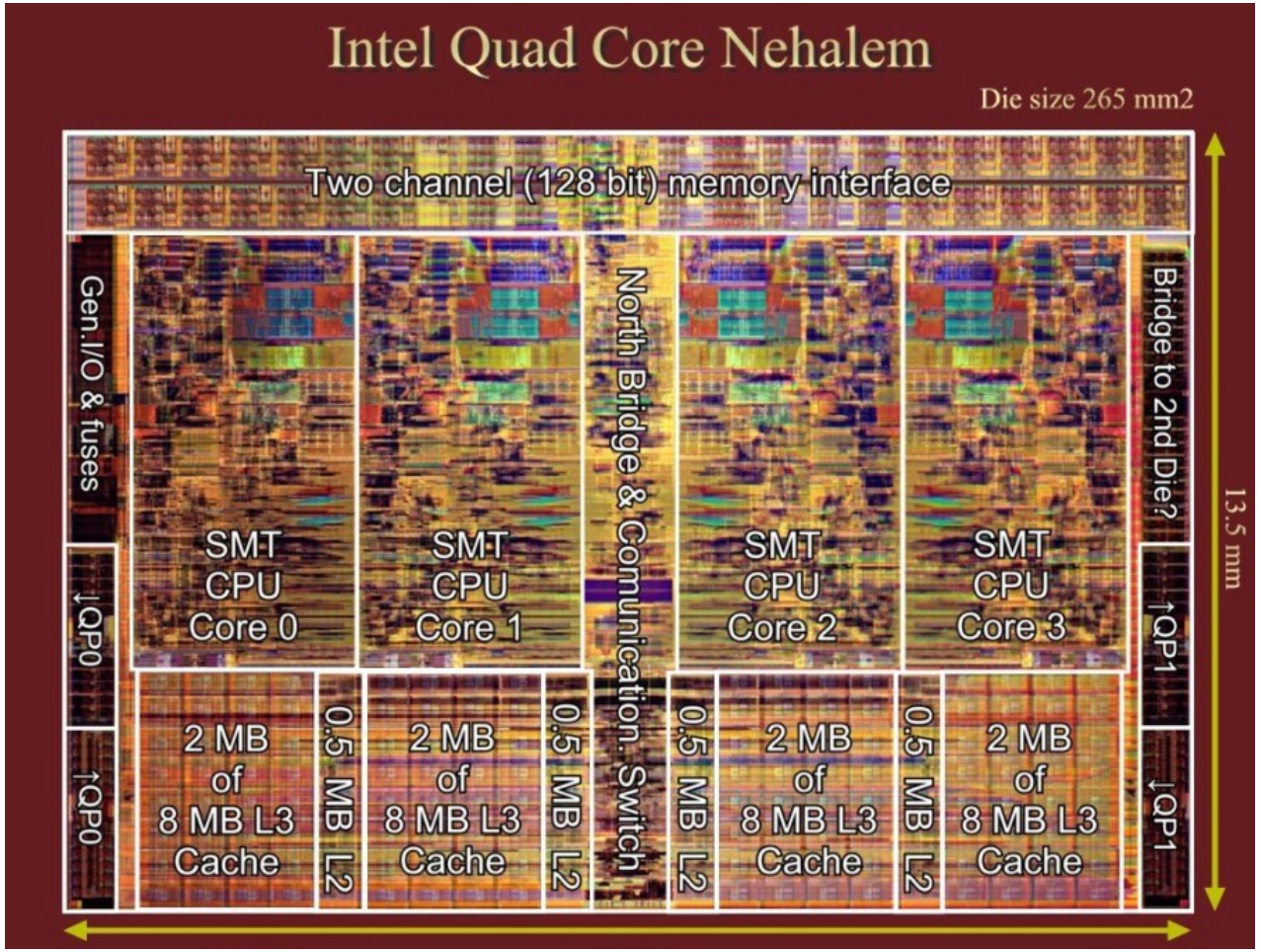
AMD cHT: coherent HyperTransport

- 12-core Magny Cours chip
- MOESI + S1 (Single sharer)
- S1: Validity, Exclusivity



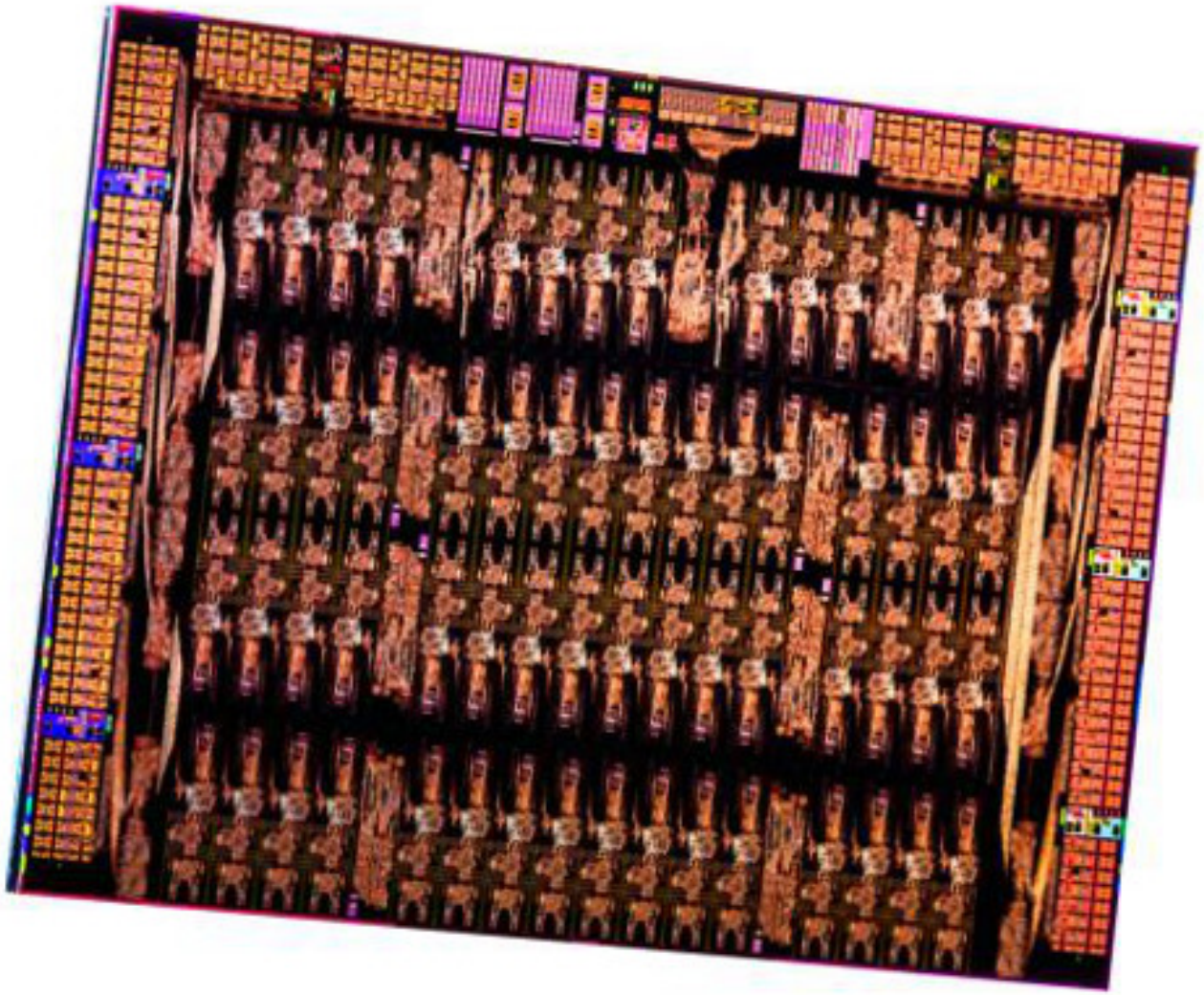
Intel QPI: Quick-Path Interconnect

- MESI + F (MESIF)
- F: Validity, Ownership



INTEL MIC: 60-CORE CO-PROCESSOR

- MESI



Recap

- The cache coherence protocol
  - hides the caches
  - works at cache-line granularity
  - provides conflict order (between two operations to the same memory block being at least one of them a store)
  - cannot provide order between different memory blocks
- The memory consistency model implementation relies on the cache coherence protocol guarantees

Outline

1. Cache coherence
2. Memory consistency
3. Coherence-Consistency Interaction
4. Open research questions

Dekker's algorithm

Dekker's algorithm

```
/* Initially X = Y = 0 */

X = 1;          |   Y = 1;
$r0 = Y;        |   $r1 = X;
```

- Can we get $r0==0, $r1==0?

Dekker's algorithm

Dekker's algorithm

```
/* Initially X = Y = 0 */

X = 1;          |   Y = 1;
$r0 = Y;        |   $r1 = X;
```

- Can we get $r0==0, $r1==0?
- Only if in a thread the load performs before the store in the same thread

Memory Consistency Model

- To argue about the correctness of a program, it is necessary to define a memory consistency model (or consistency model)
- The consistency model is the contract between the programmer and the system
  - The programmer knows which results he can expect
  - The system know how much the program can be optimized

Memory Consistency Model

- To argue about the correctness of a program, it is necessary to define a memory consistency model (or consistency model)
- The consistency model is the contract between the programmer and the system
  - The programmer knows which results he can expect
  - The system know how much the program can be optimized

Definition

The consistency model is an specification of the behaviour of multithreaded programs executing under shared memory

Memory Consistency Model

- To argue about the correctness of a program, it is necessary to define a memory consistency model (or consistency model)
- The consistency model is the contract between the programmer and the system
  - The programmer knows which results he can expect
  - The system know how much the program can be optimized

Definition

The consistency model is an specification of the behaviour of multithreaded programs executing under shared memory

- In particular, the consistency model specifies the values returned by every load and the final status of the memory when the program finishes executing

Consistency models

- There are a large number of consistency models implemented in multicores or proposed in the literature
  - Sequential Consistency (SC)
  - Total Store Order (TSO)
  - Processor Consistency (PC)
  - ARM consistency model
  - Power consistency model
  - Weak Consistency (WC)
  - Release Consistency (RC)
  - Scope Consistency (ScC)
  - Entry Consistency
- We will review some of the most popular

Sequential consistency (SC)

Formalized by Lamport[1]

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

[3] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", IEEE Transactions on Computers, 1979.

Sequential consistency (SC)

Formalized by Lamport[1]

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

Possible interleavings?

```
X = 1;
$r1 = Y;
```

```
Y = 1;
$r0 = X;
```

[3] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", IEEE Transactions on Computers, 1979.

Sequential consistency (SC)

Possible interleavings?

sx: X = 1;	sy: Y = 1;
ly: $r1 = Y;	lx: $r0 = X;

Sequential consistency (SC)

Possible interleavings?

```
sx: X = 1;
ly: $r1 = Y;
```

```
sy: Y = 1;
lx: $r0 = X;
```

Six possible interleavings for SC

sx	sx	sx	sy	sy	sy
ly	sy	sy	sx	sx	lx
sy	ly	lx	ly	lx	sx
lx	lx	ly	lx	ly	ly
(1,0)	(1,1)	(1,1)	(1,1)	(1,1)	(0,1)

- Effectively, (0,0) is not possible under SC

SEQUENTIAL CONSISTENCY (SC)

- Being <p the order in which the operations appear in the program
- Being <m the order in which the operations read/write from/to memory
  - If L(a) <p L(b) $\Rightarrow$ L(a) <m L(b) /* Load$\rightarrow$Load */
  - If L(a) <p S(b) $\Rightarrow$ L(a) <m S(b) /* Load$\rightarrow$Store */
  - If S(a) <p S(b) $\Rightarrow$ S(a) <m S(b) /* Store$\rightarrow$Store */
  - If S(a) <p L(b) $\Rightarrow$ S(a) <m L(b) /* Store$\rightarrow$Load */
  - The same rules apply to atomic operations (RMW)

Sequential consistency (SC)

- Being <p the order in which the operations appear in the program
- Being <m the order in which the operations read/write from/to memory
  - If L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
  - If L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
  - If S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
  - If S(a) <p L(b) ⇒ S(a) <m L(b) /* Store→Load */
  - The same rules apply to atomic operations (RMW)

Ordering rules for SC

		Op2		
		Load	Store	RMW
Op1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

Sequential consistency (SC)

- Being <p the order in which the operations appear in the program
- Being <m the order in which the operations read/write from/to memory
  - If L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
  - If L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
  - If S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
  - If S(a) <p L(b) ⇒ S(a) <m L(b) /* Store→Load */
  - The same rules apply to atomic operations (RMW)

Ordering rules for SC

		Op2		
		Load	Store	RMW
Op1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

- The MIPS R10000 processor provides SC

Motivation Total Store Order (TSO)

Dekker's algorithm

```
X = 1;
$r1 = Y;
```

```
Y = 1;
$r0 = X;
```

- Which results can we expect in an Intel or AMD processor?

Motivation Total Store Order (TSO)

Dekker's algorithm

```
X = 1;
$r1 = Y;
```

```
Y = 1;
$r0 = X;
```

- Which results can we expect in an Intel or AMD processor?
- We can get (0,0)! Why?

Motivation Total Store Order (TSO)

Dekker's algorithm

`X = 1;`	`Y = 1;`
`$r1 = Y;`	`$r0 = X;`

- Which results can we expect in an Intel or AMD processor?
- We can get (0,0)! Why?
  - The processor may reorder the operations to improve performance
  - The processor cannot be blocked on store operations
  - Intel or AMD implement a Store Buffer (SB)

The Store Buffer (SB)

- A store operation requires write permission to perform
- Write permission may require invalidating other copies
- It is a long-latency operation
- Solution
  1. Move the data to the store buffer, delaying the write to cache
  2. Process the next intructions
  3. When the core has write permission, perform the write
- How does this code executes in a multicore with SB?
  - ⇒ We can get (0,0)

Total Store Order (TSO)

- If L(a) <p L(b) $\Rightarrow$ L(a) <m L(b) /* Load$\rightarrow$Load */
- If L(a) <p S(b) $\Rightarrow$ L(a) <m S(b) /* Load$\rightarrow$Store */
- If S(a) <p S(b) $\Rightarrow$ S(a) <m S(b) /* Store$\rightarrow$Store */
- /* No Store$\rightarrow$Load! */
- Atomic operations (RMW) are ordered with respect to any other memory operation

Total Store Order (TSO)

- If L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
- If L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
- If S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
- /* No Store→Load! */
- Atomic operations (RMW) are ordered with respect to any other memory operation

Ordering rules for TSO

		Op2		
		Load	Store	RMW
Op1	Load	X	X	X
	Store	**B**	X	X
	RMW	X	X	X

- **B** means that a load finding a matching store in the SB takes the value from the SB, not from memory

Fences

- A fence is used to avoid allowed reorderings
- Two memory operations, having a fence in between them, cannot be reordered

Fences

- A fence is used to avoid allowed reorderings
- Two memory operations, having a fence in between them, cannot be reordered

Ordering rules for TSO with fences

		Op2			
		Load	Store	RMW	Fence
Op1	Load	X	X	X	**X**
	Store	**B**	X	X	**X**
	RMW	X	X	X	**X**
	Fence	**X**	**X**	**X**	**X**

Fences

- If we insert fences in between all memory operations, the result is SC, no matter the consistency model

Fenced code

```
X = 1;
Fence;
$r1 = Y;
```

```
Y = 1;
Fence;
$r0 = X;
```

Why TSO?

- Is TSO a good consistency model if it allows unexpected results?

Why TSO?

- Is TSO a good consistency model if it allows unexpected results?
- Yes, it work for most of the codes
- For example:

Synchronization with flags

```
/* Initially X = flag = 0 */

X = 1;          |   while (flag == 0);
flag = 1;       |   $r0 = X;
```

Motivation Weak Consistency (WC)[2]

Synchronization with flags

```
/* Initially X = Y = flag = 0 */

X = 1;
Y = 2;
flag = 1;

while (flag == 0);
$r1 = X;
$r2 = Y;
```

- Do we need to keep the order of the two load or of the two writes to X and Y?

[4] M. Dubois *et al.*, "Memory Access Buffering in Multiprocessors", ISCA, 1986.

Motivation Weak Consistency (WC)[2]

Synchronization with flags

```
/* Initially X = Y = flag = 0 */

X = 1;                  while (flag == 0);
Y = 2;                  $r1 = X;
flag = 1;               $r2 = Y;
```

- Do we need to keep the order of the two load or of the two writes to X and Y?
- No, it does not matter the order they execute
- We only need to keep the order when synchronizing the threads

[4] M. Dubois *et al.*, "Memory Access Buffering in Multiprocessors", ISCA, 1986.

Weak Consistency (WC)

- Synchronization operations act as fences
- Sequential Consistency for Data-Race-Free programs (SC for DRF)[3]
- Synchronization must be exposed to the hardware

[5] S. V. Adve and M. D. Hill. "Weak Ordering–A New Definition", ISCA, 1990

Weak Consistency (WC)

- Synchronization operations act as fences
- Sequential Consistency for Data-Race-Free programs (SC for DRF)[3]
- Synchronization must be exposed to the hardware

Ordering rules for WC

		Op2			
		Load	Store	RMW	**Sync**
Op1	Load	**A**	**A**	**A**	X
	Store	B	**A**	**A**	X
	RMW	**A**	**A**	**A**	X
	Sync	X	X	X	X

[5] S. V. Adve and M. D. Hill. "Weak Ordering–A New Definition", ISCA, 1990

Motivation Release Consistency (RC)[4]

Synchronization with flags

```
/* Initially X = Y = flag = 0 */

X = 1;
Y = 2;
flag = 1;

while (flag == 0);
$r1 = X;
$r2 = Y;
```

- Do we always need to keep the order across all synchronization operations?

[6] K. Gharachorloo *et al.*, "Memory Consistency and Event Ordering in Scalable Shared-Memory", ISCA, 1990

Motivation Release Consistency (RC)[4]

Synchronization with flags

```
/* Initially X = Y = flag = 0 */

X = 1;
Y = 2;
flag = 1;

while (flag == 0);
$r1 = X;
$r2 = Y;
```

- Do we always need to keep the order across all synchronization operations?
- No, not for all.
- We can define two kings for synchronization operations: Acquire and Release
  - Flag = 1 is an operation with Release semantics and the while loop has Acquire semantics

[6] K. Gharachorloo *et al.*, "Memory Consistency and Event Ordering in Scalable Shared-Memory", ISCA, 1990

Release Consistency (RC)

- It only ensures the orders ACQ→Load,Store & Load,Store→REL
- Acquire and Release synchronization are exposed to the hardware
- Sequential Consistency for Data-Race-Free programs (SC for DRF)

Release Consistency (RC)

- It only ensures the orders ACQ→Load,Store & Load,Store→REL
- Acquire and Release synchronization are exposed to the hardware
- Sequential Consistency for Data-Race-Free programs (SC for DRF)

Ordering rules for RC

		Op2				
		Load	Store	RMW	**ACQ**	**REL**
Op1	Load	A	A	A	**A**	X
	Store	B	A	A	**A**	X
	RMW	A	A	A	**A**	X
	ACQ	X	X	X	X	X
	REL	**A**	**A**	**A**	X	X

Consistency vs. Coherence

- Consistency and coherence are different
- Coherence
  - provides conflict order to memory operations to the same block
  - do not defines the behaviour of the programs
  - makes cache memories transparent to the programmer
  - simplifies reasoning about consistency
- Consistency
  - defines the behaviour of all accesses to different memory locations
  - can be defined with program order
  - can use coherence

Outline

1. Cache coherence
2. Memory consistency
3. Coherence-Consistency Interaction
4. Open research questions

INTEL-LIKE MULTICORE

- Cache coherence protocol
  - Invalidation-based
  - MESI states
- Memory consistency model
  - Total Store Order (TSO)
    - load→load
    - store→store
    - load→store

Performing and commiting loads and stores

- A load operation
  - enters the load queue (LQ) and re-order buffer (RoB) in order
  - performs when the data is loaded to the register
  - commits when it is retired from the LQ/RoB
- A store operation
  - enters the store queue (SQ) and re-order buffer (RoB) in order
  - commits when it is retired from the SQ/RoB and enters the store buffer (SB)
  - performs when the data is stored in cache (memory)

NAÏVE IMPLEMENTATION

- A naïve way to enforce TSO consistency, given the guarantees of the underlying coherence protocol (i.e. write-atomicity) is to enforce the load→load, store→store, and load→store ordering rules by delaying the second operation until the first one completes.
- Performance loss.

Code example

Initially X = Y = 0

```
lx: $r0 = X;
ly: $r1 = Y;
```

```
sy: Y = 1;
sx: X = 1;
```

Six possible interleavings and values for $r0 and $r1

lx	lx	lx	sy	sy	sy
ly	sy	sy	lx	lx	sx
sy	ly	sx	ly	sx	lx
sx	sx	ly	sx	ly	ly
(0,0)	(0,1)	(0,1)	(0,1)	(0,1)	(1,1)

- (1,0) is not possible under load→load & store→store

SPECULATION

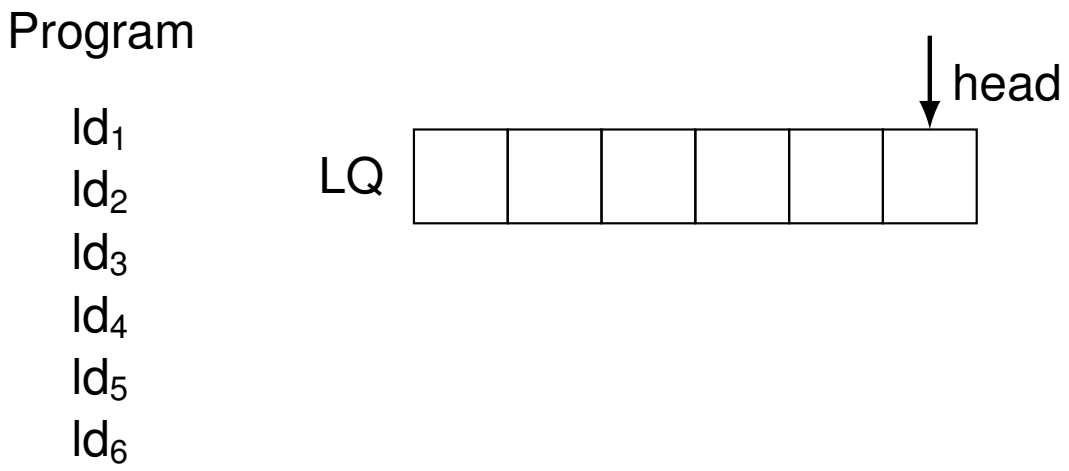
- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism

SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism

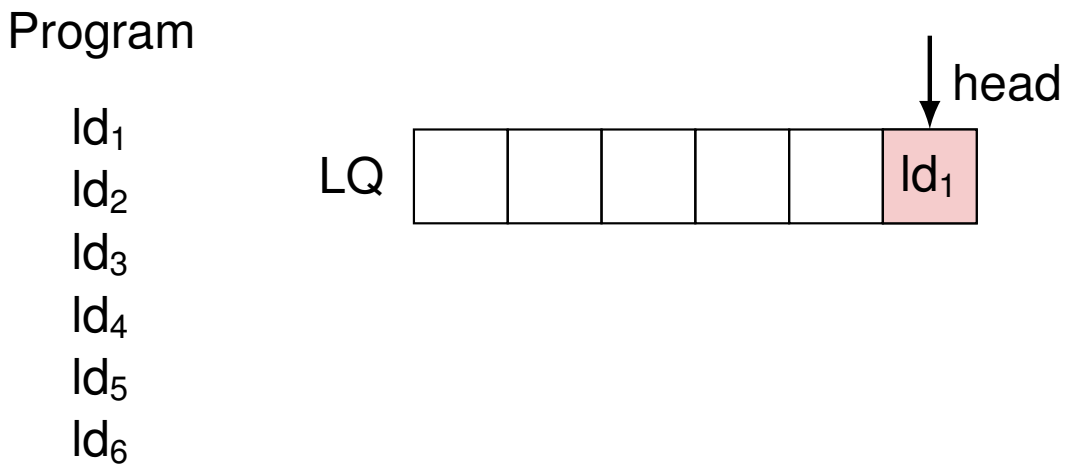
SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



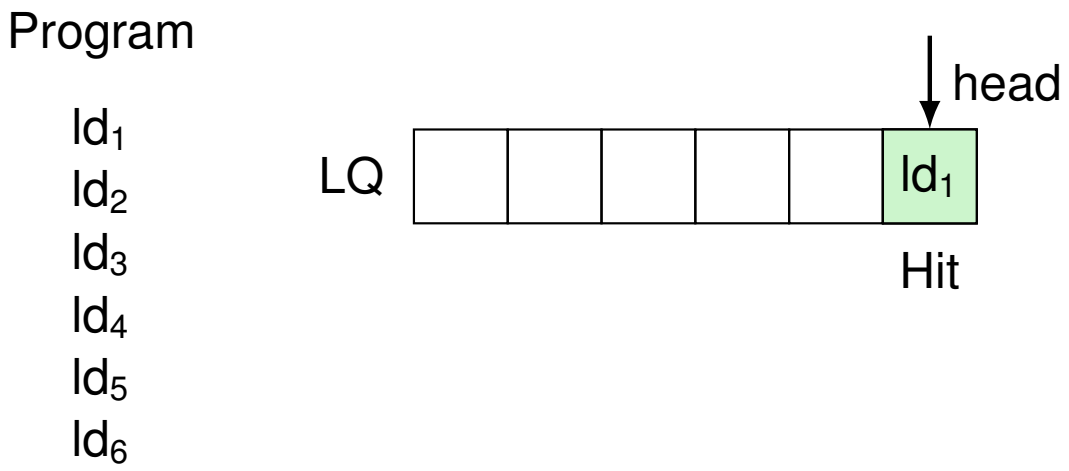
SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



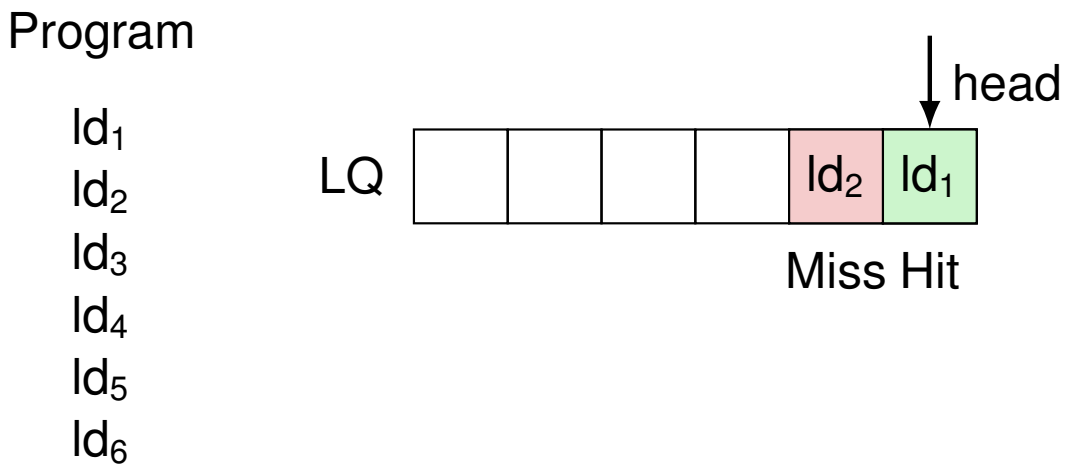
SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



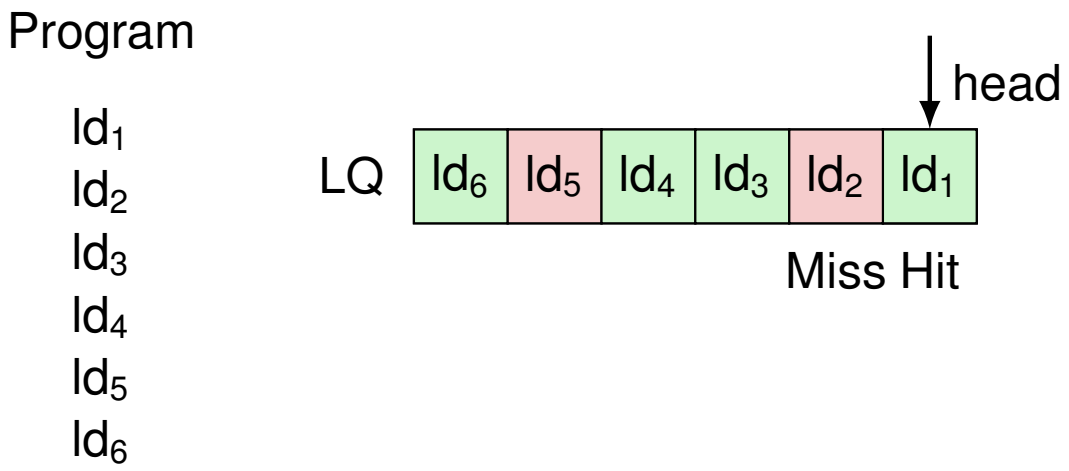
SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



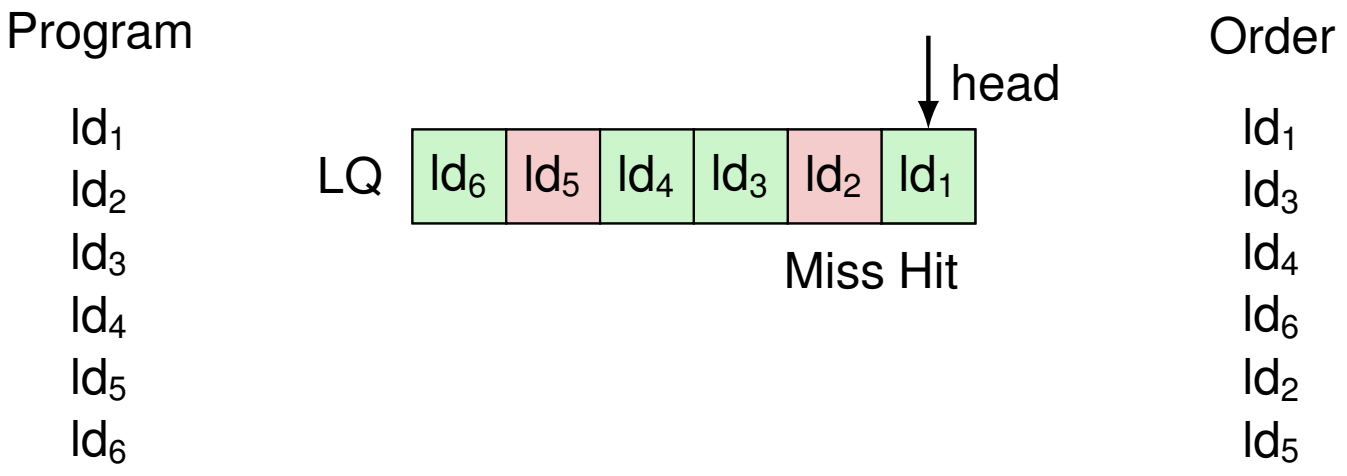
SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



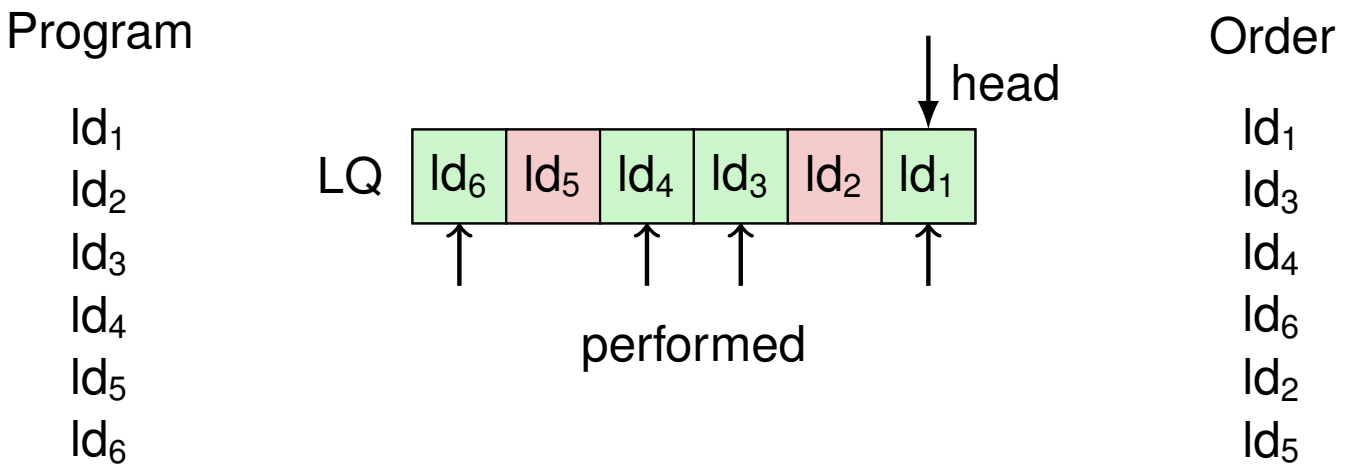
SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered
- This could violate the load→load rule (e.g. hit-miss)



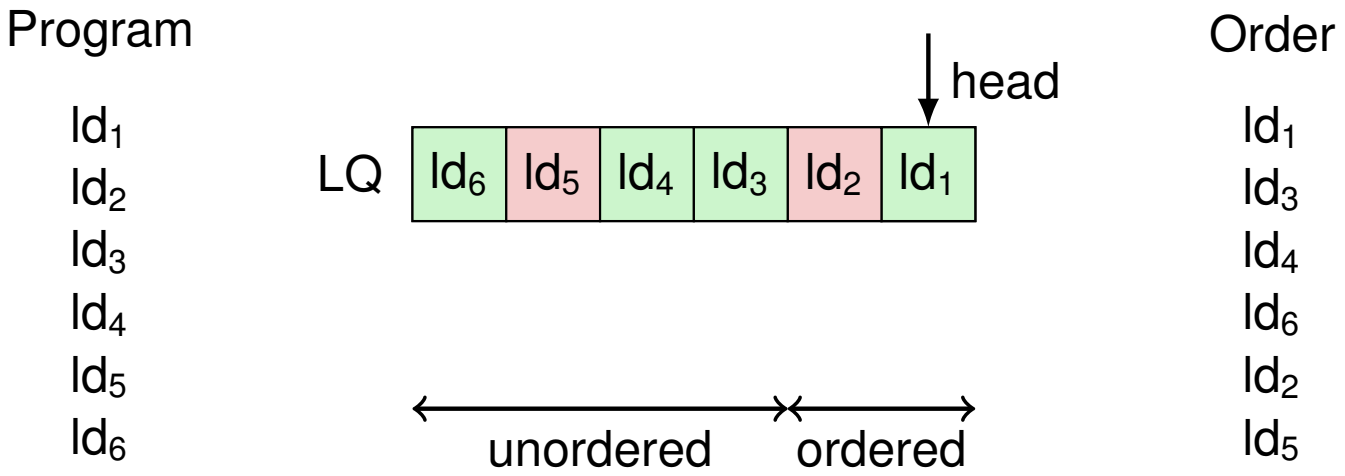
SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered
- This could violate the load→load rule (e.g. hit-miss)



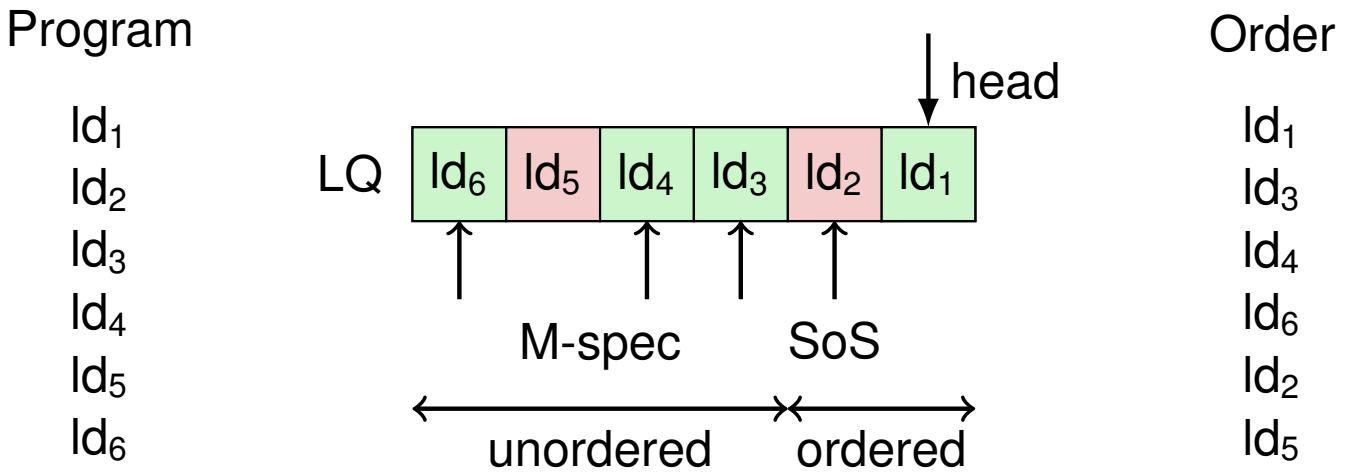
SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered
- This could violate the load→load rule (e.g. hit-miss)



SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered
- This could violate the load→load rule (e.g. hit-miss)



Breaking load→load

Initially X = Y = 0

```
lx: $r0 = X;
ly: $r1 = Y;
```

```
sy: Y = 1;
sx: X = 1;
```

Six possible interleavings breaking load→load

ly	ly	ly	sy	sy	sy
lx	sy	sy	ly	ly	sx
sy	lx	sx	lx	sx	ly
sx	sx	lx	sx	lx	lx
(0,0)	(0,0)	(1,0)	(0,1)	(1,1)	(1,1)

- (1,0) is possible when relaxing load→load

SQUASH AND RE-EXECUTE

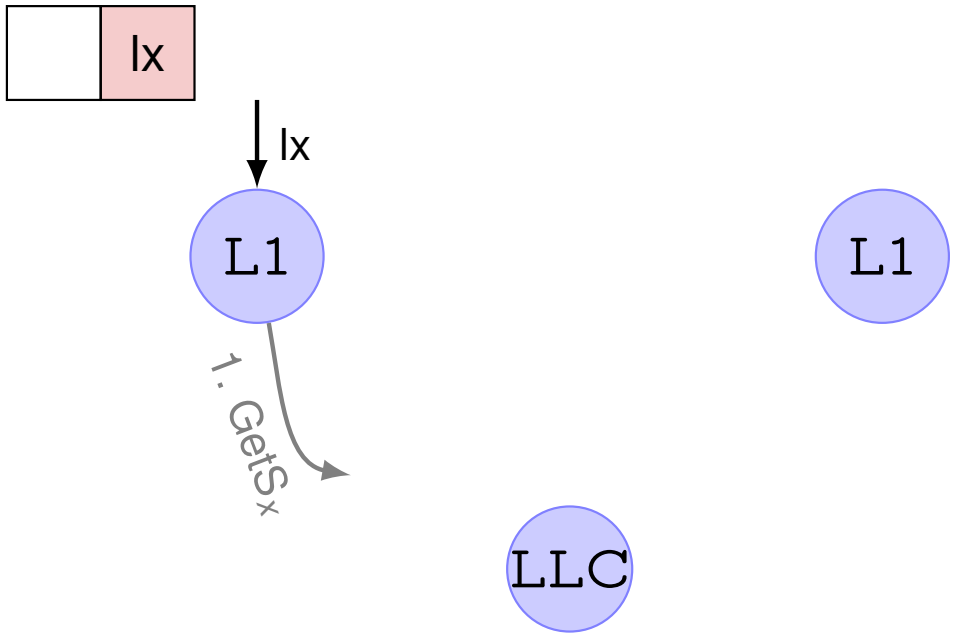
- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

```
ly
      sy
      sx
lx

 (1,0)
```

SQUASH AND RE-EXECUTE

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

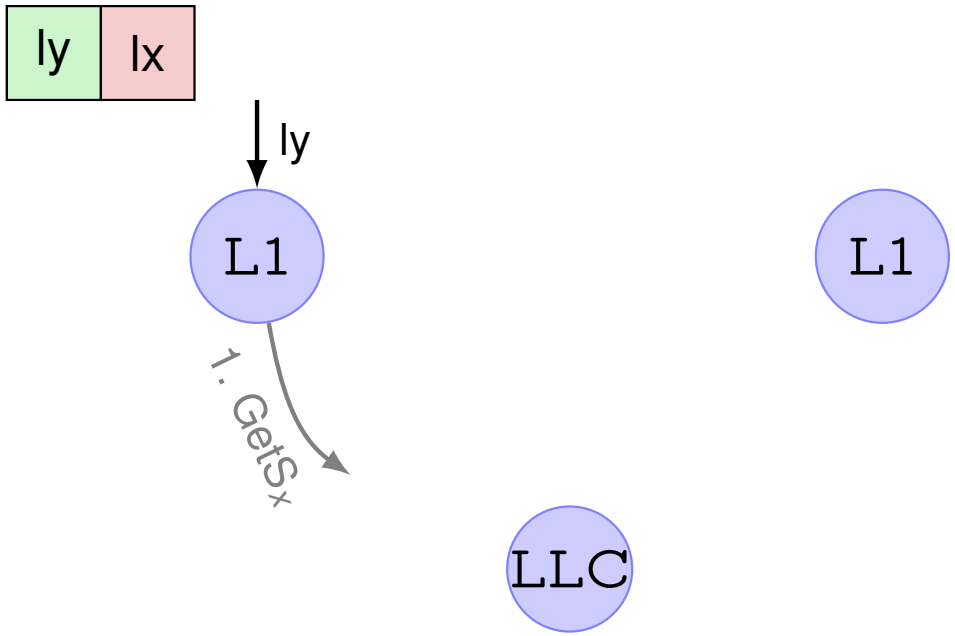
```
ly
      sy
      sx
lx

 (1,0)
```



SQUASH AND RE-EXECUTE

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

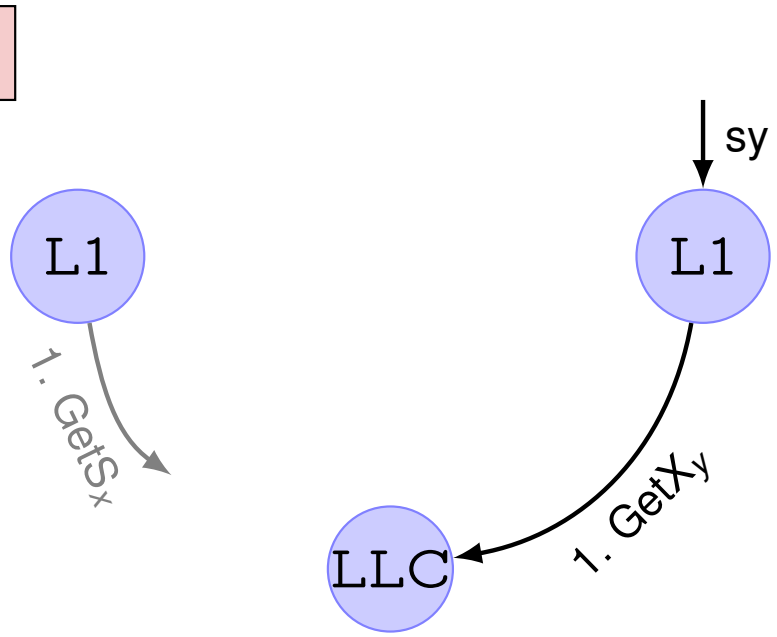
```
ly
      sy
      sx
lx

 (1,0)
```



SQUASH AND RE-EXECUTE

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

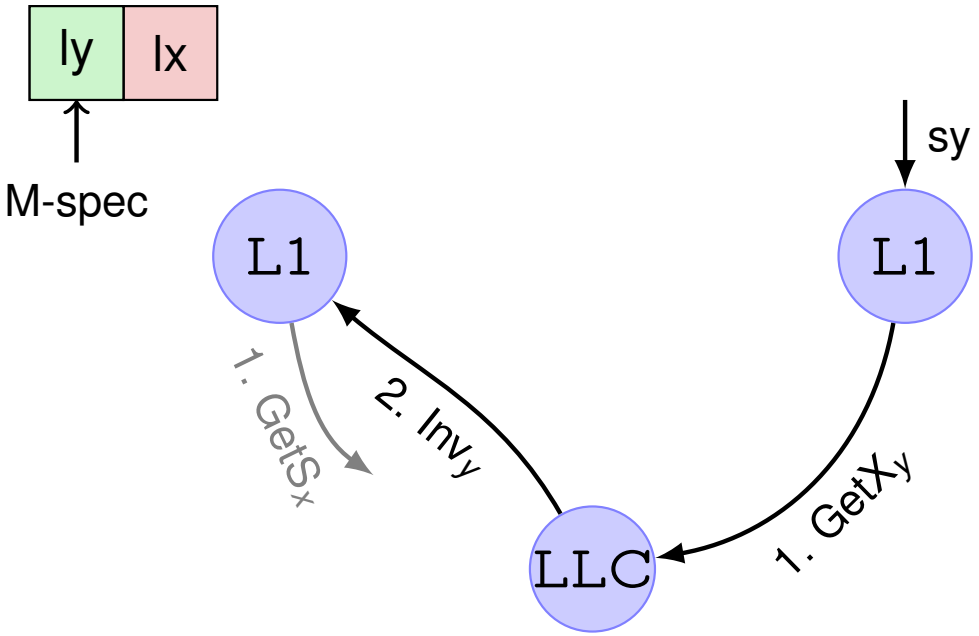
```
ly
      sy
      sx
lx

 (1,0)
```



SQUASH AND RE-EXECUTE

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

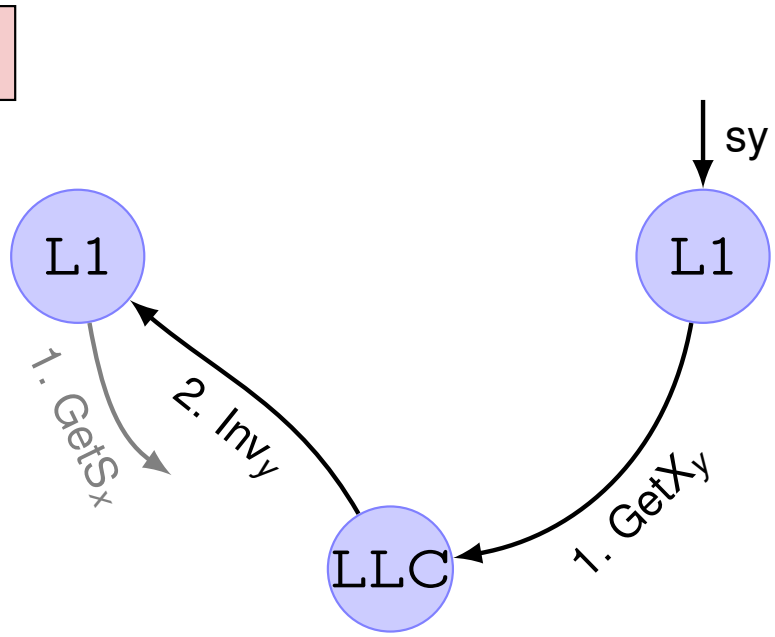
```
ly
      sy
      sx
lx

(1,0)
```



SQUASH AND RE-EXECUTE

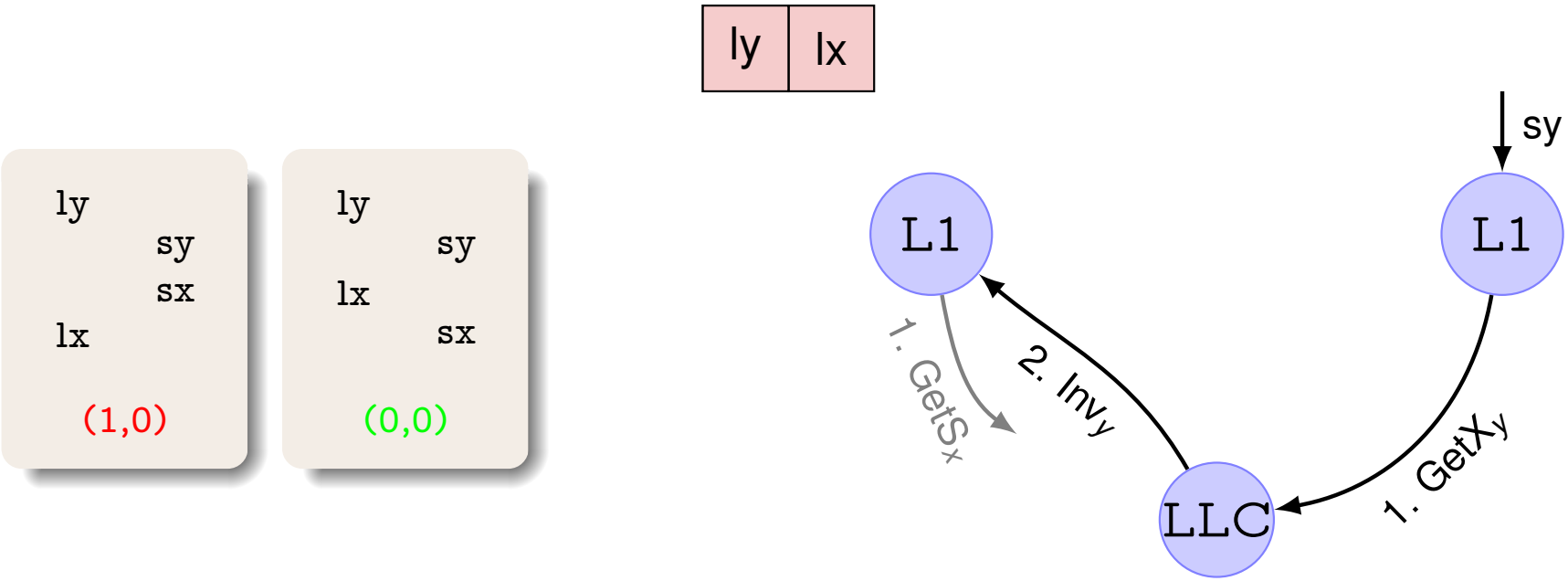
- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol
  - Squashing and re-executing on remote writes

```
ly
      sy
      sx
lx

 (1,0)
```



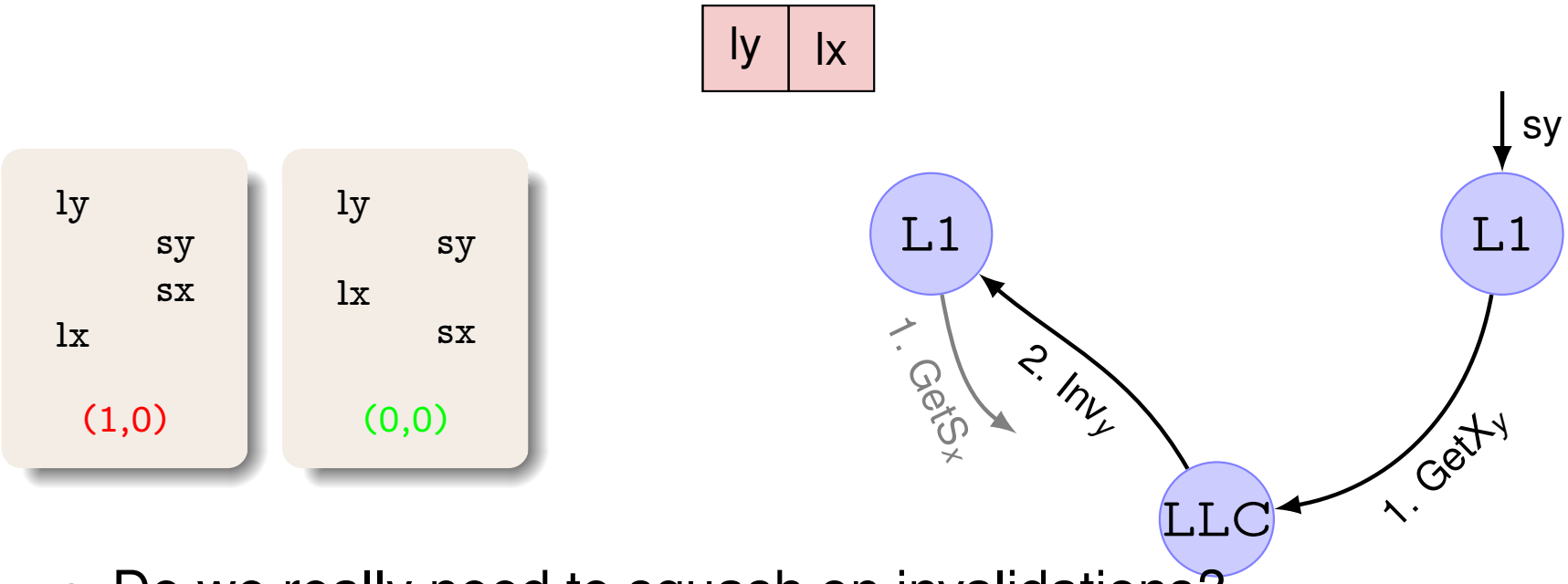
SQUASH AND RE-EXECUTE

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol
  - Squashing and re-executing on remote writes



SQUASH AND RE-EXECUTE

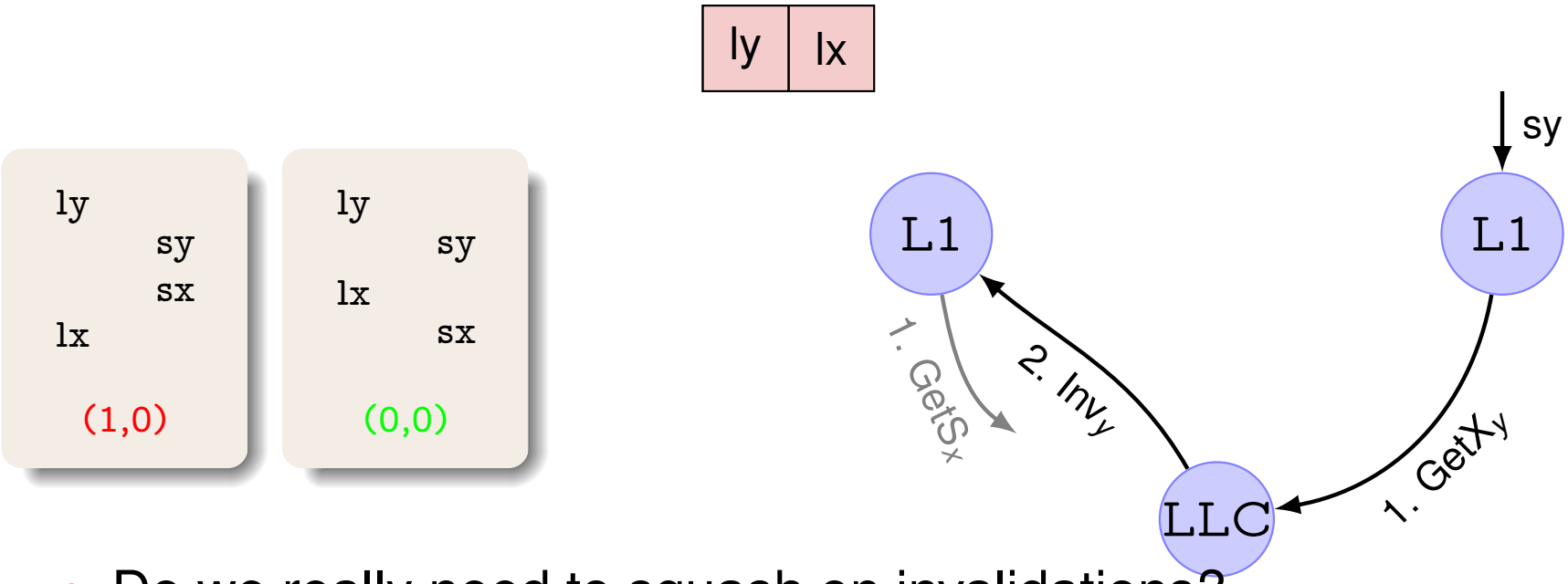
- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol
  - Squashing and re-executing on remote writes



- Do we really need to squash on invalidations?

SQUASH AND RE-EXECUTE

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol
  - Squashing and re-executing on remote writes



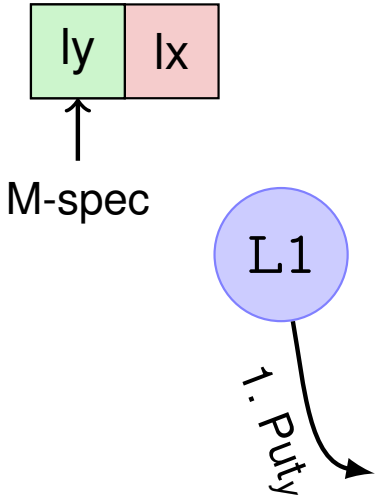
- Do we really need to squash on invalidations?
  - No, we just need to delay `sx`[7]

[7] A. Ros, T. E. Carlson, M. Alipour, S. Kaxiras, "Non-Speculative Load-Load Reordering in TSO", ISCA, 2017.

SQUASH ON CACHE EVICTIONS

- What happens if we need to evict a block used by an M-speculative load?

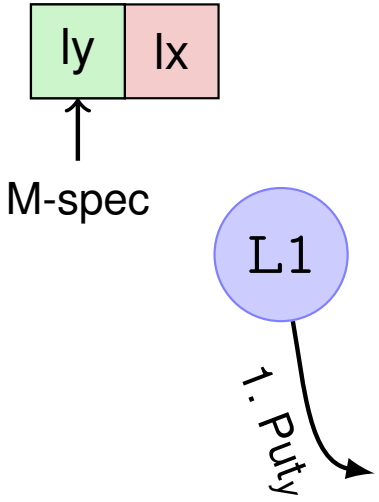
```
ly
      sy
      sx
lx

 (1,0)
```



SQUASH ON CACHE EVICTIONS

- What happens if we need to evict a block used by an M-speculative load?
- If we perform a noisy eviction, and the directory stops tracking it, we will not be able to see an invalidation

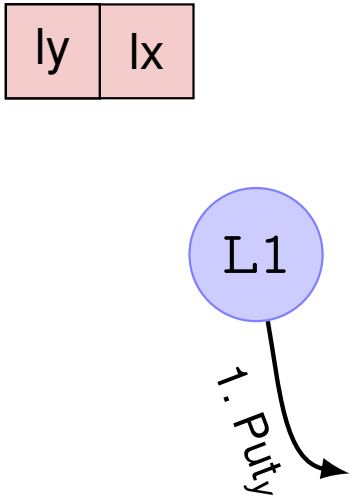
```
ly
      sy
      sx
lx

 (1,0)
```



SQUASH ON CACHE EVICTIONS

- What happens if we need to evict a block used by an M-speculative load?
- If we perform a noisy eviction, and the directory stops tracking it, we will not be able to see an invalidation
- Solution: Squashing and re-executing on evictions

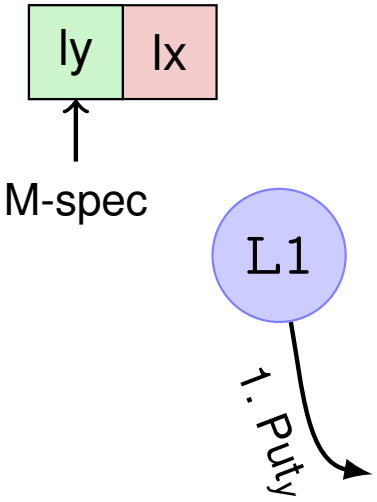
```
ly
      sy
      sx
lx

 (1,0)
```



SQUASH ON CACHE EVICTIONS

- Do we really need to squash on cache evictions?

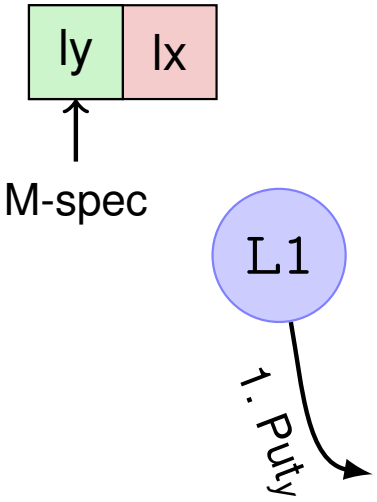
```
ly
      sy
      sx
lx

(1,0)
```



SQUASH ON CACHE EVICTIONS

- Do we really need to squash on cache evictions?
  - No, we just need to force that the invalidation will come on a write

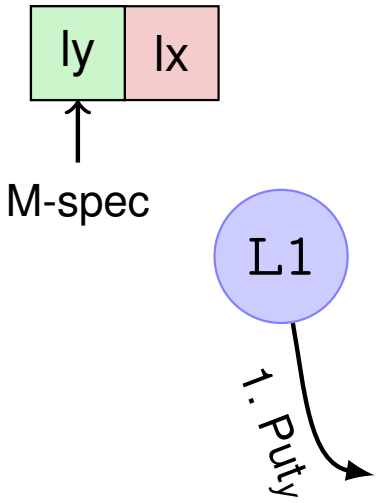
```
ly
      sy
      sx
lx

(1,0)
```



SQUASH ON CACHE EVICTIONS

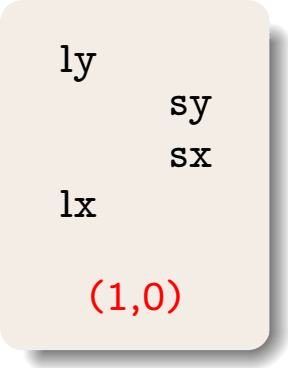
- Do we really need to squash on cache evictions?
  - No, we just need to force that the invalidation will come on a write
- Use silent evictons for clean, non-owner blocks
  - An important reason to implement silent evictions when possible

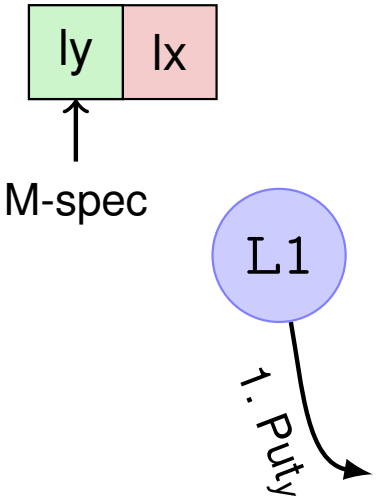
```
ly
      sy
      sx
lx

(1,0)
```



SQUASH ON CACHE EVICTIONS

- Do we really need to squash on cache evictions?
  - No, we just need to force that the invalidation will come on a write
- Use silent evictons for clean, non-owner blocks
  - An important reason to implement silent evictions when possible
- Implement noisy and *keep-track* evictions for dirty, owner blocks[7]

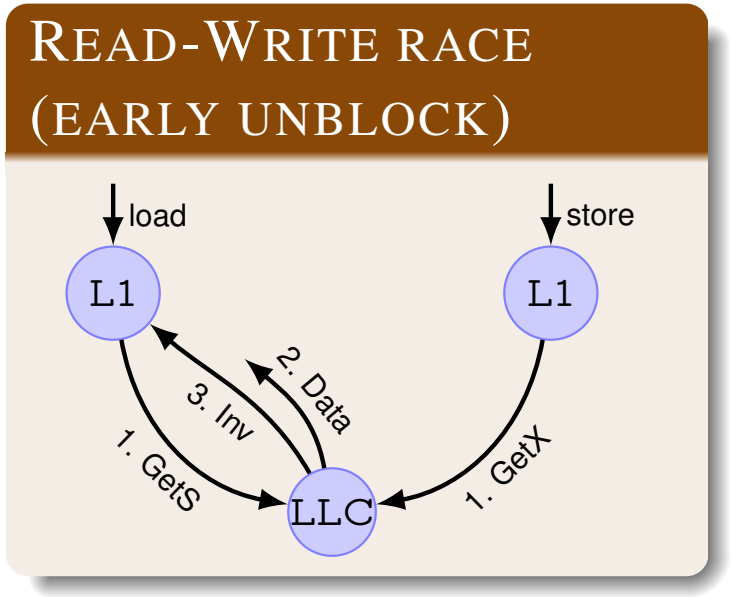




[7] A. Ros, T. E. Carlson, M. Alipour, S. Kaxiras, "Non-Speculative Load-Load Reordering in TSO", ISCA, 2017.

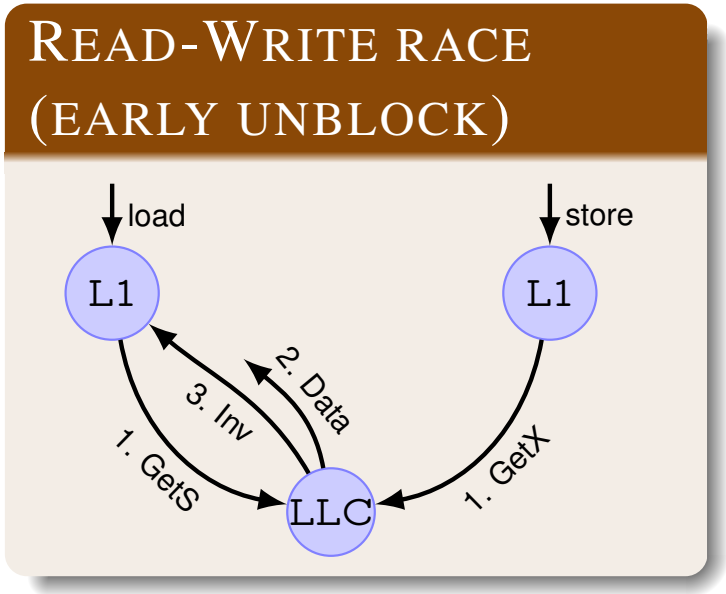
Early vs. late unblock

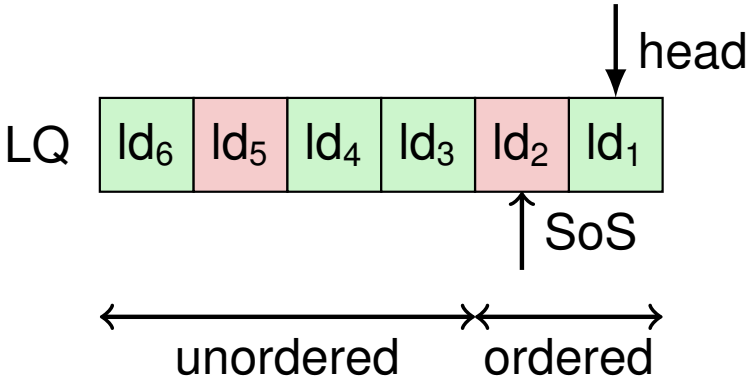
- With early unblock the core cannot infer if the load was ordered before or after the write
- The data cannot be cached, since this could violate the SWMR invariant
- Can the load can perform?



Early vs. late unblock

- With early unblock the core cannot infer if the load was ordered before or after the write
- The data cannot be cached, since this could violate the SWMR invariant
- Can the load can perform?
  - Only if it is the source of speculation (SoS)
  - It may not receive an invalidation





Outline

1. Cache coherence
2. Memory consistency
3. Coherence-Consistency Interaction
4. Open research questions

Open research questions

- What if the cache coherence protocol does not provides write atomicity?
- What if there is no cache coherence protocol or the memory system allows incoherences?
- What if the cache coherence protocol can provide stronger guarantees?
- What it we merge coherence and consistency?

Cache coherence (SC)

- Cache coherence problem studied for several decades
- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)

Consistency model
SC
Cache coherence

Cache coherence (SC)

- Cache coherence problem studied for several decades
- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)
  - Single-writer-multiple-readers (SWMR) invariant
  - Invalidation/update of the copies on every write
  - Large amount of traffic ⇒ increases energy consumption

Consistency model
SC
Cache coherence
SWMR ⇒ Energy

Cache coherence (SC)

- Cache coherence problem studied for several decades
- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)
  - Single-writer-multiple-readers (SWMR) invariant
  - Invalidation/update of the copies on every write
  - Large amount of traffic ⇒ increases energy consumption

Observation 1

Most processors offer consistency models weaker than SC

Consistency model
~~SC~~ TSO RMO
Cache coherence
SWMR ⇒ Energy

Cache coherence (SC)

- Cache coherence problem studied for several decades
- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)
  - Single-writer-multiple-readers (SWMR) invariant
  - Invalidation/update of the copies on every write
  - Large amount of traffic ⇒ increases energy consumption

Observation 1

Most processors offer consistency models weaker than SC

- Why implement protocols that provide more functionality than necessary?

Consistency model
~~SC~~ TSO RMO
Cache coherence
~~SWMR ⇒ Energy~~

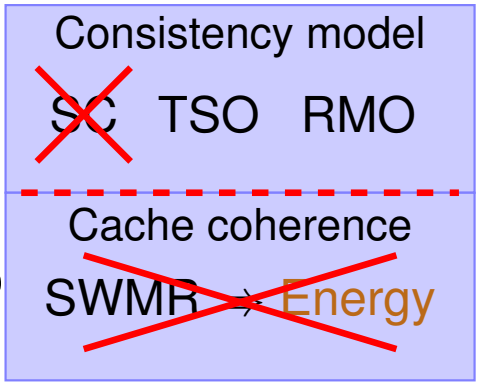
Cache coherence (SC)

- Cache coherence problem studied for several decades
- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)
  - Single-writer-multiple-readers (SWMR) invariant
  - Invalidation/update of the copies on every write
  - Large amount of traffic ⇒ increases energy consumption

Observation 1

Most processors offer consistency models weaker than SC

- Why implement protocols that provide more functionality than necessary?
- Protocol as a black box?
  - Break the layer between the consistency mo and the coherence protocol!



Cache coherence (SC-for-DRF)

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Allows virtual caches without reverse translation ⇒ Efficient
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable

Cache coherence (SC-for-DRF)

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Allows virtual caches without reverse translation ⇒ Efficient
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
- How? Synchronization exposed to the protocol

Example of DRF code

```
X = 1;          | WAIT(cond);
SIGNAL(cond);   | $r1 = X;
```

CACHE COHERENCE (SC-FOR-DRF)

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Allows virtual caches without reverse translation ⇒ Efficient
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
- How? Synchronization exposed to the protocol
  - Release: SELF-DOWNGRADE (SD)
    - ⇒ Write-through dirty blocks

EXAMPLE OF DRF CODE

SD

```
X = 1;          | WAIT(cond);
SIGNAL(cond);   | $r1 = X;
```

Cache coherence (SC-for-DRF)

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Allows virtual caches without reverse translation ⇒ Efficient
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
- How? Synchronization exposed to the protocol
  - Release: Self-downgrade (SD)
    - ⇒ Write-through dirty blocks
  - Acquire: Self-invalidation (SI)
    - ⇒ Empty the cache

Example of DRF code

```
X = 1;          | WAIT(cond);
SIGNAL(cond);   | $r1 = X;
```

SD SI

Cache coherence (SC-for-DRF)

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Allows virtual caches without reverse translation ⇒ Efficient
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
- How? Synchronization exposed to the protocol
  - Release: Self-downgrade (SD)
    - ⇒ Write-through dirty blocks
  - Acquire: Self-invalidation (SI)
    - ⇒ Empty the cache

Example of DRF code

```
X = 1;                  | WAIT(cond);
SIGNAL(cond);           | $r1 = X;
```

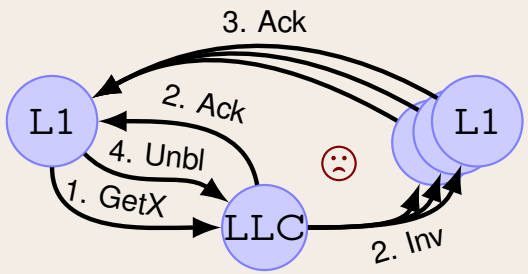
SD SI

Observation 2

SI & SD are conservatively performed because of static synchronization even if there is no actual value propagation between cores

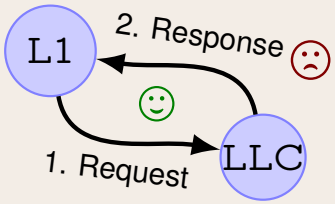
SC versus SC-for-DRF coherence

SC



☺ Works for all codes

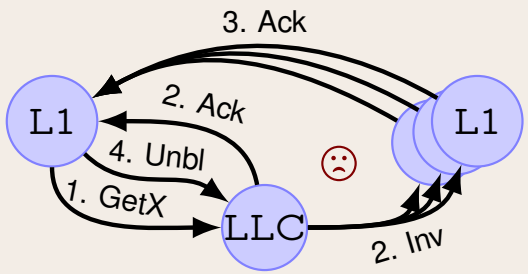
SC-for-DRF



Only works for DRF codes

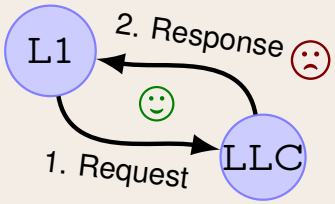
SC versus SC-for-DRF coherence

SC



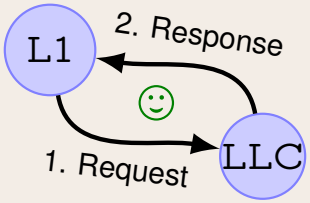
☺ Works for all codes

SC-for-DRF



Only works for DRF codes

Racer (Total-Store-Order – TSO)



☺ Works for all (TSO) codes

First efficient, request-response protocol for all codes

Racer at a glance

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level

Racer at a glance

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD

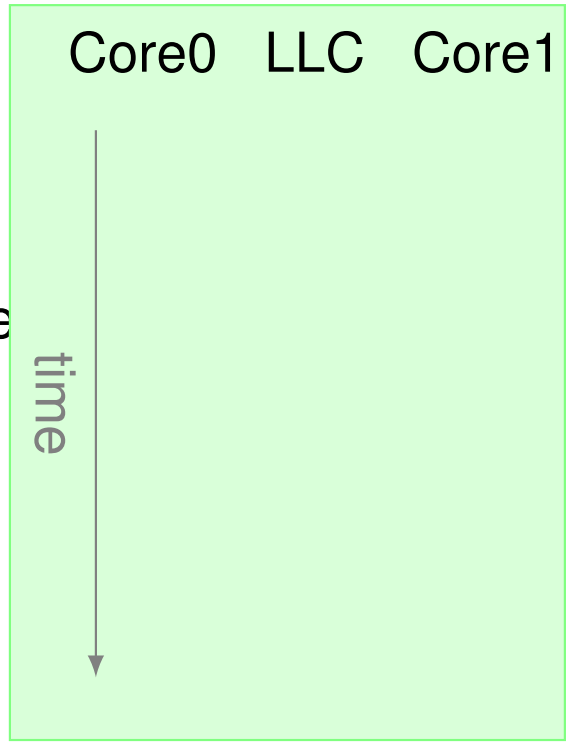
Racer at a glance

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD

- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization (Obs.2)

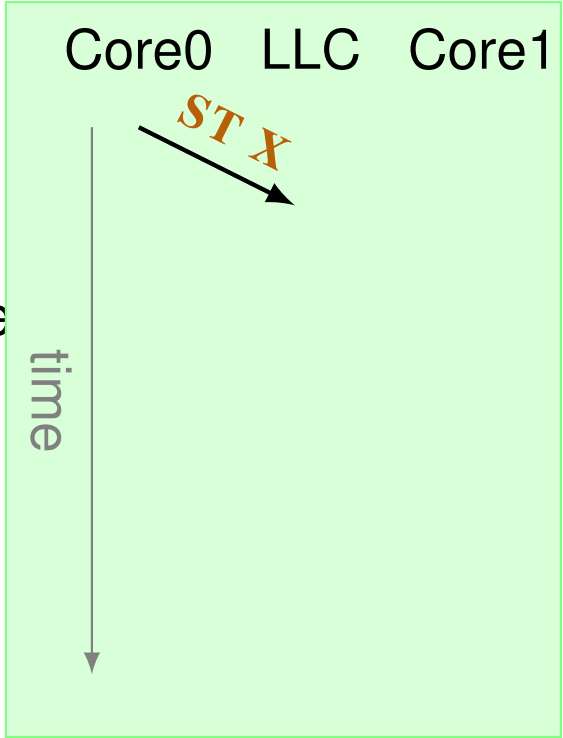
Racer at a glance

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization (Obs.2)
  - ⇒ In Racer only when it is possible to see a reordering (Obs.3)
  - ⇒ On actual (read-after-write) RAW races



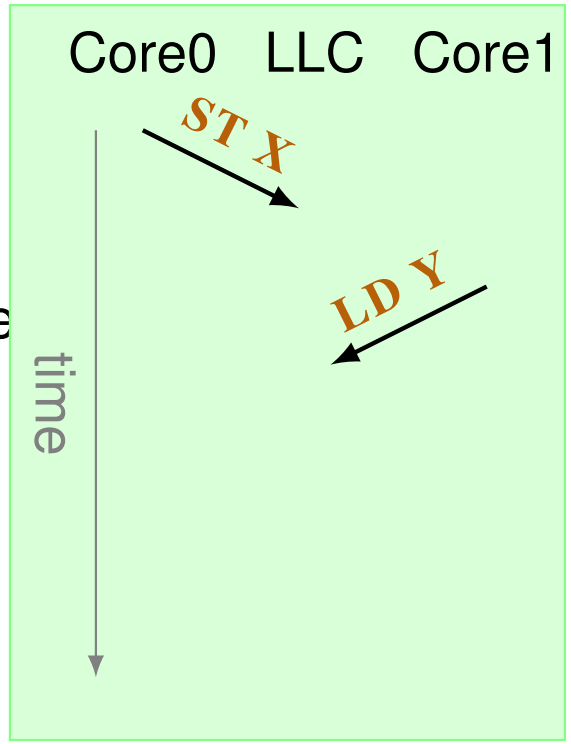
Racer at a glance

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization (Obs.2)
  - ⇒ In Racer only when it is possible to see a reordering (Obs.3)
  - ⇒ On actual (read-after-write) RAW races



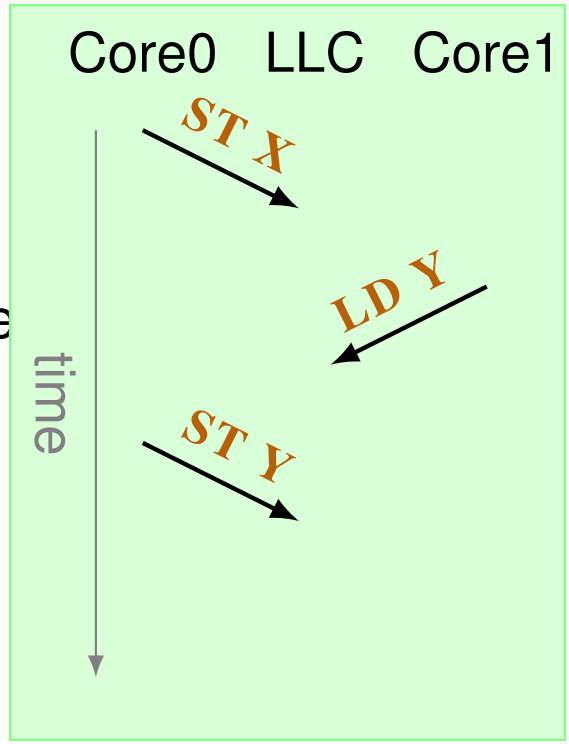
RACER AT A GLANCE

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization (Obs.2)
  - ⇒ In RACER only when it is possible to see a reordering (Obs.3)
  - ⇒ On actual (read-after-write) RAW races



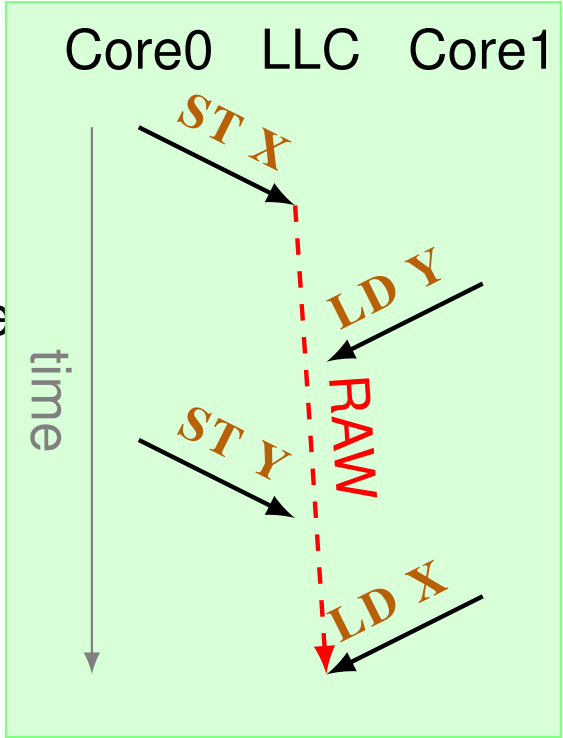
RACER AT A GLANCE

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization (Obs.2)
  - ⇒ In RACER only when it is possible to see a reordering (Obs.3)
  - ⇒ On actual (read-after-write) RAW races



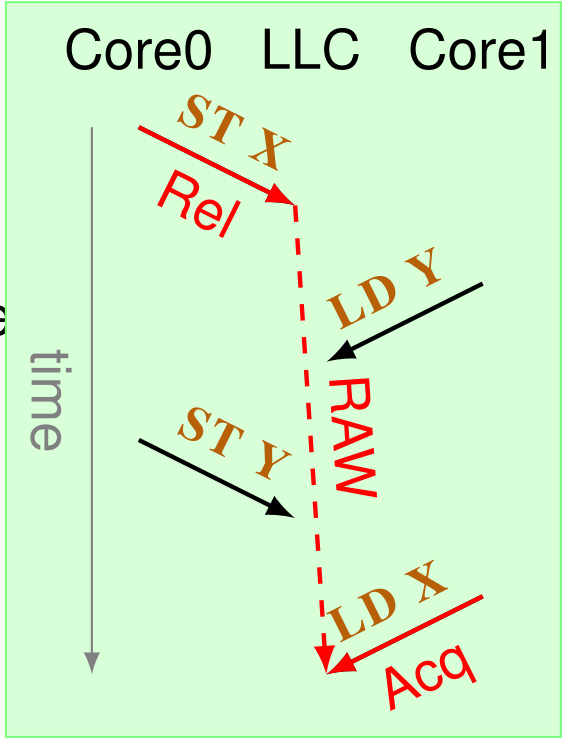
Racer at a glance

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization (Obs.2)
  - ⇒ In Racer only when it is possible to see a reordering (Obs.3)
  - ⇒ On actual (read-after-write) RAW races



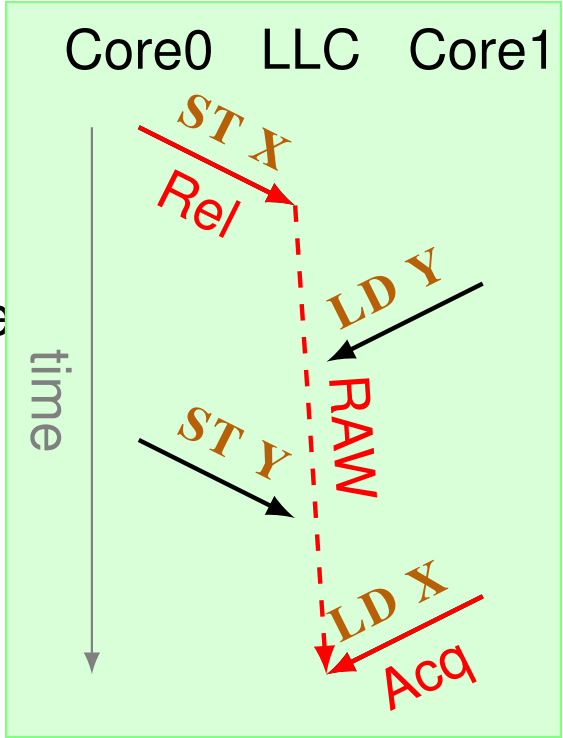
RACER AT A GLANCE

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
  - When it is necessary to enforce order?
    - ⇒ In SC-for-DRF conservatively on synchronization (Obs.2)
    - ⇒ In RACER only when it is possible to see a reordering (Obs.3)
    - ⇒ On actual (read-after-write) RAW races

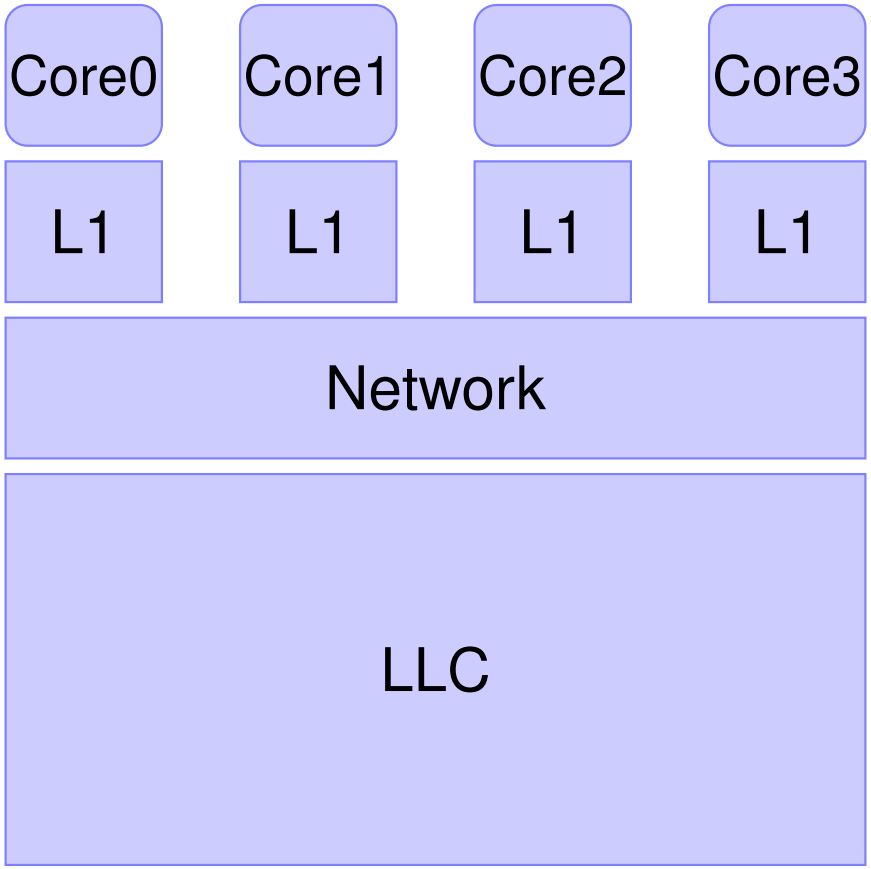


RACER AT A GLANCE

- A novel way of supporting TSO consistency (Obs.1)
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
  - When it is necessary to enforce order?
    - ⇒ In SC-for-DRF conservatively on synchronization (Obs.2)
    - ⇒ In RACER only when it is possible to see a reordering (Obs.3)
    - ⇒ On actual (read-after-write) RAW races
  - Consistency only enforced for shared data *[Singh et al. ISCA'12]*



Basic operation



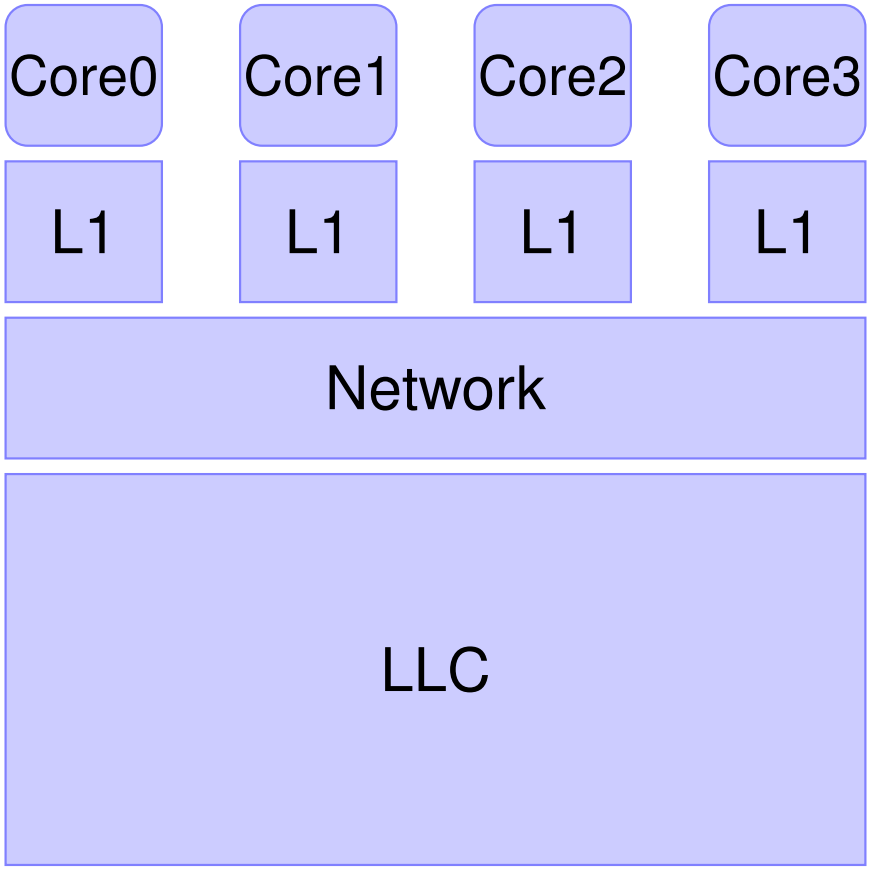
Basic operation

ore0 LLC Core1

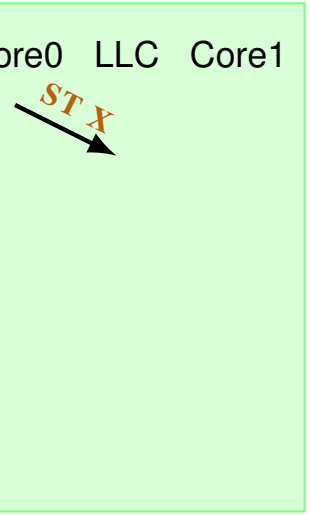
One bloom filter per core

RAWR detector

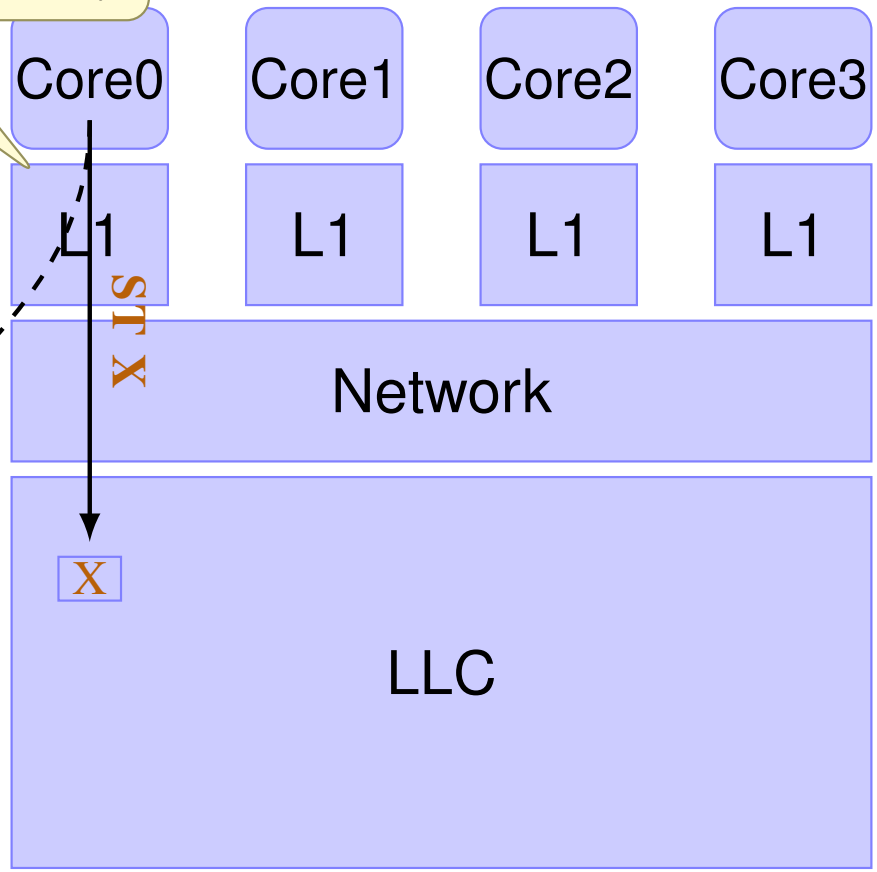
Core	Bloom filter
0	
1	
2	
3	



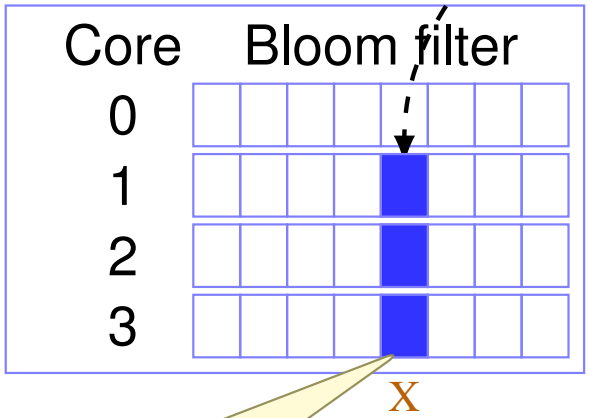
Basic operation



Stores do not require write permission (no SWMR invariant)

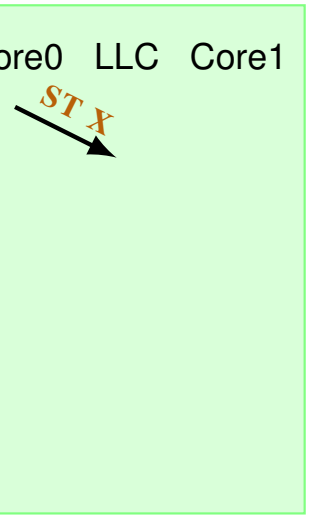


RAWR detector

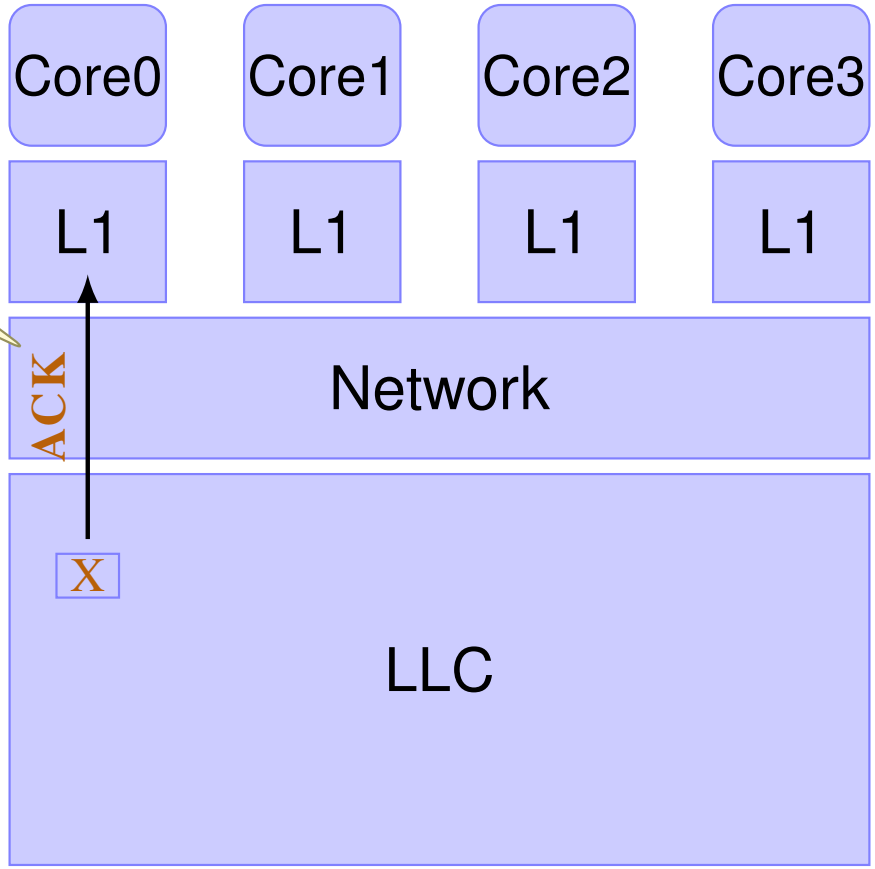


Cores 1, 2, and 3 have not seen the write to x

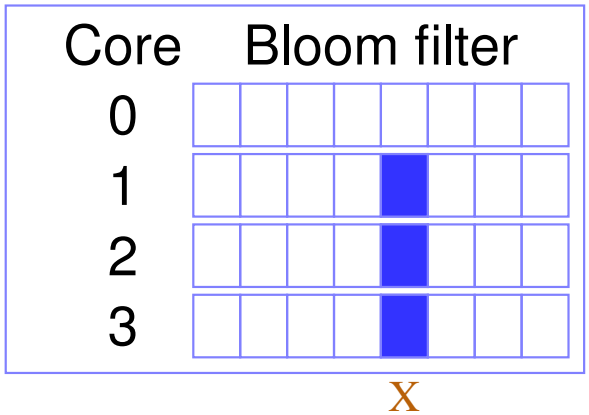
Basic operation



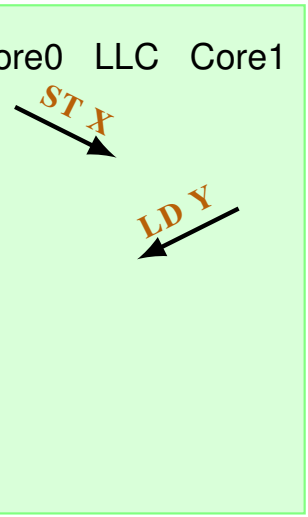
ACK to guarantee order of store operations

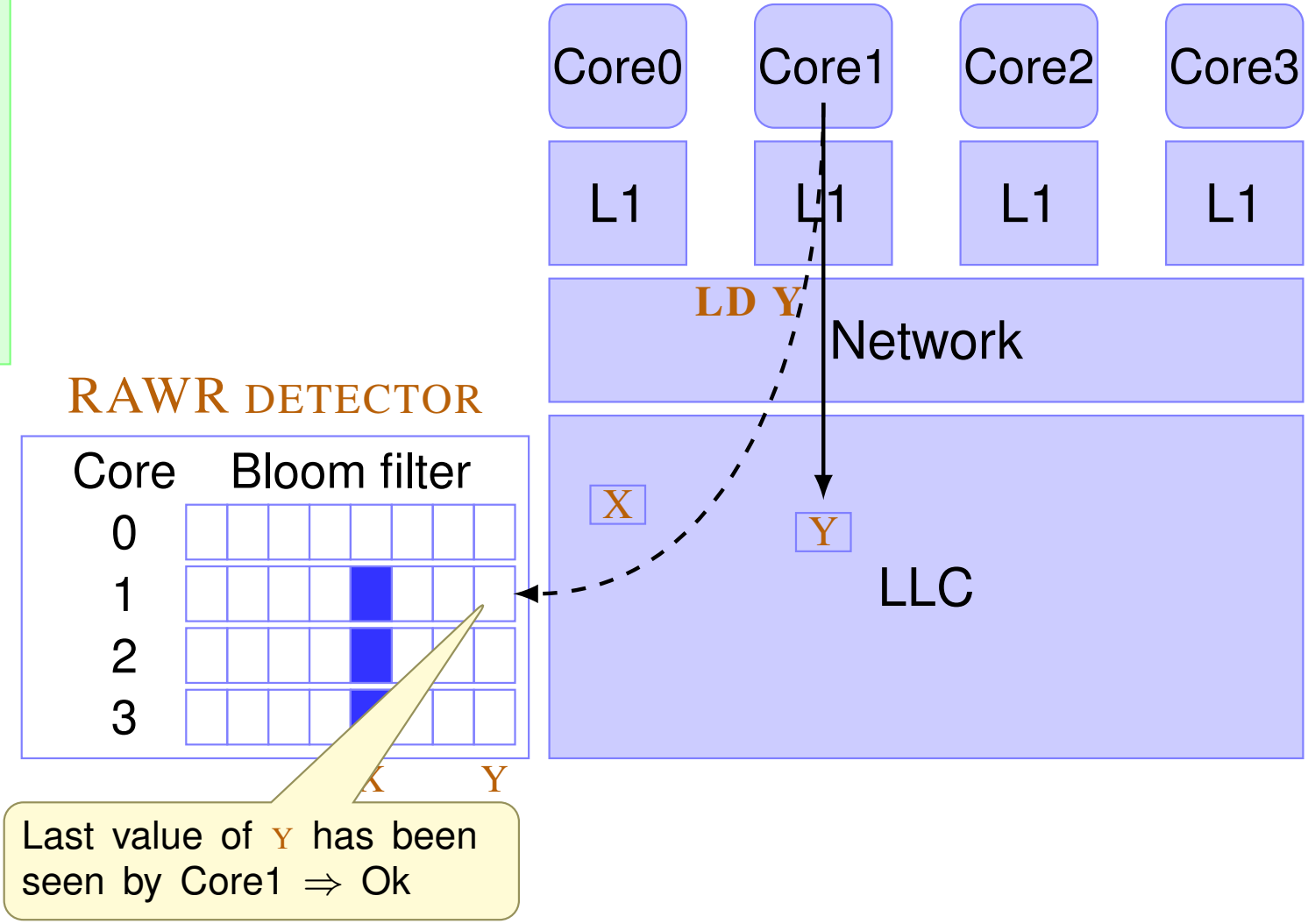


RAWR detector

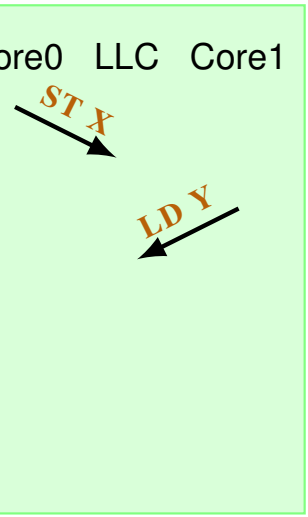


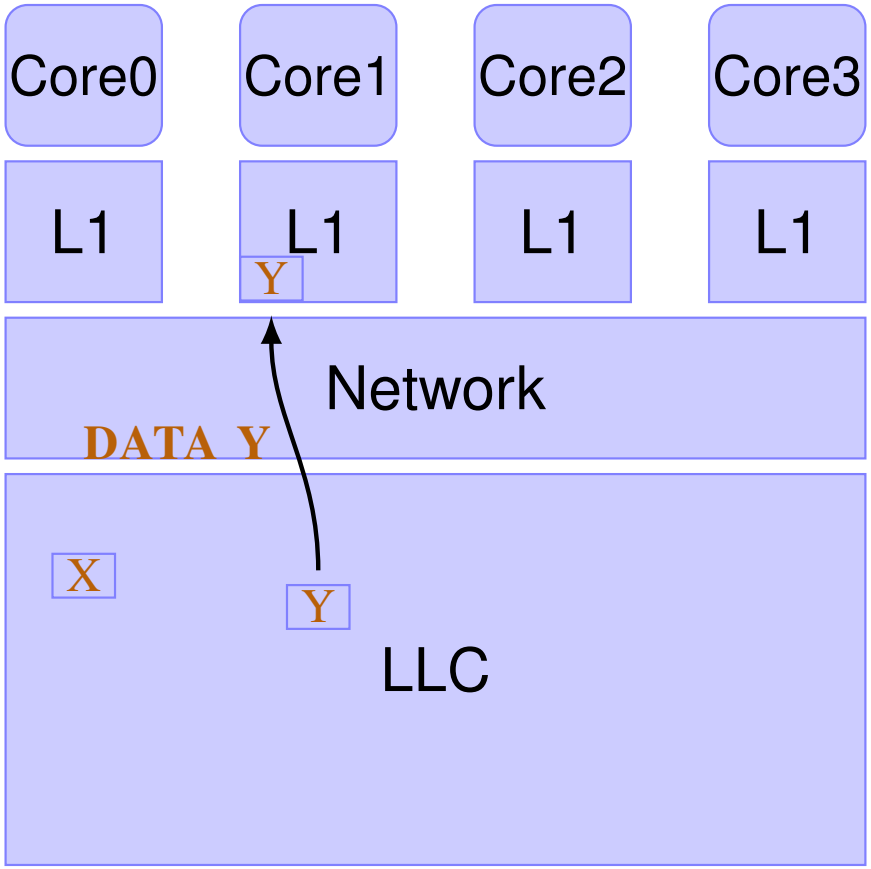
Basic operation



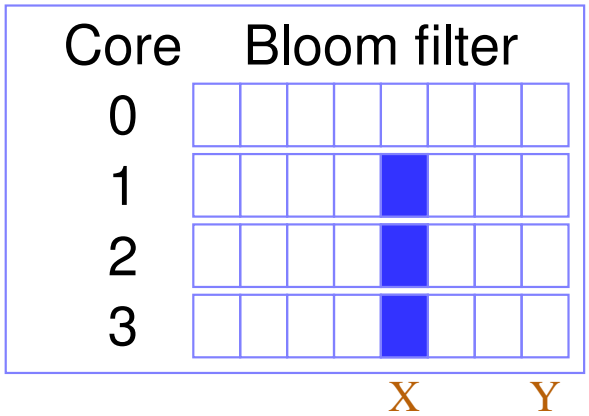


Basic operation

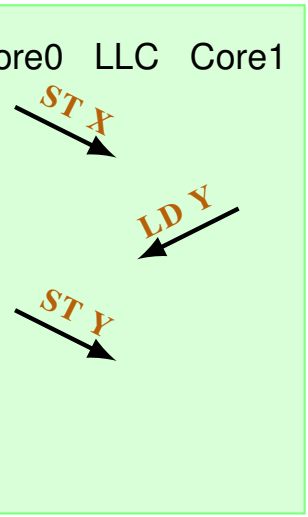


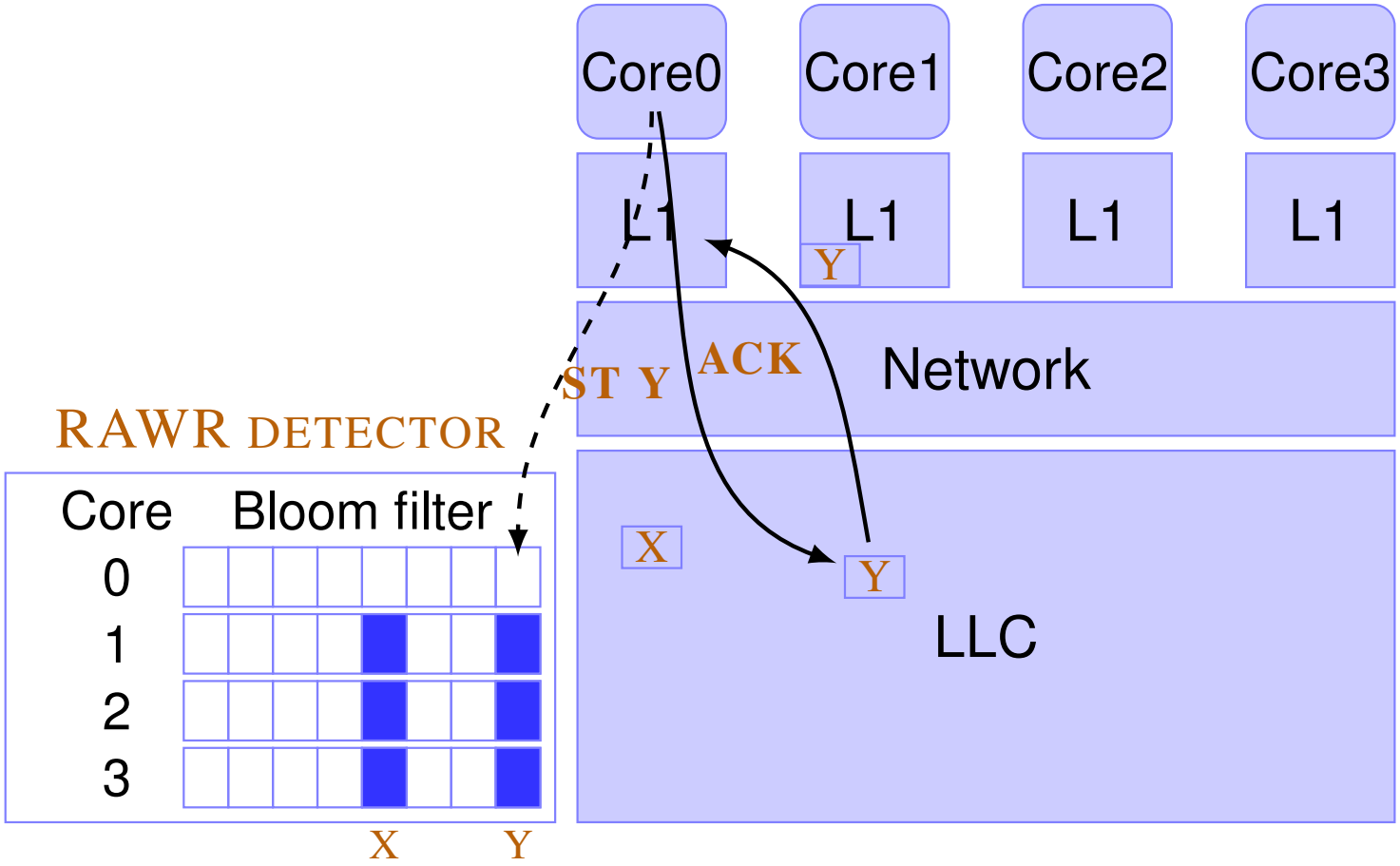


RAWR detector

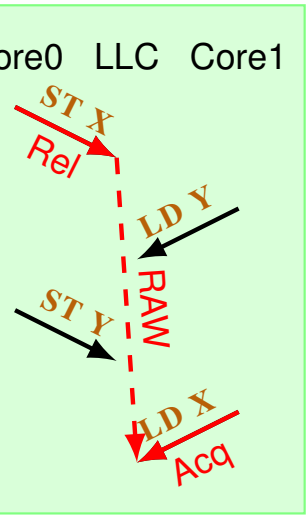


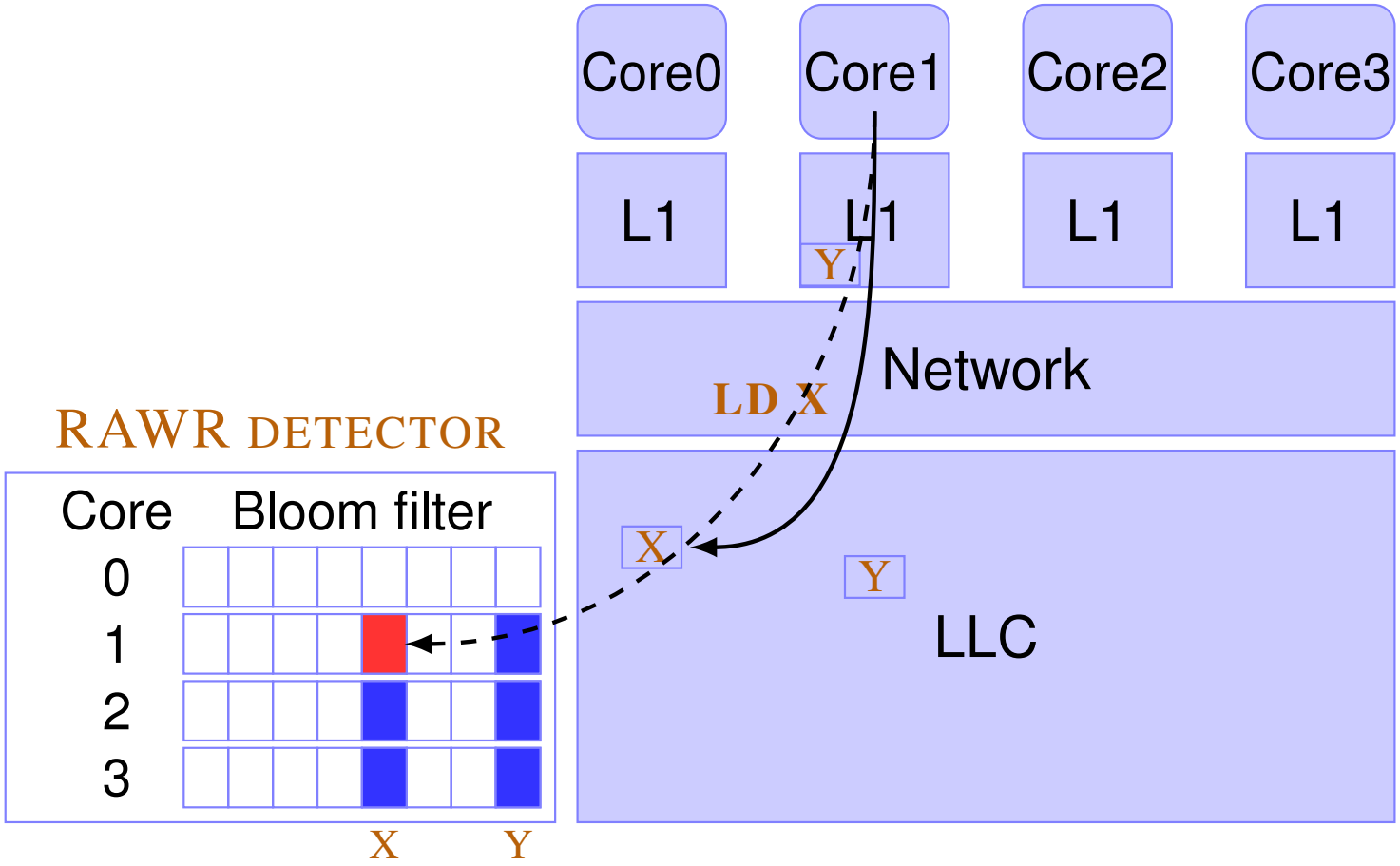
Basic operation



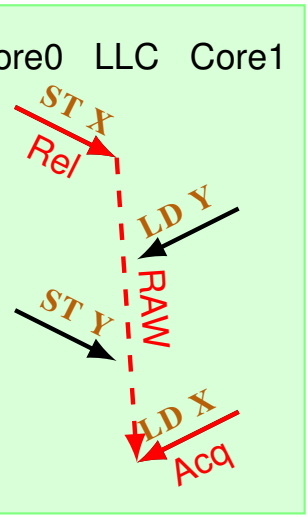


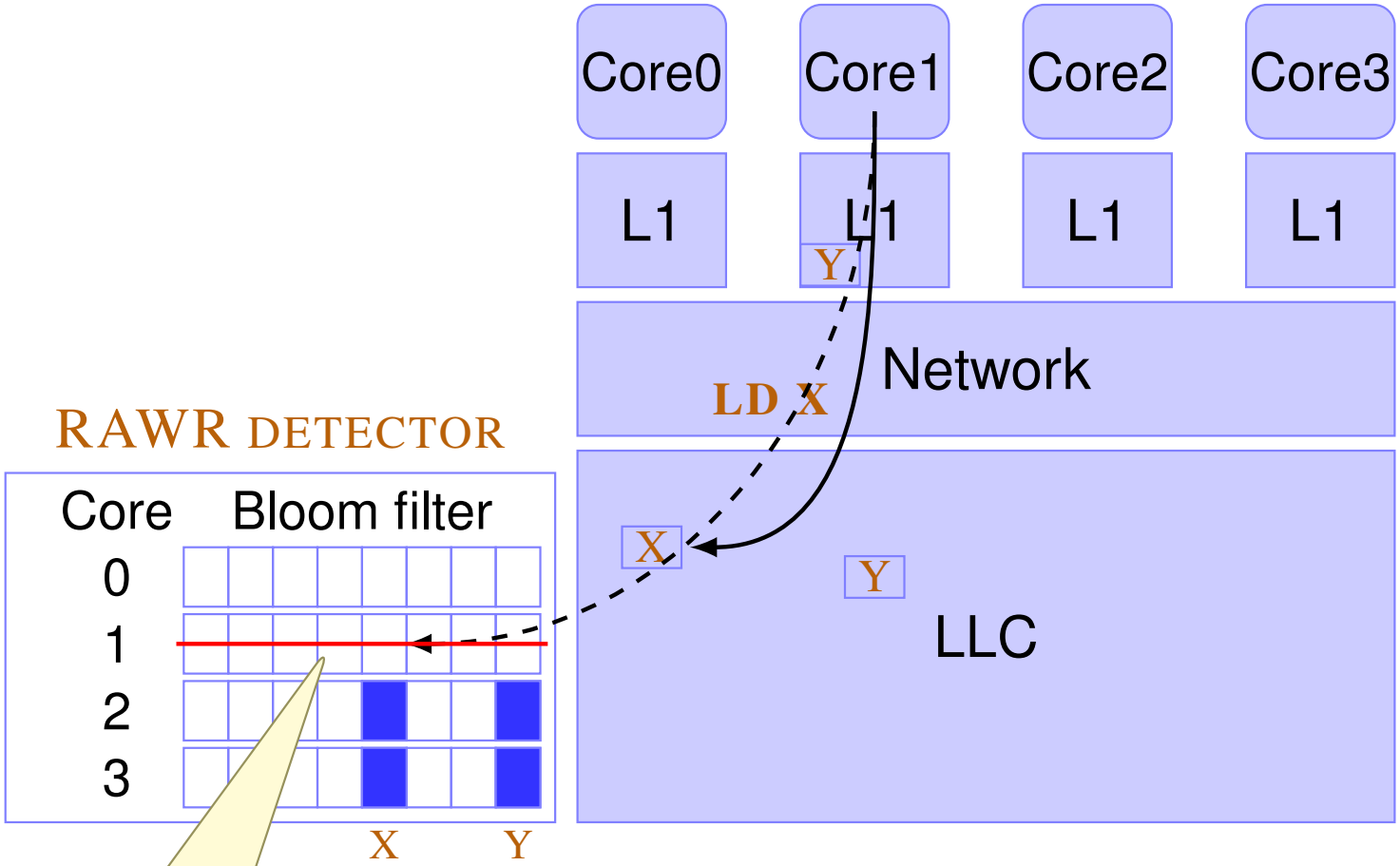
Basic operation





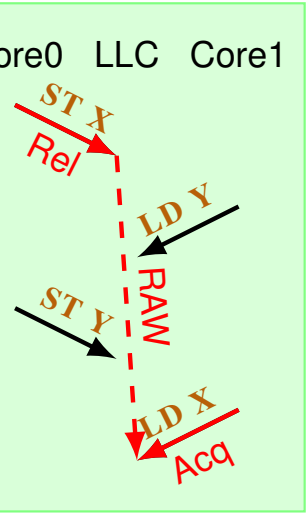
Basic operation

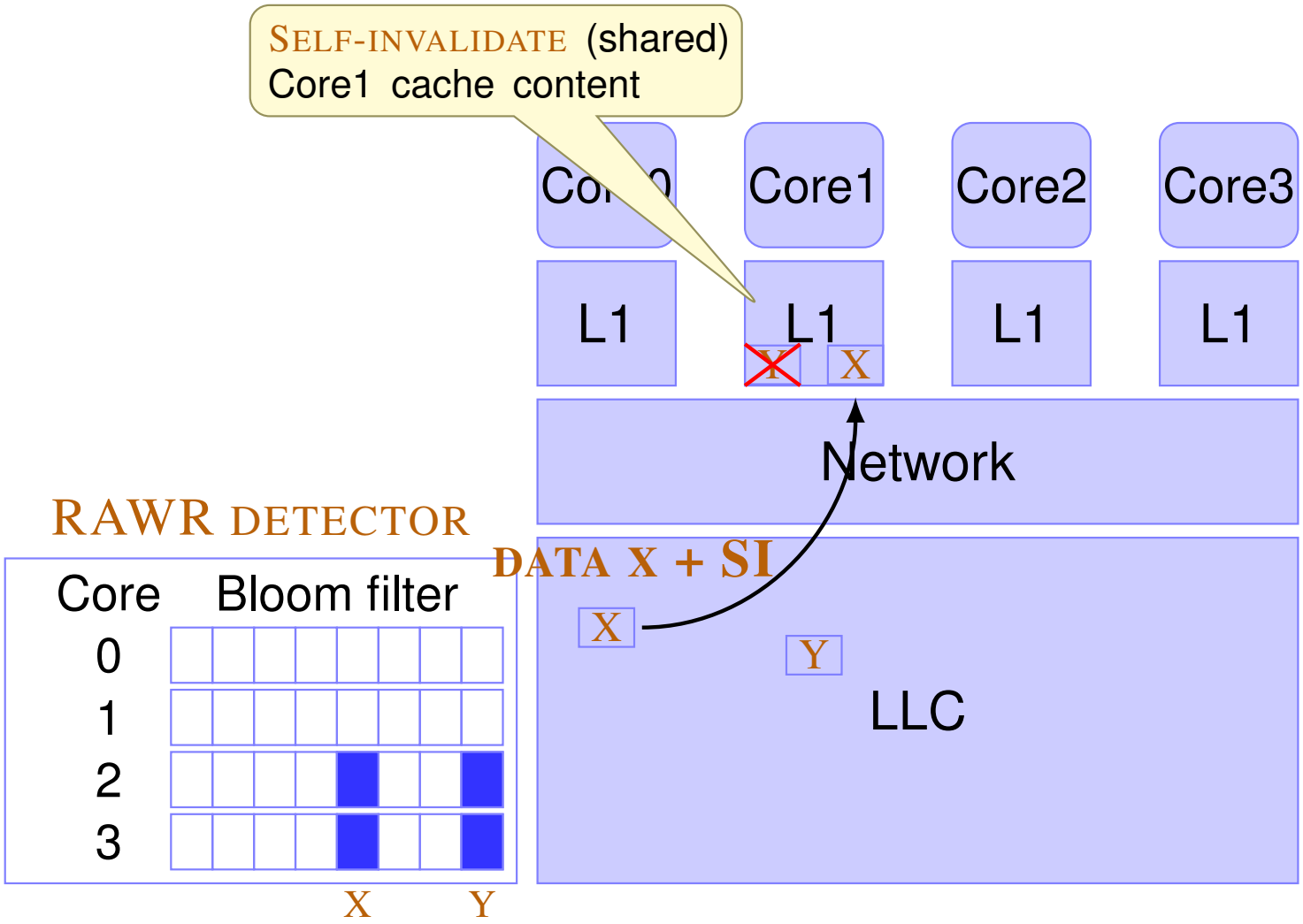




Clean Core1 bloom filter.
Filters are naturally cleared when detecting races!

Basic operation



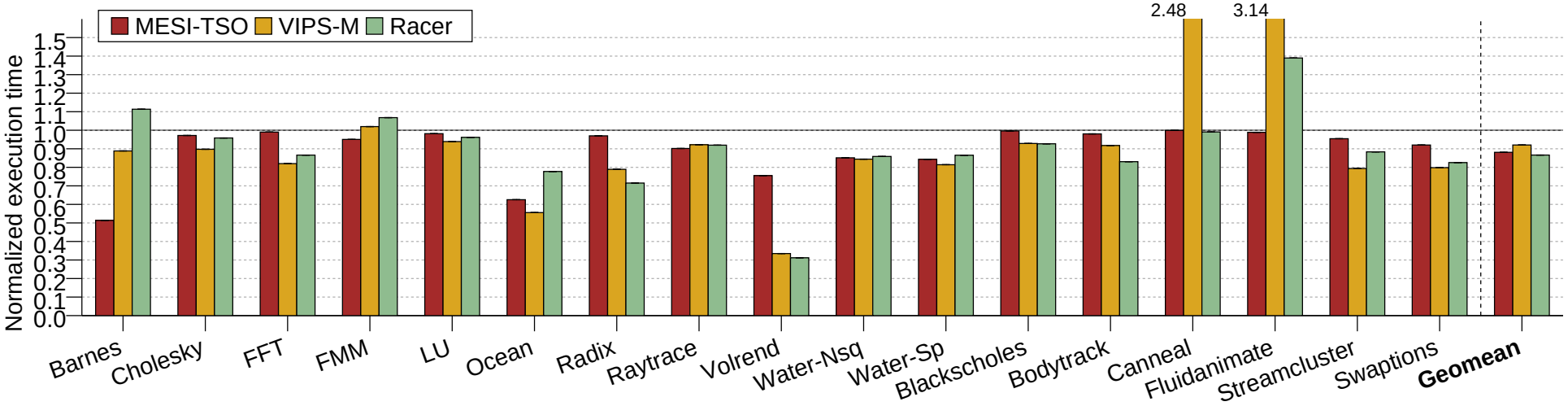


Simulation environment

- 64-core tiled-CMP (GEMS simulator)
  - L1 (private): 32KB 4-way
  - LLC (shared): 256KB 16-way (per tile)
  - RAWR detector: 256-byte bloom filter
  - Racer overhead: ≈18KB per tile
- Benchmarks: Splash-3 and Parsec-2.1
- Protocols evaluated:
  - MESI: Directory-based SC protocol
  - MESI-TSO: Directory-based TSO protocol
  - VIPS-M: SC-for-DRF protocol
  - Racer: TSO protocol

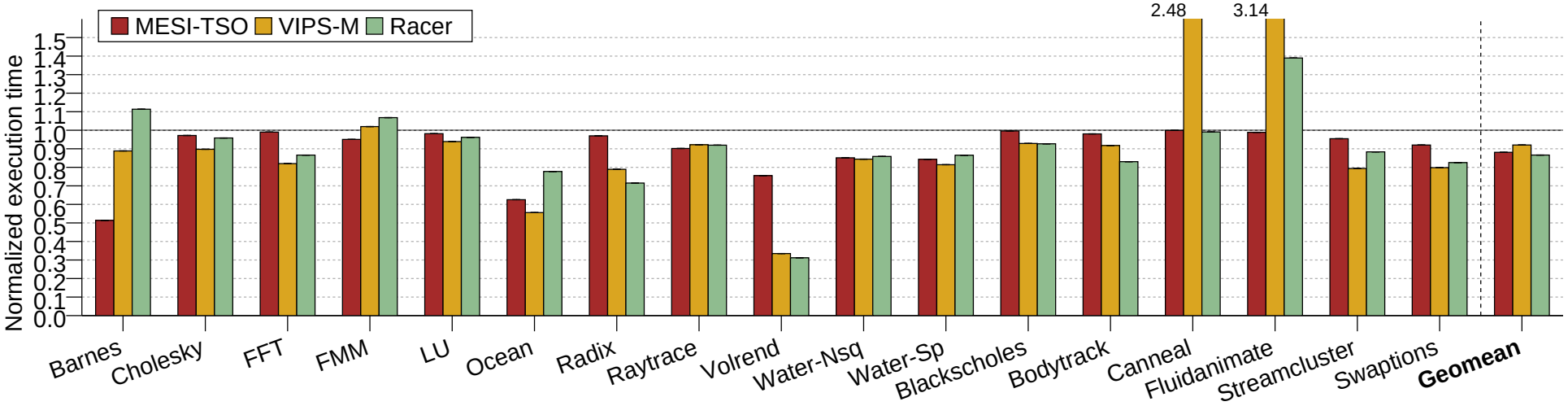
EXECUTION TIME

- Normalized to MESI



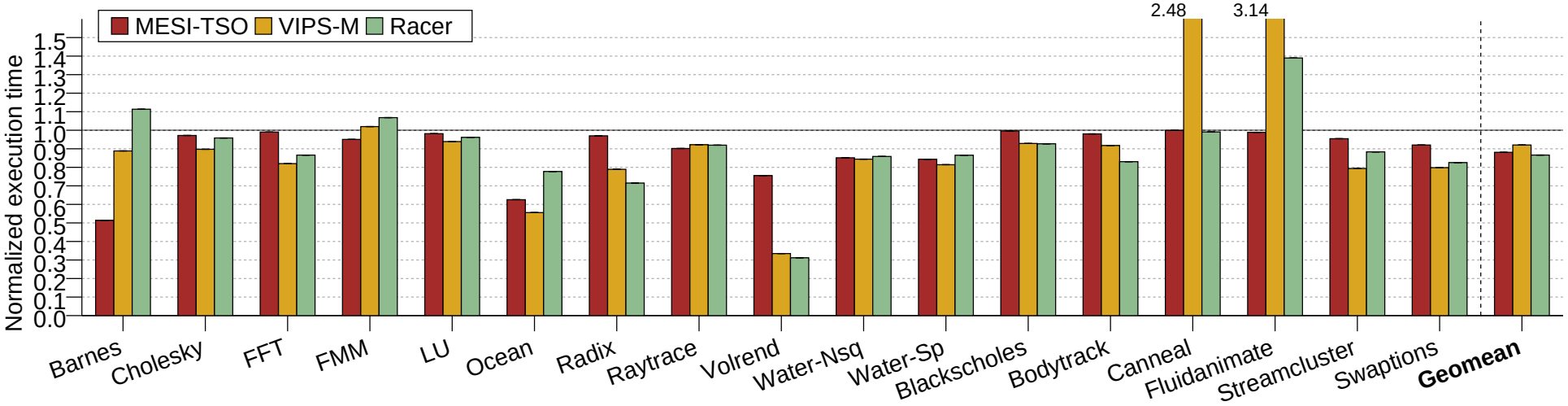
EXECUTION TIME

- Normalized to MESI
- VIPS-M: Conservative SI & SD results in dramatic slow-downs for Fluidanimate and Canneal (Obs.2)



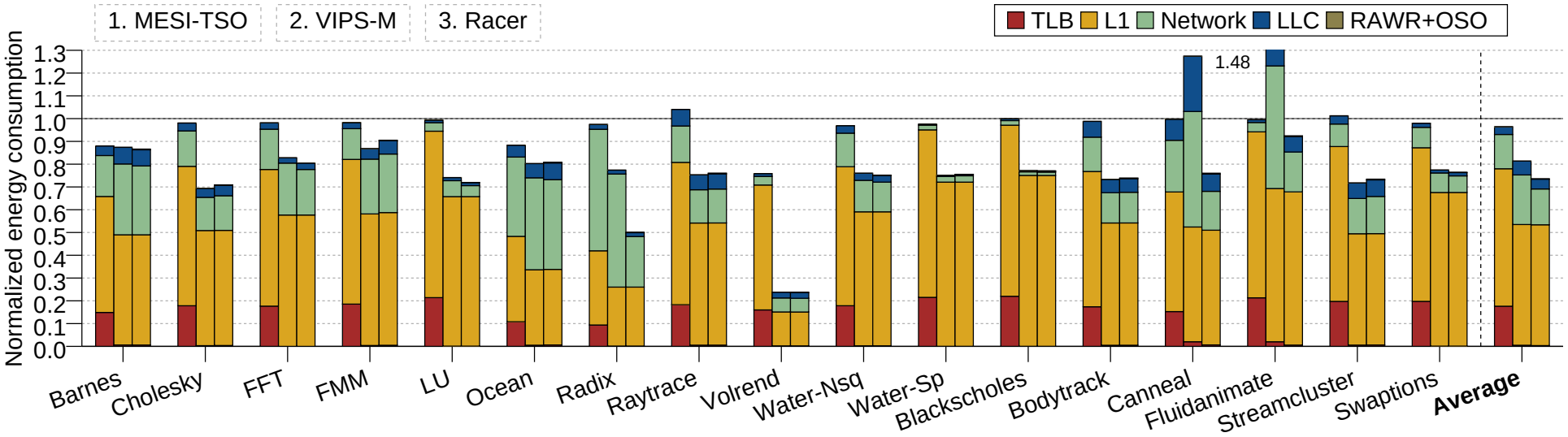
EXECUTION TIME

- Normalized to MESI
- VIPS-M: Conservative SI & SD results in dramatic slow-downs for Fluidanimate and Canneal (Obs.2)
- RACER ≈ non-scalable MESI-TSO
- RACER: better performance than VIPS-M, while providing stronger consistency, but only when needed at run time



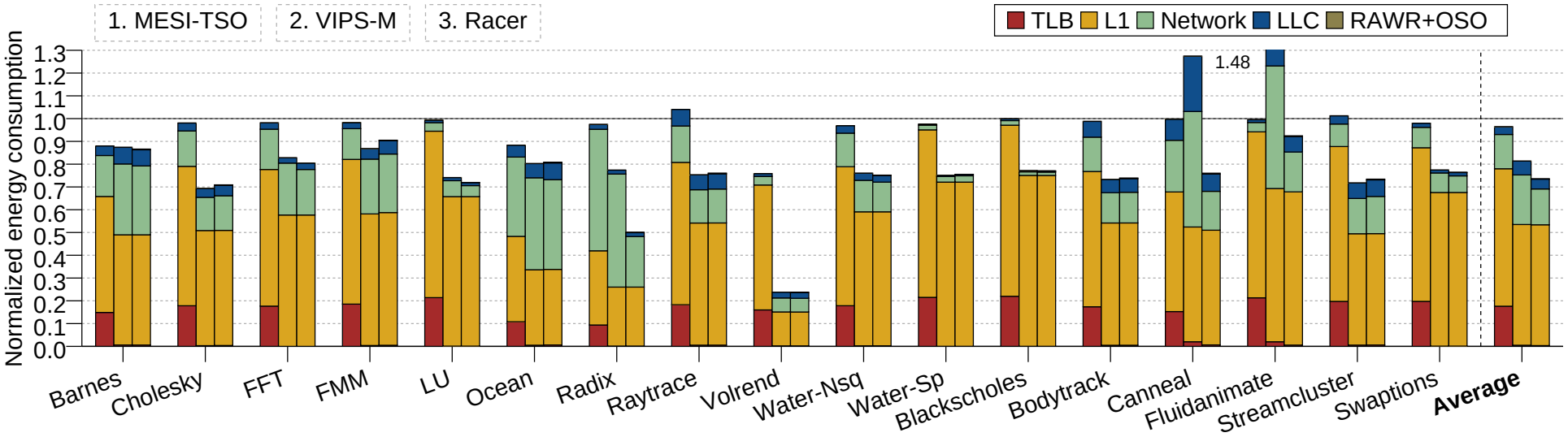
Energy consumption

- Energy of TLBs, L1 caches, network, LLC, and RAWR
- Normalized to MESI



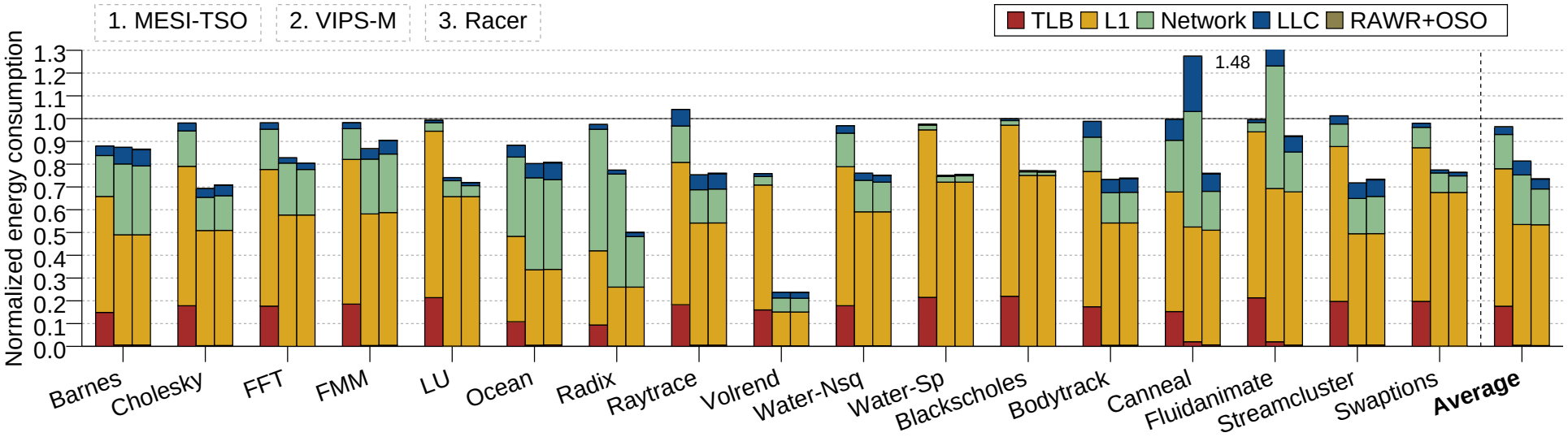
Energy consumption

- Energy of TLBs, L1 caches, network, LLC, and RAWR
- Normalized to MESI
- Racer gets the best from MESI-TSO and VIPS-M



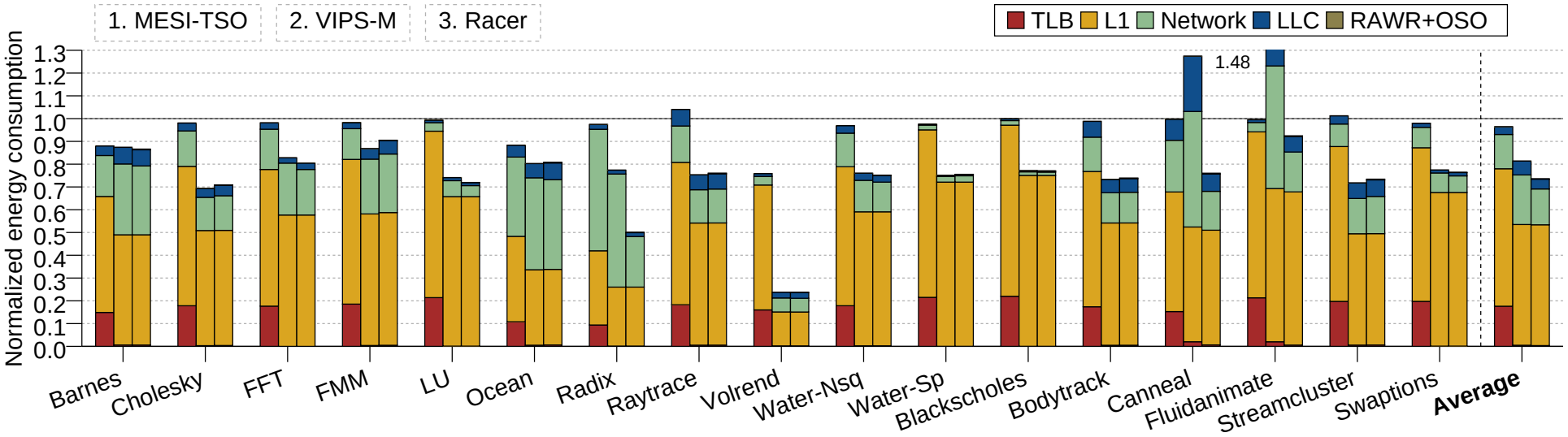
Energy consumption

- Energy of TLBs, L1 caches, network, LLC, and RAWR
- Normalized to MESI
- Racer gets the best from MESI-TSO and VIPS-M
  - TLB consumption mitigated by using virtual caches (as VIPS-M)



Energy consumption

- Energy of TLBs, L1 caches, network, LLC, and RAWR
- Normalized to MESI
- Racer gets the best from MESI-TSO and VIPS-M
  - TLB consumption mitigated by using virtual caches (as VIPS-M)
  - LLC and network consumption of MESI-TSO (runtime synchronization)



Conclusions

- Racer is a novel way of providing TSO consistency
  - ⇒ First efficient, request-response protocol for TSO

Conclusions

- Racer is a novel way of providing TSO consistency
  - ⇒ First efficient, request-response protocol for TSO
  - ⇒ No indirection: supports low-cost virtual caches
  - ⇒ Low area overhead

Conclusions

- Racer is a novel way of providing TSO consistency
  - ⇒ First efficient, request-response protocol for TSO
  - ⇒ No indirection: supports low-cost virtual caches
  - ⇒ Low area overhead
- More about Racer
  - ⇒ Coalesing write-through
  - ⇒ Race prediction
  - ⇒ Distributed RAWR
  - ⇒ OoO cores with speculation