Memory consistency models and synchronizations

Acknowledgements

- Several of the slides in the deck are from Luis Ceze (Washington), Nima Horanmand (Stony Brook), Mark Hill, David Wood, Karu Sankaralingam (Wisconsin), Abhishek Bhattacharjee(Rutgers).

- Development of this course is partially supported by Western Digital corporations.

Food for thought (assume SC)

- Answer the following questions:
  - Initially: all variables zero (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:

Core 0	Core 1
LD x LD y	ST y 1 ST x 1

Food for thought (assume SC)

- Answer the following questions:
  - Initially: all variables zero (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:

Core 0	Core 1
LD x	ST y 1
LD y	ST x 1

How about (1,0)?

Food for thought (assume SC)

- Answer the following questions:
  - Initially: all variables zero (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:

Core 0	Core 1
LD x LD y	ST y 1 ST x 1

How about (1,0)?

- What value pairs can be read by the two loads? (x, y) pairs:

Core 0	Core 1
ST y 1 LD x	ST x 1 LD y

How about (0,0)?

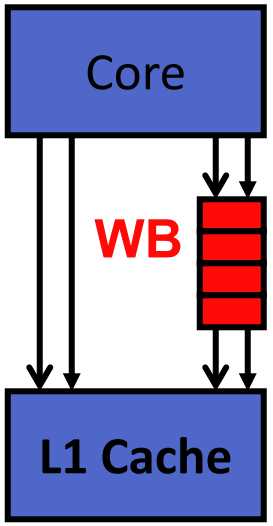
Problems with SC memory model

- Difficult to implement efficiently in hardware
  - Straight-forward implementations of SC dictate:
    - Strict ordering of memory accesses at each processors
    - Essentially precludes most out-of-order CPU benefits

  → Conflicts with common latency-hiding techniques
- Constrains compiler optimizations
  - Disallows code motion, common subexpression elimination, register allocations
- Implementations of SC which tries to extract concurrency of accesses are complex
  - e.g., MIPS R10K
- No commercial processors implement SC today

Constraints of SC: Write buffer

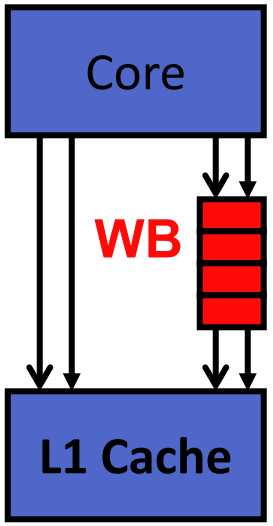
- Why have a write (store buffer) buffer?



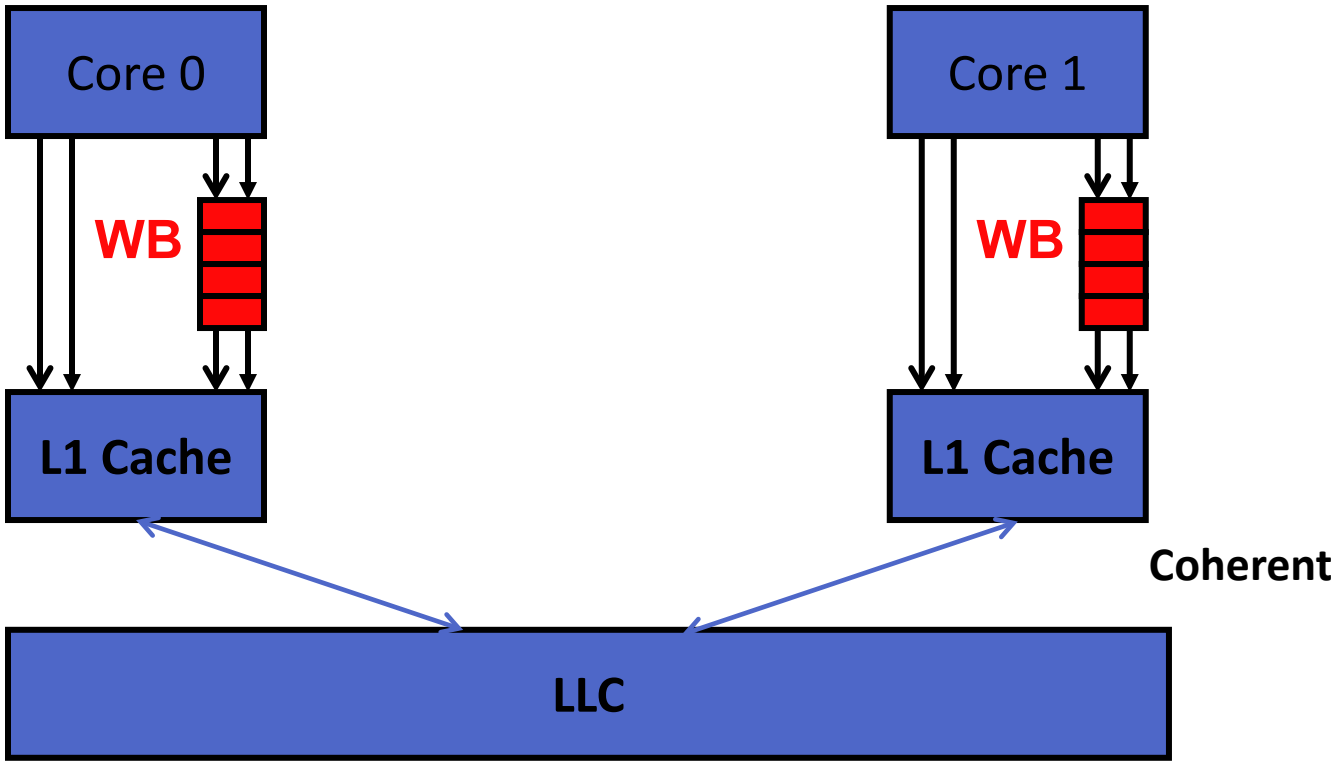
Constraints of SC: Write buffer

- Why have a write (store buffer) buffer?

- Can existence of write buffer break SC?

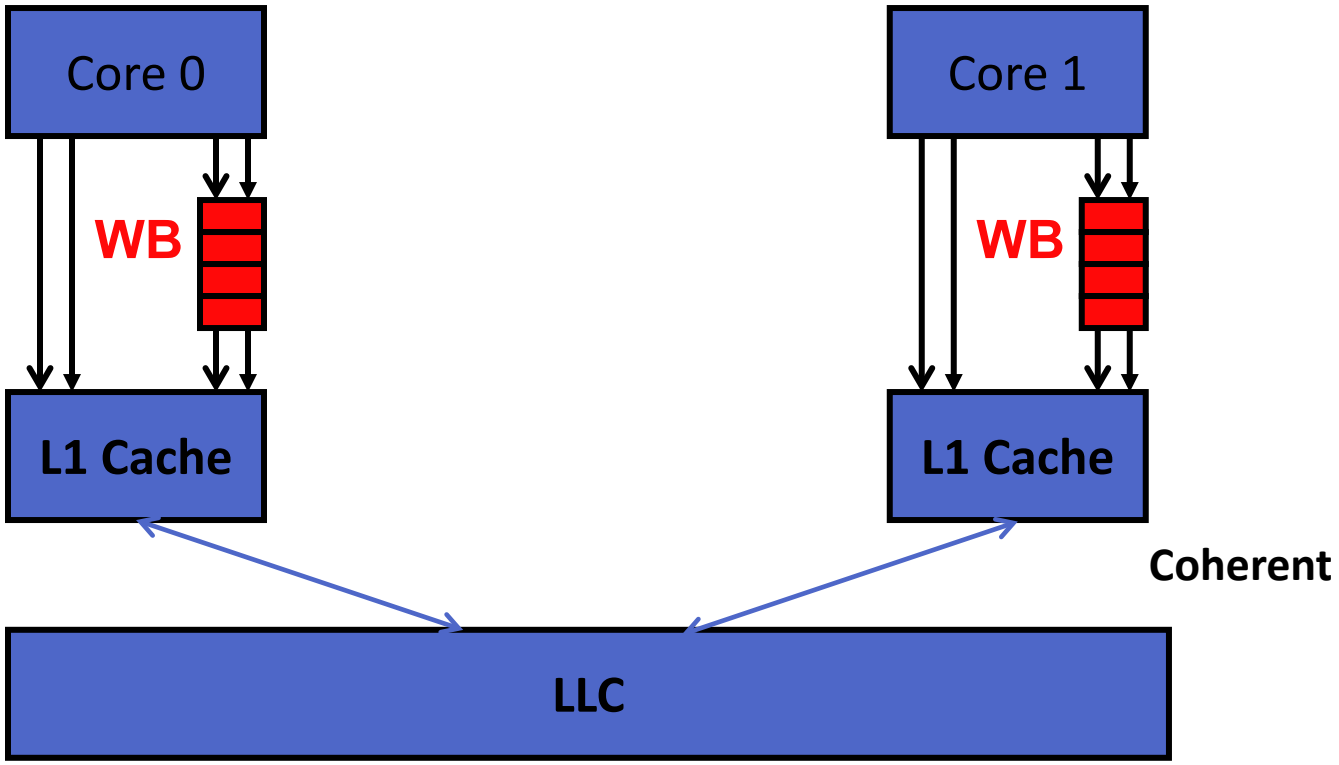


Write buffer breaks SC

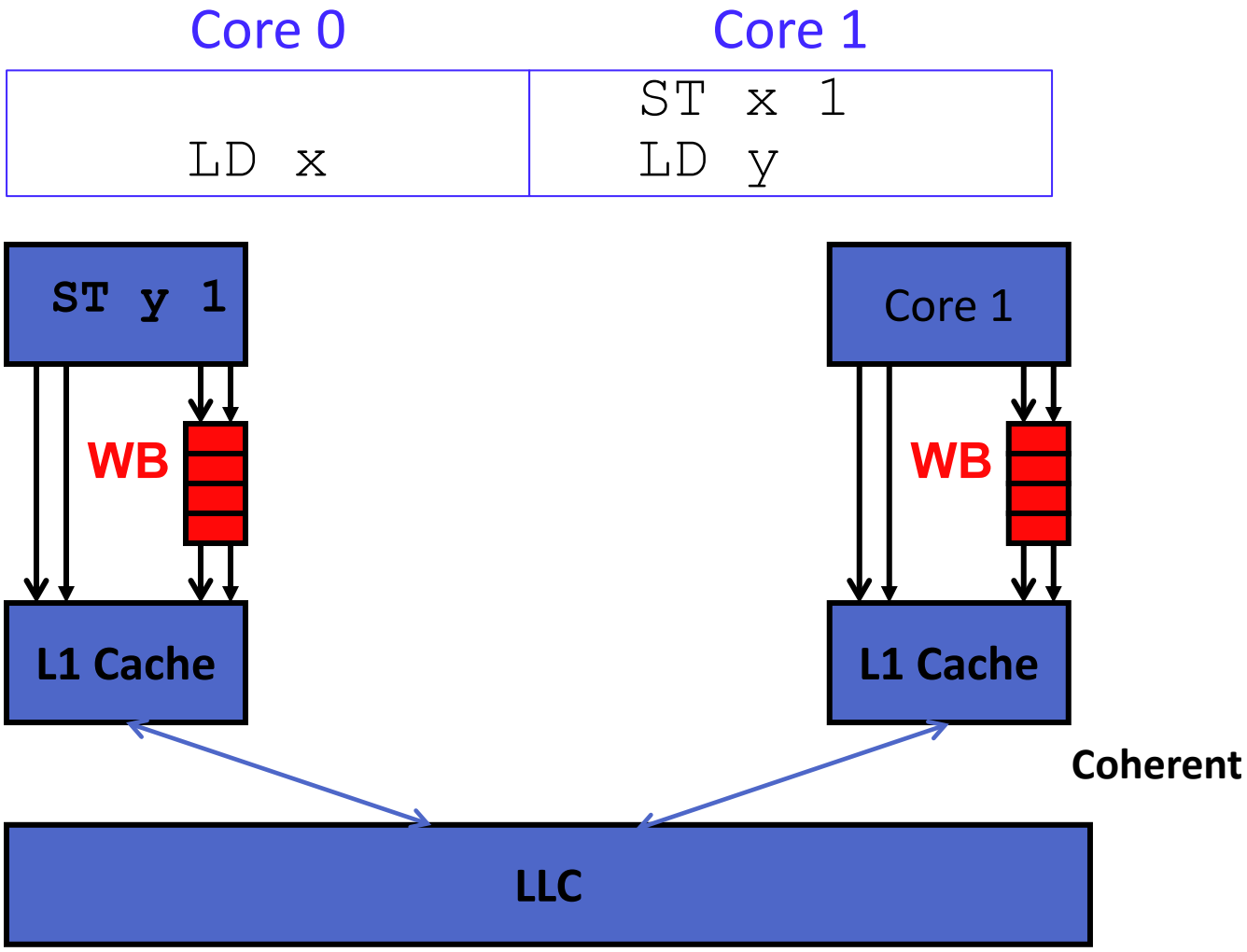


Write buffer breaks SC

Core 0	Core 1
ST y 1	ST x 1
LD x	LD y

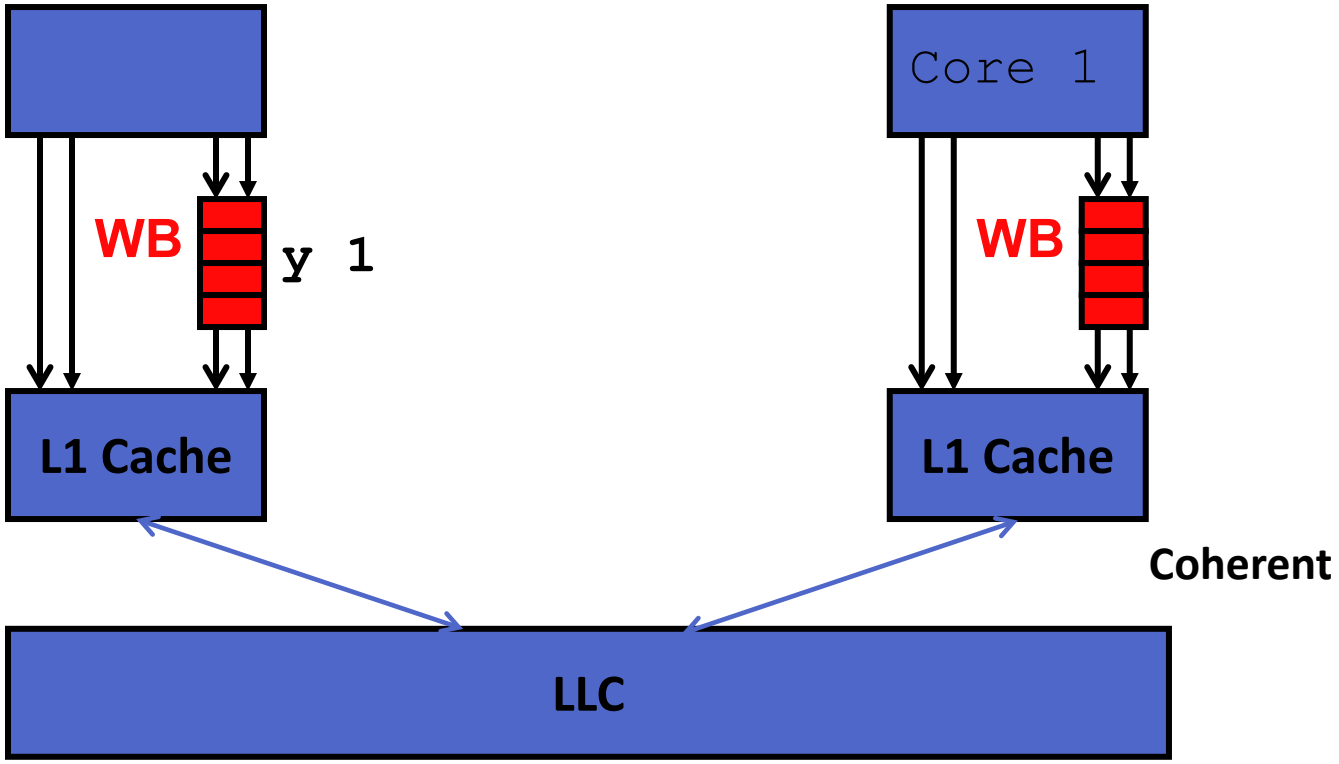


Write buffer breaks SC

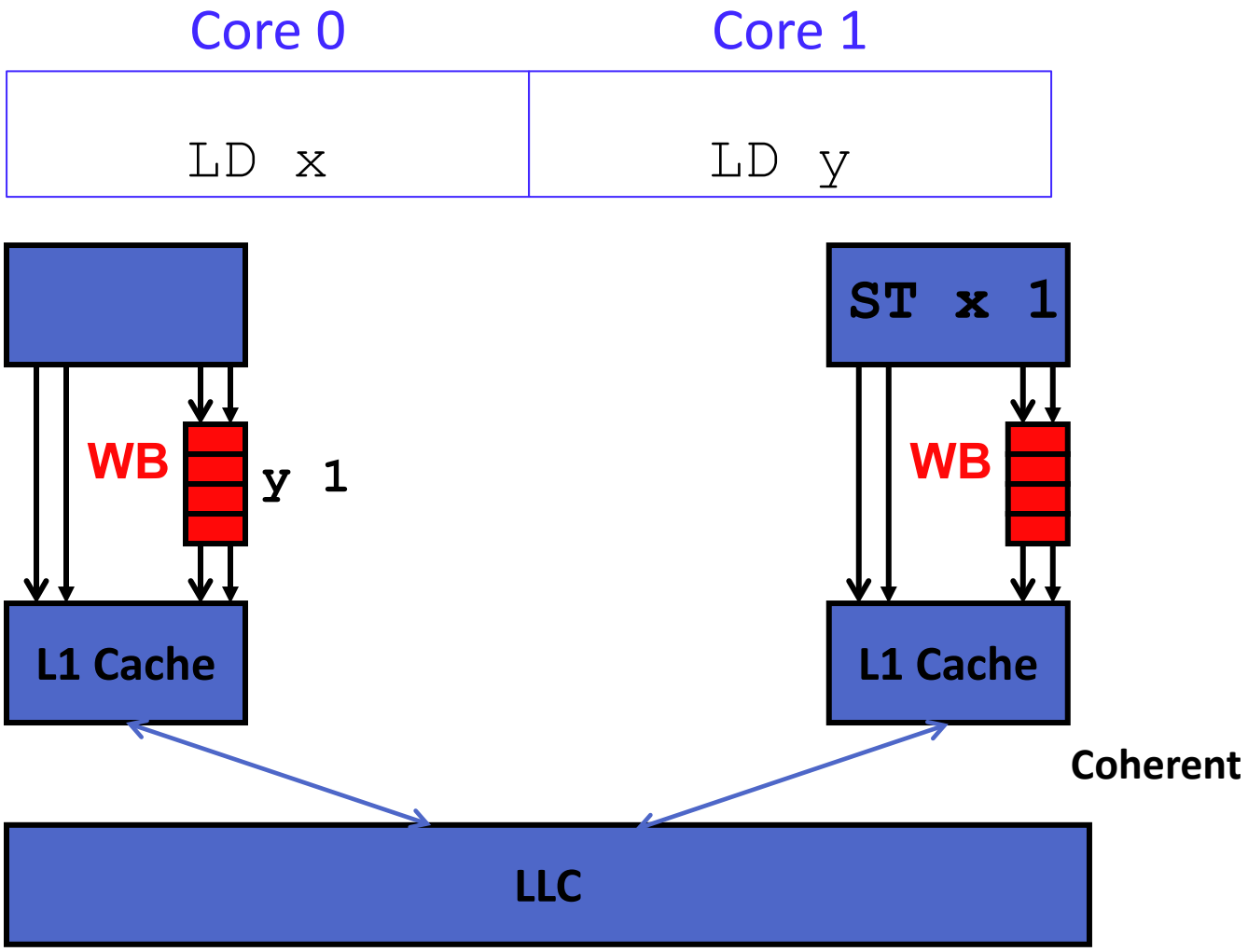


Write buffer breaks SC

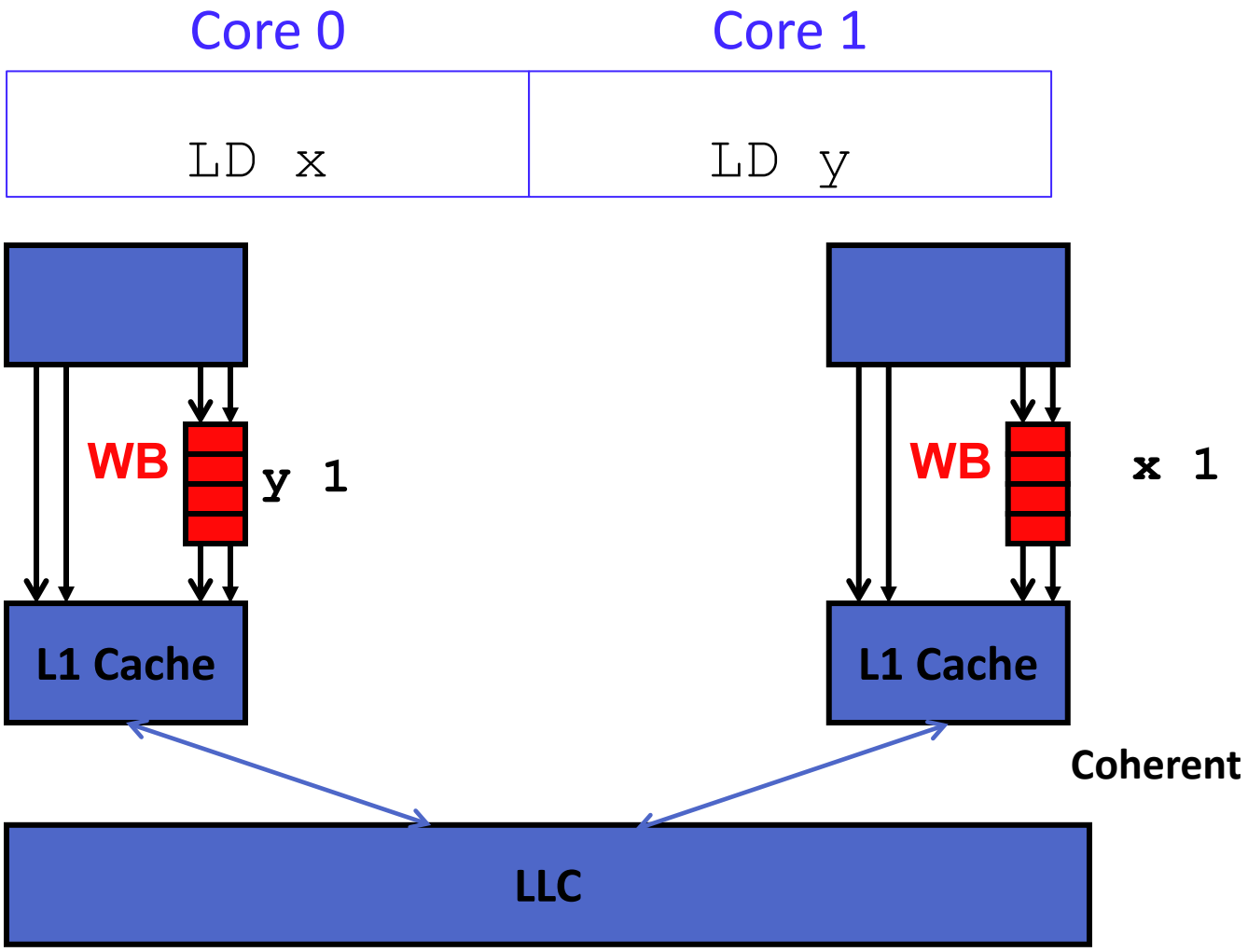
Core 0	Core 1
LD x	ST x 1 LD y



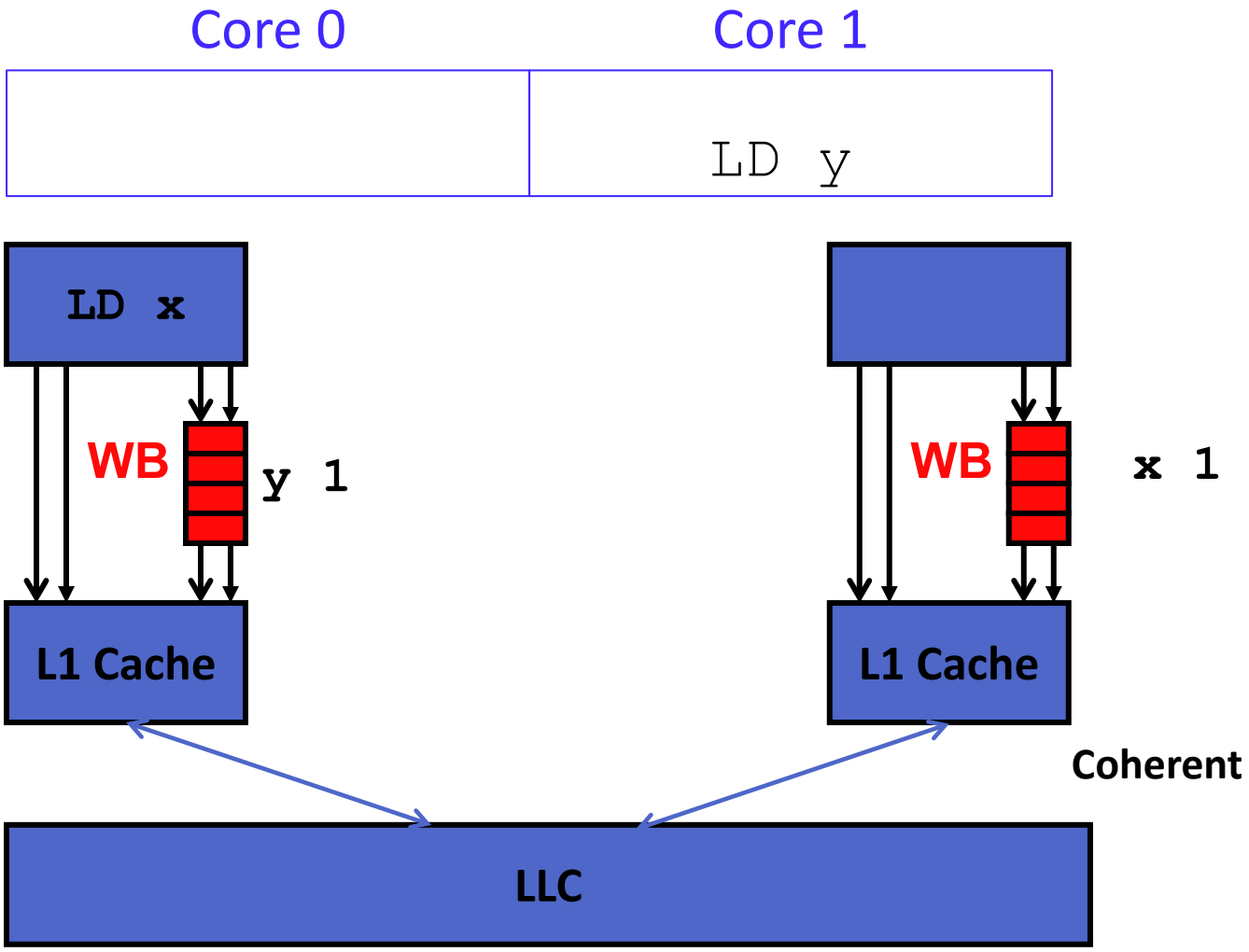
Write buffer breaks SC



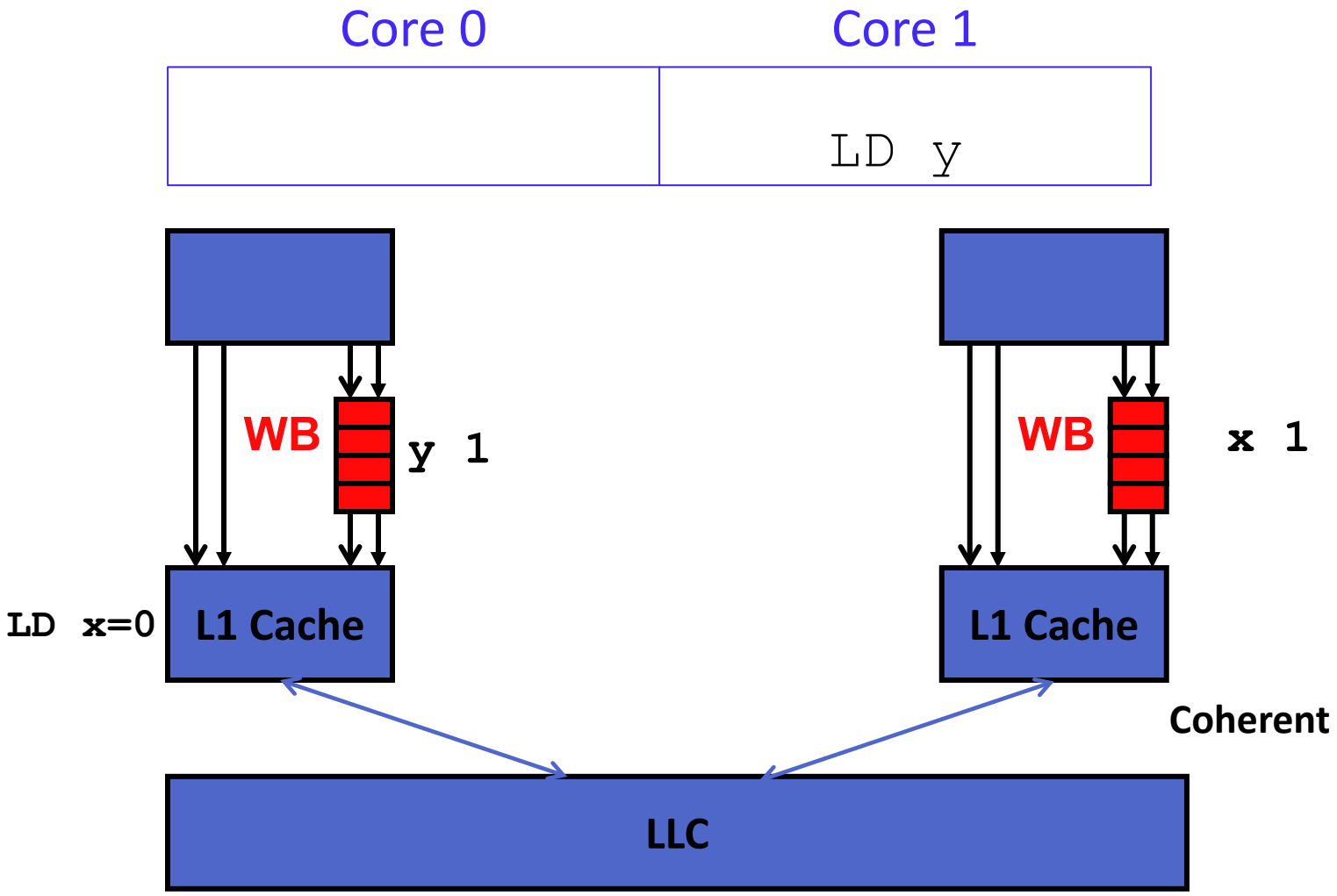
Write buffer breaks SC



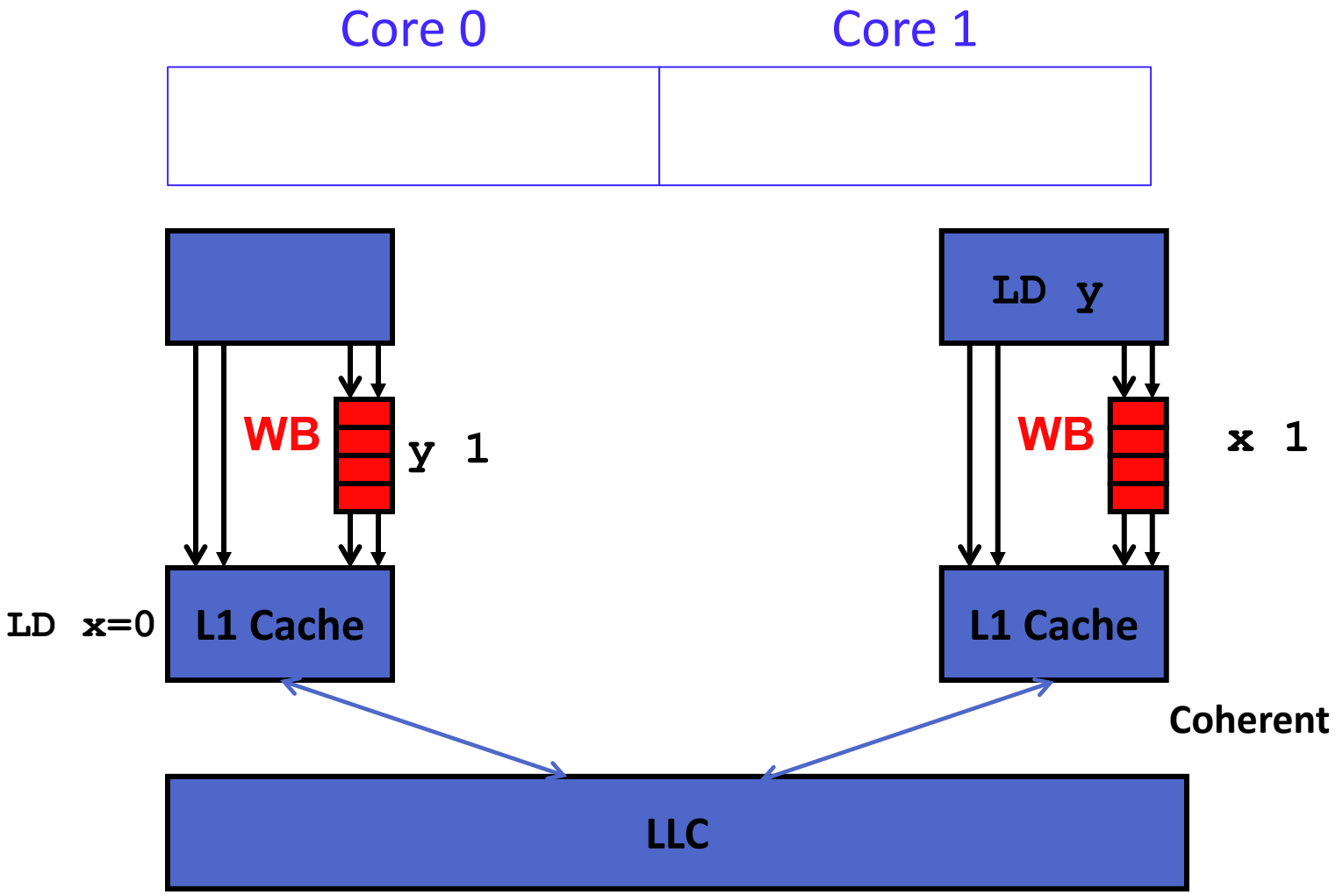
Write buffer breaks SC



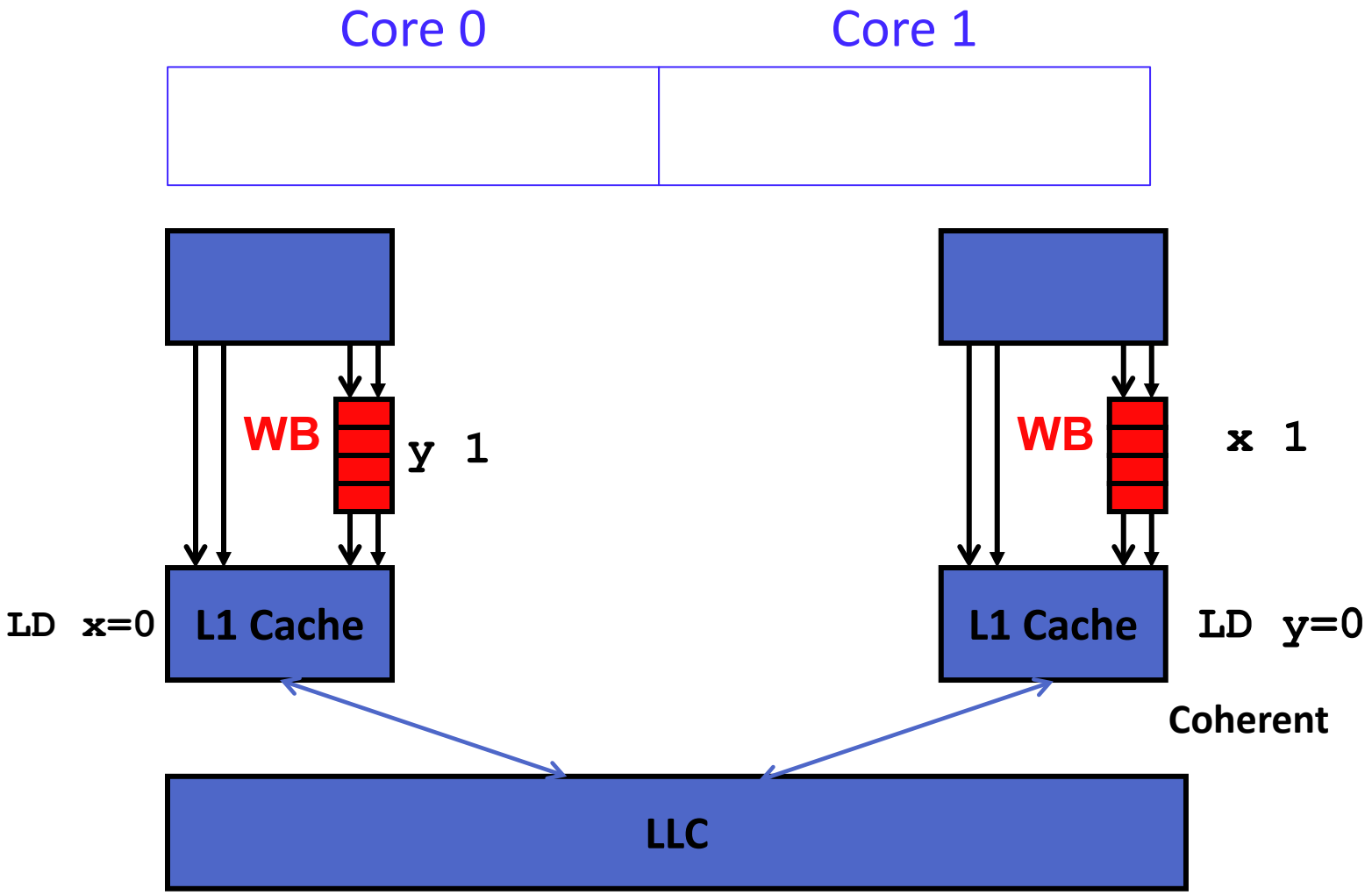
Write buffer breaks SC



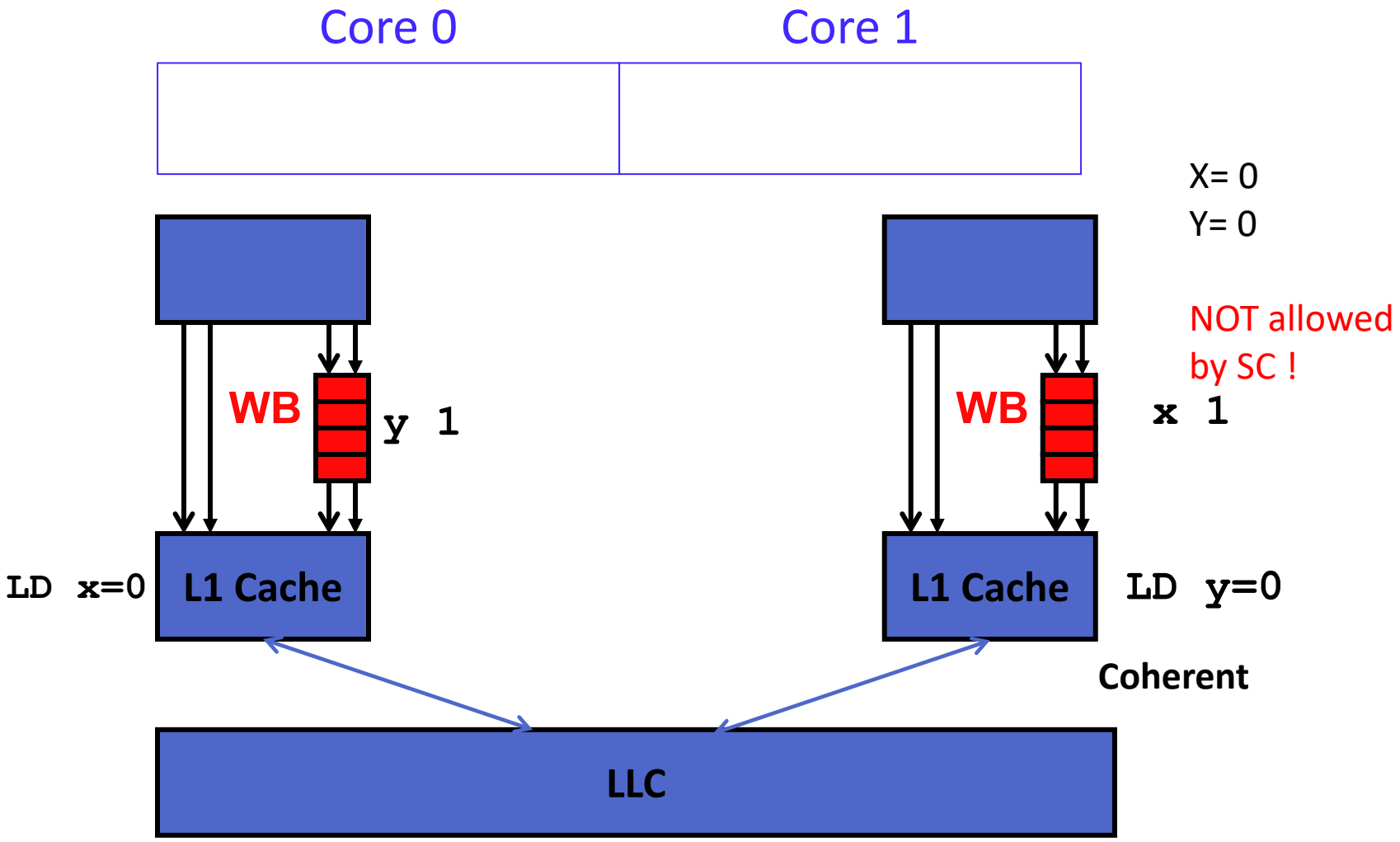
Write buffer breaks SC



Write buffer breaks SC



Write buffer breaks SC



Alternative memory model

- **Total store order (TSO)** memory model

Program order

Earlier operation / Later operation	LD	ST
LD	NO	**YES**
ST	NO	NO

Allowed reordering of load/stores under TSO

Alternative memory model

- **Total store order (TSO)** memory model

Program order

Earlier operation / Later operation	LD	ST
LD	NO	**YES**
ST	NO	NO

Allowed reordering of load/stores under TSO

Remember that reordering allowed only if ST/LD are to different addresses.

Alternative memory model

- **Total store order (TSO)** memory model
  - Implemented by Intel, AMD and Sun/Oracle's SPARC processors
  - It is sometime called processor consistency model

Program order

Earlier operation / Later operation	LD	ST
LD	NO	**YES**
ST	NO	NO

Allowed reordering of load/stores under TSO

Remember that reordering allowed only if ST/LD are to different addresses.

TSO vs. SC

- What performance optimization would TSO allow?
  - (That is not allowed by SC)

TSO vs. SC

- What performance optimization would TSO allow?
  - (That is not allowed by SC)
  - A **FIFO** write buffer
    - Still need to maintain store-to-store order

TSO vs. SC

- What performance optimization would TSO allow?
  - (That is not allowed by SC)
  - A **FIFO** write buffer
    - Still need to maintain store-to-store order
- What is disadvantage of TSO?
  - Some programs will break

What breaks under TSO?

```
                A=0   FLAG=0
C 0                          C 1
ST A 1;                      L1: LD r1 FLAG
ST FLAG 1;                   If  r1 == 0; JMP L1;// spin lock
                                LD r2 A;
```

What breaks under TSO?

```
                A=0   FLAG=0
C 0                          C 1
ST A 1;                      L1: LD r1 FLAG
ST FLAG 1;                   If  r1 == 0; JMP L1;// spin lock
                                LD r2 A;
```

- This will work as expected

What breaks under TSO?

```
                A=0   FLAG=0
C 0                          C 1
ST A 1;                      L1: LD r1 FLAG
ST FLAG 1;                   If  r1 == 0; JMP L1;// spin lock
                                LD r2 A;
```

- This will work as expected
- TSO allows only later load to bypass previous stores to different address

What breaks under TSO?

Core 0	Core 1
ST y 1 LD x	ST x 1 LD y

What breaks under TSO?

Core 0	Core 1
ST y 1 LD x	ST x 1 LD y

- Is x=0, y=0 possible?

What breaks under TSO?

Core 0	Core 1
ST y 1 LD x	ST x 1 LD y

- Is x=0, y=0 possible?
- Yes, because later load to different address can bypass earlier store

What if the programmer wants SC like ordering?

- Special **fence** instructions to explicitly introduce ordering
  - Example, **mfence** instruction in x86/x86-64
  - Programmer needs to insert them manually

What if the programmer wants SC like ordering?

- Special **fence** instructions to explicitly introduce ordering
  - Example, **mfence** instruction in x86/x86-64
  - Programmer needs to insert them manually

Earlier operation / Later operation	LD	ST	mfence
LD	NO	**YES**	**NO**
ST	NO	NO	NO
mfence	NO	NO	NO

Fences to order load/stores

Core 0	Core 1
ST y 1 **mfence** LD x	ST x 1 **mfence** LD y

Fences to order load/stores

Core 0	Core 1
ST y 1 **mfence** LD x	ST x 1 **mfence** LD y

- x=0, y=0 **not** possible anymore → SC compliant

Weak/Relaxed memory models

- What if we want even more performance?
  - TSO does not allow stores to different addresses to be re-ordered
  - TSO does not allow loads to different addresses to be re-ordered

Weak/Relaxed memory models

- What if we want even more performance?
  - TSO does not allow stores to different addresses to be re-ordered
  - TSO does not allow loads to different addresses to be re-ordered
- Weak memory models
  - No ordering guarantees for load/stores
  - Allows non-FIFO, coalescing write buffer
  - Allows load-to-load bypass to different addresses

Weak memory models

Earlier operation / Later operation	LD	ST	fence
LD	**YES**	YES	NO
ST	**YES**	**YES**	NO
fence	NO	NO	NO

Weak memory model

```
                A=0   FLAG=0
C 0                          C 1
ST A 1;                      L1: LD r1 FLAG
ST FLAG 1;                   If  r1 == 0; JMP L1;// spin lock
                                LD r2 A;
```

- Will this work (i.e., r2 gets value 1) ?

Weak memory model

```
                A=0   FLAG=0
C 0                          C 1
ST A 1;                      L1: LD r1 FLAG
ST FLAG 1;                   If  r1 == 0; JMP L1;// spin lock
                                LD r2 A;
```

- Will this work (i.e., r2 gets value 1) ?
- How to fix this?

Weak memory model

```
                A=0   FLAG=0
C 0                          C 1
ST A 1;                      L1: LD r1 FLAG
fence;                       If  r1 == 0; JMP L1;// spin lock
ST FLAG 1;                      fence;
                                LD r2 A;
```

- Would this work (i.e., r2 gets value 1) ?
- How to fix this?

Locking/Mutual exclusion

Implementing a simple lock

- Shared counter/sum update example
  - Use a mutex variable for **mutual exclusion**
  - Only one processor can own the mutex
    - **Many processors may call lock(), but only one will succeed** (others block)
    - The winner executes the critical section, then calls unlock() to release the mutex
    - Now one of the others gets it, etc.
  - But how do we implement a mutex?
    - As a shared variable (1 – owned, 0 –free)
- *How would you implement it?*

Implementing a simple lock

- Shared counter/sum update example
  - Use a mutex variable for **mutual exclusion**
  - Only one processor can own the mutex
    - **Many processors may call lock(), but only one will succeed** (others block)
    - The winner executes the critical section, then calls unlock() to release the mutex
    - Now one of the others gets it, etc.
  - But how do we implement a mutex?
    - As a shared variable (1 – owned, 0 –free)
- *How would you implement it?*

```
1. while (lock_var != 0);!
2. lock_var = 1;!
```

Locking

- Releasing a mutex is easy
  - Just set it to 0

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - What invariant do we need?

Thread 1	Thread 2
Line1: lock_var == 0	
… descheduled …	Line 1: lock_var == 0
	Line 2: Sets lock_var = 1 (Thinks it has the lock.)
Line 2: Sets lock_var = 1 (Thinks it has the lock)	… descheduled …

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become ∪
  - But when it does, others will see it, too
  - What invariant do we need?

```
1. while (lock_var != 0);!
2. lock_var = 1;!
```

Thread 1	Thread 2
Line1: lock_var == 0	
… descheduled …	Line 1: lock_var == 0
	Line 2: Sets lock_var = 1 (Thinks it has the lock.)
Line 2: Sets lock_var = 1 (Thinks it has the lock)	… descheduled …

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - Need a way to ***atomically*** see that the mutex is 0 ***and*** set it to 1 (i.e., needs read-modify-write)
  - *How?*

Atomic read -modify -write instructions

- Atomic compare and exchange instruction
  - e.g., In x86-64 ISA, **cmxcg op1 (address), op2** compares value in the address op1 with value in a specific register **eax.**
  - If the values are equal then writes op2 to address pointed by op1 and sets zero flag **ZF** =1.
  - If the values are not equal then sets conditional flag **ZF=0**

Atomic read -modify -write instructions

- Atomic compare and exchange instruction
  - e.g., In x86-64 ISA, **cmxcg op1 (address), op2** compares value in the address op1 with value in a specific register **eax.**
  - If the values are equal then writes op2 to address pointed by op1 and sets zero flag **ZF** =1.
  - If the values are not equal then sets conditional flag **ZF=0**
  - Many similar atomic RMW instructions such as **FetchAndAdd**, **TestAndSet**

Atomic read -modify -write instructions

- Atomic compare and exchange instruction
  - e.g., In x86-64 ISA, **cmxcg op1 (address), op2** compares value in the address op1 with value in a specific register **eax.**
  - If the values are equal then writes op2 to address pointed by op1 and sets zero flag **ZF** =1.
  - If the values are not equal then sets conditional flag **ZF=0**
  - Many similar atomic RMW instructions such as **FetchAndAdd**, **TestAndSet**

- How do you create lock/mutex with cmpxcg?

Atomic RMW for locking

```
spin_lock:
    xorl   %ecx, %ecx
    incl   %ecx          # Locked value = 1
lock_retry:
    xorl   %eax, %eax    # expected value of the lock= 0
    cmpxchgl (lock_addr), %ecx  # attempt to acquire lock
    jnz    spin_lock_retry # retry if failed (i.e., flag ZF not set)
    ret

spin_unlock:
    movl   $0,  (lock_addr)    # lock release
    ret
```

RMWs are not reordered in TSO

Earlier operation / Later operation	LD	ST	mfence	RMW
LD	NO	**YES**	NO	NO
ST	NO	NO	NO	NO
mfence	NO	NO	NO	NO
RMW	NO	NO	NO	NO

Allowed reordering of load/stores to different addresses under TSO

RMWs can be reordered in *some* weak models

Earlier operation / Later operation	LD	ST	fence	RMW
LD	**YES**	YES	NO	YES
ST	**YES**	**YES**	NO	YES
fence	NO	NO	NO	NO
RMW	YES	YES	NO	NO

Allowed reordering of load/stores to different addresses under weaker memory models