Non-Speculative Load→Load Reordering in TSO[1]

Alberto Ros

Universidad de Murcia

October 17th, 2017

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

Outline

Program order

- Programmer intuition: instructions execute in the order they appear in the program

Thread 1

```
$r0 = X;   // load
$r1 = Y;   // load
```

Program order

- Programmer intuition: instructions execute in the order they appear in the program

Thread 1

```
$r0 = X;   // load
$r1 = Y;   // load
```

- What happens if the order changes?

Thread 1

```
$r1 = Y;   // load
$r0 = X;   // load
```

Program order

- Programmer intuition: instructions execute in the order they appear in the program

Thread 1

```
$r0 = X;   // load
$r1 = Y;   // load
```

Thread 2

```
Y = 1;   // store
X = 1;   // store
```

- What happens if the order changes?

Thread 1

```
$r1 = Y;   // load
$r0 = X;   // load
```

Thread 2

```
Y = 1;   // store
X = 1;   // store
```

Total Store Order (TSO)

- The memory consistency model defines the behavior of the programs
  - In particular, the behavior of the memory operations: load and store

Total Store Order (TSO)

- The memory consistency model defines the behavior of the programs
  - In particular, the behavior of the memory operations: load and store
- x86 processors (Intel, AMD) implement a Total Store Order (TSO)

TSO rules

- load→load
- store→store
- load→store

TOTAL STORE ORDER (TSO)

- The memory consistency model defines the behavior of the programs
  - In particular, the behavior of the memory operations: load and store
- x86 processors (Intel, AMD) implement a Total Store Order (TSO)

TSO RULES

- load→load
- store→store[2]
- load→store

[2] A. Ros and S. Kaxiras, "Racer: TSO Consistency via Race Detection", MICRO, 2016.

TOTAL STORE ORDER (TSO)

- The memory consistency model defines the behavior of the programs
  - In particular, the behavior of the memory operations: load and store
- x86 processors (Intel, AMD) implement a Total Store Order (TSO)

TSO RULES

- load→load
- store→store[2]
- load→store

- This talk focus on the load→load order

[2] A. Ros and S. Kaxiras, "Racer: TSO Consistency via Race Detection", MICRO, 2016.

POSSIBLE RESULTS UNDER LOAD→LOAD (E.G. TSO)

INITIALLY X=0, Y=0

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

POSSIBLE RESULTS UNDER LOAD→LOAD (E.G. TSO)

INITIALLY X=0, Y=0

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

SIX POSSIBLE INTERLEAVINGS AND VALUES FOR ($R0, $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 if load→load & store→store

RELAXING LOAD→LOAD

INITIALLY X=0, Y=0

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

SIX POSSIBLE INTERLEAVINGS AND VALUES FOR ($R0, $R1)

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 by relaxing load→load

LOAD→LOAD REORDERING

- Waiting for a load to finish to start the execution of the next load is very inefficient

LOAD→LOAD REORDERING

- Waiting for a load to finish to start the execution of the next load is very inefficient
- High-performance processors execute multiple load operations simultaneously
  - Memory level parallelism
- Load operations can execute out of order
- This is correct for single-core processors

LOAD→LOAD REORDERING

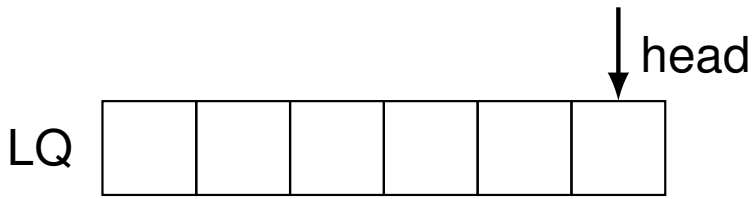
- Executing load operations out of order can relax the load→load order

LOAD→LOAD REORDERING

- Executing load operations out of order can relax the load→load order

Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6

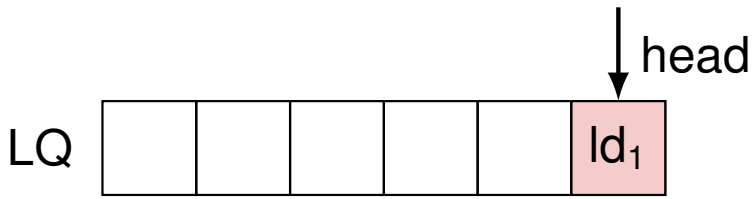


LOAD→LOAD REORDERING

- Executing load operations out of order can relax the load→load order

Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6

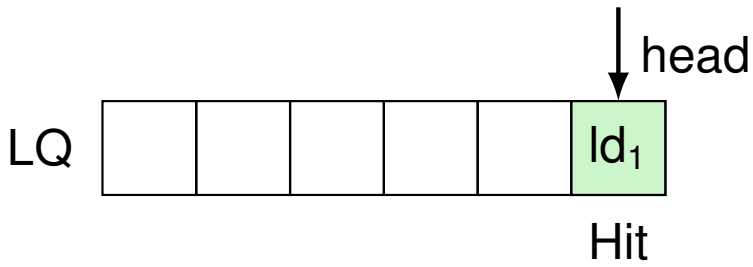


LOAD→LOAD REORDERING

- Executing load operations out of order can relax the load→load order

Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6

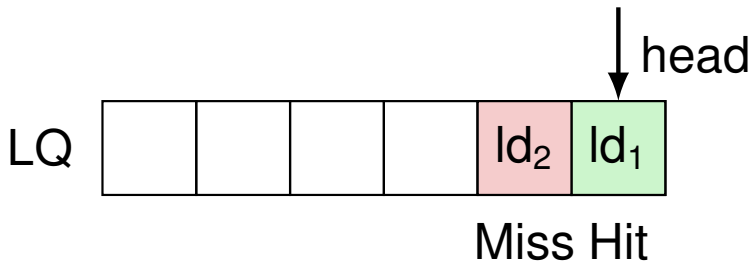


LOAD→LOAD REORDERING

- Executing load operations out of order can relax the load→load order

Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6

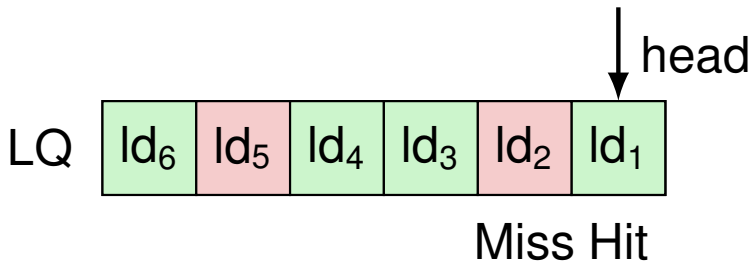


LOAD→LOAD REORDERING

- Executing load operations out of order can relax the load→load order

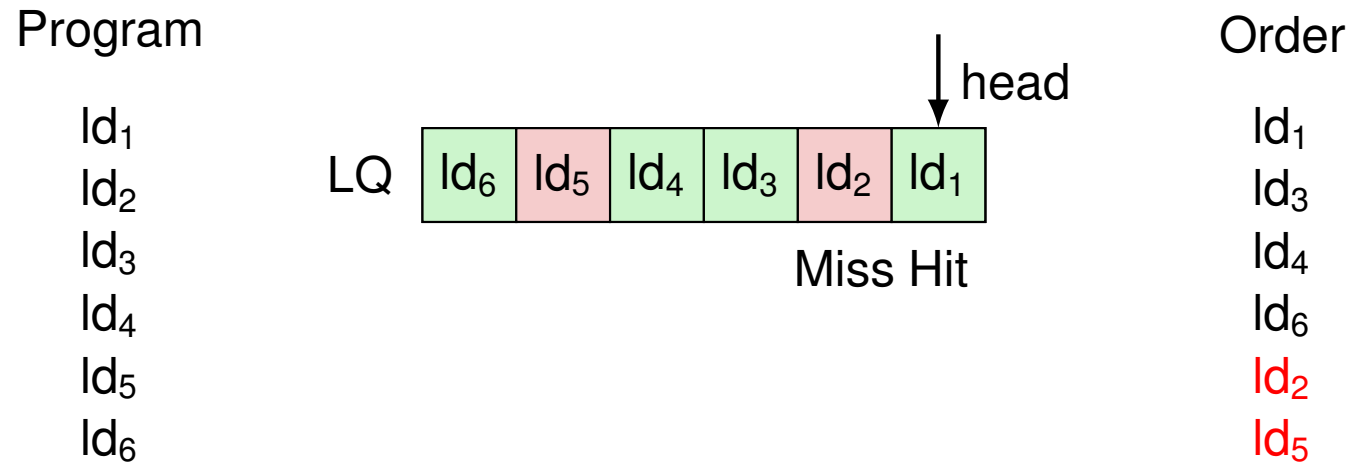
Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6



LOAD→LOAD REORDERING

- Executing load operations out of order can relax the load→load order



LOAD→LOAD REORDERING

- In multicore processors reordering loads can affect the expected result
- It is necessary to always maintain the load→load order?

LOAD→LOAD REORDERING

- In multicore processors reordering loads can affect the expected result
- It is necessary to always maintain the load→load order?

INITIALLY X=0, Y=0

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

/* (1,0) not allowed */
```

LOAD→LOAD REORDERING

- In multicore processors reordering loads can affect the expected result
- It is necessary to always maintain the load→load order?

INITIALLY X=0, Y=0

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

/* (1,0) not allowed */
```

POSSIBLE EXECUTION

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

/* (0, 0) allowed */
```

LOAD→LOAD REORDERING

- In multicore processors reordering loads can affect the expected result
- It is necessary to always maintain the load→load order?

INITIALLY X=0, Y=0

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

`/* (1,0) not allowed */`

POSSIBLE EXECUTION

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

`/* (1, 0) not allowed */`

LOAD→LOAD REORDERING

- In multicore processors reordering loads can affect the expected result
- It is necessary to always maintain the load→load order?

INITIALLY X=0, Y=0

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

/* (1,0) not allowed */
```

POSSIBLE EXECUTION

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

/* (1, 0) not allowed */
```

- No, if the other cores do not see the reordering

RELAXING LOAD→LOAD

INITIALLY X=0, Y=0

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

SIX POSSIBLE INTERLEAVINGS AND VALUES FOR ($R0, $R1)

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)

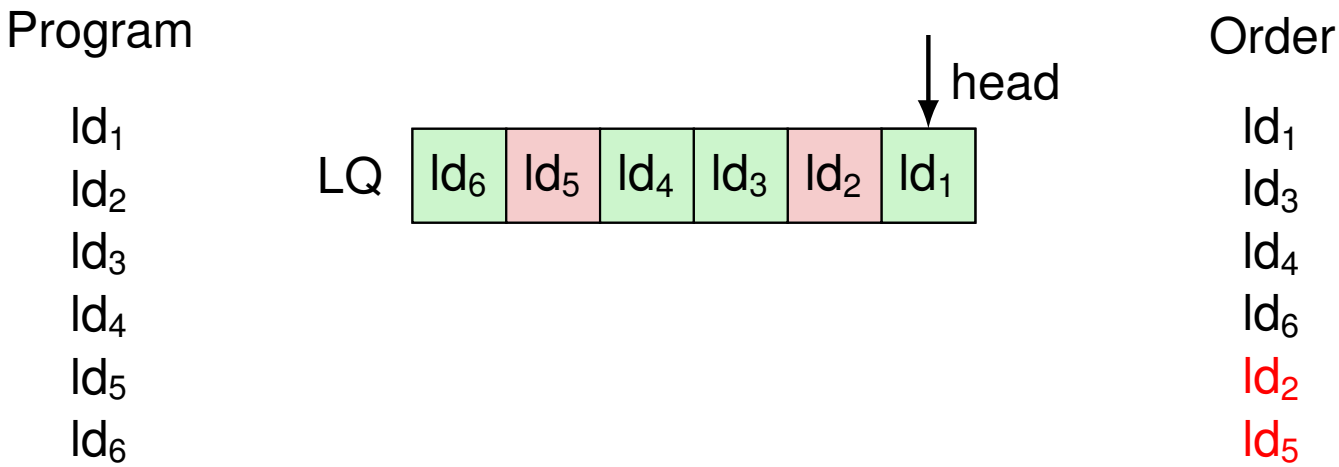
- (1,0) is possible by relaxing load→load

LOAD→LOAD SPECULATION

- Solution: To allow speculative load→load reordering
- Some definitions: performed, ordered, source of speculation (SoS)

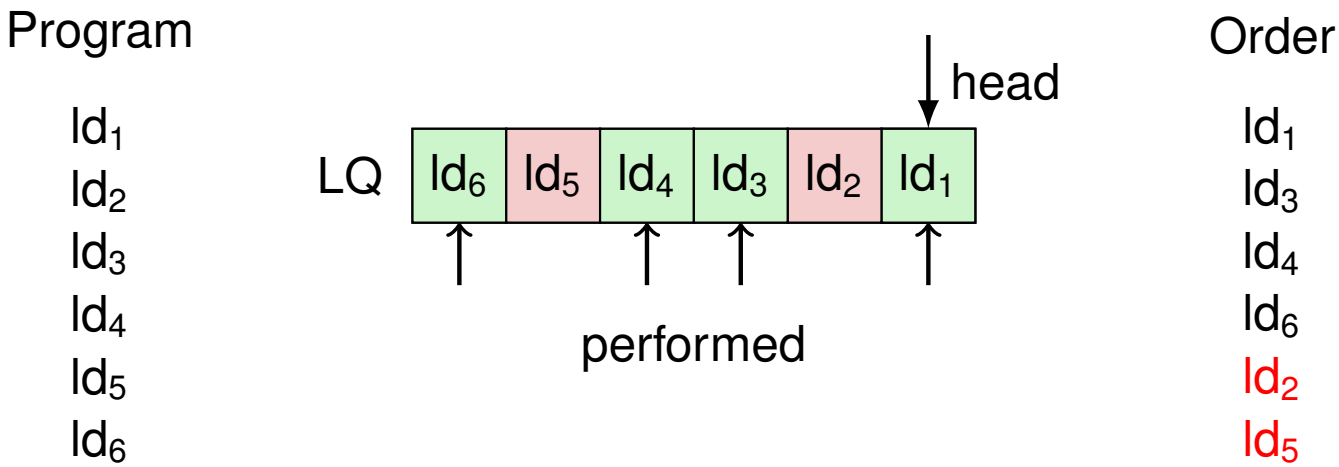
LOAD→LOAD SPECULATION

- Solution: To allow speculative load→load reordering
- Some definitions: performed, ordered, source of speculation (SoS)



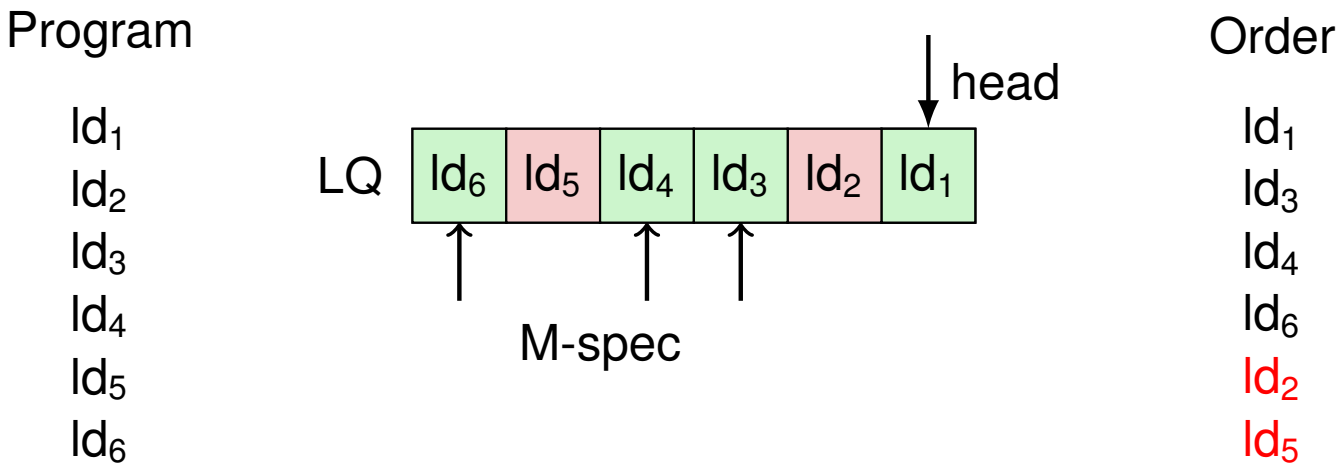
Load→load speculation

- Solution: To allow speculative load→load reordering
- Some definitions: performed, ordered, source of speculation (SoS)



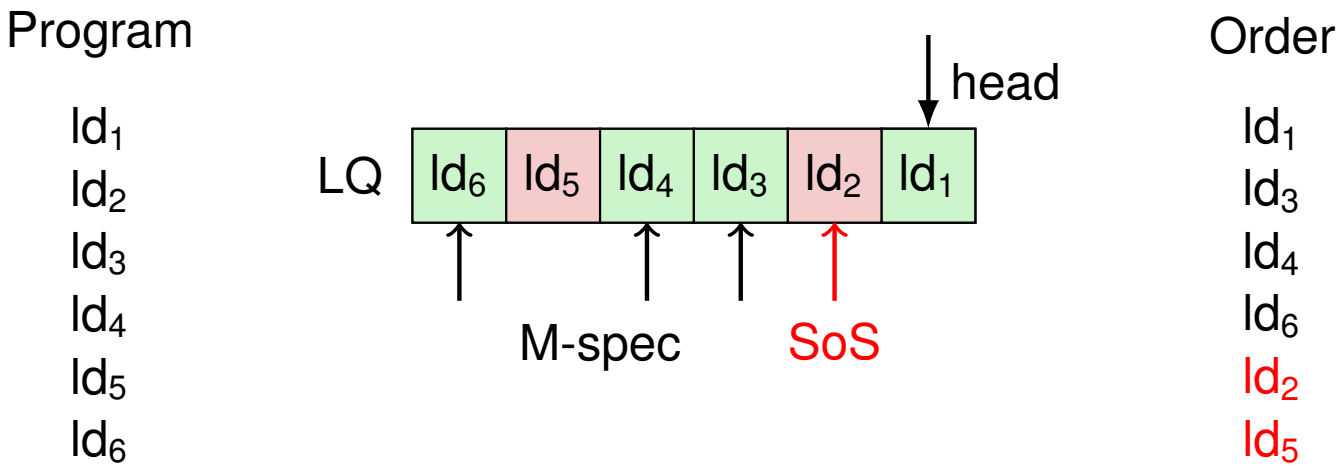
LOAD→LOAD SPECULATION

- Solution: To allow speculative load→load reordering
- Some definitions: performed, ordered, source of speculation (SoS)



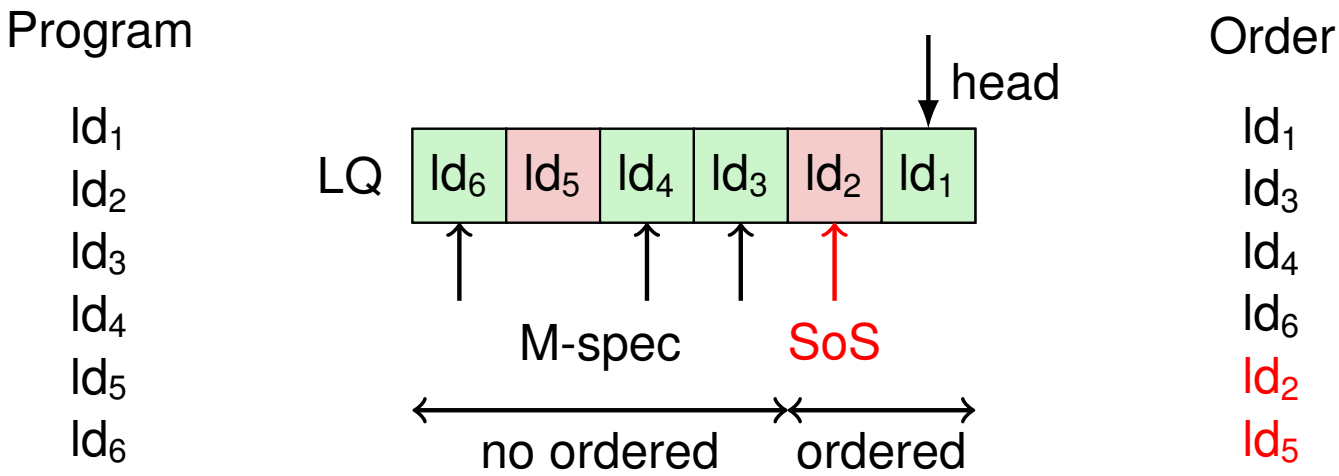
Load→load speculation

- Solution: To allow speculative load→load reordering
- Some definitions: performed, ordered, source of speculation (SoS)



Load→load speculation

- Solution: To allow speculative load→load reordering
- Some definitions: performed, ordered, source of speculation (SoS)



SQUASH AND RE-EXECUTE UPON INVALIDATION

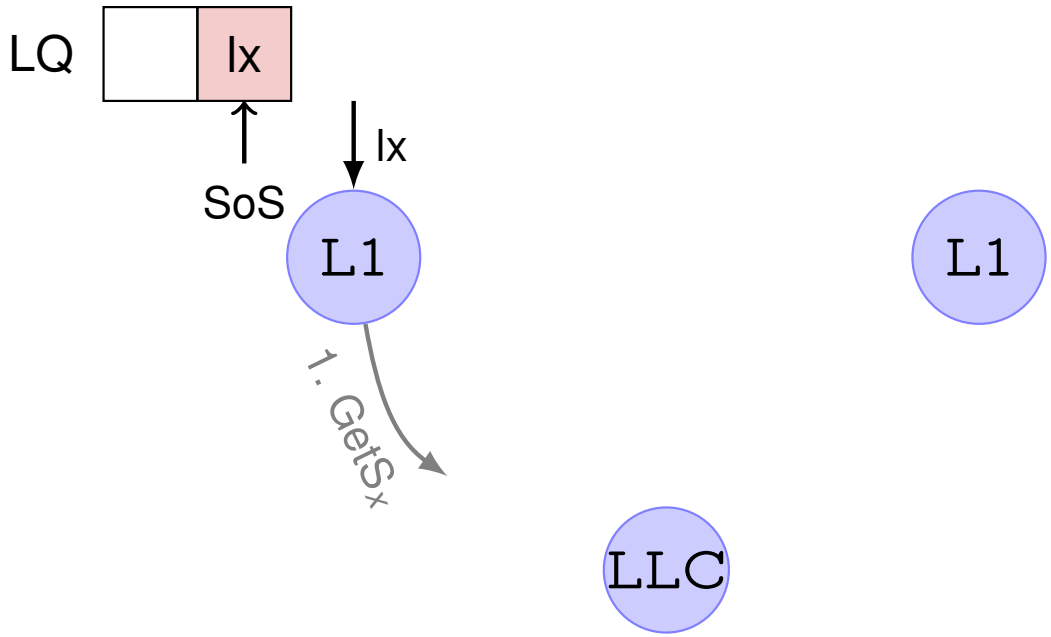
- Current multicore avoid incorrect results
  - With the help of the cache coherence protocol

```
ly
        sy
        sx
lx

  (1,0)
```

SQUASH AND RE-EXECUTE UPON INVALIDATION

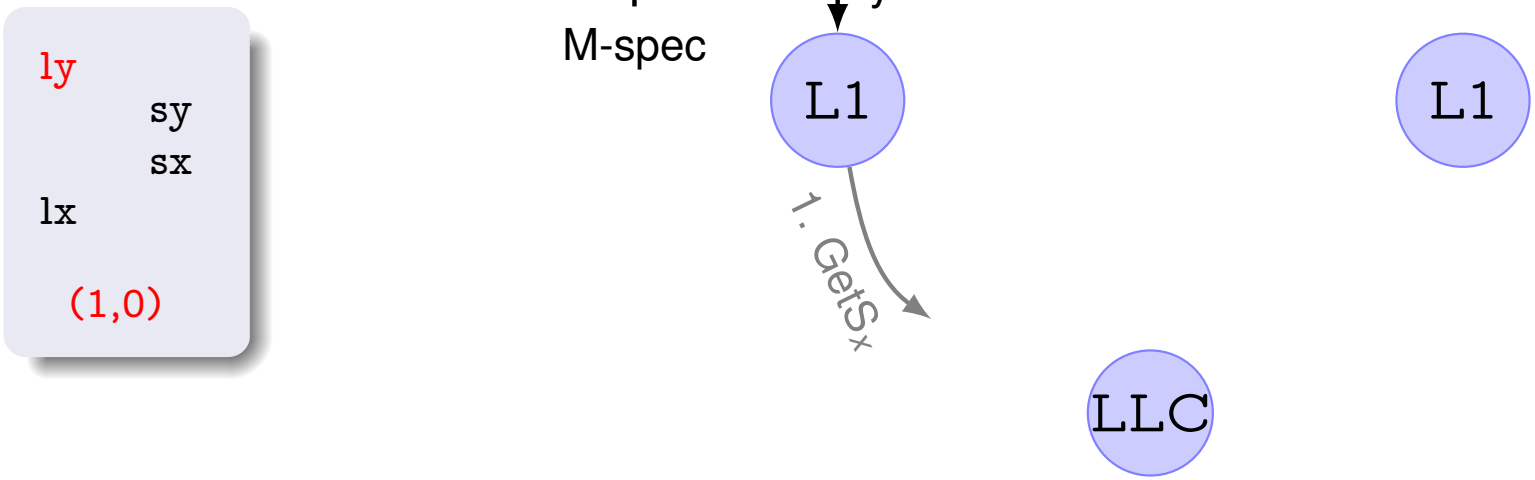
- Current multicore avoid incorrect results
  - With the help of the cache coherence protocol

```
ly
        sy
        sx
lx

  (1,0)
```



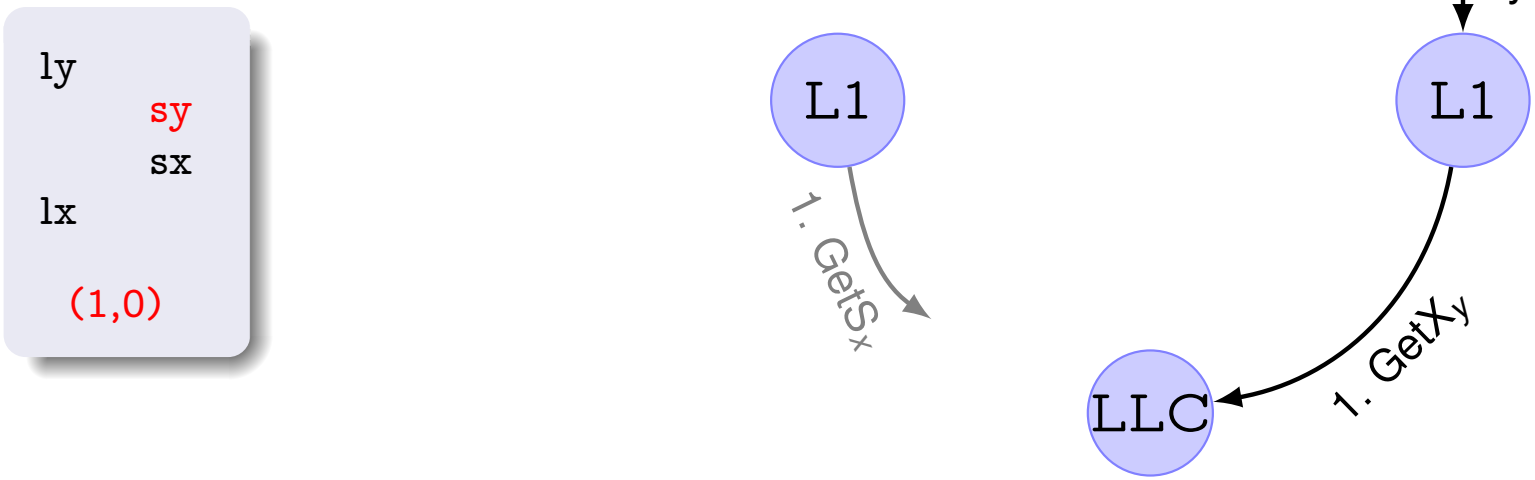
SQUASH AND RE-EXECUTE UPON INVALIDATION

- Current multicore avoid incorrect results
  - With the help of the cache coherence protocol



SQUASH AND RE-EXECUTE UPON INVALIDATION

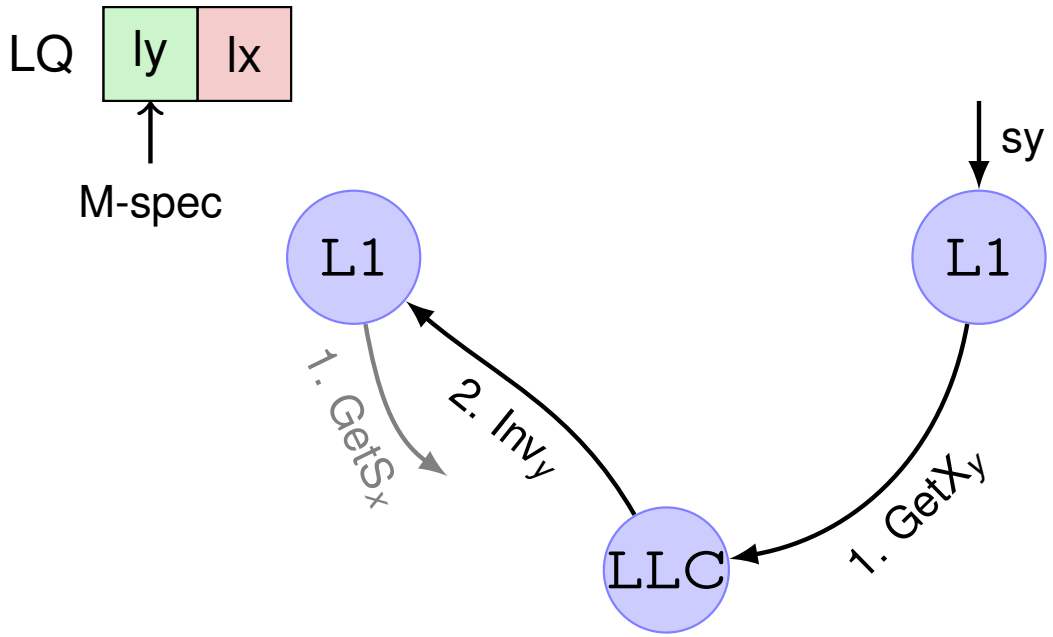
- Current multicore avoid incorrect results
  - With the help of the cache coherence protocol



SQUASH AND RE-EXECUTE UPON INVALIDATION

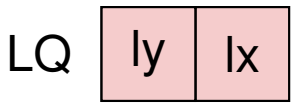
- Current multicore avoid incorrect results
  - With the help of the cache coherence protocol

```
ly
        sy
        sx
lx

  (1,0)
```

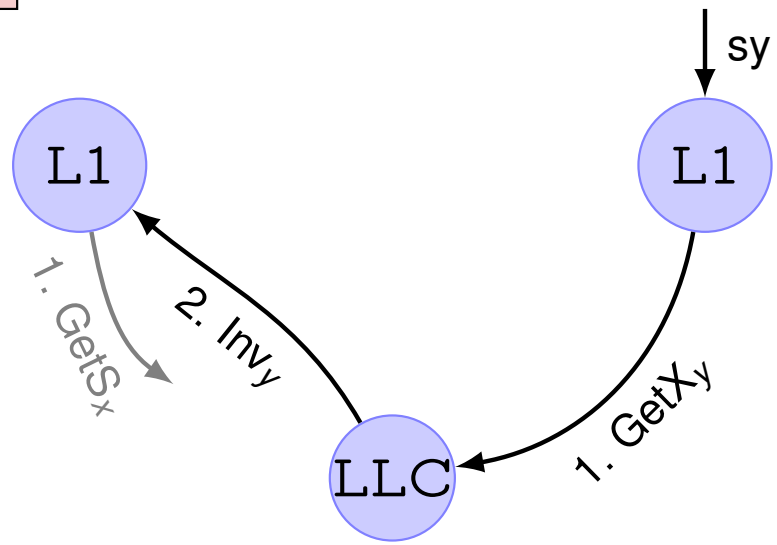


SQUASH AND RE-EXECUTE UPON INVALIDATION

- Current multicore avoid incorrect results
  - With the help of the cache coherence protocol
  - Squashing and re-executing on remote writes



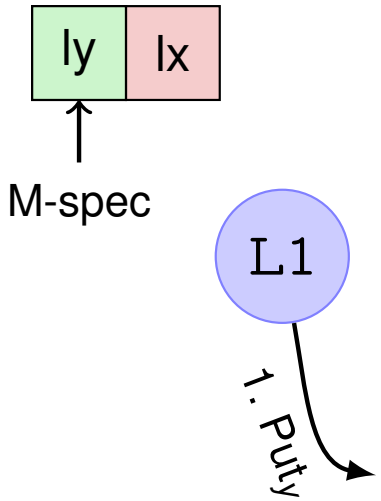

```
ly
        sy
        sx
lx

  (1,0)
```



SQUASH AND RE-EXECUTE UPON EVICTIONS

- What happens when a cache block loaded by an M-spec load is evicted?
  - If the directory stops tracking the block, the M-spec load will not receive an invalidation

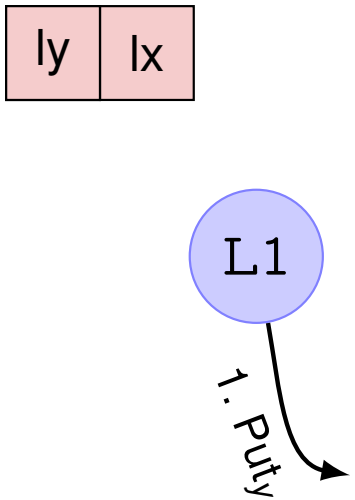
```
ly
        sy
        sx
lx

 (1,0)
```



SQUASH AND RE-EXECUTE UPON EVICTIONS

- What happens when a cache block loaded by an M-spec load is evicted?
  - If the directory stops tracking the block, the M-spec load will not receive an invalidation

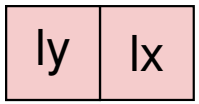
```
ly
        sy
        sx
lx

 (1,0)
```

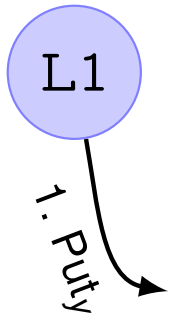


SQUASH AND RE-EXECUTE UPON EVICTIONS

- What happens when a cache block loaded by an M-spec load is evicted?
  - If the directory stops tracking the block, the M-spec load will not receive an invalidation
- Solution: Squash and re-execute upon evictions
  - This impacts the performance of sequential applications!!!

```
ly
        sy
        sx
lx

 (1,0)
```





Problems of speculation

- Memory-related speculation is the current solution to issue loads out of order while keeping the load→load order

Problems of speculation

- Memory-related speculation is the current solution to issue loads out of order while keeping the load→load order
- Why is good?
  - Because squashing is not frequent!

Problems of speculation

- Memory-related speculation is the current solution to issue loads out of order while keeping the load→load order
- Why is good?
  - Because squashing is not frequent!
- Why is bad?
  - Because speculative loads hold critical resources (LQ, RoB)
  - Because the processor needs to keep continuously the rollback path

Problems of speculation

- Memory-related speculation is the current solution to issue loads out of order while keeping the load→load order
- Why is good?
  - Because squashing is not frequent!
- Why is bad?
  - Because speculative loads hold critical resources (LQ, RoB)
  - Because the processor needs to keep continuously the rollback path

Question

Can we execute loads out of order without speculation and guaranteeing load→load?

Outline

WritersBlock achievement

- Current multicore processors speculatively execute loads out of order
- If a conflict happens, loads are squashed and re-executed

WritersBlock achievement

- Current multicore processors speculatively execute loads out of order
- If a conflict happens, loads are squashed and re-executed

WritersBlock

Makes possible removing this speculation, executing loads out of order, and making that an executed load is never squashed because of the consistency model

How?

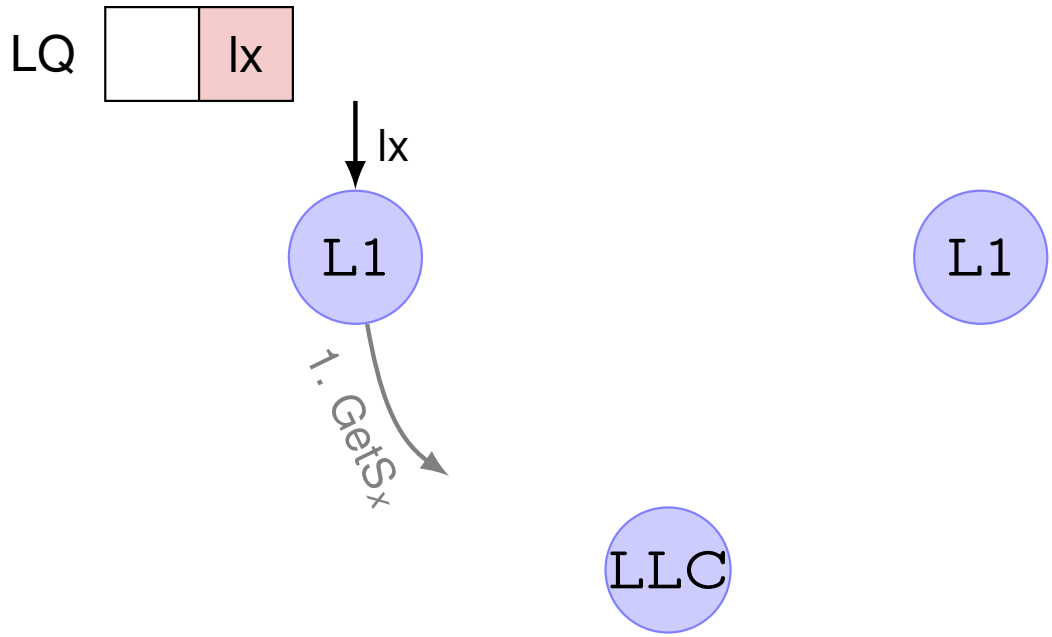
- With the help of the cache coherence protocol

```
ly
      sy
      sx
lx

  (1,0)
```

How?

- With the help of the cache coherence protocol

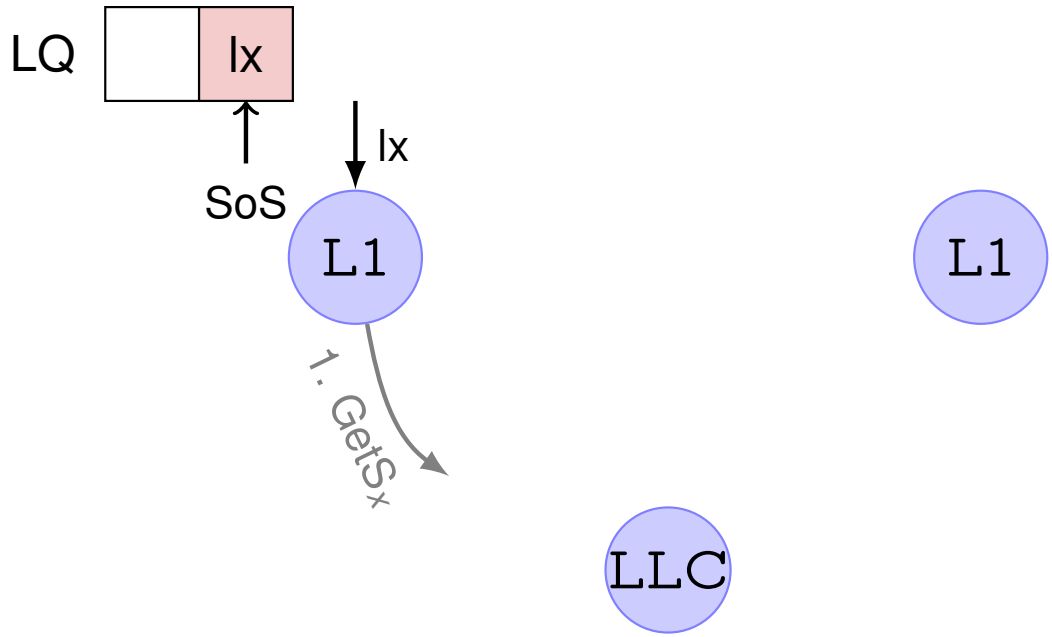
```
ly
        sy
        sx
lx

  (1,0)
```



How?

- With the help of the cache coherence protocol

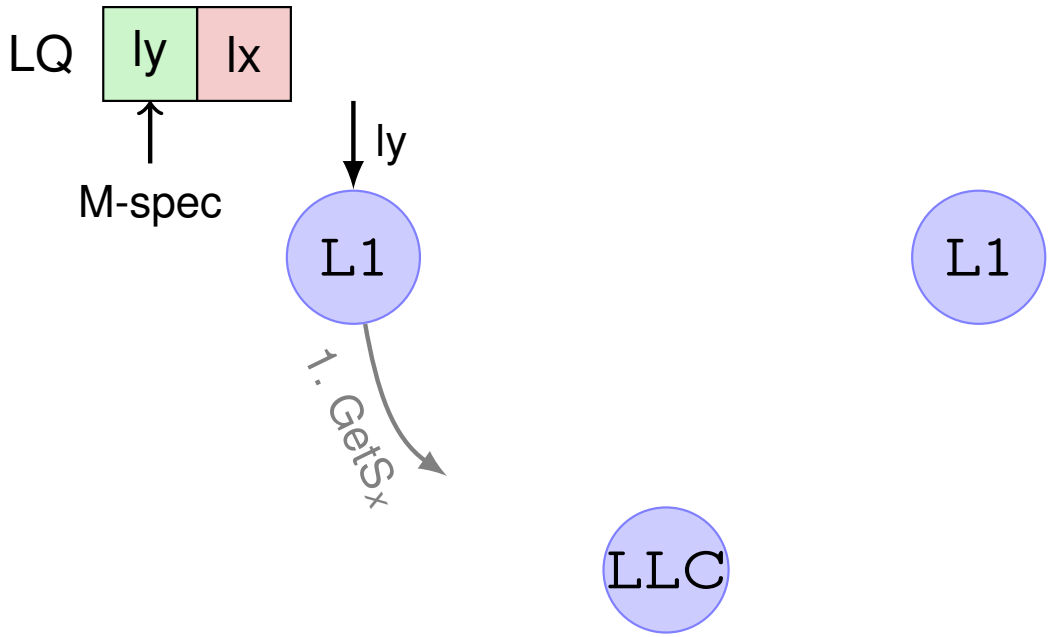
```
ly
        sy
        sx
lx

  (1,0)
```



How?

- With the help of the cache coherence protocol

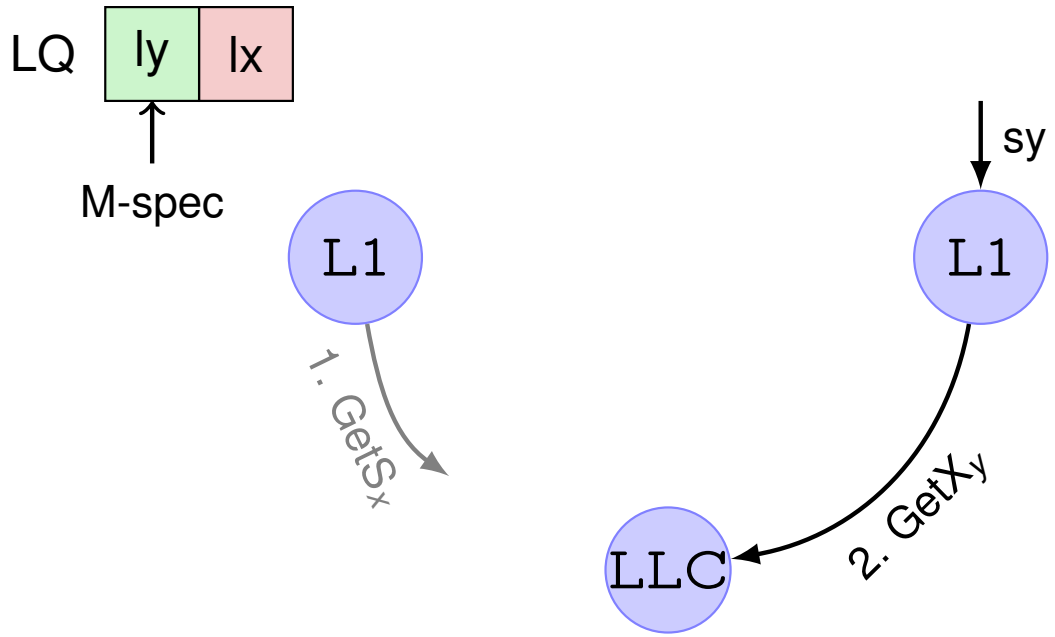
```
ly
        sy
        sx
lx

  (1,0)
```



How?

- With the help of the cache coherence protocol

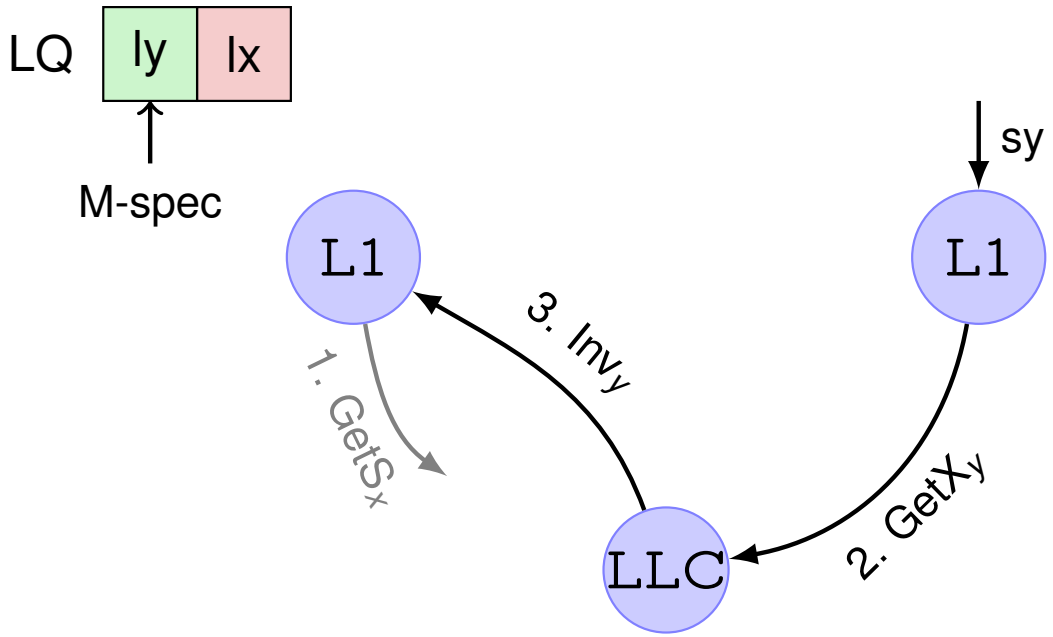
```
ly
        sy
        sx
lx

 (1,0)
```



How?

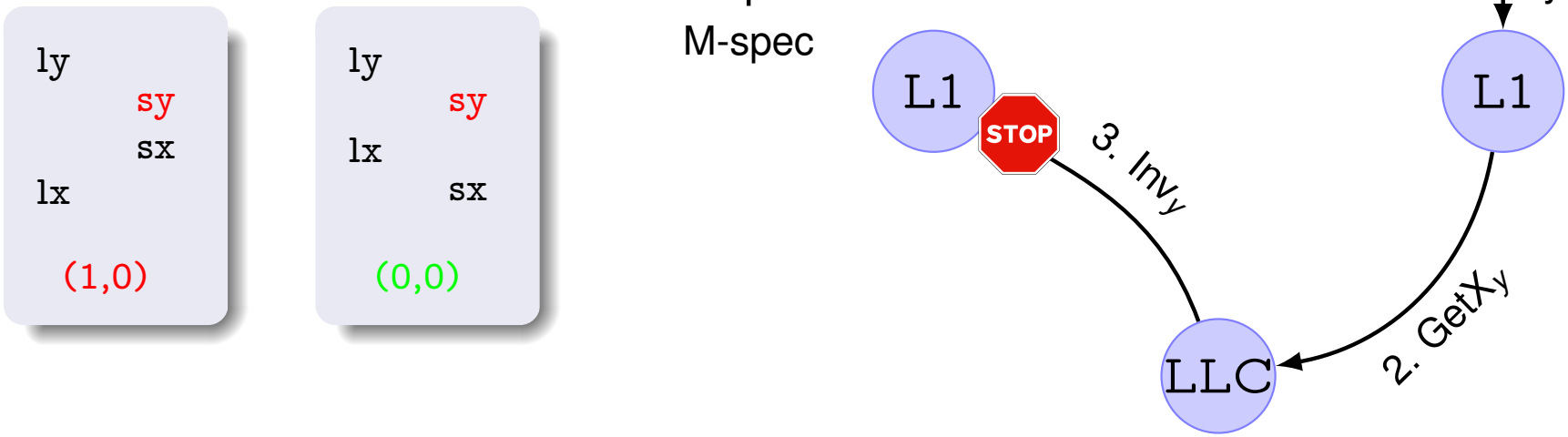
- With the help of the cache coherence protocol

```
ly
        sy
        sx
lx

 (1,0)
```



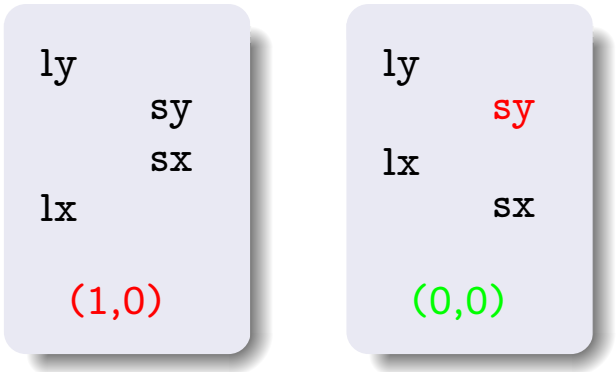
How?

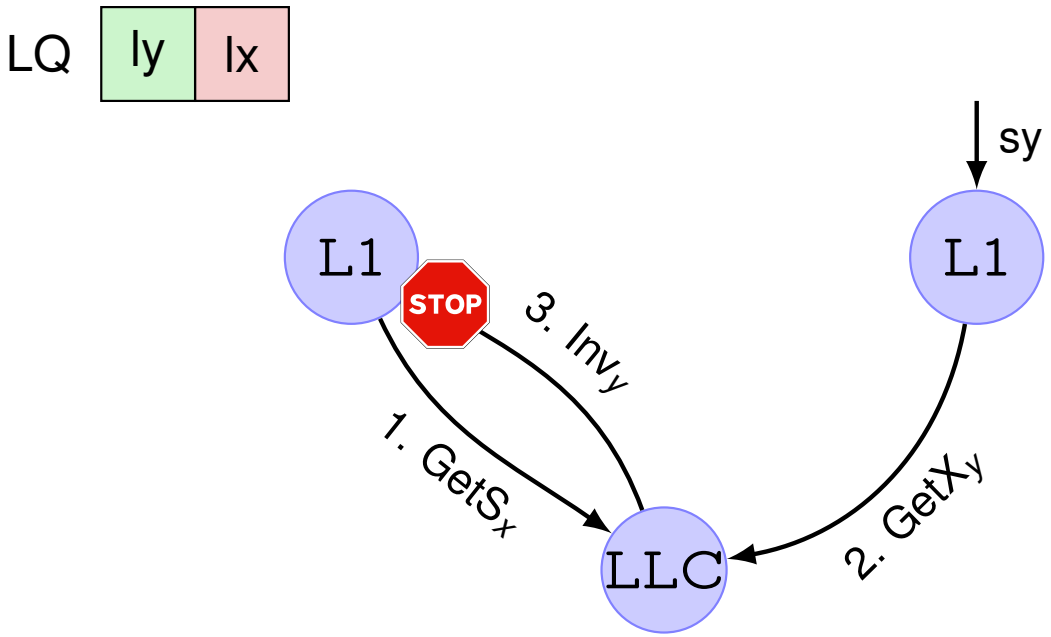
- With the help of the cache coherence protocol
  - Blocking and delaying the remote write (WritersBlock)



How?

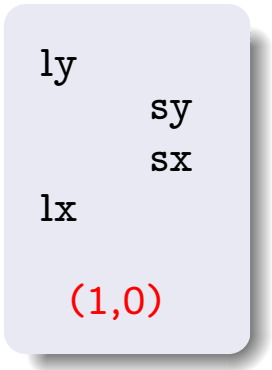
- With the help of the cache coherence protocol
  - Blocking and delaying the remote write (WritersBlock)
  - Until when? Until the load stop being M-spec

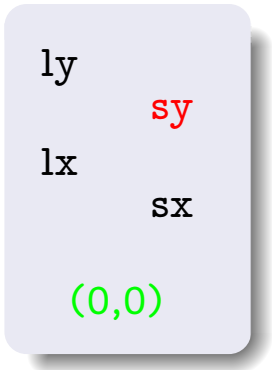


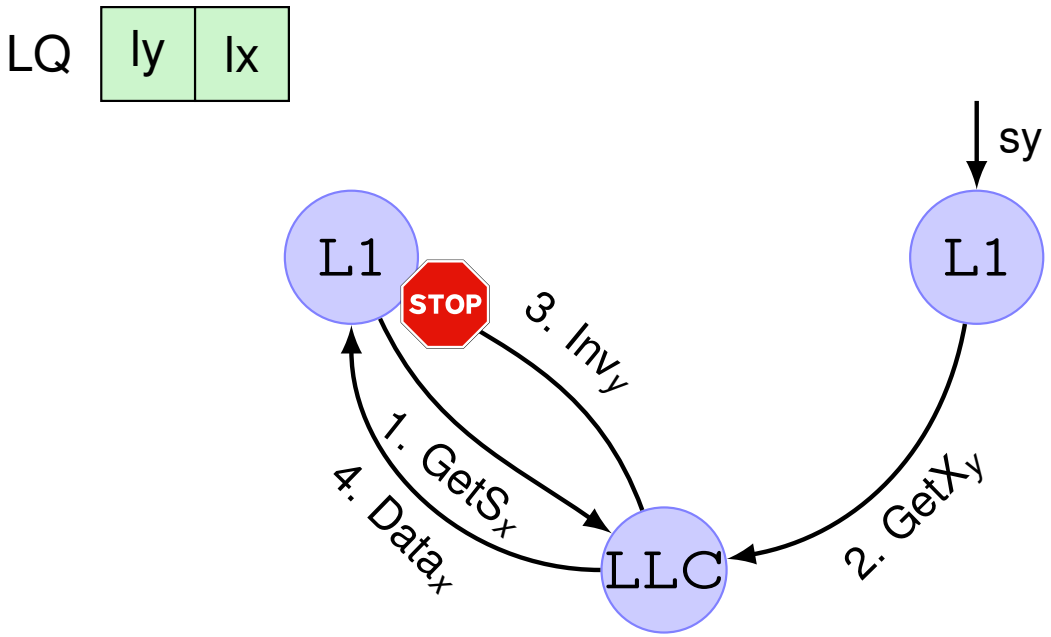


How?

- With the help of the cache coherence protocol
  - Blocking and delaying the remote write (WritersBlock)
  - Until when? Until the load stop being M-spec

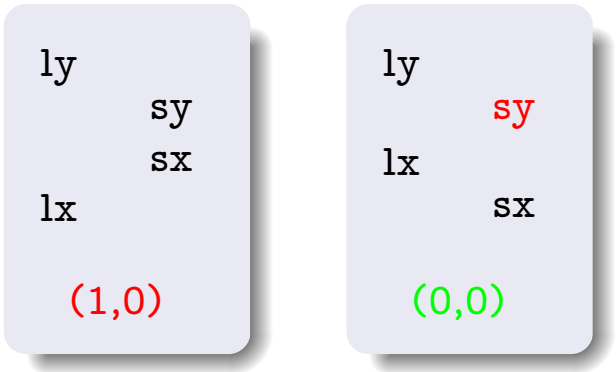


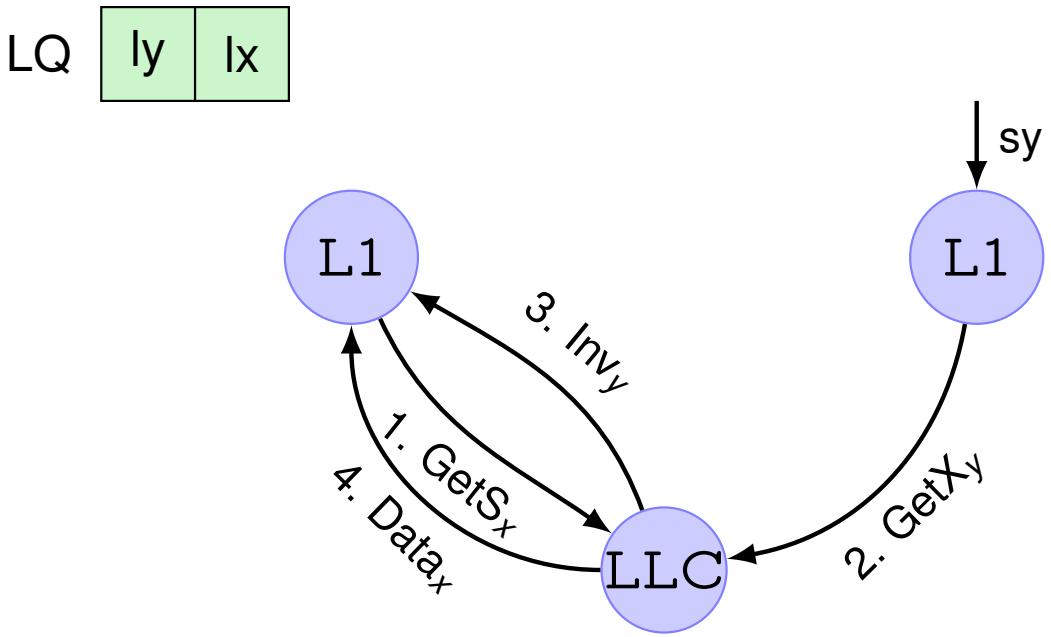




How?

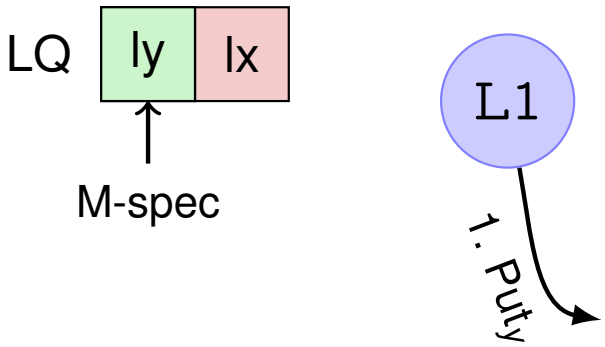
- With the help of the cache coherence protocol
  - Blocking and delaying the remote write (WritersBlock)
  - Until when? Until the load stop being M-spec





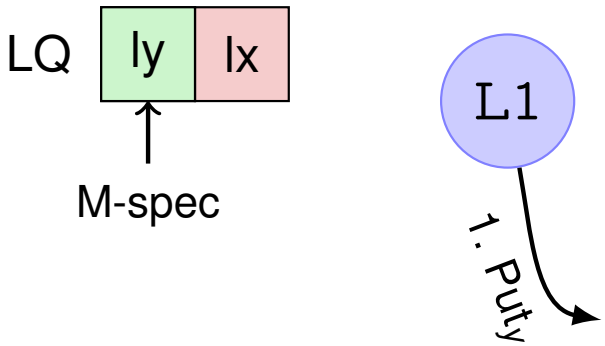
EVICTIONS

- What happens upon an eviction? Do we squash loads?



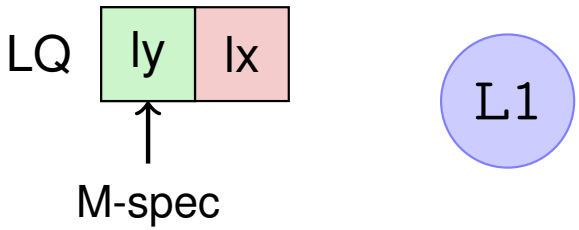
EVICTIONS

- What happens upon an eviction? Do we squash loads?
  - No, just need to guarantee that the invalidation will arrive upon a remote write



EVICTIONS

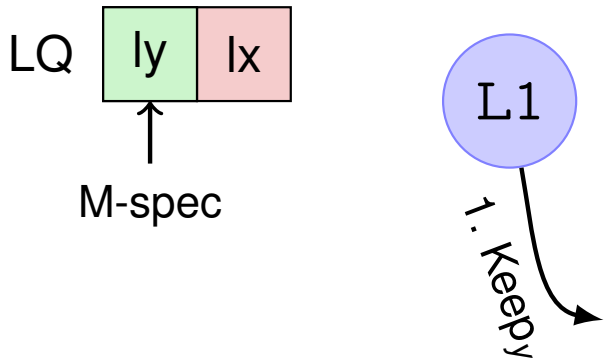
- What happens upon an eviction? Do we squash loads?
  - No, just need to guarantee that the invalidation will arrive upon a remote write
- Solution:
  - Clean blocks implement silent evictions[3]



[3] 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.

EVICTIONS

- What happens upon an eviction? Do we squash loads?
  - No, just need to guarantee that the invalidation will arrive upon a remote write
- Solution:
  - Clean blocks implement silent evictions[3]
  - Dirty blocks write back the data but the directory still keeps track

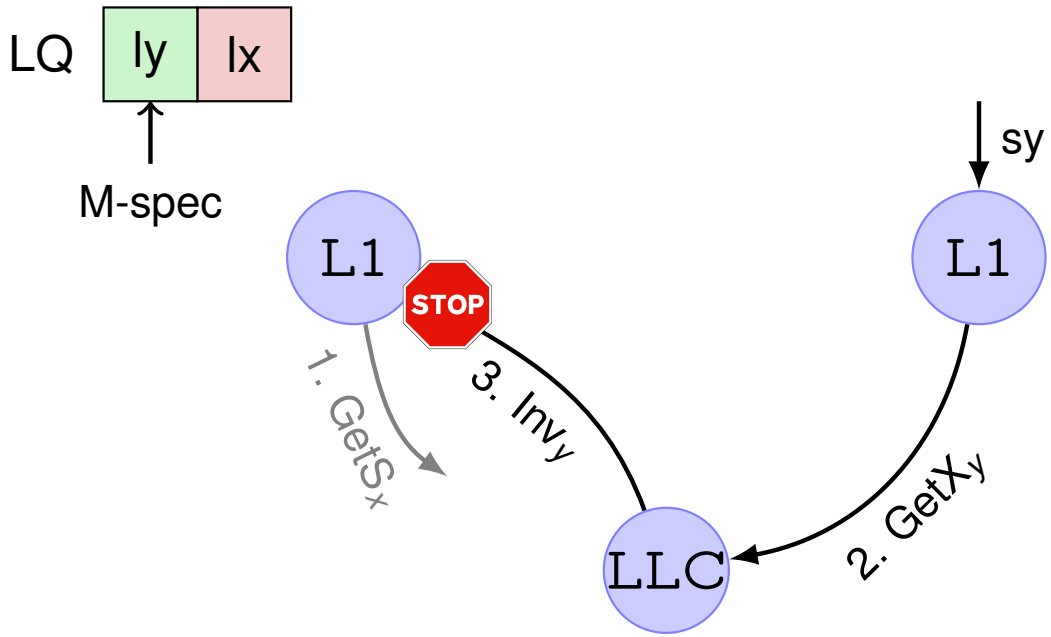


[3] R. Fernandez-Pascual, A. Ros, and M. E. Acacio, "To Be Silent or Not: On the Impact of Evictions of Clean Data in Cache-Coherent Multicores", Journal of Supercomputing, 2017.

Deadlock

- Blocking writes can cause deadlocks
  - If x and y are two words within the same cache line

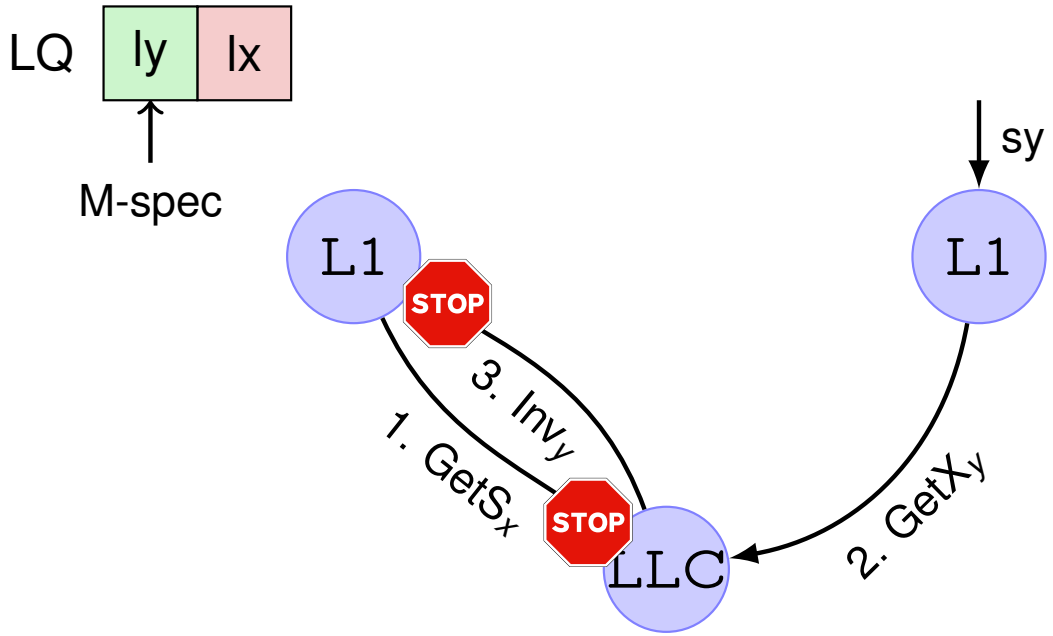
```
ly
        sy
        sx
lx

  (1,0)
```



DEADLOCK

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line

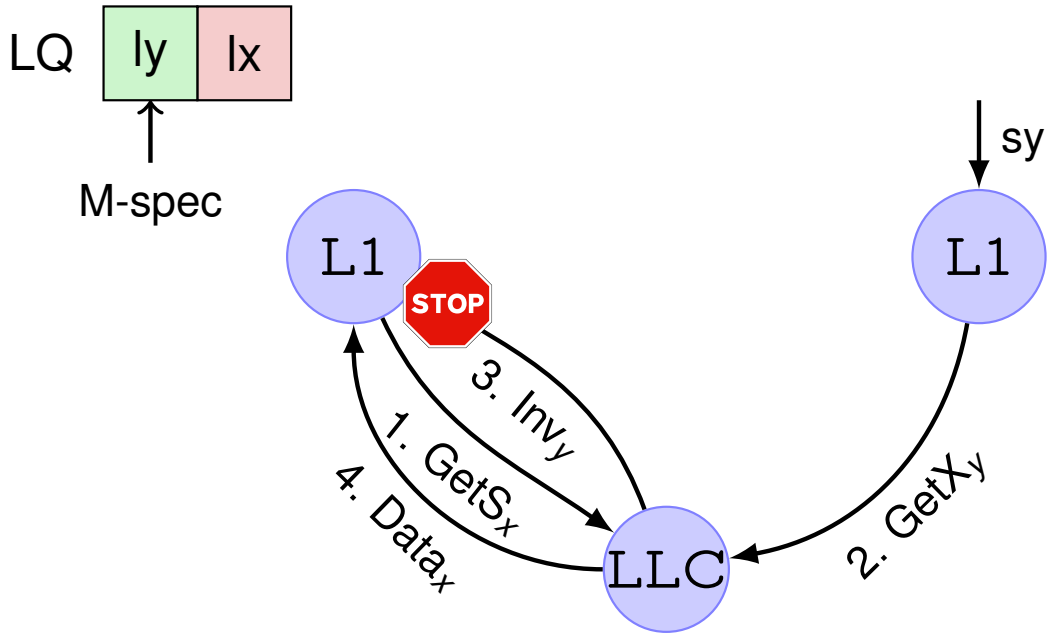
```
ly
        sy
        sx
lx

  (1,0)
```



DEADLOCK

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line
  - Solution: Blocked writes allow reads to be resolved

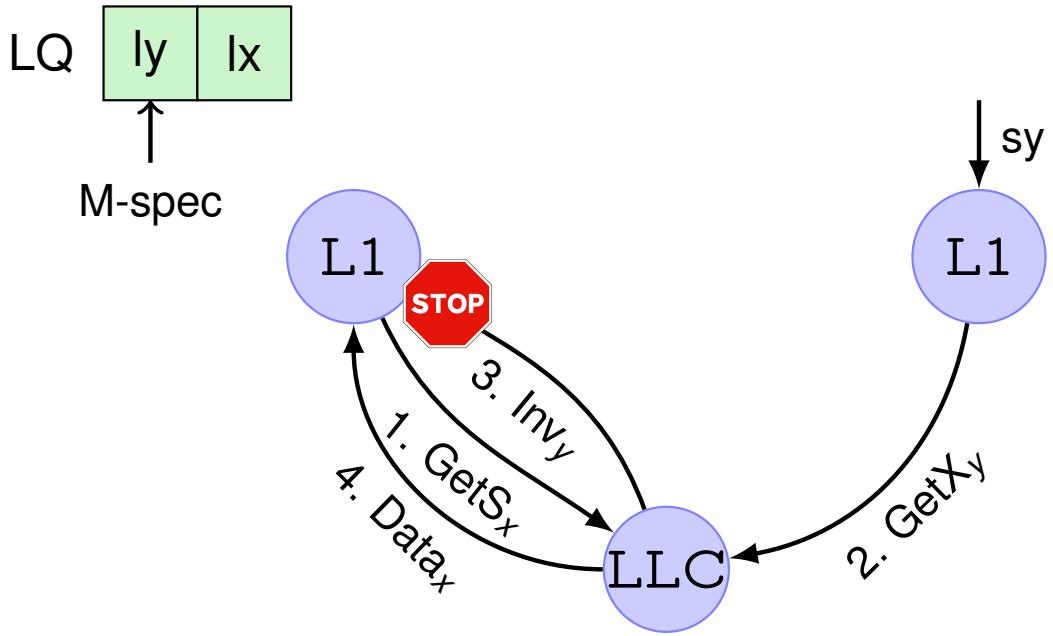
```
ly
        sy
        sx
lx

  (1,0)
```



DEADLOCK

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line
  - Solution: Blocked writes allow reads to be resolved

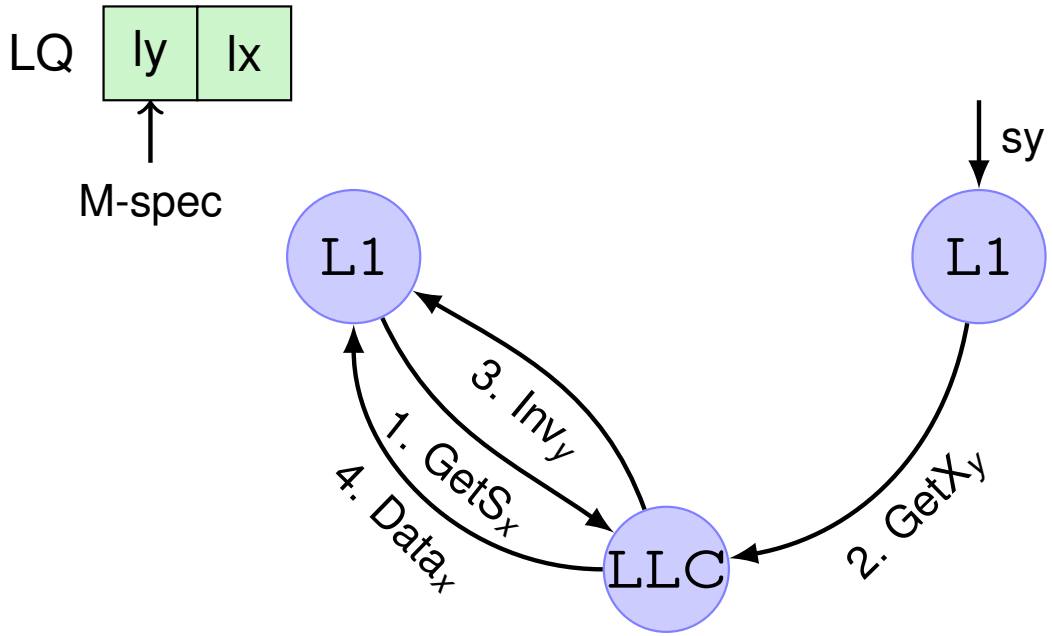
```
ly
        sy
        sx
lx

  (1,0)
```



DEADLOCK

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line
  - Solution: Blocked writes allow reads to be resolved

```
ly
        sy
        sx
lx

  (1,0)
```



LIVELOCK

- Resolving reads while blocking writes can cause livelock
  - Resolving a read once the data is invalidated will cause a second invalidation

LIVELOCK

- Resolving reads while blocking writes can cause livelock
  - Resolving a read once the data is invalidated will cause a second invalidation
- Solution
  - Reads resolved through WritersBlock must be non-cacheable
  - and cannot resolve M-spec loads (no invalidation will be received)

DEADLOCK AVOIDANCE

- WRITERSBLOCK cause writes to be blocked
  - Until a load stop being M-speculative
- Deadlocks will not happen if loads cannot be stopped by a pending write miss
- Other blocking causes:
  - MSHR address occupied by write miss $\Rightarrow$ Duplicate read-write MSHR allocation
  - Full directory/LLC $\Rightarrow$ Non-cacheable loads
  - Atomic Read-Modify-Write $\Rightarrow$ Non-speculative

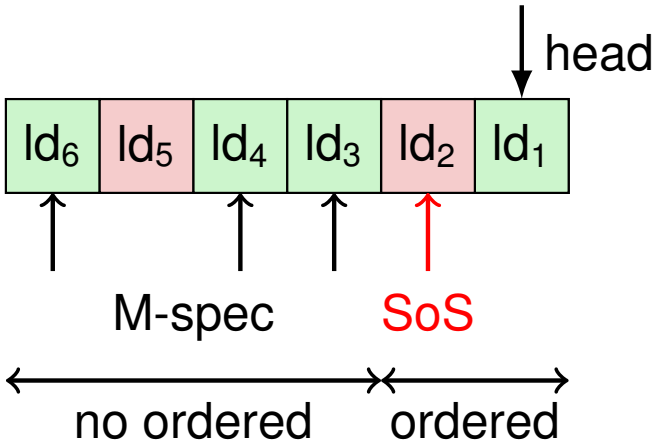
Case of use: Out-of-order commit

- Out-of-order commit[4] allows the processor to retire instructions from the reorder buffer (RoB) even if they are not at the head
- It cannot retire instructions that may squash

[4] G. B. Bell and M. H. Lipasti, “Deconstructing Commit”, ISPASS, 2004.

CASE OF USE: OUT-OF-ORDER COMMIT

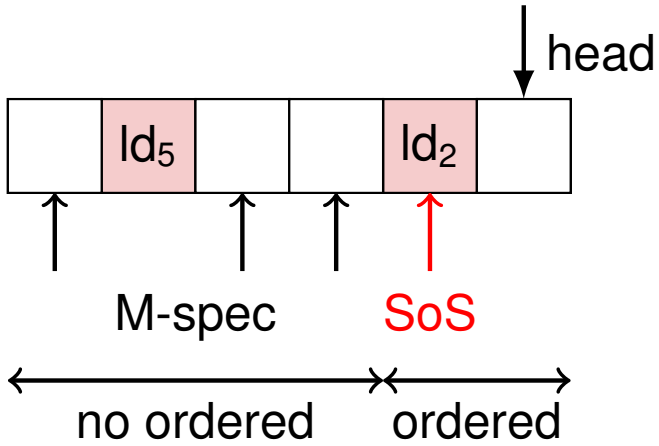
- Out-of-order commit[4] allows the processor to retire instructions from the reorder buffer (RoB) even if they are not at the head
- It cannot retire instructions that may squash
- WRITERSBLOCK allows the retirement of out-of-order loads
- Better RoB/LQ usage



[4] G. B. Bell and M. H. Lipasti, "Deconstructing Commit", ISPASS, 2004.

CASE OF USE: OUT-OF-ORDER COMMIT

- Out-of-order commit[4] allows the processor to retire instructions from the reorder buffer (RoB) even if they are not at the head
- It cannot retire instructions that may squash
- WRITERSBLOCK allows the retirement of out-of-order loads
- Better RoB/LQ usage



[4] G. B. Bell and M. H. Lipasti, "Deconstructing Commit", ISPASS, 2004.

Outline

Simulation environment

- Simulator: GEMS + OoO processor (TSO)

Simulation environment

- Simulator: GEMS + OoO processor (TSO)
- 16-core multicore
- Silvermont (32-entry RoB), Nehalem (128-entry RoB), and Haswell (192-entry RoB)

Simulation environment

- Simulator: GEMS + OoO processor (TSO)
- 16-core multicore
- Silvermont (32-entry RoB), Nehalem (128-entry RoB), and Haswell (192-entry RoB)
- Benchmarks: Splash-3[5] and Parsec-3.0

[5] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research", ISPASS, 2016.

SIMULATION ENVIRONMENT

- Simulator: GEMS + OoO processor (TSO)
- 16-core multicore
- Silvermont (32-entry RoB), Nehalem (128-entry RoB), and Haswell (192-entry RoB)
- Benchmarks: Splash-3[5] and Parsec-3.0
- Protocols
  - DIRECTORY: Directory-based MESI protocol
  - WRITERSBLOCK: Extensions to DIRECTORY

[5] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research", ISPASS, 2016.

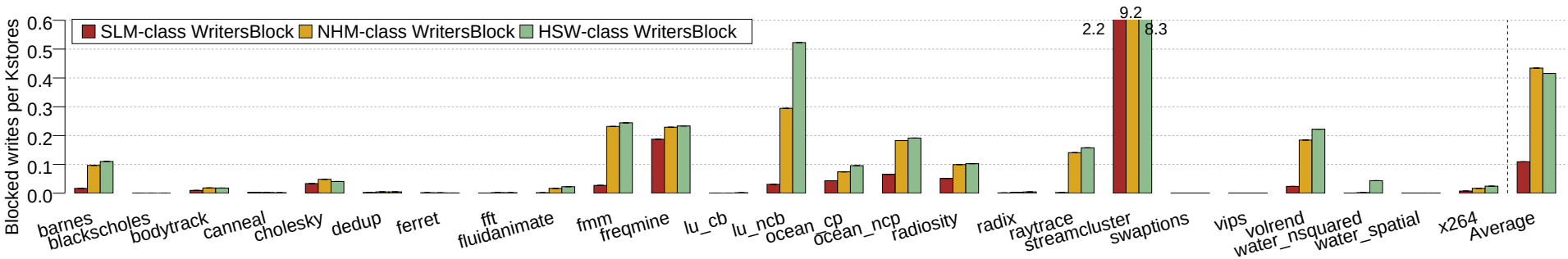
Simulation environment

- Simulator: GEMS + OoO processor (TSO)
- 16-core multicore
- Silvermont (32-entry RoB), Nehalem (128-entry RoB), and Haswell (192-entry RoB)
- Benchmarks: Splash-3[5] and Parsec-3.0
- Protocols
  - Directory: Directory-based MESI protocol
  - WritersBlock: Extensions to Directory
- Commit technique
  - InOrderCommit
  - OoOCommit

[5] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research", ISPASS, 2016.

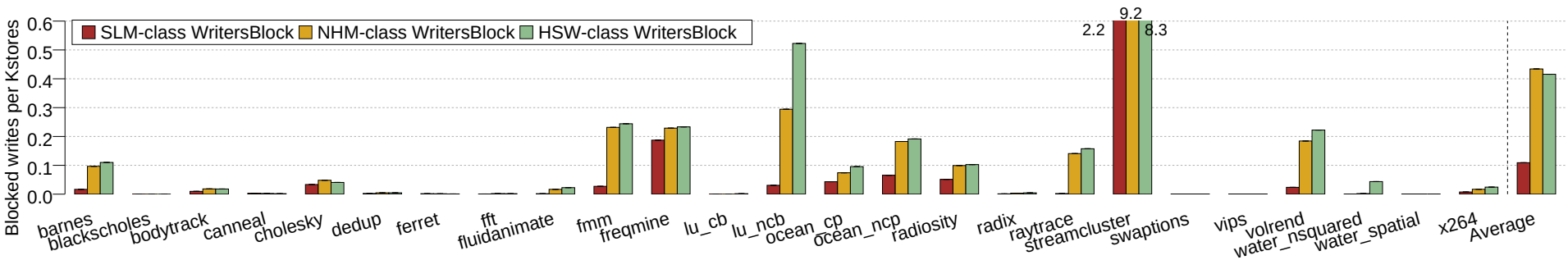
WritersBlock: Blocked writes

- Results for InOrderCommit
- Normalized to Directory



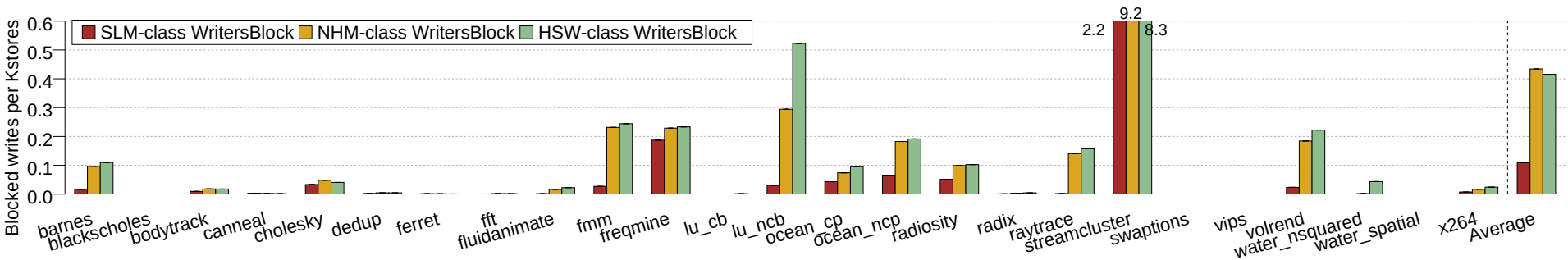
WritersBlock: Blocked writes

- Results for InOrderCommit
- Normalized to Directory
- The larger the RoB, the more misspeculations, and the more blocked writes



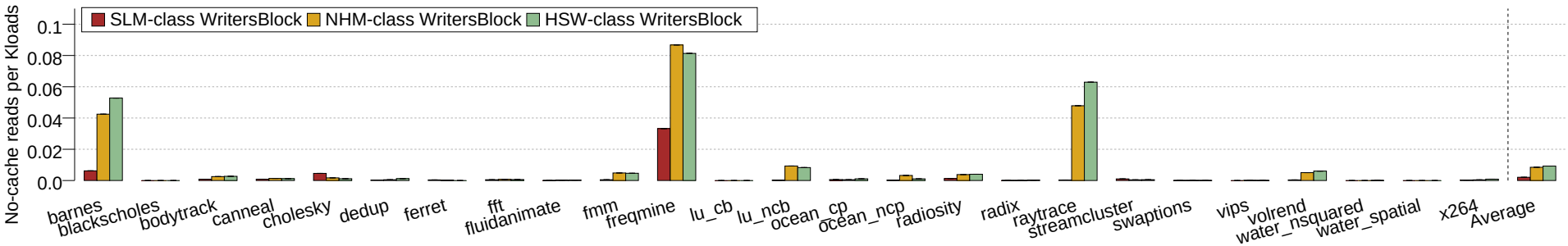
WritersBlock: Blocked writes

- Results for InOrderCommit
- Normalized to Directory
- The larger the RoB, the more misspeculations, and the more blocked writes
- Less that 5 blocks per 10,000 stores, on average



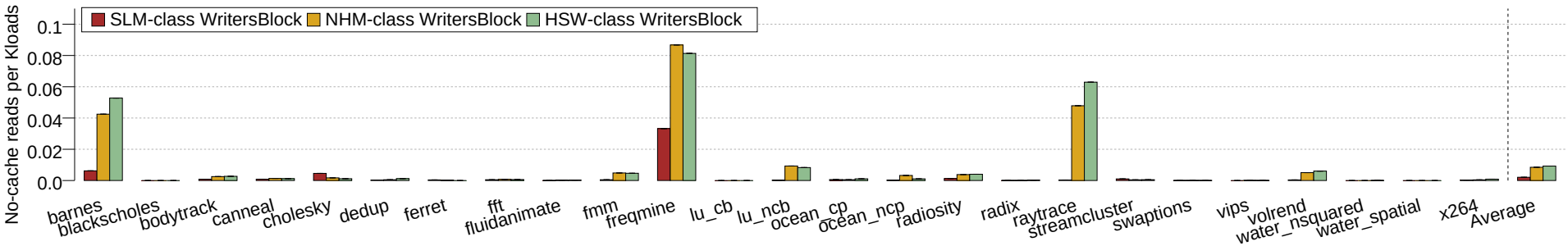
WritersBlock: Non-cacheable data

- Results for InOrderCommit
- Normalized to Directory



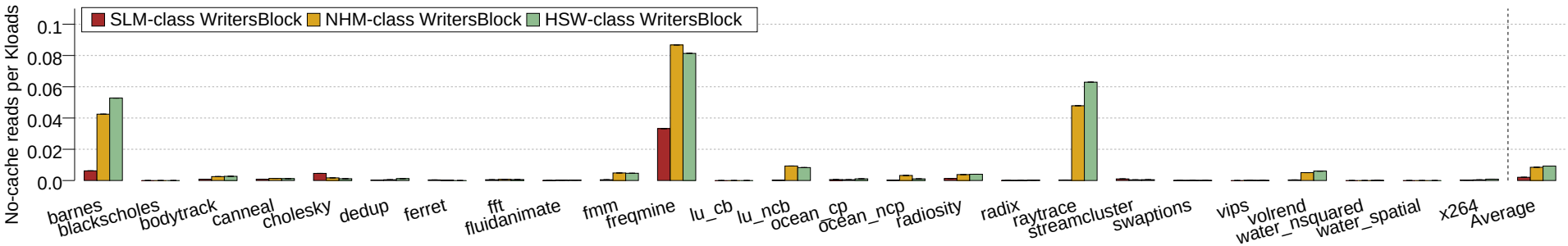
WritersBlock: Non-cacheable data

- Results for InOrderCommit
- Normalized to Directory
- The larger the RoB, the more misspeculations, and the more non-cacheable data



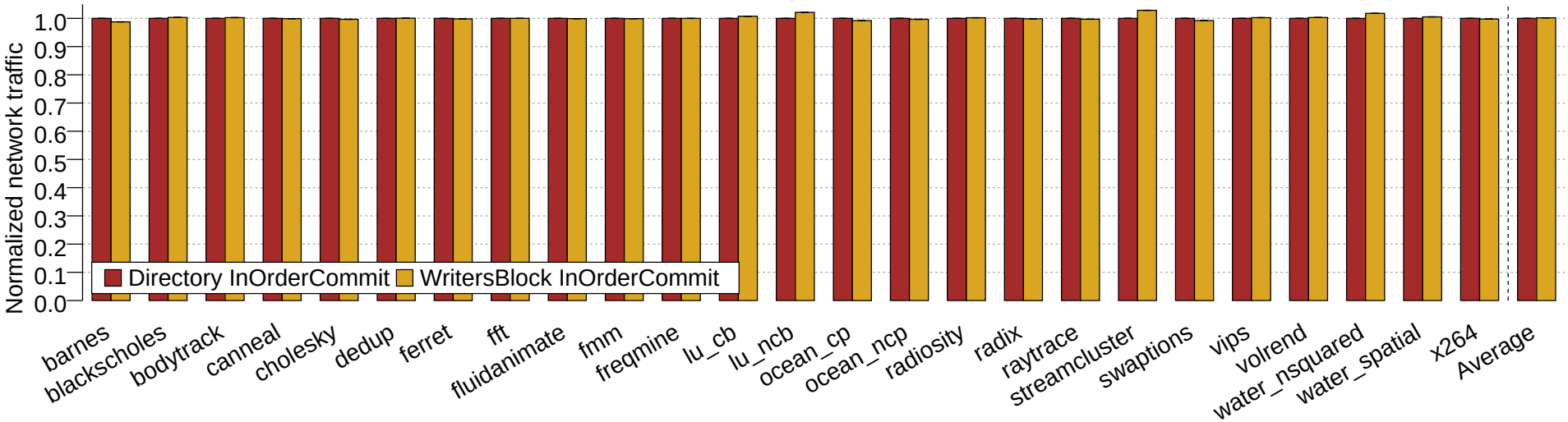
WritersBlock: Non-cacheable data

- Results for InOrderCommit
- Normalized to Directory
- The larger the RoB, the more misspeculations, and the more non-cacheable data
- $\approx$ 1 non-cacheable data per 100,000 loads, on average



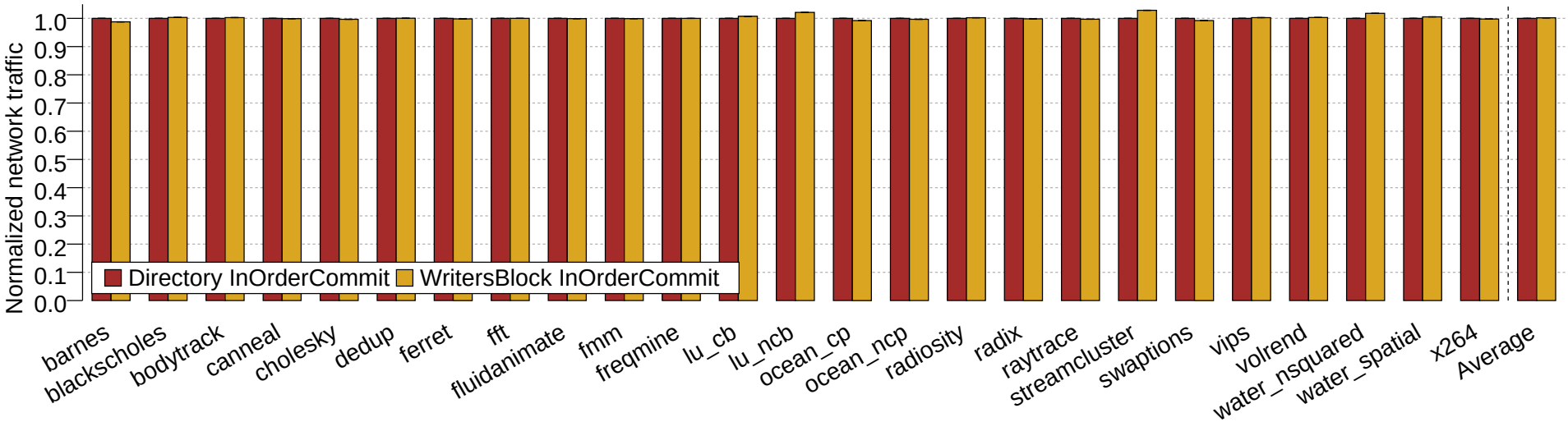
WritersBlock: Network traffic

- Results for InOrderCommit
- Normalized to Directory



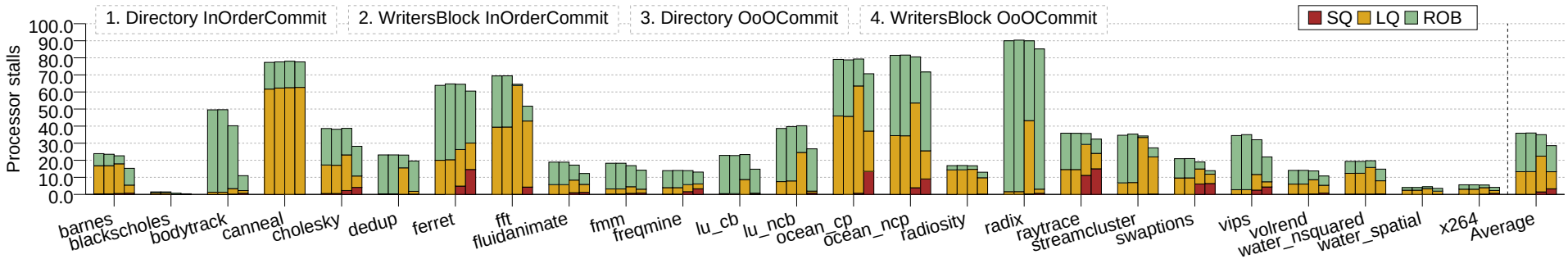
WritersBlock: Network traffic

- Results for InOrderCommit
- Normalized to Directory
- Network traffic on par



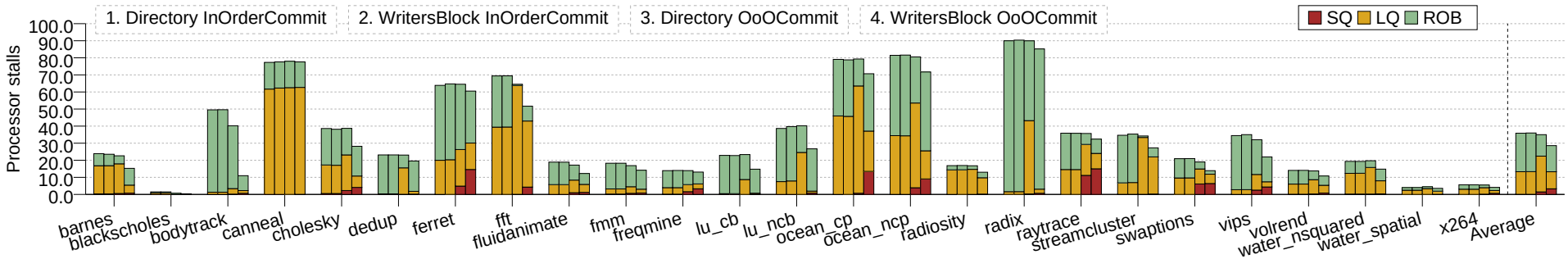
Out-of-order Commit: Processor stalls

- Normalized to DIRECTORY + INORDERCOMMIT



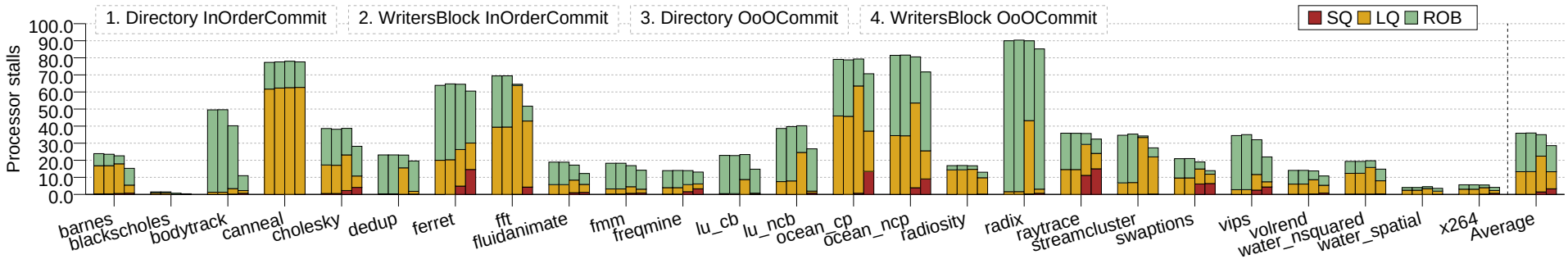
Out-of-order Commit: Processor stalls

- Normalized to Directory + InOrderCommit
- InOrderCommit
  - WritersBlock does not increases SQ stalls



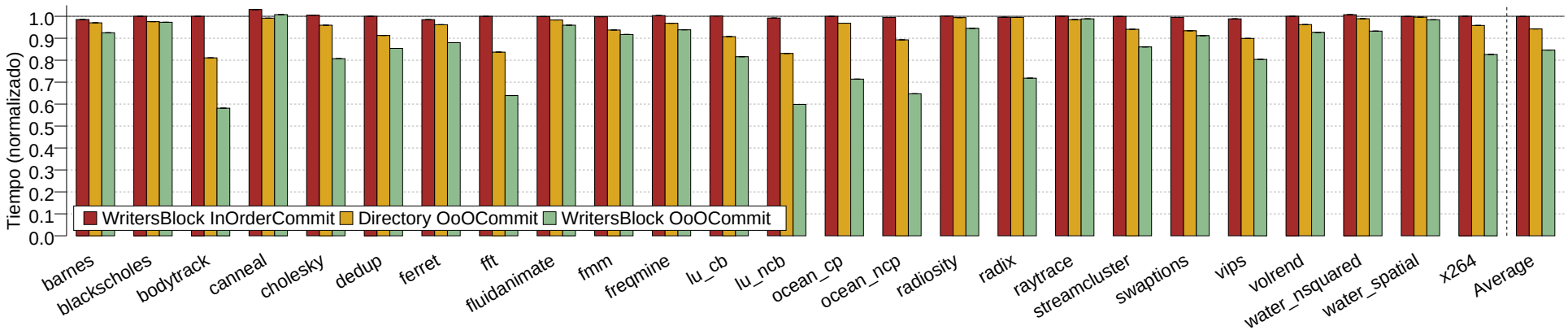
Out-of-order Commit: Processor stalls

- Normalized to Directory + InOrderCommit
- InOrderCommit
  - WritersBlock does not increases SQ stalls
- OoOCommit
  - WritersBlock reduces RoB and LQ stalls on average respect to Directory



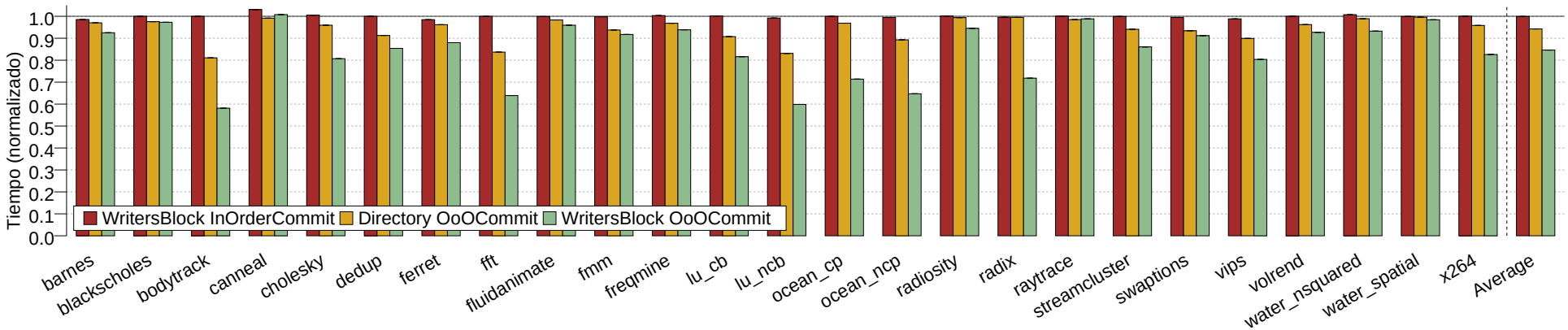
Out-of-order Commit: Execution time

- Normalized to Directory + InOrderCommit



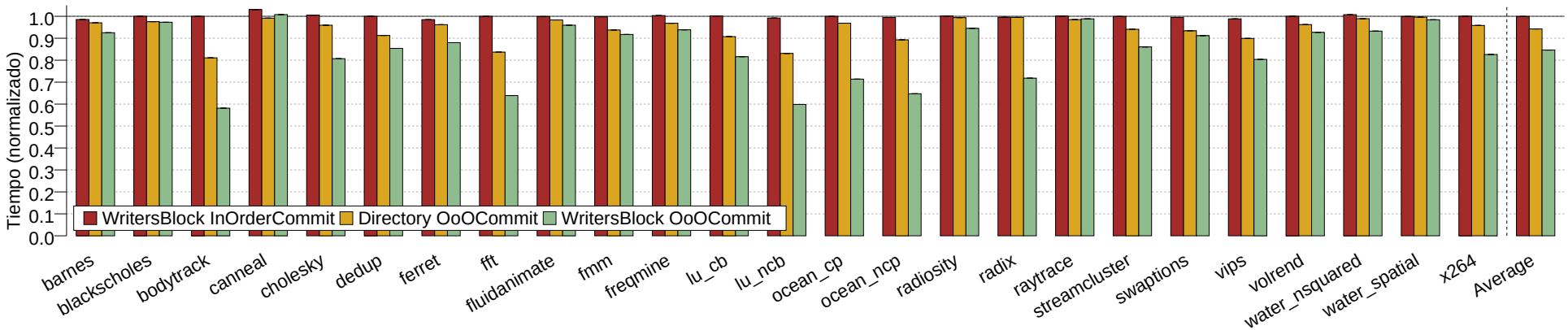
Out-of-order Commit: Execution time

- Normalized to Directory + InOrderCommit
- InOrderCommit
  - WritersBlock does not harm performance on average respect to Directory



Out-of-order Commit: Execution time

- Normalized to Directory + InOrderCommit
- InOrderCommit
  - WritersBlock does not harm performance on average respect to Directory
- OoOCommit
  - WritersBlock improves performance by 11% on average respect to Directory



Outline

Conclusions

With the help of the cache coherence protocol,
and without harming performance,
we can execute loads out of order and without speculation,
and obtaining results as if the loads were executed in order
(LOAD→LOAD)

Conclusions

With the help of the cache coherence protocol,
and without harming performance,
we can execute loads out of order and without speculation,
and obtaining results as if the loads were executed in order
(LOAD→LOAD)

Non-speculative loads can increase performance of
out-of-order commit by 11%

Non-Speculative Load→Load Reordering in TSO[1]

Alberto Ros

Universidad de Murcia

October 17th, 2017

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